



BUG HUNTER

От первых шагов до профессионального Bug Bounty

ZYCT

Практика Web Security

ZYCT 10

**Bug Hunter: От первых шагов до
профессионального Bug Bounty**

«Автор»

2026

10 Z.

Bug Hunter: От первых шагов до профессионального Bug Bounty /
Z. 10 — «Автор», 2026

BUG HUNTER. От первых шагов до профессионального Bug Bounty. Практическое руководство по безопасности веб-приложений и Bug Bounty: от основ — сети, Linux, HTTP — до систематического поиска уязвимостей, собственных инструментов на Python и профессиональных отчётов. Каждый класс разбирается по единой схеме: механизм обнаружения, подтверждение, влияние, исправление, с реальными запросами и кодом. Вся практика построена вокруг безопасной локальной лаборатории HunterMart, где каждую проблему можно найти в уязвимом режиме и убедиться, что она закрыта в исправленном. Вы разберёте IDOR, SQL-инъекции, XSS, SSRF, mass assignment, гонки, OAuth/JWT и другие классы из OWASP Top 10:2025, научитесь оценивать серьёзность по CVSS 4.0. Книга честна: без обещаний лёгкого дохода, только о легальном исследовании. Для начинающих, разработчиков и специалистов AppSec.

© 10 Z., 2026

© Автор, 2026

Содержание

BUG HUNTER	7
Об этой книге	8
Как читать	9
Правовые и этические рамки	10
Глава 1. Введение в Bug Bounty	11
1.1. Что такое Bug Bounty	12
1.2. Место исследователя в экосистеме безопасности	13
1.3. Bug Bounty, VDP, пентест, red team, vulnerability research	14
1.4. Как на самом деле выглядит работа	15
1.4.1. Как выглядит типичная сессия исследования	15
1.5. Жизненный цикл находки	16
1.6. Выбор программы: первая развилка	18
1.6.1. Экономика находок: честная картина	18
1.6.2. Платформы и прямые программы	18
1.7. Этика и легальность как фундамент	20
1.8. Как пользоваться этой книгой	21
1.8а. Задания (уровни 1–3)	22
1.9. Резюме	23
1.10. Контрольные вопросы	24
1.11. Источники	25
Глава 2. Правила программ, границы и этика	26
2.1. Почему правила — это главное	27
2.2. Как читать политику программы	28
2.2.1. Score: разрешённое не значит «всё внутри»	28
2.2.2. Типичные out-of-score находки	28
2.2.3. Worked-пример: разбор scope	28
2.3. VDP и Bug Bounty: разные обещания	30
2.4. Safe harbor и его пределы	31
2.5. Принципы безопасного исследования	32
2.5.1. Опасные классы — особые меры	32
2.6. Инструменты с техническими границами	33
2.7. Раскрытие и репутация	34
2.7.1. Worked-пример: типичная временная линия раскрытия	34
2.8. Практикум	35
2.8а. Дополнительные задания (уровни 1–5)	36
2.9. Анализ ошибок новичка	37
2.10. Защитные меры (для владельца программы)	38
2.11. Резюме	39
2.12. Контрольные вопросы	40
2.13. Источники	41
Глава 3. HTTP, браузер, cookies, сессии, TLS и веб-архитектура	42
3.1. Архитектура веб-приложения	43
3.1.1. Клиент-серверная модель и SPA	43
3.2. HTTP: протокол запрос–ответ	44
3.2.1. Анатомия обмена	44

3.2.2. Методы	44
3.2.3. Коды состояния	45
3.3. Заголовки и безопасность	46
3.3.1. Worked-пример: разбор CORS	46
3.4. Строка запроса, тело и форматы данных	48
3.5. Cookies и сессии	49
3.5.1. Worked-пример: полный цикл аутентифицированной сессии	49
3.6. Редиректы	52
3.7. Версии протокола	53
3.7.1. Worked-пример: почему стыки версий опасны	53
3.9.1. Worked-пример: ключ кэша и его рассинхрон	53
3.8. TLS: защита канала	54
3.9. Кэширование	55
3.10. Практикум	56
3.10а. Дополнительные задания (уровни 1–8)	57
3.11. Анализ ошибок новичка	58
3.12. Защитные меры (для разработчика)	59
3.13. Резюме	60
3.14. Контрольные вопросы	61
3.15. Источники	62
Глава 4. Linux, командная строка, Git и Python для исследователя	63
4.1. Почему именно Linux	64
4.2. Терминал и оболочка	65
4.3. Файловая система и права	66
4.4. Процессы, окружение, пакеты	67
4.5. Обработка текста: grep, sed, awk, jq	68
4.5.1. Worked-пример: разбор лога запросов	68
4.8.1. Worked-пример: Python для анализа JSON-ответа	69
4.6. Сетевая диагностика из терминала	71
4.7. Git: учёт экспериментов	72
4.7.1. Worked-пример: Git-рабочий процесс исследователя	72
4.8. Python для исследователя: основы	73
4.9. Изолированное окружение и зависимости	75
4.10. Тестирование кода	76
4.11. Практикум	77
4.11а. Дополнительные задания (уровни 1–6)	78
4.12. Анализ ошибок новичка	79
4.13. Резюме	80
4.14. Контрольные вопросы	81
4.15. Источники	82
Конец ознакомительного фрагмента.	83

ЗУСТ 10

Bug Hunter: От первых шагов до профессионального Bug Bounty

BUG HUNTER — От первых шагов до профессионального Bug Bounty
ЗУСТ

BUG HUNTER

От первых шагов до профессионального Bug Bounty

Автор: ZYCT

Об этой книге

Это практическое руководство по безопасности веб-приложений и Bug Bounty. Оно ведёт читателя от основ (сети, Linux, HTTP, устройство веб-приложений) к систематическому поиску уязвимостей в **разрешённых** целях, разработке собственных инструментов на Python и написанию профессиональных отчётов.

Весь практический материал рассчитан на изолированную локальную лабораторию — учебное приложение **HunterMart** — и на цели, тестировать которые у читателя есть явное разрешение. Книга не содержит инструкций по атаке на чужие системы и не обещает гарантированного дохода.

Лаборатория HunterMart, инструменты на Python и дополнительные файлы к книге доступны в репозитории: <https://github.com/ZYCT10/BugHunterBook/>

Как читать

Новичок: главы 1–5 подряд (основы, этика, HTTP, Linux/Python, разведка), затем классы уязвимостей (главы 6–11) по одному, с лабораториями.

Разработчик: главы 1–2, классы уязвимостей с разделами «Защитные меры» (главы 6–11), автоматизация на Python (глава 12).

Практикующий исследователь: разведка (глава 5), классы уязвимостей (главы 6–11), автоматизация (глава 12), триаж и отчётность (главы 13–14), лаборатории (глава 15) — как справочник и углубление.

Уровни сложности заданий обозначены от 1 (понимание терминов) до 8 (профессиональный отчёт).

Правовые и этические рамки

Любое практическое действие выполняется только: в локальной лаборатории, в специально предназначенной учебной среде, на собственных системах либо в пределах явно разрешённого scope программы. Перед началом работы пройдите «Чек-лист безопасного тестирования» (приложение). Ответственность за применение знаний несёт читатель.

Глава 1. Введение в Bug Bounty

Цель главы. Дать точную картину того, что такое Bug Bounty и исследование безопасности веб-приложений: кто этим занимается, как устроены роли и форматы работы, как выглядит реальный процесс и экономика находок, и какие ожидания реалистичны. Эта глава закладывает словарь и систему координат, которыми мы пользуемся до конца книги.

Предварительные требования. Базовое умение пользоваться компьютером. Специальных знаний по безопасности не требуется.

Результаты обучения. После главы вы сможете: (1) объяснить, чем Bug Bounty отличается от пентеста, red team, VDP и vulnerability research; (2) описать полный жизненный цикл находки — от выбора программы до выплаты; (3) трезво оценивать экономику и трудности этого занятия; (4) понимать, какое место занимает исследователь в экосистеме безопасности.

1.1. Что такое Bug Bounty

Bug Bounty (программа вознаграждения за уязвимости) — это публичное приглашение владельца системы: «найдите у нас уязвимости по этим правилам, сообщите нам, и мы заплатим за подтверждённые находки». Исследователь (**bug hunter**, или **researcher**) ищет уязвимости в разрешённых целях, оформляет отчёт, а владелец — как правило, через команду **триажа (triage)** — проверяет находку, оценивает её и выплачивает вознаграждение (**bounty**).

Ключевое слово — **разрешение**. Bug Bounty легален именно потому, что владелец заранее и публично разрешил тестирование в оговорённых границах. То же действие без разрешения — правонарушение. Мы возвращаемся к границам, score и «тихой гавани» в главе 2; здесь важно зафиксировать принцип: нет разрешения — нет исследования.

Исторически модель выросла из простого наблюдения: у любой достаточно сложной системы есть уязвимости, которых не видят её создатели, а «свежий взгляд» множества внешних исследователей находит то, что пропустили внутренние процессы. Вместо того чтобы ждать, пока уязвимость найдёт злоумышленник, организация платит добросовестным исследователям за то, что они находят её первыми и сообщают приватно.

Воспринимайте Bug Bounty не как «взлом за деньги», а как возмездный аудит по правилам. Эта рамка убережёт и от юридических проблем, и от выгорания: вы не воюете с системой, вы помогаете закрыть в ней дыры и получаете за это вознаграждение.

