Security Engineering: часть 2. Протоколы и Контроль доступа

09.06.2026 12:14 edited 10.06.2026 15:24

Протоколы

В предыдущей части мы касались того, кто мы (психология) и что мы защищаем (политика), то эта часть отвечает на вопрос: **как** стороны взаимодействуют друг с другом безопасным образом. Определим протокол безопасности как набор правил, управляющих коммуникацией между участниками (принципалами) в условиях наличия злоумышленника. Ключевая идея: безопасность системы часто ломается не из-за слабости криптографии (алгоритмов шифрования), а из-за ошибок в дизайне самого протокола взаимодействия.

1. Что такое протокол безопасности?

Это не просто передача зашифрованных данных. Это сценарий обмена сообщениями, который должен гарантировать:

- **Аутентификацию** — уверенность в том, с кем вы говорите.
- **Целостность** — уверенность, что сообщение не было изменено.
- **Свежесть (freshness)** — уверенность, что сообщение не является повтором старого (replay attack).
- **Неотрекаемость (non-repudiation)** — невозможность для отправителя отрицать факт отправки.

Подчеркнем: проектирование протоколов — это «программирование компьютера Сатаны». Противник будет пытаться нарушить логику обмена, подделывая, перехватывая или изменяя сообщения.

2. Простая аутентификация и Challenge-Response

Самый базовый механизм — проверка пароля. Но передача пароля в открытом виде опасна (сниффинг).
Решение: **Challenge-Response (запрос-ответ)**.

- Сервер присылает случайное число (**nonce**).
- Клиент шифрует это число своим секретным ключом (паролем) и отправляет обратно.
- Сервер проверяет результат.

Плюс: пароль никогда не передаётся по сети.
Минус: уязвимость к атакам «человек посередине» (Man-in-the-Middle, MITM), если взаимное доверие реализовано неправильно.

3. Классические атаки на протоколы

Подробно разбирем хитроумные способы взлома, казалось бы, надёжных схем.

Атака «человек посередине» (Man-in-the-Middle)

Злоумышленник встаёт между Алисой и Бобом. Он притворяется Бобом для Алисы и Алисой для Боба, ретранслируя и модифицируя сообщения.
Пример: история южноафриканских лётчиков во время войны в Анголе. Кубинские МиГи перехватывали сигналы опознавания «свой-чужой» (IFF) от самолётов SAAF, передавали их своей ПВО, получали правильный ответ и ретранслировали его обратно, успешно проникая в воздушное пространство.

Атака отражения (reflection attack)

Возникает при взаимной аутентификации. Злоумышленник начинает сеанс с жертвой, получает запрос (challenge) и, вместо того чтобы вычислять ответ сам, открывает второй сеанс с той же жертвой, пересылая ей её же запрос. Жертва отвечает на свой же запрос, и злоумышленник использует этот ответ для завершения первого сеанса.
Защита: использовать разные ключи для разных направлений связи или включать имя отправителя в шифруемое сообщение.

Атаки на свежесть (replay attacks)

Запись легального сообщения и его повторная отправка позже.
Защита: временные метки (timestamps) или одноразовые номера (nonces / счётчики). Но временные метки требуют синхронизации часов, что само по себе уязвимость.

4. Управление ключами (key management)

Как безопасно распределить ключи между участниками?

- **Протокол Нидэма-Шрёдера (Needham-Schroeder):** классический протокол с использованием доверенной третьей стороны (TTP).
  Уязвимость: обнаружена спустя 17 лет после публикации. Протокол позволял возобновить старые сессионные ключи, если долгосрочный ключ был скомпрометирован в прошлом. Это показывает, насколько сложна верификация протоколов.
- **Kerberos:** практическая реализация идей Нидэма-Шрёдера, широко используемая в Windows и Unix-сетях. Использует билеты (tickets) и временные метки для предотвращения повторного воспроизведения.

5. Формальная верификация

Можно ли доказать математически, что протокол безопасен?

- **Логика BAN (Burrows-Abadi-Needham):** формальный метод для анализа убеждений участников протокола («Алиса верит, что ключ K свежий и принадлежит Бобу»).
- **Ограничения:** логика работает, только если исходные предпосылки верны. Она может доказать корректность абстрактной модели, но не учтёт ошибки реализации или физические атаки. Пример с Needham-Schroeder показал, что даже «доказанные» протоколы могут иметь скрытые изъяны.

Формальная верификация — мощный фильтр для отсеивания логических ошибок, но она не заменяет тщательный инженерный анализ реализации и контекста использования.

