ChatGPT

Безопасность интернет-магазина: руководство для начинающих разработчиков

Введение

Интернет-магазины обрабатывают большие объёмы конфиденциальных данных (личные данные пользователей, пароли, платежная информация), поэтому являются привлекательной мишенью для хакеров [1]. Успешная атака может привести к утечке данных клиентов, финансовым потерям и потере доверия. По оценкам, до **60%** мелких компаний, подвергшихся взлому, закрываются в течение полугода [2]. Кроме того, последствия взлома могут повлечь юридическую ответственность и штрафы (например, несоблюдение стандартов безопасности PCI DSS при утечке данных платежей) [3]. Поэтому задача веб-разработчика – обеспечить всестороннюю защиту интернет-магазина. В этом материале мы рассмотрим основные понятия веб-безопасности и практические методы защиты данных, заказов, сессий и других компонентов интернет-магазина.

> **Важно:** Главное правило – **никогда не доверяйте входящим данным от клиента**. Любые данные из браузера (формы, параметры URL, cookies, заголовки, файлы) должны считаться потенциально опасными. Всегда проверяйте и очищайте (санитизируйте) любые вводимые пользователем данные, предполагая худший сценарий [4]. Это ключевой принцип, который мы будем применять во всех рассмотренных техниках защиты.

Ключевые понятия веб-безопасности

Перед тем как перейти к практическим советам, разберём основные понятия, связанные с безопасностью веб-приложений, особенно актуальные для интернет-магазинов. Понимание этих терминов поможет осознать, какие уязвимости существуют и как их предотвращать.

Сессии

Сессия – это механизм сохранения данных о пользователе между запросами. Поскольку HTTP-протокол **без сохранения состояния** (stateless), сервер не помнит пользователя между обращениями. Сессии решают эту проблему: при входе на сайт сервер создаёт уникальный идентификатор сессии и сохраняет данные о пользователе (например, его ID или права) на сервере, ассоциируя их с этим идентификатором. Идентификатор сессии обычно передаётся клиенту в виде cookie (например, `PHPSESSID`) и автоматически отправляется браузером с каждым запросом к серверу [5]. Получив cookie с идентификатором, сервер «понимает», какой это пользователь, и может извлечь данные его сессии.

Безопасность сессии: Идентификатор сессии – это своего рода «ключ» к аккаунту пользователя. Если злоумышленник его похитит (например, через XSS-уязвимость или перехват трафика), он сможет выдавать себя за этого пользователя. Поэтому важно:

- Генерировать сложные, криптостойкие ID сессий (это обеспечивается движком, например PHP генерирует достаточно случайные идентификаторы по умолчанию).
- Передавать сессионные cookie только по защищённому соединению HTTPS (устанавливать флаг **Secure**), чтобы предотвратить их перехват в сети [6].
- Ограничить срок действия сессий и выполнять **разлогин** (инвалидировать сессию) после длительного бездействия.
- Устанавливать флаг **HttpOnly** для сессионного cookie, чтобы снизить риск кражи сессионного ID через XSS (браузер запретит скриптам читать такой cookie) [7].
- При повышении привилегий (например, после логина) – пересоздавать сессию (менять её ID), чтобы затруднить фиксацию сессии.

Таким образом, сессии – удобный способ сохранять состояние между запросами, но их нужно защищать, чтобы злоумышленник не «угнал» сессию пользователя. В традиционных веб-приложениях сессии на основе cookie являются стандартным решением для аутентификации и хранения данных корзины, фильтров и т.п.

Токены доступа, refresh-токены и JWT

Токены – альтернативный или дополнительный способ аутентификации, часто используемый в современных веб-API и SPA (Single Page Applications). Токен – это некий секрет (строка), который выдается пользователю после успешного логина и предъявляется при каждом последующем запросе для подтверждения его личности.

Access Token (токен доступа) – короткоживущий токен, дающий доступ к ресурсам. Обычно действует недолго (несколько минут или часов). При запросе к серверу токен доступа передаётся, например, в заголовке `Authorization: Bearer ...`. Если он валиден, сервер допускает запрос. Когда токен доступа истекает, требуется его обновить. Тут вступает в игру **Refresh Token**.