1.2. Место исследователя в экосистеме безопасности

Чтобы понять, где находится bug hunter, разведём несколько ролей и дисциплин. Они часто пересекаются, но образ мышления и рамки у них разные.

Оборонительная безопасность (Defensive Security, «blue team»). Построение и эксплуатация защиты: безопасная разработка, мониторинг, обнаружение и реагирование на инциденты. Их метрика — устойчивость.

Наступательная безопасность (Offensive Security, «red team»). Моделирование действий реального противника, чтобы найти слабые места раньше него. Метрика — насколько глубоко удалось продвинуться.

Безопасность приложений (Application Security, AppSec). Дисциплина на стыке разработки и безопасности: моделирование угроз, ревью кода, тестирование, исправления на уровне конкретного приложения. Многие исследователи приходят в Bug Bounty из AppSec и наоборот.

Bug hunter. Работает на стыке offensive и AppSec, но всегда в легальных рамках программы. Находит, подтверждает, документирует и помогает исправить.

Ценность исследователя удваивается, когда он понимает не только как сломать, но и как починить. Поэтому в этой книге за разбором каждой уязвимости следует раздел с защитными мерами и исправленным кодом: это не «довесок для галочки», а часть профессии.

1.3. Bug Bounty, VDP, пентест, red team, vulnerability research

Эти форматы постоянно путают. Они различаются тем, кто инициирует работу, насколько широки и публичны границы, как устроена оплата и какова главная цель.

Формат	Кто инициирует	Границы (scope)	Оплата	Главная цель
Bug Bounty	Владелец публикует программу	Опубликованы, часто широкие	За подтверждённые находки	Поток находок от многих исследователей
VDP	Владелец публикует политику	Опубликованы	Обычно без денег, с safe harbor	Легальный канал для сообщений
Пентест	Заказчик нанимает исполнителя	Жёстко оговорён в договоре	Фиксированная (контракт)	Оценка защищённости за отведённое время
Red team	Крупная организация	Широкий, скрытый	Контракт	Проверка систем И обнаружения/реагирования
Vulnerability research	Исследователь/вендор	Продукт или класс технологий	Разное (гранты, CVE, репутация)	Поиск ранее неизвестных классов уязвимостей

Несколько важных различий подробнее.

Bug Bounty против пентеста. Пентест ограничен временем и людьми: нанятая команда за, скажем, две недели проходит оговорённый периметр и сдаёт отчёт. Bug Bounty — непрерывный процесс с неограниченным числом исследователей, которые смотрят на систему годами и под разными углами. Пентест даёт полноту за фиксированную цену; Bug Bounty даёт «длинный хвост» нестандартных находок и платит только за результат.

Bug Bounty против red team. Red team проверяет не только систему, но и способность организации **замечить** и **остановить** атаку; это скрытная операция с имитацией конкретного противника. В Bug Bounty скрытность не нужна — наоборот, вы действуете открыто и по правилам.

VDP (Vulnerability Disclosure Program). Это «дверь» для сообщений: «если вы что-то нашли, сообщите сюда безопасно, и мы не станем вас преследовать». Часто без денег. Bug Bounty можно считать VDP с вознаграждением. Понятие **safe harbor** («тихая гавань») — раздел политики, где владелец обязуется не инициировать преследование добросовестного исследователя, действующего в рамках правил. Его читают дословно (глава 2).

Vulnerability research. Поиск ранее неизвестных уязвимостей или целых их классов — часто в продуктах, библиотеках, протоколах. Результат может оформляться как **CVE (Common Vulnerabilities and Exposures)** — запись в публичном каталоге уязвимостей. Это соседняя дисциплина; многие приёмы общие.

1.4. Как на самом деле выглядит работа

Разберём честно, без романтики, потому что завышенные ожидания — главная причина, по которой новички бросают занятие в первые месяцы.

Это не «запустил сканер — получил деньги». Автоматические сканеры находят шум: тысячи «потенциальных» проблем, из которых подавляющее большинство — ложные срабатывания или находки нулевой ценности. Настоящая работа — это проверка гипотез о логике приложения, а её сканер не делает. Мы посвящаем мышлению исследователя главу 5 и возвращаемся к нему в каждой главе части про уязвимости.

Конкуренция высокая. Популярные программы изучают тысячи людей одновременно. Частая судьба отчёта — **дубликат (duplicate)**: находка валидна, но кто-то сообщил о ней раньше, и вознаграждения не будет. Это нормально и случается даже с опытными исследователями.

Много ложных срабатываний. Значительная часть времени уходит на то, чтобы отсеять то, что лишь выглядит уязвимостью. Умение уверенно сказать «это не баг» — профессиональный навык, а не признак слабости. Мы выделяем ложные срабатывания отдельными врезками в каждой главе.

Доход негарантирован и неравномерен. Эта книга не обещает заработок и не приводит «средних цифр дохода» — они вводят в заблуждение, потому что распределение выплат крайне неравномерно: небольшая доля исследователей получает большую часть вознаграждений. Правильнее думать о Bug Bounty как о способе развить навык, который монетизируется по-разному: собственно выплаты, работа в AppSec, пентест, исследования, преподавание.

Обучение не заканчивается. Технологии, фреймворки и защиты меняются. Приём, работавший вчера, сегодня закрыт новой версией библиотеки. Устойчивый результат даёт не коллекция «трюков», а понимание механизмов — поэтому книга построена вокруг причин уязвимостей, а не рецептов.

Между «обнаружил уязвимость» и «получил вознаграждение» лежит большая дистанция: подтверждение, оценка влияния, качественный отчёт, триаж, возможный дубликат и решение программы. Главы 13 и 14 целиком посвящены этому «последнему километру».

1.4.1. Как выглядит типичная сессия исследования

Чтобы заземлить ожидания, опишем обобщённый ход одной сессии (без романтики):

Выбор и подготовка (15–30 мин). Открыть программу, перечитать score и правила, поднять заметки и инструменты, проверить, что цель в границах.

Разведка и карта (часто дольше всего). Пройтись по функциям, собрать эндпоинты, роли, точки ввода (глава 5). Здесь рождаются гипотезы.

Проверка гипотез. Аккуратные эксперименты: «что если запросить чужой id?», «что если поле роли примут?». Большинство гипотез **не** подтверждается — это норма.

Отсев шума. Отличить реальную находку от ложного срабатывания (контрпроверка, глава 13). Часть времени уходит именно сюда.

Оформление. Для подтверждённой находки — минимальный PoC, доказательства, отчёт (главы 13–14).

Обратите внимание на пропорции: собственно «эксплуатация» — малая часть; больше всего времени занимают разведка, проверка гипотез и отсев шума. Это не «взлом за минуту», а методичная работа. Понимание этого отличает тех, кто остаётся в профессии, от тех, кто бросает после первых недель без мгновенного результата.

1.5. Жизненный цикл находки

Чтобы увидеть процесс целиком, пройдем его по шагам. Это каркас, который мы детализируем в главах 5, 13 и 14; здесь — обзорная карта.

Выбор программы. Оценка границ, технологий, активности, конкуренции (раздел 1.6 и глава 2).

Чтение правил и score. Что разрешено, что запрещено, что вне границ, какова safe harbor.

Картирование поверхности атаки. Домены, приложения, API, роли, точки ввода (глава 5).

Разведка. Пассивная и активная, строго в пределах score (глава 5).

Формирование гипотез. «Что здесь может пойти не так и почему» — на основе наблюдаемого поведения.

Проверка гипотез. Аккуратные эксперименты, минимально необходимые для подтверждения.

Исключение ложных срабатываний. Отделить реальную уязвимость от шума и реакций фильтров.

Оценка влияния и серьезности. Какой актив затронут, каковы последствия (глава 13; CVSS — глава 14).

Подготовка доказательств (PoC). Минимально достаточная демонстрация (глава 13).

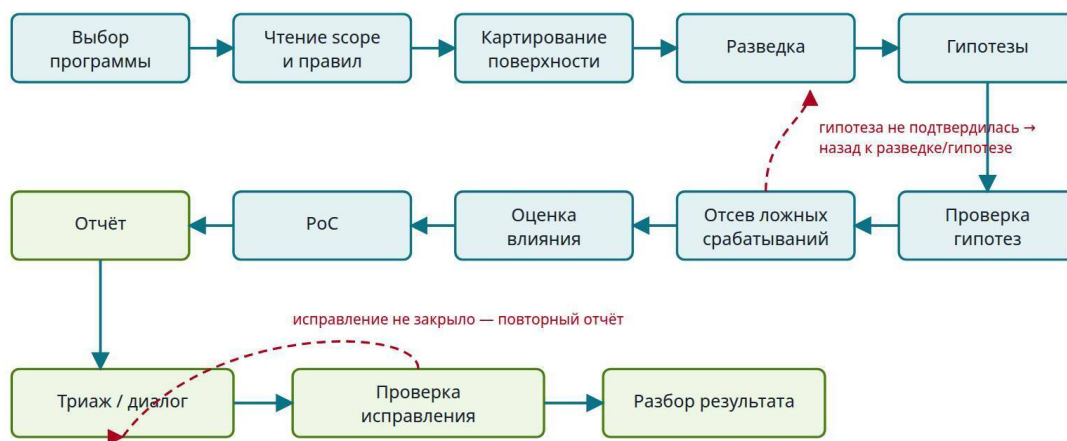
Написание отчёта. Воспроизводимо, честно, по структуре (глава 14).