6. Атаки на составные протоколы (chosen protocol attacks)

Одна из самых коварных проблем современных систем — взаимодействие нескольких протоколов. Протокол, безопасный сам по себе, может стать уязвимым, если его ключи или сообщения используются в другом контексте.

Атака «мафия посередине» (mafia-in-the-middle)

Это развитие идеи MITM, но с акцентом на переиспользование легитимных действий пользователя для мошенничества.
Сценарий: злоумышленник создаёт фиктивный сайт (порно-сайт или бесплатный сервис), который требует от пользователя подтверждения личности с помощью той же смарт-карты или токена, что используется для банковских операций.

Механизм:
1. Жертва заходит на сайт злоумышленника.
2. Сайт инициирует протокол аутентификации, пересылая запрос банку от имени жертвы.
3. Банк присылает challenge.
4. Сайт передаёт этот challenge жертве.
5. Жертва вводит ПИН в свой токен / смарт-карту и получает ответ.
6. Ответ передаётся обратно банку.
7. Банк считает, что жертва авторизовала транзакцию (например, перевод денег), хотя жертва думала, что просто входит на веб-сайт.

Причина уязвимости: отсутствие привязки контекста. Протокол не различает «вход на сайт» и «подтверждение платежа», если оба используют одну и ту же криптографическую операцию с одними и теми же ключами.
Защита: явное включение типа операции, суммы, получателя и других контекстных данных в подписываемое сообщение. Ключи для разных целей должны быть разделены.

7. Проблемы реальных реализаций: EMV и другие стандарты

Теория часто расходится с практикой из-за необходимости обратной совместимости, сложности стандартов и человеческого фактора.

Уязвимости EMV (Chip and PIN)

- **Атака отказа от ПИНа (No-PIN attack):** скомпрометированный терминал может не запрашивать ПИН у карты, но отправлять банку сообщение, что ПИН был введён верно. Если банк доверяет терминалу больше, чем карте, транзакция проходит.
- **Проблема интерфейса доверия:** пользователь вводит ПИН в терминал, но не имеет способа проверить, действительно ли этот терминал защищён. Карта доверяет терминалу, а терминал может быть злонамеренным.
- **Релейная атака (relay attack):** даже если ПИН защищён, можно ретранслировать сигналы между картой в кармане жертвы и терминалом в магазине, заставляя карту думать, что она находится рядом с терминалом.

Проблема обратимости и совместимости

Многие системы сохраняют поддержку старых, менее защищённых режимов работы (fallback modes) для совместимости со старым оборудованием. Злоумышленники часто атакуют именно эти режимы, принудительно переключая систему в небезопасное состояние. Пример: принудительное использование магнитной полосы вместо чипа в системах EMV, если терминал «якобы» не поддерживает чипы.

8. Фундаментальные принципы проектирования безопасных протоколов

На основе анализа многочисленных неудач выделяется несколько принципов, нарушение которых почти гарантированно ведёт к уязвимостям:

1. **Явность (explicitness).** Все существенные параметры (имена участников, тип сообщения, nonce, временные метки, контекст) должны быть явно указаны в сообщении. Не полагайтесь на неявный контекст или позицию поля в пакете.
2. **Свежесть (freshness).** Всегда используйте надёжные механизмы предотвращения replay-атак (nonce, таймстампы с защитой от рассинхронизации, счётчики). Убедитесь, что значение свежести нельзя предсказать или переиспользовать.
3. **Разделение ключей (key separation).** Никогда не используйте один и тот же ключ для разных целей (шифрование, аутентификация, подпись) или в разных протоколах без явного разделения (например, добавлением префикса к данным перед шифрованием).
4. **Минимизация доверия (least privilege in protocols).** Протокол должен раскрывать минимум информации, необходимой для выполнения задачи. Избегайте передачи избыточных данных, которые могут быть использованы для анализа трафика или будущих атак.

5. **Защита от активного противника.** Проектируйте протокол, предполагая, что противник может создавать, модифицировать, удалять и задерживать любые сообщения в сети. Не предполагайте, что канал связи надёжен или что участники ведут себя честно.
6. **Простота.** Чем сложнее протокол, тем выше вероятность ошибки. Избегайте излишней сложности и большого количества опций. Каждая дополнительная ветвь логики — потенциальная уязвимость.

9. Эволюция угроз и адаптация протоколов

Безопасность протоколов — это гонка вооружений.