Refresh Token (обновляющий токен) – более «долгоиграющий» токен, используемый исключительно для получения нового access-токена без повторного ввода логина и пароля. Refresh-токен хранится более длительное время (например, дни или недели) и тоже должен храниться безопасно. При истечении токена доступа клиент может отправить refresh-токен серверу (например, на специальный endpoint обновления), и если refresh-токен действителен, сервер выдаст новый токен доступа (а иногда и новый refresh-токен). Таким образом достигается баланс безопасности и удобства: короткоживущий access-token затрудняет злоумышленнику долгосрочное использование украденного токена, а refresh-token позволяет пользователю оставаться залогиненным без постоянного ввода пароля.

JWT (JSON Web Token) – популярный формат токенов, представляющий собой зашифрованный или подписанный JSON. JWT состоит из трех частей (header.payload.signature), кодированных в Base64. В payload обычно помещают **претензии** (claims) – данные о пользователе и срок действия токена. Подпись обеспечивает целостность: сервер может проверить, что токен не был подделан. JWT часто выступает в роли access-токена. Его плюс – **статус автономности**: сервер может аутентифицировать запрос, просто проверив подпись JWT, без поиска токена в базе (в отличие от классических сессий) [8]. Минус – отозвать JWT до истечения срока сложно, если не хранить где-то состояние (например, «чёрный список» отозванных токенов). Поэтому JWT хорошо подходят для API, где сервер не хранит сессии, а доверяет информации, заложенной в токене.

Как хранить токены: - В веб-приложении токен доступа (особенно JWT) часто хранят в памяти приложения или `localStorage`. Однако хранение в `localStorage` уязвимо для XSS – вредоносный скрипт может украсть токен. Безопаснее хранить токены (особенно refresh-токен) в

HttpOnly cookie, чтобы JavaScript не мог их прочитать [9]. - Refresh-токен **никогда** не должен храниться в местоположении, доступном JavaScript (HttpOnly cookie – хороший вариант). Токен доступа, если он краткоживущий, иногда хранят в памяти (не в персистентном хранилище) и обновляют при перезагрузке страницы из refresh-токена, полученного через cookie.

Сессии vs токены: В классических веб-сайтах (как интернет-магазин с серверной генерацией страниц) часто используют сессии с cookie – это проще и безопасно «из коробки». Токены (особенно JWT) популярны в микро-сервисах, мобильных приложениях и SPA, где нужен автономный метод проверки на стороне API без хранения сессий. Считайте токен аналогом «удостоверения», которое клиент предъявляет на каждый запрос. При правильном использовании токены и сессии обеспечивают сопоставимый уровень безопасности, однако требуются меры защиты: например, **не хранить JWT в небезопасном месте**, **проверять срок действия токена**, **использовать HTTPS для передачи** и пр.

Cookies (куки)

Cookie – это небольшой фрагмент данных (пары «имя=значение»), который сервер отправляет браузеру, а браузер сохраняет и автоматически добавляет в каждый последующий запрос к тому же домену. Cookie – основной способ хранения сессионного идентификатора на стороне клиента. Кроме того, cookies могут использоваться для хранения предпочтений, меток корзины для гостя, трекинга активности и т.п.

Безопасность cookie: Поскольку cookie автоматически передаются браузером, они становятся целью для атак. Для защиты cookie предусмотрены специальные атрибуты:
- **Secure:** помечает cookie как «отправлять только по HTTPS». Это предотвращает перехват cookie в незашифрованном виде по сети (например, в открытом Wi-Fi) [10] [6]. **Все чувствительные cookie (сессии, токены)** должны иметь флаг Secure, т.к. в противном случае их можно украсть с помощью «прослушки» трафика.
- **HttpOnly:** запрещает доступ к cookie из JavaScript. Таким образом, даже если на странице выполняется вредоносный скрипт (XSS), он не сможет прочитать значение HttpOnly cookie [7]. Этот флаг рекомендуется для всех аутентификационных cookie (сессионные идентификаторы, JWT refresh-token и пр.), чтобы затруднить их кражу через XSS.
- **SameSite:** ограничивает отправку cookie в межсайтовых запросах. Значение `SameSite=Lax` (по умолчанию во многих браузерах) означает, что для обычной навигации (GET-запросы) cookie будут отправляться, но не будут при POST-запросах с внешних сайтов. `SameSite=Strict` – самый строгий режим, cookie не пойдут даже при переходе по ссылке с другого сайта. `SameSite=None` отключает этот механизм (обычно применять не нужно, а если применяют, то **только вкупе с Secure**, иначе браузеры игнорируют атрибут [11]). Правильная настройка SameSite может **смягчить риск CSRF-атак**, не позволяя посторонним сайтам незаметно отправлять ваши cookies, но полагаться только на SameSite недостаточно – лучше сочетать его с другими методами (CSRF-токены, проверка Referer и пр.).
- **Домен и путь:** определяют, к какому диапазону URL относится cookie. По умолчанию, cookie привязан к текущему домену. Можно указать поддомен или путь. Обычно в целях безопасности не стоит выставлять домен шире, чем нужно (например, если приложение на `shop.example.com`, не делать cookie доступным для всего `.example.com`, если в этом нет необходимости).