Коммуникация с триажем. Ответы на вопросы, дополнительные доказательства.

Повторная проверка исправления. Если программа это предусматривает.

Разбор результата. Что сработало, что нет, чему научились — развитие навыка.

На схеме ниже этот цикл показан как поток с обратными связями: отрицательный результат проверки гипотезы возвращает к разведке или к новой гипотезе, а не завершает работу.



Жизненный цикл находки в Bug Bounty

1.6. Выбор программы: первая развилка

Выбор программы сильнее влияет на результат, чем техника. Несколько ориентиров (подробно — глава 2 и 5):

Широта score. Чем больше разрешённых активов (домены, поддомены, API, мобильные приложения), тем больше неисследованной поверхности. Широкий score часто продуктивнее «одной страницы входа».

Зрелость программы. Давно работающие программы крупных компаний отполированы тысячами исследователей — там выше конкуренция и больше дубликатов. Молодые программы рискованнее по выплатам, но менее «вытоптаны».

Технологии. Выбирайте то, что понимаете глубже других: ваш фон в конкретном стеке — преимущество.

Активность и отзывчивость. Как быстро программа отвечает, как оценивает, нет ли массовых споров — это видно по публичной статистике и отзывам сообщества.

«Большой бренд = лёгкие деньги» — опасное заблуждение. Чем известнее цель, тем плотнее её уже изучили. Иногда разумнее менее раскрученная программа с широким score и адекватным триажем.

1.6.1. Экономика находок: честная картина

Распределение вознаграждений в Bug Bounty сильно неравномерно — это важно понимать, чтобы не строить нереалистичных ожиданий. Несколько устойчивых закономерностей (без конкретных цифр, которые быстро устаревают и разнятся между платформами):

Выплаты зависят от серьёзности и влияния, а не от объёма усилий. Один отчёт об уязвимости высокой серьёзности часто стоит больше, чем десятки «информационных» находок.

Дубликаты неизбежны. На активных программах заметная доля валидных находок приходит повторно и не вознаграждается. Это не брак в вашей работе, а свойство среды.

Время до первого результата может быть долгим. Освоение целей и методологии занимает недели и месяцы; доход на старте непредсказуем и часто близок к нулю.

Навык переносим. Умение находить и описывать уязвимости ценится в AppSec, пентесте, исследованиях и преподавании — это снижает зависимость от конкретных выплат.

Практический вывод: относитесь к Bug Bounty как к инвестиции в навык с отложенной и неравномерной отдачей, а не как к источнику стабильного дохода. Эта книга не обещает заработка и намеренно не приводит «средних доходов» — такие цифры вводят в заблуждение из-за перекошенного распределения.

1.6.2. Платформы и прямые программы

Программы существуют в двух формах: на **платформах-посредниках** (которые агрегируют программы, ведут триаж и выплаты) и как **прямые программы** компаний (собственная страница безопасности, свой канал приёма). Для начинающего платформы удобнее: единый

интерфейс, история репутации, стандартизированный процесс. Прямые программы бывают менее «вытоптаны», но требуют больше самостоятельности. В любом случае правила, score и safe harbor читаются по конкретной программе (глава 2) — не переносите условия одной платформы на другую.

1.7. Этика и легальность как фундамент

Мы подробно разбираем это в главе 2, но принцип вводим сразу, потому что он не опционален.

Исследование безопасности легально при одном из условий: (1) это ваша собственная система; (2) это специально предназначенная учебная среда или локальная лаборатория (как HunterMart в этой книге); (3) вы действуете в пределах **явно разрешённого score** программы и по её правилам. Во всех остальных случаях — нельзя, какими бы благими ни были намерения.

Вся практика в книге построена на локальной лаборатории HunterMart, которую вы запускаете у себя. Это не ограничение «для вида»: безопасная среда позволяет свободно экспериментировать, ломать и чинить, не рискуя навредить чужим системам и не нарушая закон.

Доступность системы из интернета — не разрешение её тестировать. «Я же только посмотрел» — не оправдание. Разрешение даёт либо владелец (в score программы), либо тот факт, что система ваша или локальная.

1.8. Как пользоваться этой книгой

Маршрут новичка. Главы 1–5 подряд (основы, этика, HTTP, инструменты, разведка), затем классы уязвимостей (6–11) по одному, обязательно с лабораториями HunterMart, затем триаж и отчётность (13–14).

Маршрут разработчика. Главы 1, 3, разделы «Защитные меры» в главах 6–11, затем 14. Цель — понять, как думает атакующий, чтобы писать устойчивый код.

Маршрут практикующего исследователя. Главы 5, 6–11 как углубление, 12 (автоматизация), 13–14 как рабочий справочник.

Уровни сложности заданий обозначены от 1 (понимание терминов) до 8 (профессиональный отчёт). Решения практикумов вынесены в отдельную часть (глава 15), чтобы вы могли сначала попробовать сами.

1.8a. Задания (уровни 1–3)

Уровень 1. Своими словами объясните, что такое Bug Bounty и чем он отличается от «взлома».

Уровень 1. Составьте таблицу различий Bug Bounty / VDP / пентест / red team по инициатору, границам, оплате и цели.

Уровень 2. Опишите полный жизненный цикл находки (13 шагов) и отметьте, какие шаги возвращают назад.

Уровень 3. Выберите (как учебное чтение) публичную программу и оцените её по критериям раздела 1.6 (ширина score, зрелость, технологии, отзывчивость). Обоснуйте, подошла бы она новичку.

1.9. Резюме

Bug Bounty — это возмездный аудит по правилам: владелец публично разрешает искать у него уязвимости и платит за подтверждённые находки. Исследователь работает на стыке offensive и AppSec, всегда в легальных рамках. Форматы (Bug Bounty, VDP, пентест, red team, research) различаются инициатором, границами, оплатой и целью. Реальная работа — это проверка гипотез, борьба с ложными срабатываниями и дубликатами, неравномерный доход и непрерывное обучение; поэтому книга делает ставку на понимание механизмов. Полный цикл находки — от выбора программы до проверки исправления — задаёт структуру всей книги.

1.10. Контрольные вопросы

Чем Bug Bounty отличается от пентеста по модели работы и оплаты? А от red team — по цели?

Что такое VDP и safe harbor и почему последнее читают дословно?

Почему «запустить сканер» — не стратегия поиска уязвимостей?

Что такое дубликат и почему он не означает, что находка была невалидной?

Назовите три условия, при которых исследование безопасности легально.

Перечислите шаги жизненного цикла находки и объясните, почему это цикл, а не прямая линия.

1.11. Источники

OWASP, общие материалы о Bug Bounty и раскрытии уязвимостей — https://owasp.org/www-community/Vulnerability_Disclosure_Cheat_Sheet (проверено 2026-10-19).

disclose.io — шаблоны политик и safe harbor (справочно) — <https://disclose.io/> (проверено 2026-10-19).

CVE Program — <https://www.cve.org/> (проверено 2026-10-19).

OWASP Top 10:2025 (контекст классов уязвимостей) — https://owasp.org/Top10/2025/0x00_2025-Introduction/ (проверено 2026-10-19; часть материалов какое-то время публиковалась в статусе preview — сверяйте финальные формулировки по owasp.org).

Глава 2. Правила программ, границы и этика

Цель главы. Научиться читать правила Bug Bounty так, чтобы не выйти за границы закона и программы, понимать score и safe harbor, вести исследование безопасно и этично, и правильно обращаться с найденными данными. Это не «юридический довесок», а операционная основа профессии.

Предварительные требования. Глава 1.

Результаты обучения. Вы сможете: (1) разобрать политику программы и выделить разрешённое, запрещённое и вне-score; (2) объяснить safe harbor и его пределы; (3) провести тест, не навредив системе и чужим данным; (4) настроить инструменты так, чтобы они технически не выходили за границы; (5) корректно обращаться с конфиденциальной информацией и раскрытием.

2.1. Почему правила — это главное

В исследовании безопасности техника вторична по отношению к рамкам. Самая изящная находка ничего не стоит и может обернуться против вас, если получена за пределами разрешённого. Поэтому перед любым взаимодействием с целью исследователь отвечает на три вопроса: *что мне разрешено?, что запрещено?, где проходит граница, за которой я теряю защиту?*

Эта глава даёт систему, которая переводит эти вопросы в чеклист. В конце — «Чек-лист безопасного тестирования» (в приложениях книги), который вы проходите каждый раз.

Ничто в этой главе не является юридической консультацией. Законы различаются по юрисдикциям; при сомнениях консультируйтесь с юристом. Книга даёт инженерную дисциплину, а не правовую гарантию.

2.2. Как читать политику программы

Политика (program policy, brief) — это договор, по которому вы действуете. Читать её нужно целиком и буквально. Ключевые разделы:

Score (границы). Что именно разрешено тестировать: домены, поддомены, приложения, API, иногда диапазоны. Бывает «wildcard» score (*.example.com) и точечный (конкретные хосты).

Out of scope (вне границ). Что запрещено прямо: отдельные поддомены, сторонние сервисы, инфраструктура провайдера. Активы вне score не тестируются даже если «технически доступны».

Запрещённые действия. Обычно: DoS/нагрузочное тестирование, спам, социальная инженерия сотрудников и пользователей, физические атаки, автоматическое агрессивное сканирование.

Правила раскрытия (disclosure). Можно ли и когда публиковать write-up; как правило — только с разрешения и после исправления.

Safe harbor. Обязательство не преследовать добросовестного исследователя (раздел 2.4).