- **Вычислительная мощность.** Алгоритмы, считавшиеся безопасными 20 лет назад (DES, RSA с коротким ключом), сегодня легко взламываются. Протоколы должны позволять гибкое обновление криптографических примитивов.
- **Новые векторы атак.** Появление квантовых компьютеров угрожает асимметричной криптографии (RSA, ECC). Протоколы будущего должны учитывать постквантовую криптографию.
- **Изменение моделей угроз.** Протоколы, разработанные для закрытых корпоративных сетей, часто оказываются небезопасными в открытом интернете, где любой узел может быть враждебным.

Главный вывод

Безопасность нельзя достичь только мощным шифрованием. Если логика обмена сообщениями имеет изъяны, злоумышленник может обойти криптографию, манипулируя потоком данных. Инженер должен думать как противник, предвидя все возможные пути искажения диалога между системами.
Ошибка в дизайне протокола катастрофична: её нельзя исправить патчем на уровне приложения — требуется замена самого протокола. Контекст решает всё: один и тот же протокол может быть безопасным в одной среде и полностью уязвимым в другой. Даже идеально спроектированный протокол может быть скомпрометирован из-за ошибок реализации, плохого управления ключами или социальной инженерии.

Контроль доступа

Если мы ранее обсуждали, кто мы (психология) и как общаемся (протоколы), то сейчас мы ответим на вопрос: **что** конкретному пользователю или процессу разрешено делать с ресурсами системы.
Подчеркнем: контроль доступа — традиционный центр тяжести компьютерной безопасности, где теория встречается с практикой. Однако чем выше уровень абстракции (от железа к приложениям), тем сложнее становится управление доступом и тем больше вероятность ошибок.

1. Уровни контроля доступа

Опишем иерархию механизмов защиты:

- **Аппаратный уровень:** защита памяти, сегментация адресного пространства (процессор не даёт одному процессу читать память другого).
- **Уровень ОС:** управление файлами, портами, устройствами (пользователь `alice` может читать `file.txt`, но не может писать в `/etc/passwd`).
- **Уровень middleware (ПО промежуточного слоя):** базы данных (кто может выполнять запросы к таблицам), системы очередей сообщений.
- **Прикладной уровень:** бизнес-логика (менеджер может одобрить платёж до $10 000, но свыше требуется подпись директора).

Главная проблема: большинство реальных мошенничеств происходит на прикладном уровне, когда сотрудники злоупотребляют доверенными функциями или находят лазейки в логике, которую программисты не защитили так тщательно, как ядро ОС.

2. Модели управления доступом

Матрица доступа и её реализация

Теоретически права можно представить как матрицу: строки — субъекты (пользователи), столбцы — объекты (файлы), ячейки — права (чтение, запись, исполнение). Для больших систем эта матрица слишком огромна и разрежена. Поэтому используют два основных способа её сжатия.

Списки контроля доступа (ACL – Access Control Lists). Хранение прав «по столбцам» (привязка к объекту). Пример: у файла `secret.doc` есть список: `{Alice: rw, Bob: r}`.
Плюсы: легко понять, кто имеет доступ к конкретному файлу; легко отозвать доступ у всех сразу, удалив файл.
Минусы: трудно узнать, к каким файлам имеет доступ конкретный пользователь; сложно делегировать права.
Реализация: Unix (упрощённо — владелец/группа/остальные), Windows (полные ACL), NFS.

Возможности (capabilities / tickets). Хранение прав «по строкам» (привязка к субъекту). Пример: у пользователя Alice есть «билет», дающий право на чтение `file.txt`. Если она передаст этот билет Бобу, Боб тоже сможет читать.
Плюсы: лёгкое делегирование; высокая производительность проверки (не нужно сканировать списки).
Минусы: сложно отозвать право (нужно найти все выданные билеты); сложно узнать, кто ещё имеет доступ к файлу.
Реализация: Kerberos (частично), некоторые микроядра, современные токены в веб-приложениях (JWT).

Роли и группы (RBAC – Role-Based Access Control)

Чтобы не управлять правами для каждого человека отдельно, используются группы и роли.

- **Группа:** просто список пользователей (например, «Бухгалтеры»).
- **Роль:** набор функций, которые может выполнять человек в определённый момент (например, «Дежурный врач»). Один человек может иметь несколько ролей, но активна обычно одна.

Отмечаем: RBAC стала стандартом де-факто в корпоративных системах, так как она отражает организационную структуру бизнеса.

