ГЛАВА 15

Эксплуатация сторонних зависимостей

Не секрет, что современный софт часто строится на основе ПО с открытым исходным кодом. Такой подход практикуется даже в коммерческой сфере. Многие из крупных и прибыльных продуктов создаются на основе OOS, написанного множеством сторонних разработчиков.

Вот примеры продуктов, созданных на основе открытого исходного кода:

- Reddit (BackBoneJS, Bootstrap);
- Twitch (Webpack, Nginx);
- YouTube (Polymer);
- LinkedIn (EmberJS);
- Microsoft Office Web (AngularJS);
- Amazon DocumentDB (MongoDB).

Кроме того, многие компании теперь сами открывают исходный код своих основных продуктов, получая доход за счет поддержки или услуг, а не за счет прямой продажи. Вот некоторые примеры:

- Automattic Inc. (WordPress);
- Canonical (Ubuntu);
- Chef (Chef);
- Docker (Docker);
- Elastic (Elasticsearch);

- Mongo (MongoDB);
- GitLab (GitLab).

Веб-приложение BuiltWith анализирует другие веб-приложения, пытаясь определить, на базе каких технологий они построены (рис. 15.1).

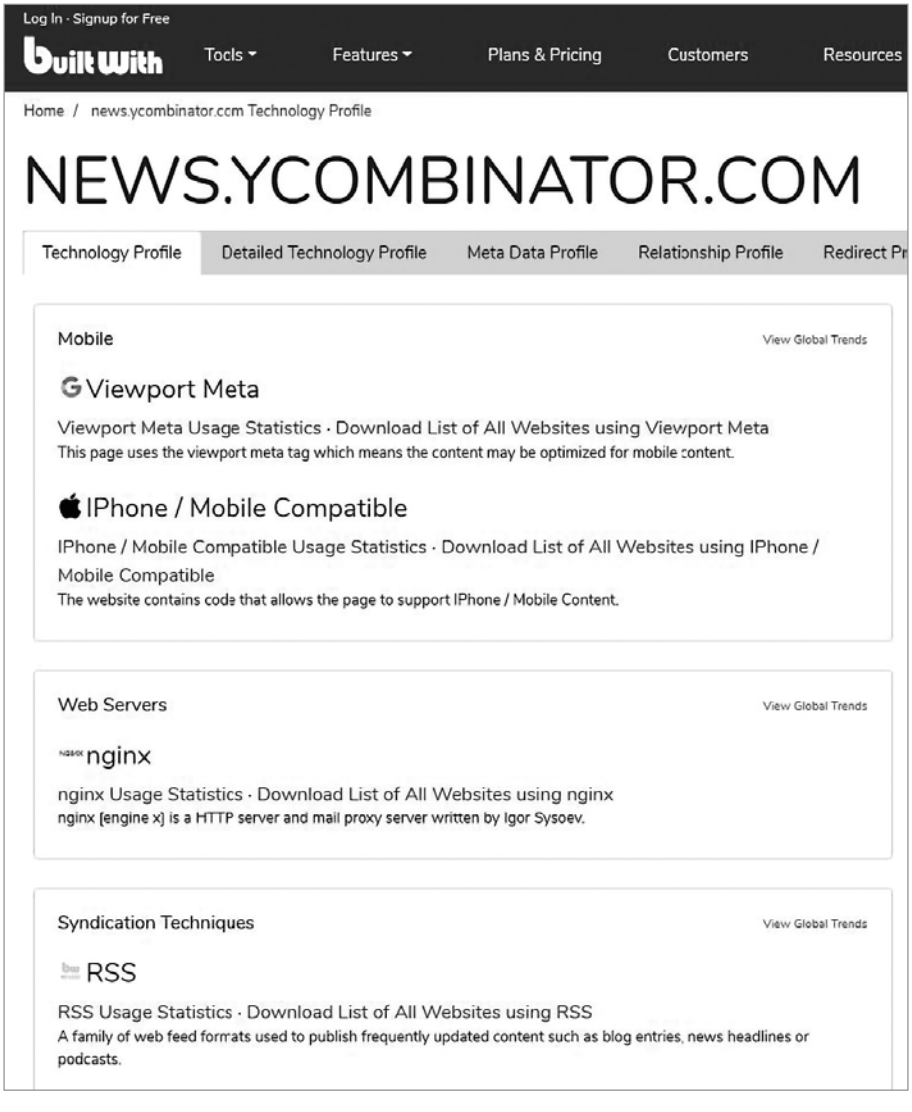

Рис. 15.1. Веб-приложение BuiltWith

Хотя применение открытого исходного кода и удобно, зачастую оно создает значительный риск для безопасности. Появившимися дырами в защите могут воспользоваться стратегически мыслящие хакеры. Существует ряд причин, по которым открытый open source может представлять угрозу безопасности вашего приложения, и обо всех них следует помнить.