Вознаграждения и оценка. Как оценивается серьёзность, какие классы принимаются, что считается дубликатом.

Выпишите score и запреты в свои заметки перед началом и держите их перед глазами. Это дисциплинирует и защищает: при споре у вас есть зафиксированное понимание правил на момент работы.

2.2.1. Score: разрешённое не значит «всё внутри»

Даже внутри разрешённого домена могут быть исключения: отдельные пути, функции «только для внутренних пользователей», интеграции со сторонними сервисами. Wildcard *.example.com не включает сервисы партнёров на других доменах и обычно не включает инфраструктуру хостинга. Правило простое: **сомнение трактуется как «нельзя»**, а спорную границу уточняют у программы до тестирования.

2.2.2. Типичные out-of-score находки

Многие программы заранее объявляют, что НЕ принимают (это не уязвимости в их модели риска): отсутствие отдельных security-заголовков без доказанного влияния, самоподписанные сертификаты на непроизводственных хостах, перечисление пользователей на страницах, где это по дизайну, «теоретические» CSRF на формах без значимого действия, сообщения об устаревших версиях без подтверждённой эксплуатации. Знание этого списка экономит время: такие отчёты закрывают как «informative» без вознаграждения.

2.2.3. Worked-пример: разбор score

Разберём типовую (вымышленная, обобщённая) формулировку score, как её читает исследователь:

```

In scope:
  *.shop.example           (все поддомены)
  api.shop.example
  HunterMart mobile app (iOS/Android)
Out of scope:
  blog.shop.example        (на стороннем движке)
  *.partners.shop.example  (инфраструктура партнёров)
  Любые DoS/нагрузочные тесты, социальная инженерия, физический доступ
Rewards: по CVSS, Critical/High приоритетны. Informative – без вознаграждения.
Safe harbor: действия в рамках политики не повлекут юр. претензий.

```

Что отсюда следует практически:

*.shop.example включает, например, admin.shop.example, но **не** включает blog.shop.example (явно исключён) и **не** включает api.shop.example? — включает, поддомен подходит под wildcard; при этом api.shop.example упомянут отдельно, что лишь подтверждает его в scope.

*.partners.shop.example — это другой узел дерева, явно вне scope: трогать нельзя, даже если доступен.

Мобильное приложение в scope — значит, его бэкенд-API (если на доменах в scope) тоже исследуемы, но сверяйтесь, какие именно хосты он использует.

DoS и соц. инженерия запрещены вообще — даже «лёгкая» проверка нагрузкой выходит за правила.

Safe harbor привязан к соблюдению политики: шаг на partners.* или нагрузочный тест выводит из-под неё.

Выпишите разрешённые и запрещённые узлы в заметки **до** работы и сверяйтесь с ними при каждой новой находке хоста в разведке (глава 5). Граница, зафиксированная заранее, защищает вас при спорах.

2.3. VDP и Bug Bounty: разные обещания

VDP (Vulnerability Disclosure Program) — это канал «сообщите об уязвимости безопасно», как правило без денежного вознаграждения, но с обещанием не преследовать. **Bug Bounty** добавляет к этому выплаты. Практический вывод: на VDP не стоит рассчитывать на доход, но это легальный и полезный способ сообщить о проблеме и выстроить репутацию. Не выдавайте правила одной платформы или программы за универсальные: формулировки score, safe harbor и оценки различаются.

2.4. Safe harbor и его пределы

Safe harbor («тихая гавань») — раздел политики, где владелец обязуется не инициировать юридическое преследование исследователя, действующего добросовестно и в рамках правил. Это важная защита, но у неё есть пределы, которые нужно понимать:

Она действует **только в границах программы**. Шаг за score — и защита не применяется.

Она касается отношений с **этой** организацией. Она не отменяет законы и не связывает третьи стороны (например, провайдера или партнёра, чьи системы вы заделали).

Формулировки различаются: одни программы дают широкий safe harbor, другие — узкий и с оговорками. Читайте дословно, что именно обещано и при каких условиях.

«У них есть safe harbor» — не индульгенция. Защита обусловлена добросовестностью и соблюдением правил. Разрушение данных, выгрузка чужих персональных данных сверх необходимого, тестирование вне score выводят вас из-под неё.

2.5. Принципы безопасного исследования

Даже имея разрешение, действуйте так, чтобы не навредить. Это и этика, и условие сохранения safe harbor.

Минимальное воздействие. Подтверждайте уязвимость минимально достаточным PoC. Нашли SQL-инъекцию — достаточно извлечь одну «несанкционированную» строку, а не выгружать базу (сравните с лабораторией L-SQLI, где мы ограничиваемся парой записей для доказательства).

Без разрушения. Не удаляйте и не портите данные, не меняйте чужие записи, не выводите сервис из строя. Если PoC требует деструктивного шага — остановитесь и опишите риск словами.

Чужие данные — бережно. Не собирайте и не храните персональные данные сверх необходимого; в доказательствах маскируйте их. Не выносите чужие данные за пределы задачи.

Без чужих учётных данных. Не вводите реальные чужие логины/пароли/токены. Для тестов используйте свои тестовые аккаунты (в HunterMart — предусмотренные учебные учётки).

Не в URL. Не помещайте персональные или секретные данные в URL и query-строки.

2.5.1. Опасные классы — особые меры

При тестировании SSRF, XXE, инъекций команд и десериализации легко случайно задеть внутреннюю инфраструктуру. Поэтому (подробно — главы 9 и 11):
используйте безопасные маркеры и, где можно, локальную лабораторию;

исключайте реальные внутренние адреса, облачные metadata-эндпоинты и чужие секреты;

не разворачивайте нагрузку, которая может нарушить работу сервиса.

В этой книге все такие демонстрации выполняются только в HunterMart на синтетических данных и локальных адресах. Это осознанный принцип, а не ограничение «для вида».

2.6. Инструменты с техническими границами

Дисциплина «не выходить за score» должна быть не только в голове, но и в коде инструментов. Автоматика, способная обратиться к произвольной цели, рано или поздно это сделает — по ошибке в списке целей, из-за редиректа или опечатки. Поэтому инструменты книги построены вокруг **обязательного allowlist**.

Пример — `huntkit.safe_client` (глава 12): клиент выполняет запрос, только если хост и порт входят в явный allowlist; пустой allowlist не разрешает ничего.

```
from huntkit.safe_client import (
    Allowlist, SafeHTTPClient, DisallowedTargetError,
)

with SafeHTTPClient(Allowlist.from_iterable(["127.0.0.1:8000"])) as c:
    r = c.get("http://127.0.0.1:8000/health")  # разрешено
    try:
        c.get("http://example.test/")          # не в allowlist
    except DisallowedTargetError as e:
        print("заблокировано:", e)
```

Помимо allowlist, инструмент обеспечивает таймауты, ограничение частоты запросов, контроль редиректов (каждый хоп проверяется против allowlist) и журналирование. Эти меры — часть безопасного исследования: они не дают автоматике зашуметь цель или уйти за её пределы. Полный разбор — в главе 12.

Allowlist по хосту (`example.com`) не покрывает его поддомены автоматически — это осознанный выбор в пользу явности. Если вам нужен поддомен, добавьте его явно. «Умное» сопоставление поддоменов чаще расширяет границу, чем помогает.

2.7. Раскрытие и репутация

Приватность по умолчанию. Не раскрывайте уязвимость публично до исправления и явного разрешения программы. Преждевременное раскрытие вредит пользователям и вашей репутации.

Публичный write-up. После разрешённого раскрытия качественный разбор — ценный вклад и элемент портфолио. Отделяйте факты (что было) от интерпретаций и не раскрывайте лишних деталей о клиентах.

Коммуникация. С триажем общайтесь предметно и вежливо, даже при несогласии с оценкой. Репутация в этой сфере накапливается годами и открывает доступ к приватным программам и работе (глава 14).

2.7.1. Worked-пример: типичная временная линия раскрытия

Чтобы понимать, что и когда можно делать, полезно видеть процесс целиком (конкретные сроки задаёт политика программы — здесь обобщённая схема):

Обнаружение и подтверждение находки (в score, по правилам).

Приватный отчёт через канал программы. С этого момента вы не раскрываете детали публично.

Триаж (программа подтверждает/уточняет/отклоняет; возможен дубликат).

Исправление уязвимости владельцем (может занять недели/месяцы).

Повторная проверка исправления (если программа предусматривает).

Согласованное раскрытие (coordinated disclosure): публичный write-up — только с разрешения и обычно после исправления.

Ключевые правила на этой линии: не раскрывать до исправления и разрешения (это навредит пользователям и вашей репутации); не «торопить» угрозами публикации; быть терпеливым к срокам фикса, но вправе вежливо уточнять статус. Некоторые программы задают предельные сроки раскрытия — их тоже читают в политике.

Молчание программы какое-то время — норма, а не повод для публичного раскрытия. Эскалация — через канал программы/платформы, а не через публикацию деталей.

2.8. Практикум

Уровень 1. Возьмите публичную политику любой известной VDP/Bug Bounty программы (как учебный материал для чтения, без какого-либо тестирования) и выпишите: score, out-of-score, запрещённые действия, наличие и формулировку safe harbor. Отметьте пункты, которые трактуете неоднозначно.

Уровень 2. Для HunterMart составьте собственный «score»: какие эндпоинты существуют (см. карту лаборатории в главе 15), какие учётные записи вам разрешено использовать, какие действия вы считаете допустимыми в лаборатории. Это тренировка дисциплины на безопасной среде.