3. Реализация в популярных ОС

Unix / Linux

- Исторически простая модель: владелец / группа / остальные (rwx).
- Проблема: слишком грубая гранулярность.
- Механизм `setuid`: программа запускается с правами владельца файла (часто root), а не того, кто её запустил. Это мощный, но опасный механизм — источник многих уязвимостей (если программа с `setuid` содержит баг, злоумышленник получает root).
- Современные расширения: **SELinux, AppArmor** (принудительный контроль доступа, MAC), которые позволяют задавать сложные политики («браузер может писать только в /tmp, но не может читать /home»).

Windows

- Сложные ACL с наследованием прав.
- Понятие доменов и доверительных отношений между ними.
- **UAC (User Account Control):** появился в Vista и Windows 7. Даже администратор работает в обычном режиме, а для критических действий требуется подтверждение (элевация прав). Это попытка уйти от модели «всегда работай под root», которая была нормой в XP.

4. Что идёт не так? Уязвимости

Посвятим немного времени тому, как ломаются эти системы.

Переполнение стека (smashing the stack)

Классическая атака: злоумышленник переполняет буфер ввода данными, содержащими исполняемый код, и перезаписывает адрес возврата в стеке, заставляя процесс выполнить вредоносный код с правами программы.

Эволюция защиты:
* **StackGuard:** специальный «канареечный» значение (canary) помещается перед адресом возврата. Если буфер переполняется, канарейка изменяется, и программа аварийно завершается.
* **NX-bit (No-Execute):** аппаратная защита, помечающая области памяти (стек и кучу) как неисполняемые.
* **ASLR (Address Space Layout Randomization):** случайное размещение областей памяти усложняет предсказание адресов для внедрения кода.

Гонки данных (race conditions / TOCTTOU)

Атаки типа **Time-of-Check to Time-of-Use (TOCTTOU)** эксплуатируют временной разрыв между проверкой прав доступа и фактическим использованием ресурса.

Механизм: злоумышленник создаёт символическую ссылку (symlink) на целевой файл сразу после того, как программа проверила права на исходный файл, но до того, как открыла его для записи.
Классический пример с `/tmp`: если программа с привилегиями root создаёт временный файл предсказуемым именем (например, `/tmp/pid123.tmp`) без безопасных системных вызовов (типа `mkstemp` с флагом `O_EXCL`), злоумышленник может заранее создать symlink с этим именем, указывающий на `/etc/passwd`. Когда привилегированная программа запишет данные, она перезапишет файл паролей.
Защита: атомарные операции ядра, случайные имена файлов, открытие файлов по дескрипторам (а не по именам), минимизация времени между проверкой и использованием.

Человеческий фактор и интерфейсы

Пользователи часто дают программе лишние права («Запустить от имени администратора»), не понимая последствий. Разработчики ленятся реализовывать тонкий контроль доступа, давая программе полные права «на всякий случай».

5. Песочницы (sandboxing) и виртуализация

Как ограничить вред от ненадёжного кода?

- **Песочницы:** изолированная среда, где код может работать, но его возможности строго ограничены (нет доступа к сети, диску и т.д.). Пример: Java Applets, плагины браузеров.
- **Виртуализация:** запуск целых ОС внутри другой ОС. Позволяет изолировать приложения на уровне ядра. Если вирус заразил гостевую ОС, хост-система остаётся чистой.
- **Trusted Computing (доверенные вычисления):** инициатива Microsoft и Intel (TPM-чипы). Попытка создать аппаратно защищённую среду для выполнения критического кода (DRM, защита ключей), изолированную даже от самой ОС. Можно относится к этому скептически, указывая на проблемы с удобством и реальным уровнем доверия.

6. Мандатный контроль доступа (MAC) и SELinux

В то время как дискреционный контроль (DAC), используемый в стандартных Unix/Windows, полагается на владельца файла, **мандатный контроль доступа (MAC)** принудительно навязывается системой на основе глобальной политики, которую пользователь не может изменить.

Type Enforcement и SELinux

Одна из самых мощных реализаций MAC — **Type Enforcement**, лежащая в основе **SELinux** (Security-Enhanced Linux), разработанного АНБ и интегрированного в Linux.

- **Принцип:** каждому субъекту (процессу) и объекту (файлу, порту) присваивается тип (security context). Политика определяет матрицу разрешений: какие типы процессов могут обращаться к каким типам объектов и каким образом.
- **Преимущество перед DAC:** даже если злоумышленник получит права root (UID 0), он всё равно ограничен политикой SELinux. Процесс Apache, даже запущенный от root, не сможет читать файлы домашнего каталога пользователя или изменять системные конфиги, если это явно не разрешено.
- **Domain and Type Enforcement (DTE):** расширение, позволяющее группировать процессы в домены и определять правила перехода между ними. Это позволяет создавать «песочницы» и защищённые конвейеры обработки данных.