Cookie – простой, но мощный механизм. Грамотно применяя упомянутые атрибуты, вы защищаете cookies интернет-магазина от распространённых атак: перехвата (Secure), XSS-кражи (HttpOnly) и CSRF (SameSite).

CSRF (Cross-Site Request Forgery) – межсайтовая подделка запроса

CSRF – тип атаки, при котором злоумышленник заставляет браузер жертвы выполнить непреднамеренный запрос к доверенному сайту от имени авторизованного пользователя, **без его ведома** [12] [13]. Браузер по умолчанию прикрепляет cookies (в том числе сессионные) ко **всем** запросам к сайту, даже если запрос инициирован сторонней страницей или скриптом [14] [15]. Эту особенность и эксплуатирует CSRF.

Как происходит атака: Представим, что пользователь вошёл в интернет-банк и его сессия действительна. Злоумышленник (назовём его Джон) знает, что в банке есть URL для перевода денег: например, запрос `POST /transfer` с параметрами `account=номер_счёта&amount=100`. Джон создаёт на своём сайте или в письме HTML-форму: `<form action="https://bank.com/transfer" method="POST">` и вставляет в неё скрытые поля: `<input type="hidden" name="account" value="СЧЁТ_ДЖОНА">` и `<input type="hidden" name="amount" value="10000">`. Также он делает автоматическую отправку формы (через JavaScript или `<body onload="form.submit()">`). Когда пользователь, будучи авторизованным в банке, откроет злонамеренную страницу Джона, его браузер **сам отправит** запрос на перевод 10000 со счёта пользователя на счёт Джона, прикрепив к запросу сессионные cookie пользователя. Банк, получив запрос с корректными cookies, подумает, что это законное действие самого пользователя, и выполнит перевод [16] [17]. В результате деньги пользователя будут переведены Джону – атака удалась, хотя жертва ничего не подозревает.

Защита от CSRF: Основной подход – **добавление токенов-вкладок (anti-CSRF токенов)** в критичные запросы. Суть в том, что каждый защищённый HTML-формуляр содержит скрытое поле с уникальным случайным токеном, сгенерированным сервером для данного пользователя и данной сессии. Сервер при приёме POST-запроса проверяет, что вместе с запросом пришёл правильный токен. Злоумышленнику трудно угадать или узнать этот токен, поэтому он не сможет создать корректно подписанный запрос [18] [19]. Например, при отображении страницы с формой заказа сервер сгенерирует токен `csrf_token` и сохранит его значение в сессии. В форму он вставит: `<input type="hidden" name="csrf_token" value="...случайное значение...">`. Когда пользователь отправит форму, сервер сравнит полученный `csrf_token` со значением в сессии. Если токен отсутствует или не совпадает – запрос отклоняется. Таким образом, даже если злоумышленник заставит браузер отправить запрос на оформление заказа, он не сможет подменить **правильный токен**, и сервер отклонит такой запрос как подозрительный.

Дополнительные меры:
- **SameSite=Lax/Strict** для сессионного cookie – снижает вероятность автоматической отправки cookie в кросс-доменных запросах (хотя полностью не исключает CSRF, например, при прямом переходе по ссылке SameSite=Lax cookie всё равно отправятся).
- **Проверка заголовка Origin/Referer:** Сервер может проверять, с какого домена пришёл запрос. Браузеры обычно указывают заголовок `Origin` для POST-запросов. Если он есть и не соответствует вашему домену – запрос отклонять. Этот метод – вспомогательный, его стоит использовать вместе с токенами.
- **Отдельные авторизационные механизмы для критических операций:** Например, при изменении пароля или проведении платежа запрашивать у пользователя повторный ввод пароля или 2FA-код – это защитит даже если сессия пользователя скомпрометирована для CSRF.