Уровень 4. Настройте `huntkit.safe_client` с `allowlist` только на `127.0.0.1:8000` и убедитесь, что попытка обратиться к любому другому хосту завершается `DisallowedTargetError`. Объясните, почему такая техническая граница надёжнее устной договорённости «я буду аккуратен».

2.8a. Дополнительные задания (уровни 1–5)

Уровень 1. Выпишите из учебной политики (как чтение) scope, out-of-scope и запреты.

Уровень 2. Для *.shop.example определите, что входит и что не входит в scope, с обоснованием.

Уровень 3. Сформулируйте, какие действия выводят вас из-под safe harbor, и почему.

Уровень 4. Настройте huntkit.safe_client на лабораторию и покажите блокировку чужого хоста.

Уровень 5. Опишите временную линию согласованного раскрытия для гипотетической находки.

2.9. Анализ ошибок новичка

Тестируют «немного за score». Границы бинарны: либо разрешено, либо нет.

Полагаются на safe harbor как на индульгенцию. Он обусловлен добросовестностью и соблюдением правил.

Выгружают чужие данные «для убедительности». Это и вредно, и выводит из-под защиты; хватает PoC.

Раскрывают находку публично до исправления. Вредит пользователям и репутации.

Доверяют дисциплину только памяти. Границы должны быть и в коде инструментов (allowlist).

2.10. Защитные меры (для владельца программы)

Пишите score и out-of-score однозначно; явно перечисляйте запрещённые действия.

Давайте внятный safe harbor — это повышает качество и количество отчётов.

Предусмотрите тестовые аккаунты и окружение, чтобы исследователи не трогали реальные данные.

Быстро и предметно реагируйте на отчёты: отзывчивость программы напрямую влияет на её результаты.

2.11. Резюме

Правила программы — это договор, который читают буквально и целиком: scope, out-of-scope, запреты, раскрытие, safe harbor. Safe harbor защищает, но только в границах и при добросовестности. Безопасное исследование — это минимальное воздействие, бережное обращение с данными, отказ от чужих учётных данных и деструктивных действий, а также технические границы в инструментах (обязательный allowlist). Приватное раскрытие и корректная коммуникация формируют репутацию, которая в этой профессии стоит дорого.

2.12. Контрольные вопросы

Какие разделы политики вы читаете до начала работы и почему именно их?

Что входит и что не входит в wildcard-scope *.example.com?

Назовите три предела safe harbor.

Почему минимальное воздействие — это одновременно этика и защита вашей «тихой гавани»?

Чем технический allowlist надёжнее устного обещания «быть аккуратным»?

Когда допустимо публиковать write-up и что в нём не стоит раскрывать?

2.13. Источники

OWASP Vulnerability Disclosure Cheat Sheet — https://cheatsheetseries.owasp.org/cheatsheets/Vulnerability_Disclosure_Cheat_Sheet.html (проверено 2026-10-19).

disclose.io (шаблоны политик, safe harbor, терминология) — <https://disclose.io/> (проверено 2026-10-19).

ISO/IEC 29147 (Vulnerability disclosure) и 30111 (Vulnerability handling) — стандарты процессов раскрытия и обработки; см. каталог ISO <https://www.iso.org/> (проверено 2026-10-19).

«Чек-лист безопасного тестирования» — в приложениях книги.

Глава 3. HTTP, браузер, cookies, сессии, TLS и веб-архитектура

Цель главы. Дать рабочее понимание веб-стека на уровне, достаточном для исследователя: как устроена архитектура веб-приложения, как работает HTTP во всех значимых деталях, что такое cookies и сессии, как TLS защищает канал, и как всё это читать в трафике. Это фундамент для всех глав про уязвимости.

Предварительные требования. Главы 1–2. Базовое представление о том, что такое веб-страница.

Результаты обучения. Вы сможете: (1) описать путь запроса от браузера до сервера и обратно; (2) разобрать HTTP-обмен по частям и объяснить роль ключевых заголовков; (3) понимать cookies, сессии и их атрибуты безопасности; (4) объяснить, что даёт TLS; (5) анализировать security headers инструментом.

3.1. Архитектура веб-приложения

Современное веб-приложение — это распределённая система. Между вашим браузером и «сервером» обычно стоит несколько компонентов, и границы между ними — частый источник уязвимостей (главы 9, 11).

Типичный путь запроса:

```
Браузер → DNS → [CDN] → [обратный прокси / балансировщик]
→ веб-сервер приложения → [кэш] → база данных
```

Браузер — доверенная для пользователя, но **недоверенная для сервера** сторона. Всё, что приходит от браузера, сервер обязан считать потенциально враждебным (граница доверия, глава 1).

DNS — преобразует имя (huntermart.example) в IP-адрес.

CDN / обратный прокси — кэширование, балансировка, терминация TLS. Здесь рождаются классы вроде cache poisoning и request smuggling (главы по продвинутым темам).

Сервер приложения — исполняет бизнес-логику (в HunterMart — FastAPI).

База данных — хранилище (в HunterMart — SQLite).

Рисуйте для исследуемой системы диаграмму потоков данных с границами доверия (как DFD HunterMart в главе 1). Большинство уязвимостей живёт на границах, где данные переходят из менее доверенной зоны в более доверенную.

3.1.1. Клиент-серверная модель и SPA

Классическое приложение рендерит HTML на сервере. **SPA (Single-Page Application)** переносит рендеринг в браузер: сервер отдаёт JavaScript, который обращается к API (REST/GraphQL) за данными. Для исследователя это означает: много логики и эндпоинтов «спрятано» в JS-бандле (его анализ — глава 5), а безопасность смещается к API (глава 11) и клиентскому коду (глава 8).

3.2. HTTP: протокол запрос–ответ

HTTP (HyperText Transfer Protocol) — текстовый по происхождению прикладной протокол. Он **без состояния (stateless)**: сам по себе не помнит предыдущие запросы; «память» (сессию) надстраивают поверх через cookies и токены (раздел 3.5).

3.2.1. Анатомия обмена

Любое HTTP-сообщение: **стартовая строка**, **заголовки**, пустая строка, опциональное **тело**. Пример на HunterMart:

```
GET /api/orders/1001 HTTP/1.1
Host: 127.0.0.1:8000
Authorization: Bearer 1.9f86d0...
Accept: application/json

--- ответ ---
HTTP/1.1 200 OK
content-type: application/json
content-length: 173

{"id":1001,"user_id":1,"status":"paid","shipping_address":"..."}
```

Части запроса: **метод** (GET), **цель** (/api/orders/1001 + возможная строка запроса ?a=1), **версия** (HTTP/1.1), **заголовки** (Host обязателен в 1.1), **тело** (у GET обычно нет). Имена заголовков **регистронезависимы**.

3.2.2. Методы

Метод	Назначение	Идемпотентный	«Безопасный»*
GET	получить ресурс	да	да
HEAD	как GET без тела	да	да
POST	создать/выполнить	нет	нет
PUT	заменить целиком	да	нет
PATCH	изменить частично	нет	нет
DELETE	удалить	да	нет
OPTIONS	допустимые методы/CORS	да	да

* «Безопасный» здесь — термин HTTP (метод не меняет состояние), не про защищённость. Нельзя полагаться, что приложение честно соблюдает семантику: иногда GET меняет данные (проблема — CSRF, кэширование). Проверяйте фактическое поведение.

3.2.3. Коды состояния

2xx успех (200, 201, 204); **3xx** перенаправление (301, 302, 304, 307, 308); **4xx** ошибка клиента (400, 401, 403, 404, 429); **5xx** ошибка сервера (500, 502, 503).

Для исследователя коды — **сигнал**, а не истина. 401 («не аутентифицирован») против 403 («нельзя»); 404 вместо 403 на чужом объекте — осознанная защита от перечисления (глава 6); разные коды/размеры на «есть/нет пользователя» — канал для перечисления учётных записей.

3.3. Заголовки и безопасность

Заголовки несут и метаданные, и защитные механизмы. Ключевые защитные заголовки ответа:

Заголовок	Что делает	Глава
Strict-Transport-Security (HSTS)	запрет понижения до http	3, 10
Content-Security-Policy (CSP)	ограничение источников, смягчение XSS	8
X-Content-Type-Options: nosniff	запрет «угадывания» типа	8, 9
X-Frame-Options / CSP frame-ancestors	защита от clickjacking	8
Referrer-Policy	ограничение утечки URL в Referer	13
Set-Cookie с Secure/HttpOnly/SameSite	защита cookie	3.5, 8, 10

Некоторые заголовки, наоборот, **раскрывают** лишнее (Server, X-Powered-By — технология и версия). Проверять security headers вручную утомительно; инструмент `huntkit.headers` (глава 12) делает это по рекомендациям OWASP Secure Headers Project:

```
from huntkit.headers import analyze, parse_header_block

raw = "HTTP/1.1 200 OK\nContent-Type: text/html\nServer: Apache\n"
for f in analyze(parse_header_block(raw), is_https=True).findings:
    print(f.severity, f.header)
# medium Strict-Transport-Security
# medium Content-Security-Policy
# low X-Content-Type-Options ...
```

Отсутствие security-заголовка — ослабление защиты, но ещё не эксплуатируемая уязвимость. «Нет CSP» значимо, если есть точка для XSS. Пустой список отсутствующих заголовков без контекста многие программы считают информационным пунктом.