Во-первых, вы полагаетесь на кодовую базу, которая вряд ли подвергалась такой же тщательной проверке, как ваш собственный код. Проводить подобные проверки для большой базы открытого кода нецелесообразно, так как для этого сначала придется увеличить количество инженеров по безопасности, а потом потратить время на глубокий анализ. Это очень дорогостоящий процесс.

Кроме того, анализ вы проведете единожды, но кодовая база открытого ПО постоянно обновляется. Поэтому в идеале потребуется оценивать безопасность каждого запроса на включение изменений. Это тоже очень дорого, поэтому большинство компаний предпочитает брать на себя риск и пользоваться относительно незнакомым софтом.

По этой причине места интеграции и разнообразные зависимости служат отличной отправной точкой для хакера. Напомню, что прочность цепи определяется ее самым слабым звеном, а в приложениях самым слабым звеном часто оказывается то, которое подвергалось наименее жесткому контролю качества.

Первым шагом при поиске зависимостей, которыми вы потенциально можете воспользоваться, является разведка. После предварительного сбора данных возможны разные варианты их применения.

Давайте подробнее рассмотрим, как происходит интеграция со сторонними программами. Только поняв ее механизм, мы получим возможность оценивать риски, которые она несет, и эксплуатировать обнаруженные уязвимости.

Методы интеграции

С точки зрения архитектуры существует несколько способов интеграции сторонних фрагментов кода с приложением.

Важно знать структуру интеграции веб-приложения со сторонними пакетами, поскольку часто именно она определяет тип перемещаемых между ними данных, метод, с помощью которого осуществляется обмен, и уровень привилегий, который дает коду стороннего пакета основное приложение.

Интеграция со сторонними пакетами настраивается разными способами. Это может быть как прямая интеграция в основной код приложения, так и запуск

стороннего кода на стороннем сервере с настройкой API для односторонней связи с основным приложением (децентрализованный подход). Каждый подход имеет свои плюсы и минусы и создает собственные проблемы для лиц, отвечающих за безопасность приложения.

Ветви и вилки

Сегодня большая часть программ с открытым исходным кодом размещена в системах контроля версий (VCS) на основе Git. Это основное отличие современных веб-приложений от более старых версий. Ведь 10 лет назад для размещения могли использовать Perforce, Subversion или даже Microsoft Team Foundation Server.

В отличие от многих устаревших VCS, Git — распределенная система управления версиями. То есть изменения вносятся не централизованно: каждый разработчик загружает собственную копию программного обеспечения и редактирует ее локально. Завершив работу над кодом в ветке основной сборки, разработчик может слить изменения в master-ветку.

Разработчики, которые берут для своих приложений программы с открытым исходным кодом, иногда создают отдельную ветку для этого ПО и запускают ее вместо основной. Это позволяет им вносить собственные изменения и одновременно втягивать чужие. Но такая модель сопряжена с рисками. Например, можно случайно перенести непроверенный код из основной ветки в свою производственную.

Больший уровень разделения предлагают вилки, поскольку они представляют собой новые репозитории, начинающиеся с последнего коммита, переданного в основную ветку до создания вилки.

В вилке можно реализовать собственную систему разрешений и собственные хуки, чтобы избежать случайного добавления в ваш код небезопасных изменений.

К сожалению, у модели вилок есть недостатки. При внедрении сторонних программ слияние кода из исходного репозитория со временем становится довольно сложным, а коммиты начинают требовать тщательного отбора. А если после создания вилки происходит серьезная реорганизация кода, коммиты из основного репозитория могут и вовсе стать несовместимыми с ней.

Приложения с собственным сервером

Иногда приложения с OOS поставляются в виде пакетов с простыми установщиками. Яркий пример — система WordPress (рис. 15.2), которая начиналась

как платформа для ведения блогов на основе PHP с широкими возможностями настройки, а теперь одним кликом ее можно установить на большинство серверов на базе Linux.