В современных фреймворках защита CSRF часто встроена (например, в Laravel, Django, ASP.NET есть генерация и проверка токенов из коробки) [20]. Разработчику важно не отключать эти

механизмы и правильно включать токены во все формы, которые совершают состояние изменяющие действия (transfer, order, profile update и т.п.).

XSS (Cross-Site Scripting) – межсайтовый скриптинг

XSS – один из самых распространённых видов уязвимостей веб-приложений [21]. Эта атака позволяет злоумышленнику **внедрить вредоносный клиентский скрипт** на страницы вашего сайта, который затем выполнится в браузерах пользователей. Так как скрипт загружен с доверенного сайта, браузер **доверяет** ему все привилегии этого сайта: скрипт может получать доступ к cookies, выполнять действия от имени пользователя, отображать фальшивый контент и т.д. [12]. Например, вредоносный код, получив доступ к `document.cookie`, способен отправить сессионный идентификатор пользователя злоумышленнику. Имея этот cookie, атакующий сможет войти на сайт под аккаунтом жертвы и делать всё, что разрешено пользователю – просматривать личные данные, номера карт, оформлять заказы [22].

Как XSS проникает в приложение: Через вывод пользовательских данных **без должной обработки**. XSS бывает:
- **Reflected XSS (отражённый)** – вредоносный ввод сразу «отражается» в ответе. Например, поиск по сайту выводит фразу, которую ввёл пользователь. Если просто взять параметр `q` из URL и вставить в HTML, злоумышленник может передать в ссылке скрипт: `https://site.com/search?q=<script>...`. Если сайт не экранирует спецсимволы, этот скрипт встроится в страницу результатов и выполнится у пользователя, кликнувшего по такой ссылке [23] [24].
- **Stored XSS (хранимый)** – опаснее, т.к. вредоносный код **сохраняется** на сервере (например, в базе) и затем показывается многим пользователям. Это возможно в разделах с пользовательским контентом: отзывы, комментарии, названия товаров, профили и т.п. Если сайт не фильтрует HTML-теги, атакующий может оставить комментарий вида: `<script>...кража cookie...</script>`. Этот комментарий сохранится в БД, и при просмотре страницы всеми пользователями скрипт будет выполняться у каждого, крадя их данные [25].

Существуют и другие типы (DOM-based XSS – когда уязвимость на стороне JavaScript фронтенда), но суть одна: **неочищенные данные пользователя интерпретируются браузером как код**.

Защита от XSS:
1. **Экранирование (escaping) вывода.** Никогда напрямую не выводите в HTML данные, полученные от пользователя (через формы, параметры URL, БД), не преобразовав специальные символы. Все `<`, `>`, `&`, `'`, `"` должны быть заменены на безопасные HTML-сущности (`<`, `>` и т.д.), чтобы пользовательский ввод воспринимался как текст, а не как тег или скрипт [26] [27]. В PHP для этого есть функции `htmlspecialchars()` и `htmlentities()`. Первая конвертирует базовые опасные символы (' " < > &), вторая – все символы с HTML-представлением [26]. Например:

```
$comment = $_POST['comment'];
echo htmlspecialchars($comment, ENT_QUOTES, 'UTF-8');
```

Даже если `$comment` содержит `<script>alert('xss')</script>`, после `htmlspecialchars` вывод будет `&lt;script&gt;alert('xss')&lt;/script&gt;` – браузер отобразит это как простой текст, и скрипт **не выполнится** [27] [28]. Экранирование нужно делать **всегда при выводе** переменных в HTML, за исключением случаев, когда вы намеренно вставляете

доверенный HTML. Это простое правило устраняет подавляющее большинство XSS. В шаблонизаторах и фреймворках по умолчанию переменные часто экранируются автоматически.