3.3.1. Worked-пример: разбор CORS

CORS (Cross-Origin Resource Sharing) — механизм, которым сервер *разрешает* чужим origin читать свои ответы (послабление Same-Origin Policy). Его часто путают с защитой от CSRF — это разные вещи: CORS управляет **чтением ответа** кросс-доменно, а не отправкой запроса. Ключевые заголовки ответа:

```
Access-Control-Allow-Origin: https://trusted.example
Access-Control-Allow-Credentials: true
```

Опасная мисконфигурация — отражение Origin запроса в Access-Control-Allow-Origin вместе с Allow-Credentials: true:

```
--- запрос ---
Origin: https://evil.test
--- ответ (уязвимо) ---
Access-Control-Allow-Origin: https://evil.test    <- отражён чужой origin
Access-Control-Allow-Credentials: true
```

Это позволяет странице атакующего читать ответы с учётными данными жертвы — утечка данных. Проверка: подставить произвольный Origin и посмотреть, отражается ли он в Allow-Origin при Allow-Credentials: true. Защита — **allowlist** доверенных origin, никогда не отражать произвольный Origin, не сочетать * с Allow-Credentials.

Access-Control-Allow-Origin: * без Allow-Credentials на публичном ресурсе (например, открытый API без сессии) обычно не уязвимость — читать там нечего приватного. Подтверждение требует доступа к **приватным** данным кросс-доменно.

3.4. Строка запроса, тело и форматы данных

Строка запроса (query string) — `?key=value&key2=value2`. Видна в логах и истории — не место для секретов (глава 2).

Тело — у POST/PUT/PATCH. Частые типы: `application/json`, `application/x-www-form-urlencoded`, `multipart/form-data` (загрузка файлов, глава 9).

JSON — основной формат API. **REST** — архитектурный стиль над HTTP; **GraphQL** — единый эндпоинт с языком запросов; **WebSocket** — полнодуплексный канал. Все три разбираются в главе 11.

3.5. Cookies и сессии

Поскольку HTTP без состояния, сервер выдаёт cookie заголовком Set-Cookie, а браузер возвращает её в Cookie:

```
--- ответ ---
Set-Cookie: session=abc123; Path=/; HttpOnly; Secure; SameSite=Lax

--- следующий запрос ---
Cookie: session=abc123
```

Атрибуты cookie — это механизмы безопасности:

HttpOnly — недоступна из JavaScript (смягчает кражу сессии при XSS, глава 8).

Secure — отправляется только по HTTPS.

SameSite (Lax/Strict/None) — ограничивает отправку в межсайтовых запросах (ключ к защите от CSRF, глава 8).

Path/Domain — область действия; слишком широкий Domain — риск.

Сессии и токены. Идентификатор сессии хранят в cookie (серверная сессия) или в виде токена в Authorization (как в HunterMart). Учебная модель HunterMart упрощена (без срока жизни и ротации) — её ограничения разбираются в главе 10.

3.5.1. Worked-пример: полный цикл аутентифицированной сессии

Разберём на HunterMart цикл «вход → аутентифицированный запрос» целиком, по трафику. Шаг 1 — вход:

```
POST /api/login HTTP/1.1
Host: 127.0.0.1:8000
Content-Type: application/json

{"username":"alice","password":"alice-pass"}

--- ответ ---
HTTP/1.1 200 OK
content-type: application/json

{"token":"1.9f86d0...", "role":"customer", "user_id":"1"}
```

Сервер проверил учётные данные (аутентификация) и выдал токен сессии. HunterMart передаёт токен в теле ответа, и клиент предъявляет его в заголовке Authorization (другие приложения кладут сессию в Set-Cookie — тогда браузер возвращал бы её автоматически, см. 3.5). Шаг 2 — аутентифицированный запрос:

```
GET /api/orders/1001 HTTP/1.1
Host: 127.0.0.1:8000
Authorization: Bearer 1.9f86d081884c7d659a2feaa0c55ad015...

--- ответ ---
HTTP/1.1 200 OK
content-type: application/json

{"id":1001,"user_id":1,"status":"paid","shipping_address":"..."}
```

Что здесь важно исследователю: (1) токен — это и есть «состояние», настроенное над HTTP без состояния; (2) предъявление токена подтверждает личность (аутентификация), но **не гарантирует** право на объект 1001 — это отдельная проверка (авторизация, глава 6); (3) если убрать Authorization, сервер должен вернуть 401 — проверьте это, чтобы понять, где граница аутентификации. Воспроизвести цикл в терминале:

```
TOKEN=$(curl -s http://127.0.0.1:8000/api/login \
  -H 'Content-Type: application/json' \
  -d '{"username":"alice","password":"alice-pass"}' | jq -r .token)
curl -s http://127.0.0.1:8000/api/orders/1001 \
  -H "Authorization: Bearer $TOKEN" | jq .
curl -s -o /dev/null -w 'без токена: %{http_code}\n' \
  http://127.0.0.1:8000/api/orders/1001
```

3.6. Редиректы

Ответ 3xx с Location перенаправляет браузер. 301/308 — постоянный, 302/307 — временный; 307/308 сохраняют метод и тело, 302 исторически часто меняет метод на GET. Редиректы — источник Open Redirect (если цель берётся из управляемого параметра без проверки). В HunterMart это эндпоинт /go (разобран как лаборатория L-OPENREDIR; подробности класса — в части про уязвимости):

```
curl -s -o /dev/null -D - \
    "http://127.0.0.1:8000/go?next=https://evil.test/phish" \
    | grep -i '^location'
# location: https://evil.test/phish (уязвимый режим)
```

3.7. Версии протокола

HTTP/1.1 — текстовый, один активный обмен на соединение (с keep-alive). Обязателен Host. Расхождения в разборе 1.1 между прокси и сервером порождают request smuggling.

HTTP/2 — бинарный, мультиплексирование потоков, сжатие заголовков (HPACK). Семантика та же.

HTTP/3 — поверх QUIC (на UDP), TLS 1.3 встроенно, устраняет head-of-line blocking на транспорте.

Для большинства находок уровня приложения версия транспорта вторична (семантика сохраняется), но на стыках версий и реализаций возникают smuggling и рассинхрон кэша.

3.7.1. Worked-пример: почему стыки версий опасны

Проблема рождается, когда два компонента на пути (например, фронтенд-прокси/CDN и бэкенд-сервер) **по-разному интерпретируют** одно и то же сообщение. Классический пример — **HTTP Request Smuggling**: прокси и сервер расходятся в том, где заканчивается один запрос и начинается следующий (из-за разной трактовки Content-Length и Transfer-Encoding в HTTP/1.1, или при трансляции HTTP/2→HTTP/1.1). В результате «протасканный» фрагмент одного запроса влияет на другой — например, на запрос другого пользователя.

Для исследователя практический смысл в том, что такие классы живут **не в коде приложения**, а в расхождениях реализаций на границе компонентов. Поэтому важно знать, что перед приложением стоит (CDN, обратный прокси) и как устроена трансляция версий. Это продвинутая тема; здесь достаточно понимать принцип: **единая семантика HTTP не гарантирует единую интерпретацию байтов разными реализациями**, и именно расхождение интерпретаций — источник уязвимости.

3.9.1. Worked-пример: ключ кэша и его рассинхрон

Кэш (браузер, прокси, CDN) хранит ответ под **ключом**, обычно складывающимся из метода, хоста и пути (иногда части заголовков через Vary). Уязвимости возникают, когда **ключ не совпадает с тем, что реально влияет на ответ**:

Cache Poisoning. Если на ответ влияет заголовок, не входящий в ключ (например, нестандартный заголовок, который сервер отражает в теле), атакующий «отравляет» кэш: кладёт туда вредный ответ, который затем отдаётся другим пользователям под обычным ключом.

Cache Deception. Атакующий заставляет кэш сохранить **приватный** ответ под ключом, который потом запросит он сам (например, добавив к приватному URL «статичное» расширение, которое кэш считает кэшируемым).

Вопрос исследователя всегда один: *из чего складывается ключ кэша и совпадает ли он с факторами, влияющими на ответ?* Защита — согласовать ключ с реальными факторами (корректный Vary), не кэшировать приватные ответы (Cache-Control: private, no-store), нормализовать то, что влияет на ответ.

3.8. TLS: защита канала

TLS (Transport Layer Security) обеспечивает конфиденциальность и целостность канала и аутентификацию сервера сертификатом. **HTTPS = HTTP поверх TLS**. Что важно исследователю:

Рукопожатие согласует версию (TLS 1.2/1.3) и шифры, сервер предъявляет **сертификат**, цепочка которого проверяется до доверенного корня.

SNI (Server Name Indication) — имя хоста в рукопожатии (нужно для виртуального хостинга).

Certificate Transparency — публичные логи выданных сертификатов; ценный источник для разведки (поиск поддоменов, глава 5).

Без HTTPS данные видны и изменяемы на пути (MITM); отсюда важность HSTS (раздел 3.3).

Проверка TLS из командной строки:

```
openssl s_client -connect example.com:443 -servername example.com \  
</dev/null 2>/dev/null \  
| openssl x509 -noout -subject -issuer -dates
```

Любой практический разбор в книге — на локальном HunterMart (<http://127.0.0.1:8000>) или на явно разрешённых целях. Обращение к чужим системам — только в рамках scope (глава 2).