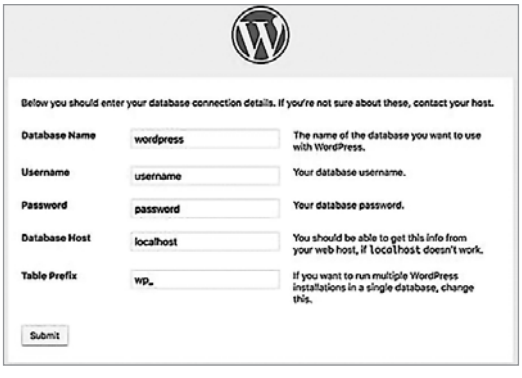

Рис. 15.2. WordPress — самая популярная CMS

Разработчики вместо исходного кода предлагают сценарий, который автоматически установит WordPress на ваш сервер. Он позволяет выбрать корректную конфигурацию базы данных и на основе нее создает файлы.

Интеграция с приложениями такого типа наиболее опасна. Может показаться, что простое ПО для ведения блога, устанавливаемое одним кликом, не может принести много проблем, но чаще всего в дальнейшем это сильно усложняет поиск и устранение уязвимостей. Чтобы определить местоположение всех файлов, требуются значительные усилия по обратному проектированию сценария установки. Поэтому от подобных программ лучше держаться подальше. Но если их применения не избежать, найдите репозиторий, в котором хранится это ПО, и тщательно проанализируйте сценарий установки и весь запускаемый в вашей системе код.

Пакеты такого типа требуют повышенных привилегий и могут легко оставить закладку для управления с удаленного доступа. Такие ситуации наносят большой ущерб организации, ведь сценарий, скорее всего, работает на ее сервере от имени администратора или пользователя с повышенным уровнем доступа.

Интеграция на уровне кода

Интеграцию программы с открытым исходным кодом в приложение можно выполнить и на уровне кода. Фактически это процедура копирования/встав-

ки, но часто она осложняется тем, что, например, крупная библиотека требует интеграции уже своих собственных зависимостей и ресурсов.

Так что в случае с крупными библиотеками OSS интеграция таким способом требует серьезной предварительной подготовки, а вот для короткого сценария из 50–100 строк такой метод, вероятно, можно считать идеальным. Для небольших утилит или вспомогательных функций такая интеграция — зачастую лучший выбор.

Для пакетов большего размера процесс не только осуществляется сложнее, но и сопряжен с большим риском. Слияния веток и вилок могут случайно привнести в ваше приложение небезопасный код из сторонней программы.

Кроме того, при интеграции путем прямого копирования кода не приходят уведомления об устраненных уязвимостях. И даже если вы узнали об этом, установка обновления может оказаться делом трудоемким и долгим.

Словом, у каждого из методов интеграции есть свои плюсы и минусы, и не существует метода, подходящего всем приложениям. Обязательно оценивайте код, который вы собираетесь интегрировать, по ряду показателей, включая его размер, цепочку зависимостей и восходящую активность в главной ветке.

Диспетчеры пакетов

В современном мире интеграция программ с открытым исходным кодом в коммерческие приложения часто происходит с помощью промежуточного приложения, называемого диспетчером пакетов. Такое приложение гарантирует, что ваше ПО будет загружать правильные зависимости из надежных источников и корректно их настраивать, обеспечивая возможность использовать их из вашего приложения на любом устройстве.

Диспетчеры пакетов полезны по ряду причин. Они позволяют абстрагироваться от сложных деталей интеграции, уменьшают начальный размер репозитория и при правильной настройке позволяют втягивать только зависимости, необходимые для текущей разработки. В небольшом приложении все эти вещи не особо нужны, но для крупного корпоративного программного пакета с сотнями зависимостей это порой экономит гигабайты пропускной способности и часы времени сборки.

Для каждого из основных ЯП существует по крайней мере один диспетчер пакетов, многие из которых следуют одним и тем же архитектурным шаблонам. У каждого крупного диспетчера пакетов есть свои особенности, средства защиты

и уязвимости. Рассмотреть все диспетчеры в рамках одной главы невозможно, поэтому проанализируем только самые популярные программы.

JavaScript

До недавнего времени экосистема разработки JavaScript (и Node.js) почти полностью основывалась на диспетчере пакетов npm (рис. 15.3).