1. **Очистка HTML (sanitization).** Если приложение позволяет вводить HTML-разметку (например, отзыв с форматированием), следует *очищать* её, разрешая только безопасный подмножество тегов. Для PHP существуют библиотеки вроде HTMLPurifier, которые удаляют опасные конструкции (скрипты, on<Event> обработчики, встроенные объекты и др.). Никогда не доверяйте «белому списку» от пользователя (например, попытке вырезать только `<script>`), так как XSS можно вставить и через другие теги (например, `<img onerror="...">`). Лучше либо не позволять ввод HTML вообще, либо использовать проверенные библиотеки для его очистки [29].

2. **Content Security Policy (CSP).** Это специальный заголовок, который можно настроить на сервере, ограничивающий выполнение скриптов на странице. Например, CSP может запретить выполнение встроенных (`<script>` в HTML) скриптов, разрешив только загрузку из доверенных источников. CSP – мощный механизм, который сильно затрудняет эксплуатацию XSS, хотя требует тщательной настройки. На первых порах достаточно знать о нём: по мере роста приложения стоит рассмотреть внедрение CSP для дополнительной защиты.

3. **Дополнительные меры:** Используйте современные фреймворки, которые **по умолчанию экранируют вывод**. Всегда проверяйте пользовательский ввод (валидируйте типы, длины – это косвенно тоже помогает, отсекая явно «левые» данные). Не забывайте, что XSS возможен не только в HTML, но и, к примеру, в атрибутах, URL и даже в JavaScript, работающем с DOM. Принцип прост: **любые данные из внешнего мира – не доверять, обрабатывать перед выводом.**

Итог: защита от XSS крайне важна. Без неё злоумышленник может похищать сессии пользователей или совершать действия от их имени, что особенно критично для интернет-магазина (кража личных данных, адресов, подмена реквизитов оплаты). Благо, методы защиты несложны – главное, соблюдать дисциплину кодирования.

SQL-инъекция

SQL-инъекция – ещё одна критическая уязвимость, при которой злоумышленник внедряет произвольный SQL-код в ваши запросы к базе данных [30]. Если не фильтровать и не параметризовать пользовательские данные в SQL-запросах, атакующий может изменить логику запросов: получить доступ к данным, которые ему не должны быть доступны, либо внести разрушающие изменения в базу.

Пример инъекции: Пусть при входе на сайт выполняется такой PHP-код:

```
$user = $_POST['username'];
$pass = $_POST['password'];
$sql = "SELECT * FROM users WHERE username='$user' AND password='$pass'";
$result = mysqli_query($conn, $sql);
```

На первый взгляд всё нормально, но если злоумышленник в поле `username` введёт:

```
' OR '1'='1
```

то запрос станет:

```
SELECT * FROM users WHERE username='' OR '1'='1' AND password='...';
```

Логика изменится: условие `OR '1'='1'` всегда истинно, в результате запрос вернёт **всех пользователей**, и проверка пароля может быть обойдена. Это упрощённый пример. Более разрушительный случай – **конкатенация нескольких команд**. Например: в поле `username` введён

```
a'; DROP TABLE users; --
```

Запрос станет:

```
SELECT * FROM users WHERE username='a'; DROP TABLE users; -- ' AND
password='...';
```

Первый оператор выберет пользователей с именем 'a', **далее вторая команда** `DROP TABLE` **будет выполнена** – таблица пользователей удалена [31] [32]. Комментарий `--` затем закомментирует остаток запроса. Это демонстрирует, как SQL-инъекция может уничтожить данные.

Последствия SQL-инъекции: Атакующий может вытащить конфиденциальную информацию (списки клиентов, хеши паролей, заказы), изменить цены в БД, создать административного пользователя, или просто удалить/повредить данные [30]. Для интернет-магазина это катастрофично: утечка базы клиентов, компрометация паролей, уничтожение заказов и т.д.

Защита от SQL-инъекций:
1. **Параметризованные запросы (Prepared Statements).** Это стандартный и надёжный способ. Вместо вставки переменных напрямую в SQL используйте параметры. Практически во всех современных API для БД (PDO в PHP, mysqli, ORM) есть поддержка prepared statements. Пример с PDO:

```
$stmt = $pdo->prepare("SELECT * FROM users WHERE username = ? AND password
= ?");
$stmt->execute([$user, $pass]);
$result = $stmt->fetchAll();
```

Здесь `?` – плейсхолдеры параметров. Значения `$user` и `$pass` будут привязаны (bind) и экранированы драйвером базы автоматически, исключая возможность инъекции. Такой подход не только безопаснее, но и удобен (не нужно вручную ставить кавычки, экранировать). **Всегда**

используйте параметризованные запросы для взаимодействия с БД – это золотой стандарт защиты от инъекций [33].