3.9. Кэширование

Заголовки Cache-Control, ETag, Last-Modified, Vary управляют кэшированием (браузер, прокси, CDN). Условные запросы (If-None-Match) дают 304. Ключевой для безопасности вопрос: **из чего складывается ключ кэша** и совпадает ли он с тем, что реально влияет на ответ — отсюда cache poisoning и cache deception (продвинутые темы).

3.10. Практикум

Уровень 2. Поднимите HunterMart, выполните вход и запрос заказа через `curl -v`. Выпишите метод, путь, три значимых заголовка запроса, код ответа и `content-type`; объясните роль каждого заголовка.

Уровень 3. Снимите заголовки ответа любого эндпоинта и прогоните через `huntkit.headers.analyze`. Для каждой находки решите: это ослабление защиты или эксплуатируемая проблема? Что нужно, чтобы находка стала реальной уязвимостью?

Уровень 4. Проверьте TLS любого публичного сайта (как чтение сертификата, без тестирования) командой из раздела 3.8 и объясните, что означают поля `subject/issuer/dates`.

3.10а. Дополнительные задания (уровни 1–8)

Уровень 1. Разметьте части HTTP-запроса и ответа из раздела 3.2.1 (метод, цель, заголовки, тело).

Уровень 2. Через `curl -v` выполните вход и запрос заказа; выпишите три значимых заголовка.

Уровень 3. Снимите заголовки ответа и прогоните через `huntkit.headers`; классифицируйте находки.

Уровень 4. Проверьте разницу 401/403/404 на эндпоинтах HunterMart и объясните её.

Уровень 5. Разберите cookie-атрибуты гипотетического Set-Cookie и назначение каждого.

Уровень 6. Объясните на примере, почему GET, меняющий состояние, — проблема (CSRF/кэш).

Уровень 7. Опишите, как расхождение интерпретации на границе «прокси — сервер» порождает smuggling.

Уровень 8. Предложите набор security-заголовков для HunterMart и обоснуйте каждый.

3.11. Анализ ошибок новичка

Доверяют коду ответа как истине. 200 ≠ «всё хорошо», 404 ≠ «ничего нет».

Считают отсутствие security-заголовка готовой уязвимостью. Нужен контекст и влияние.

Путают HTTP-«безопасный метод» с защищённостью.

Игнорируют регистр и повторы заголовков. Повторяющиеся заголовки стороны объединяют по-разному — это бывает каналом атаки.

Забывают, что браузер — недоверенная сторона. Клиентские проверки не защищают.

3.12. Защитные меры (для разработчика)

Соблюдайте семантику методов; не меняйте состояние по GET.

Выставляйте security headers осознанно; не раскрывайте версии (Server, X-Powered-By).

Корректные атрибуты cookie (HttpOnly, Secure, SameSite); HSTS над HTTPS.

Явная политика кэширования приватных ответов (Cache-Control: private, no-store).

3.13. Резюме

Веб-приложение — распределённая система с границами доверия между браузером, прокси/CDN, сервером и базой. HTTP — протокол запрос–ответ без состояния с единой семантикой поверх версий транспорта; состояние надстраивают cookies и токены с атрибутами безопасности. TLS защищает канал и аутентифицирует сервер. Исследователь читает обмен по частям, не доверяет кодам и заявленному поведению, видит в заголовках и защиту, и каналы утечки. Эти основы используются во всех последующих главах.

3.14. Контрольные вопросы

Опишите путь запроса от браузера до базы и назовите, где проходят границы доверия.

Из каких частей состоит HTTP-сообщение? Чем обязателен Host в HTTP/1.1?

В чём разница 401/403 и почему их путаница помогает атакующему?

Зачем cookie-атрибуты HttpOnly, Secure, SameSite? Против каких атак каждый?

Что даёт TLS и почему важен HSTS?

Почему «нет CSP» — ослабление защиты, но не всегда готовая уязвимость?

3.15. Источники

RFC 9110 (HTTP Semantics), 9111 (Caching), 9112 (HTTP/1.1), 9113 (HTTP/2), 9114 (HTTP/3) — <https://www.rfc-editor.org/> (проверено 2026-10-19).

RFC 6265 (Cookies) — <https://www.rfc-editor.org/rfc/rfc6265.html> (проверено 2026-10-19; атрибут SameSite описан в последующих обновлениях — сверяйте статус).

RFC 8446 (TLS 1.3) — <https://www.rfc-editor.org/rfc/rfc8446.html> (проверено 2026-10-19).

OWASP Secure Headers Project — <https://owasp.org/www-project-secure-headers/> (проверено 2026-10-19).

MDN HTTP — <https://developer.mozilla.org/en-US/docs/Web/HTTP> (проверено 2026-10-19).

Глава 4. Linux, командная строка, Git и Python для исследователя

Цель главы. Собрать рабочий инструментарий исследователя: уверенная работа в Linux и терминале, обработка текста, Git для учёта экспериментов, и Python как язык автоматизации. Это прикладная глава: её навыки используются во всех практикумах и во всей части про инструменты (глава 12).

Предварительные требования. Главы 1–3. Доступ к Linux-окружению (или WSL/виртуальной машине).

Результаты обучения. Вы сможете: (1) уверенно двигаться по файловой системе и правам; (2) строить конвейеры обработки текста (grep/sed/awk/jq); (3) вести эксперименты в Git; (4) писать и запускать Python-скрипты в изолированном окружении (venv) с тестами.

4.1. Почему именно Linux

Большинство серверов, инструментов безопасности и примеров в сообществе — под Linux. Терминал даёт воспроизводимость (команду можно записать, повторить, автоматизировать) и прозрачность (видно, что именно происходит). Если у вас Windows — используйте WSL или виртуальную машину; если macOS — многое совпадает, но отдельные утилиты (sed, grep) отличаются флагами (BSD против GNU).

Заведите отдельную изолированную среду (ВМ или контейнер) для исследований: так вы не смешиваете рабочие инструменты с личными данными и можете безопасно «ломать» лабораторию.

4.2. Терминал и оболочка

Оболочка (shell, обычно bash) исполняет команды. Базовая модель: команда, аргументы, флаги.

```
ls -la /var/log      # команда ls, флаги -l -a, аргумент — путь
```

Конвейеры (pipes) соединяют вывод одной команды со входом другой — основа мощи терминала:

```
cat access.log | grep "POST" | wc -l      # сколько POST-запросов в логе
```

Перенаправления: > (в файл, перезапись), >> (дописать), < (из файла), 2> (поток ошибок).

```
mytool > out.txt 2> err.txt      # stdout в out.txt, stderr в err.txt
```

4.3. Файловая система и права

Linux — дерево от корня /. Важные места: /etc (конфигурация), /var/log (логи), /home (домашние каталоги), /tmp (временное). Пути бывают абсолютные (/home/user) и относительные (./file, ../up).

Права (ls -l показывает -rwxr-xr--): три группы (владелец, группа, остальные) по три бита (чтение r, запись w, исполнение x).

```
chmod 750 script.sh    # rwx владельцу, r-x группе, --- остальным
chown user:group file  # сменить владельца/группу
```

Для исследователя важно: по правам видно, кто и что может; неверные права на файлы с секретами — отдельный класс проблем (security misconfiguration).

4.4. Процессы, окружение, пакеты

Процессы: ps aux, top/htop, завершение — kill <pid>. Фоновый запуск — &, управление — jobs, fg, bg.

Переменные окружения: export NAME=value; env показывает текущие. Секреты часто передают через окружение — поэтому env и файлы вроде .env интересны (но чужие — только в score!).

Пакеты: в Debian/Ubuntu — apt, обновления важны для безопасности самого рабочего места.

Скачанные файлы и распакованные архивы — недоверенные данные. Кладите каждый в отдельный пустой каталог, не запускайте интерпретаторы и сборщики «из» него (интерпретатор может подхватить код из текущего каталога), передавайте пути аргументами. Это гигиена вашего рабочего места.

4.5. Обработка текста: grep, sed, awk, jq

Исследование — это во многом обработка текста (логи, ответы, списки URL).

grep — поиск по шаблону (регулярные выражения):

```
# рекурсивно, без регистра, расширенные regex:
grep -rniE "password|secret|api[_-]?key" .
```

sed — потоковое редактирование (замены):

```
sed 's/http:/https:/g' urls.txt # заменить схему во всех строках
```

awk — обработка по полям (столбцам):

```
awk '{print $1, $7}' access.log # IP и путь из лога
```

jq — работа с JSON (незаменима для API):

```
curl -s http://127.0.0.1:8000/api/products/search?q=micro \
  | jq '.results[].title'
```

sort | uniq -c | sort -rn — частотный анализ (топ значений):

```
awk '{print $1}' access.log | sort | uniq -c | sort -rn | head
```

4.5.1. Worked-пример: разбор лога запросов

Пусть есть файл access.log в типовом формате (IP, время, «метод путь протокол», код, размер):

```
10.0.0.5 - - [19/Oct/2026:10:00:01] "GET /api/orders/1001 HTTP/1.1" 200 173
10.0.0.9 - - [19/Oct/2026:10:00:02] "GET /api/orders/1002 HTTP/1.1" 200 171
10.0.0.5 - - [19/Oct/2026:10:00:03] "POST /api/login HTTP/1.1" 401 52
10.0.0.5 - - [19/Oct/2026:10:00:04] "POST /api/login HTTP/1.1" 401 52
```

Полезные выжимки:

```
# Топ IP по числу запросов (кто активнее всех)
awk '{print $1}' access.log | sort | uniq -c | sort -rn | head

# Все 4xx/5xx (ошибки – часто интересные места)
awk '$(NF-1) ~ /^[45]/ {print}' access.log

# Повторные неудачные входы с одного IP (признак перебора)
grep '"POST /api/login' access.log \
  | awk '$(NF-1)==401 {print $1}' | sort | uniq -c | sort -rn

# Уникальные запрашиваемые пути
awk -F'"' '{print $2}' access.log | awk '{print $2}' | sort -u
```

Эти приёмы — повседневный инструмент при анализе логов, ответов и списков URL; те же awk/sort/uniq применяются к выводу разведки (глава 5) и к результатам инструментов (глава 12).