Проблема сложности

Главный недостаток MAC (особенно SELinux) — сложность настройки. Неправильная политика может заблокировать легитимную работу приложений. Поэтому многие дистрибутивы по умолчанию запускают SELinux в режиме `permissive` (только логирование нарушений), что снижает реальную защиту.

7. Проблема доверенного интерфейса (trusted interface)

Даже самая надёжная система контроля доступа бесполезна, если пользователь не может доверять тому, что он видит на экране.
Суть проблемы: пользователь вводит пароль или подтверждает транзакцию, полагая, что взаимодействует с легитимной системой (банковским приложением или диалогом входа ОС). Однако вредоносное ПО (троян) может подделать этот интерфейс (подмена окна, UI redressing).

- **Пример с банкоматами и смарт-картами:** злоумышленник устанавливает накладку на клавиатуру или подменяет дисплей. Пользователь видит запрос ПИН-кода от «банка», но на самом деле вводит его в устройство злоумышленника.
- **Пример с ПК (trusted path):** в Windows комбинация `Ctrl+Alt+Del` реализуется на уровне ядра, чтобы ни одно приложение не могло перехватить её и подделать окно входа. Однако в сложных графических средах и веб-приложениях обеспечить такой доверенный путь крайне сложно.

- **Последствия:** без доверенного интерфейса механизмы вроде двухфакторной аутентификации или цифровых подписей могут быть скомпрометированы — пользователь подписывает или подтверждает не то, что думает.

8. Экономика и стимулы в контроле доступа

Вновь обращаем внимание на то, что технические меры часто проигрывают из-за неправильных экономических стимулов.

- **Дампинг рисков (risk dumping).** Производители ПО часто проектируют системы контроля доступа так, чтобы переложить ответственность на пользователя. Пример: приложения запрашивают права администратора для любых действий, чтобы упростить разработку. Если происходит взлом, производитель заявляет: «Пользователь сам дал права».
- **UAC в Windows.** Хотя User Account Control пытается ограничить права по умолчанию, пользователи привыкли автоматически нажимать «Разрешить» на любые запросы, сводя эффективность механизма к нулю.
- **Конфликт удобства и безопасности.** Системы с жёстким контролем доступа (SELinux, строгие корпоративные политики) часто воспринимаются как препятствие для работы. Это приводит к появлению «теневых IT» решений, обходу правил и отключению защитных механизмов.
- **Ответственность за ошибки.** В отличие от физической безопасности (где производитель замка несёт репутационные риски за лёгкий взлом), в ПО ошибки контроля доступа часто считаются «особенностями» до тех пор, пока не произойдёт крупный инцидент. Отсутствие юридической ответственности за небезопасный код снижает мотивацию делать системы действительно защищёнными.

9. Виртуализация как новый периметр

С развитием виртуализации (VMware, Xen, Hyper-V) концепция контроля доступа расширяется.

- **Изоляция вместо прав:** вместо тонкой настройки прав доступа к файлам внутри одной ОС разные уровни доверия разделяются на разные виртуальные машины.
- **Проблема гипервизора:** гипервизор становится новой основой доверенной вычислительной базы (TCB). Если злоумышленник сможет выполнить побег из виртуальной машины (VM escape) и получить контроль над гипервизором, он получит доступ ко всем изолированным системам.
- **Ковертные каналы (covert channels):** виртуализация создаёт новые возможности для скрытой передачи информации между изолированными VM через общие ресурсы (кэш процессора, использование памяти), что требует новых методов контроля.

Главный вывод

Контроль доступа — это не просто настройка списков ACL или прав пользователей. Это постоянная борьба между:

- теоретической моделью (которая может быть математически строгой, как Bell-LaPadula или Biba);
- реализацией (уязвимой для гонок, переполнений и логических ошибок);
- человеческим фактором (пользователи обходят ограничения ради удобства);
- экономикой (производители не хотят нести издержки за безопасность).

Надёжная система контроля доступа требует сочетания:
- минимизации привилегий (principle of least privilege);
- использования мандатных механизмов (MAC / SELinux) для критических компонентов;
- защиты от классов уязвимостей памяти (StackGuard, ASLR);
- обеспечения доверенного пути ввода-вывода;
- правильных стимулов для разработчиков и пользователей.