1. **Экранирование данных вручную.** Если по каким-то причинам вы не можете использовать prepared statements, то минимум – применять функцию экранирования, предоставляемую API, например `mysqli_real_escape_string()` в PHP для всех вставляемых в запрос строковых значений. Эта функция добавит обратный слэш перед опасными символами (например, `'` превратит в `\'`), не позволяя разорвать строковой литерал в SQL [34]. Однако manual escaping – более хрупкий способ, легко что-то забыть. Предпочтительны параметризованные запросы.

2. **Ограничение прав в БД.** У аккаунта БД, от имени которого работает ваше приложение, должны быть только необходимые привилегии. Например, для сценария чтения/записи данных магазина не нужна возможность удалять таблицы или управлять пользователями БД. Ограничив права, вы снизите ущерб от возможной инъекции (например, атака не сможет удалить таблицы, если нет права `DROP`).

3. **Валидация вводимых данных.** Хотя основная защита – параметризация, логическая валидация тоже важна. Например, если ожидется числовой идентификатор, имеет смысл проверить, что это именно число. Это не стопроцентная защита, но хорошая практика, затрудняющая генерацию «неожиданного» вредоносного ввода.

Никогда не конкатенируйте строки SQL с непроверенными данными. SQL-инъекция – одна из самых опасных уязвимостей (в OWASP Top 10 всегда занимает лидирующие места), но полностью предотвращается правильным использованием параметров и экранированием.

HTTPS

HTTPS – это **защищённая версия протокола HTTP**, шифрующая весь трафик между клиентом и сервером с помощью SSL/TLS. В контексте интернет-магазина HTTPS не просто желателен, а обязателен: он обеспечивает конфиденциальность передаваемых данных (например, логинов, паролей, номеров карт) и защиту от «подслушивания» и вмешательства в трафик.

Если сайт работает по нешифрованному HTTP, злоумышленник, находясь в той же сети (например, публичный Wi-Fi), может с лёгкостью перехватывать данные (сниффить трафик). Так он сможет украсть пароли, сессионные cookie и любую другую информацию, передаваемую формами. Кроме того, без HTTPS атакующий способен выполнить **Man-in-the-Middle (MitM)** атаку – подменить содержимое страницы на лету, вставить вредоносный скрипт или изменить цены на витрине, незаметно для пользователя.

Почему HTTPS защищает: При установлении HTTPS-соединения браузер проверяет цифровой сертификат сервера (убеждаясь, что это именно ваш домен) и обменивается с сервером ключами для шифрования. Весь дальнейший трафик шифруется этими ключами, поэтому посторонний не сможет его прочитать или изменить. Это предотвращает перехват паролей и cookies, а также гарантирует целостность данных. Современные браузеры также помечают сайты без HTTPS как «Not Secure».

Рекомендации по HTTPS:
- Получите SSL-сертификат для вашего домена. Сейчас это просто – существуют бесплатные сертификаты (например, Let's Encrypt), которые можно настроить автоматизированно.

- **Включите HTTPS по всему сайту**, а не только на странице логина. Перенаправляйте все запросы HTTP на HTTPS (HTTP 301 redirect). Особенно важны страницы входа, регистрации, оформления заказа, ввода платежных данных.
- Установите заголовок HSTS (HTTP Strict Transport Security) – он сообщает браузерам, что ваш сайт должен открываться только по HTTPS, предотвращая даже попытку обращения по незащищённому протоколу [10]. Но будьте внимательны: HSTS следует включать, когда убедитесь, что HTTPS настроен правильно, иначе пользователи не смогут открыть ваш сайт по HTTP вовсе.
- Позаботьтесь о современной конфигурации TLS: отключите устаревшие протоколы SSLv3, TLS1.0/1.1, используйте сильные шифры. Большинство хостингов и веб-серверов имеют адекватные настройки по умолчанию, но проверить не помешает (существуют онлайн-тесты типа SSL Labs).