4.8.1. Worked-пример: Python для анализа JSON-ответа

Типичная задача — обработать JSON-ответ API. Пусть ответ сохранён в resp.json:

```
import json
from pathlib import Path

raw = Path("resp.json").read_text(encoding="utf-8")
data = json.loads(raw)

# Безопасный доступ: ключа может не быть
results = data.get("results", [])
print(f"найдено: {len(results)}")
for item in results:
    # Берём поля с запасным значением, чтобы не упасть на неполных данных
    print(item.get("id"), item.get("title", "(без названия)"))
```

Обратите внимание на оборонительный стиль: `.get(...)` с запасным значением вместо `data["results"]` — данные из внешнего источника неполны и враждебны по умолчанию (принцип границы доверия, глава 1). Этот же стиль мы применяем в инструментах `huntkit` (глава 12): любые внешние данные обрабатываются так, будто они могут быть некорректными.

4.6. Сетевая диагностика из терминала

curl -v — отправить запрос и увидеть весь обмен (основной инструмент, глава 3).

dig — DNS-запросы (записи A/AAAA/CNAME/MX/TXT), основа разведки (глава 5).

openssl s_client — ручной разбор TLS-рукопожатия и сертификата (глава 3).

ss -tlnp — какие порты слушает ваша машина (полезно для локальной лаборатории).

```
dig +short huntermart.example A           # (в лаборатории — локально)
curl -s -o /dev/null -w '%{http_code}\n' http://127.0.0.1:8000/health
```

4.7. Git: учёт экспериментов

Git — система контроля версий. Для исследователя это не только про код: это журнал гипотез, результатов и инструментов, который можно откатить и сравнить.

```
git init                # создать репозиторий
git add -A && git commit -m "baseline" # зафиксировать состояние
git diff                # что изменилось с последнего коммита
git log --oneline       # история
git checkout -- file    # откатить файл
```

Практика этой книги хранится в Git именно так: каждый блок работы — отдельный коммит, что даёт `git diff` для проверки и возможность отката (мы использовали это при переименовании проекта).

В `.gitignore` держите секреты, виртуальные окружения и кэши (`.venv/`, `.env`, `__pycache__/`). Случайный коммит секрета — частая и опасная ошибка.

4.7.1. Worked-пример: Git-рабочий процесс исследователя

Покажем минимальный цикл, которым удобно вести и код инструментов, и журнал экспериментов:

```
git init
# не коммитим окружение и секреты:
printf '.venv/\n__pycache__/\n.env\n' > .gitignore
git add -A && git commit -m "baseline: окружение и заметки"

# ... поработали: написали инструмент, заметку о гипотезе ...
git add -A && git commit -m "tool: парсер заголовков + заметка IDOR"

git log --oneline    # история: видно, что и когда делали
git diff HEAD~1      # что изменилось в последнем шаге
```

Зачем это исследователю: (1) **воспроизводимость** — к любому состоянию можно вернуться; (2) **журнал** — коммиты фиксируют ход мысли и находки; (3) **откат** — ошибочное изменение легко отменить. Именно так ведётся репозиторий этой книги: каждый блок работы — отдельный коммит, что даёт `git diff` для проверки и возможность отмены (мы использовали это при переименовании учебного проекта). Правило гигиены: секреты и окружение — всегда в `.gitignore`, проверяйте `git status` перед коммитом.

4.8. Python для исследователя: основы

Python — язык автоматизации в безопасности: читаемый, с богатой стандартной библиотекой и экосистемой. Книга использует Python 3.12. Минимум, нужный, чтобы начать (полноценно — глава 12):

```
# переменные и типы
name: str = "alice"
count: int = 3
items: list[str] = ["a", "b"]
config: dict[str, int] = {"timeout": 10}

# управляющие конструкции
for item in items:
    if item == "a":
        print("нашли a")

# функции с типизацией
def greet(who: str) -> str:
    return f"Привет, {who}"

# исключения
try:
    value = int("не число")
except ValueError as exc:
    print("ошибка:", exc)
```

Работа с JSON и файлами (типичные задачи исследователя):

```
import json
from pathlib import Path

data = json.loads('{"results": [1, 2, 3]}') # строка JSON -> объект Python
print(len(data["results"]))                # 3
Path("out.json").write_text(json.dumps(data, ensure_ascii=False))
```

4.9. Изолированное окружение и зависимости

Никогда не ставьте пакеты глобально для проектов — используйте виртуальное окружение (venv):

```
python3 -m venv .venv
source .venv/bin/activate      # Windows: .venv\Scripts\activate
pip install -r requirements.txt
```

Зависимости фиксируйте с версиями (requirements.txt), чтобы примеры были **воспроизводимы**. Книга фиксирует версии и дату проверки совместимости (2026-10-19).

4.10. Тестирование кода

Инструмент без тестов легко ломается незаметно. Минимальный тест на pytest:

```
# test_demo.py
def add(a: int, b: int) -> int:
    return a + b

def test_add() -> None:
    assert add(2, 3) == 5
```

```
pytest -q          # запустить тесты
ruff check .       # линтер (стиль и частые ошибки)
mypy mymodule.py  # проверка типов
```

Инструменты книги (huntkit.*) сопровождаются автоматическими тестами и проверкой стиля и типов; это стандарт качества, которого мы придерживаемся во всех листингах (глава 12).

4.11. Практикум

Уровень 2. Соберите конвейер: из файла со списком URL оставьте только уникальные хосты, отсортируйте по частоте. (Подсказка: `awk -F/ '{print $3}' | sort | uniq -c | sort -rn.`)

Уровень 3. Напишите Python-скрипт, который читает JSON-ответ `/api/products/search` из файла и печатает названия товаров. Добавьте обработку случая «ключа results нет».

Уровень 4. Инициализируйте Git в учебном каталоге, сделайте два коммита с изменением файла и покажите `git diff` между ними. Объясните, зачем исследователю история экспериментов.

4.11a. Дополнительные задания (уровни 1–6)

Уровень 1. Объясните, что делает конвейер и перенаправление на примере `cmd > out`
`2> err`.

Уровень 2. Из лог-файла (раздел 4.5.1) выведите топ-5 IP и все 4xx/5xx.

Уровень 2. Прочитайте права `-rwxr-x---` и запишите эквивалентный `chmod`.

Уровень 3. Напишите Python-скрипт, читающий JSON-ответ и печатающий поля с безопасным доступом.

Уровень 4. Инициализируйте Git, сделайте два коммита и покажите `git diff` между ними.

Уровень 5. Напишите функцию с типами и тестом на `pytest`; прогоните `ruff` и `муру`.

Уровень 6. Соберите мини-конвейер из `curl + jq + sortuniq` -с для анализа ответа API.

4.12. Анализ ошибок новичка

Ставят пакеты глобально и получают конфликты версий — используйте `venv`.

Коммитят секреты и `.venv` — настройте `.gitignore` заранее.

Путают BSD и GNU утилиты (macOS/Linux) — проверяйте флаги.

Запускают код «из» каталога со скачанными данными — риск подхвата чужого кода.

Пишут инструменты без тестов — незаметные регрессии.

4.13. Резюме

Рабочее место исследователя — это Linux и терминал (навигация, права, процессы, окружение), мощная обработка текста (grep/sed/awk/jq), сетевая диагностика (curl/dig/openssl), Git для учёта экспериментов и Python для автоматизации в изолированном окружении с тестами. Эти навыки — несущая конструкция всех практикумов и главы 12.

4.14. Контрольные вопросы

Что делает конвейер `cat f | grep X | wc -l` и как заменить `cat` более идиоматично?

Как прочитать права `-rwxr-x---` и что означает `chmod 750`?

Какие инструменты вы примените для JSON, для полей лога, для замен в тексте?

Зачем `venv` и почему зависимости фиксируют с версиями?

Почему историю экспериментов полезно вести в Git?

Почему скачанные данные держат в отдельном каталоге и не запускают интерпретатор «из» него?

4.15. Источники

The Linux Documentation Project — <https://tldp.org/> (проверено 2026-10-19).

GNU Coreutils / grep / sed / gawk manuals — <https://www.gnu.org/manual/> (проверено 2026-10-19).

jq manual — <https://jqlang.github.io/jq/manual/> (проверено 2026-10-19).

Pro Git (книга) — <https://git-scm.com/book>

Конец ознакомительного фрагмента.

Текст предоставлен ООО «Литрес».

Прочитайте эту книгу целиком, [купив полную легальную версию](#) на Литрес.

Безопасно оплатить книгу можно банковской картой Visa, MasterCard, Maestro, со счета мобильного телефона, с платежного терминала, в салоне МТС или Связной, через PayPal, WebMoney, Яндекс.Деньги, QIWI Кошелек, бонусными картами или другим удобным Вам способом.