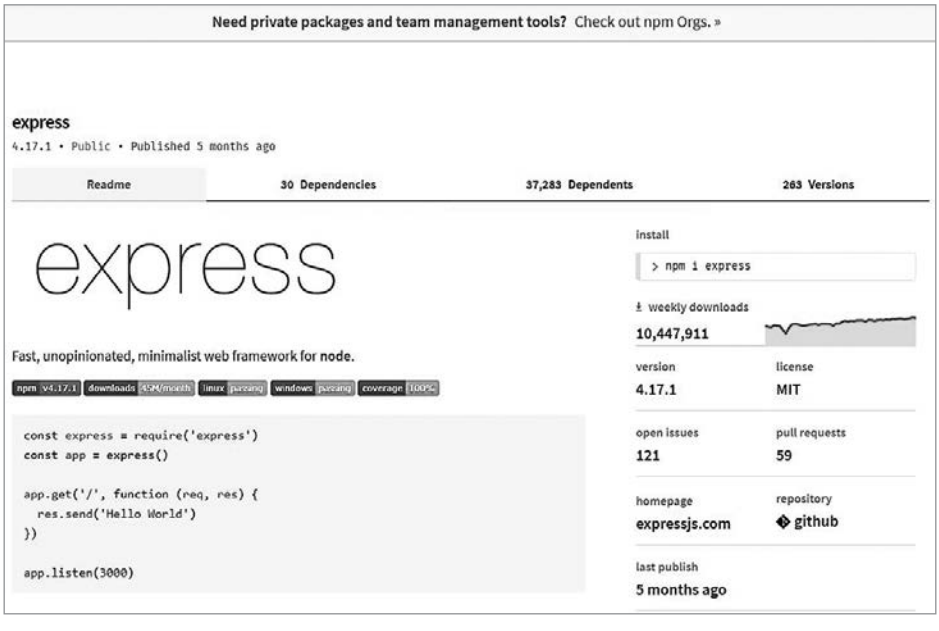

Рис. 15.3. npm — самый крупный диспетчер пакетов, написанный на JavaScript

Хотя на рынке уже появились альтернативы, npm по-прежнему поддерживает подавляющее большинство веб-приложений, написанных на JavaScript. В большинство приложений npm (https://www.npmjs.com/) встроен интерфейс командной строки для доступа к надежной базе данных библиотек с открытым исходным кодом, которые бесплатно размещаются компанией npm, Inc.

Скорее всего, вам приходилось сталкиваться с приложениями, добавляющими зависимости с помощью npm. Ключевыми признаками таких приложений являются файлы `package.json` и `package.lock` в корневом каталоге. Они сообщают интерфейсу командной строки, какие зависимости и версии нужно добавить в приложение во время сборки.

Как и большинство современных диспетчеров пакетов, npm допускает не только зависимости верхнего уровня, но и рекурсивные дочерние зависимости. Это означает, что если у добавляемой зависимости есть какая-то своя зависимость, npm внесет их во время сборки.

В прошлом слабые механизмы безопасности делали npm мишенью для злоумышленников. А так как этот диспетчер пакетов применялся довольно широко, некоторые атаки вызывали отказ работы миллионов приложений.

В качестве примера можно вспомнить `left-pad` — простую служебную библиотеку, которую поддерживает один человек. В 2016 году ее стерли из npm, что нарушило процесс сборки миллионов приложений, которые полагались на эту однострaничную утилиту. В ответ npm отменило разрешение удалять из реестра пакеты по прошествии определенного времени с момента их публикации.

В 2018 году неизвестные лица взломали разработчика популярной JavaScript-библиотеки `eslint-scope`. Злоумышленники внедрили в библиотеку вредоносный код, похищавший учетные данные пользователей. Это доказало, что библиотеки npm можно использовать в качестве векторов атаки. После инцидента компания npm увеличила объем документации по безопасности, но риск повторной компрометации учетных данных разработчика сопроводительного пакета все равно существует. И может привести к потере исходного кода компании, IP-адреса или другим проблемам.

Позже, в 2018 году, аналогичная атака произошла с библиотекой `event-stream` после добавления зависимости `flatmap-stream`. Эта зависимость содержала вредоносный код для кражи биткоин-кошельков. В результате были украдены кошельки многих пользователей, которые, сами того не зная, полагались на библиотеку `flatmap-stream`.

Как видите, диспетчер пакетов npm во многих отношениях готов к эксплуатации уязвимостей. Он представляет значительный риск для безопасности, поскольку на уровне исходного кода практически невозможно оценить каждую зависимость и подзависимость большого приложения.

Простая интеграция бесплатного пакета npm в коммерческое приложение может стать вектором атаки, способным привести к полной компрометации IP-адреса компании или к еще более тяжелым последствиям.

Я привожу эти примеры только для того, чтобы можно было правильно оценить и снизить такие риски. Если вы хотите использовать библиотеки npm для поиска уязвимостей в бизнес-приложениях, делайте это только при наличии письменного разрешения владельцев и только на базе сценария тестирования в стиле Red Team.