HTTPS и производительность: Раньше считалось, что HTTPS медленный. Сейчас это не так: с аппаратным ускорением шифрования и HTTP/2 задержки минимальны. Пользователь скорее заметит баннер «Not Secure», чем микрозадержку от TLS. Для SEO HTTPS тоже важен – поисковики дают сайту с HTTPS небольшой бонус рейтинга.

Итог: Всегда используйте HTTPS для интернет-магазина. Он защищает данные клиентов на пути к серверу и является базовым требованием безопасности. После включения HTTPS не забывайте и о других мерах (шифрование не спасёт, если у вас XSS, но по крайней мере затруднит некоторые атаки).

CORS (Cross-Origin Resource Sharing)

CORS – механизм браузера, позволяющий серверу контролировать, какие другие домены могут обращаться к его ресурсам посредством AJAX-запросов. Браузеры по умолчанию придерживаются политики **Same-Origin Policy (политика одного источника)**, запрещающей скриптам из одного домена читать ответы от другого домена по соображениям безопасности [35]. Это предотвращает, например, чтобы вредоносный сайт со стороннего домена не смог через AJAX получить ваши данные.

Однако иногда нужен легитимный кросс-доменный доступ – например, фронтенд-приложение на домене `shop-client.com` должно вызывать API `shop-api.com`. Вот тут и вступает CORS. Сервер, расположенный на `shop-api.com`, может отправлять специальные заголовки (например, `Access-Control-Allow-Origin`) в ответах, чтобы сообщить браузеру, что он разрешает запросы с определённых доменов. Если запрос приходит с разрешённого origin, браузер допускает скрипту доступ к ответу. Иначе – блокирует ответ, даже если сам запрос до сервера дошёл.

Пример: Ваш магазин на `https://store.com` имеет отдельный сервер изображений на `https://cdn.store.com`. Чтобы скрипт на странице `store.com` мог получить, скажем, список товаров через fetch-запрос к `api.store.com`, сервер API должен включить заголовок:

```
Access-Control-Allow-Origin: https://store.com
```

Браузер увидит, что ответ от `api.store.com` явно разрешает origin `store.com`, и позволит скрипту прочитать данные. Если бы заголовка не было или был другой домен, JavaScript получил бы ошибку CORS и не смог бы использовать ответ.

CORS для безопасности: CORS сам по себе не является защитой от атак со стороны вашего сайта – скорее это механизм **защиты других сайтов от ваших ресурсов**. Но разработчику API важно настроить CORS правильно, чтобы:
- **Не открыть доступ посторонним:** Например, не ставить `Access-Control-Allow-Origin: *` для приватных API, особенно в сочетании с `Allow-Credentials: true`, иначе любой сайт мог бы (при известном REST-интерфейсе) послать запрос от имени залогиненного пользователя и прочитать ответ.
- Разрешить только нужные методы и заголовки. Например, если ваш API не предполагает `PUT` или `DELETE` с других доменов, не включайте их в `Access-Control-Allow-Methods`.
- Включить `Access-Control-Allow-Credentials: true` только если используете cookies или идентификацию через cookie, и в этом случае Origin не должен быть `*` (по стандарту браузеры не примут cookie если origin звездочка).
- Если клиент – JavaScript-приложение (SPA), аутентификация может быть через Bearer-токен. В таком случае убедитесь, что **не** включено лишнего: например, не нужно разрешать креденшалы, а origin можно ограничить конкретным списком.

В целом, **CORS – это про настройку заголовков на сервере**. Библиотеки и фреймворки часто имеют встроенные средства: например, в Express (Node.js) middleware `cors()`, в PHP можно самостоятельно посылать заголовки. На стадии разработки можно временно разрешить всё (для отладки), но перед продакшеном **сконфигурируйте CORS минимально необходимым образом**.

Важно понимать: CORS защищает от чтения данных другим сайтом, **но не предотвращает сам запрос**. Например, CSRF-атака – это кросс-доменный запрос. CORS её не заблокирует, потому что CSRF не пытается прочитать ответ – ему достаточно, что запрос выполнен. Поэтому, даже настроив CORS, **не полагайтесь на него как на замену CSRF-защиты** – это разные уровни. CORS нужен, чтобы посторонний JS не украл ваши данные, CSRF-токены же нужны, чтобы посторонний сайт не заставил ваш браузер выполнить нежелательное действие.

Brute-force атаки (перебор паролей)

Brute-force – это метод взлома, при котором злоумышленник автоматически перебирает множество вариантов пароля или других секретных данных, пока не найдёт подходящий. В контексте интернет-магазина классический сценарий brute-force – скрипт пытается подобрать пароль к учётной записи, перебирая самые распространённые или все возможные комбинации. Также brute-force может применяться к промокодам, токенам восстановления пароля и т.п.

Опасность: Многие пользователи, к сожалению, ставят простые пароли (типа "123456", "password" и т.д.), которые легко угадываются перебором популярнейших вариантов. Даже если пароль сложнее, автоматизированный перебор (особенно с помощью ботнета) может попытаться тысячи и миллионы комбинаций. Без защиты злоумышленник рано или поздно может подобрать пароль, особенно если учетная запись без двухфакторной защиты.

Защита от brute-force:
- **Ограничение числа попыток входа.** После нескольких неуспешных попыток подряд временно блокируйте возможность входа для этого пользователя или IP. Например, можно разрешить не более 5 неверных вводов пароля, после чего аккаунт временно замораживается на 15 минут, либо показывается капча. Либо блокировать по IP, если оттуда идёт поток попыток (но осторожнее, чтобы не допустить DoS на легального пользователя).
- **CAPTCHA при подозрительной активности.** Например, после нескольких неудачных логинов

выводить капчу, требующую ручного ввода. Боты с этим справляются хуже. Однако капча ухудшает UX, включайте её только при подозрении на автоматизированный подбор.
- **Двухфакторная аутентификация (2FA).** Даже если пароль подберут, без дополнительного кода из SMS/приложения злоумышленник не войдёт. Для интернет-магазина 2FA можно предложить хотя бы для администраторов или при совершении важных операций (изменение email, привязка карты). MDN также рекомендует рассмотреть 2FA для повышения защиты аккаунтов [36] [37].
- **Политика паролей.** Требуйте от пользователей надежные пароли при регистрации: минимум 8-10 символов, включающие разные регистры, цифры, символы. И запретите несколько самых популярных паролей вовсе. Надёжный пароль усложнит и замедлит подбор.
- **Отслеживание аномалий.** Логируйте все неудачные попытки входа (подробнее о логировании – в соответствующем разделе). Если видите сотни попыток для одного аккаунта или с одного IP – принимаете меры: блокировка, уведомление админа.

Брутфорс – атака «в лоб», но удивительно эффективная, если не принять мер. Особенно опасен **credential stuffing** – когда злоумышленники берут огромные списки логинов/паролей, утёкшие с других сайтов, и массово пробуют на вашем (пользователи часто повторно используют пароли). Поэтому храните пароли безопасно (см. далее) и внедряйте ограничения на попытки входа.

Clickjacking (кликджекинг)

Clickjacking – техника, при которой злоумышленник обманывает пользователя, заставляя его кликнуть по элементу, которого тот не видит, но который выполняет нежелательное действие. Обычно это достигается с помощью скрытого фрейма: атакующий встраивает ваш сайт или страницу в `<iframe>` на своём сайте, накладывает поверх какие-то элементы, делает фрейм частично прозрачным и т.д., так что пользователь думает, что нажимает на видимый интерфейс, а на самом деле кликает по спрятанной кнопке на вашем сайте [38]. Например: на поддельном сайте может отображаться, как будто обычная страница, но на самом деле под курсором находится невидимая кнопка «Подтвердить покупку» или «Перевести деньги» с вашего сайта. Пользователь нажимает – и совершает действие, которого не хотел.

Другой вариант: злоумышленник показывает реальный интерфейс вашего сайта через `<iframe>` (например, интернет-банк), а поверх размещает невидимый слой, перехватывающий клики или ввод данных (например, вместо ввода пароля в настоящую форму, пользователь вводит его в поле, контролируемое злоумышленником).

Защита от clickjacking: Основная защита – **запретить загрузку вашего сайта во фрейме на сторонних доменах**. Делается это с помощью специальных HTTP-заголовков:
- **X-Frame-Options:** старый, но поддерживаемый вариант. Можно вернуть заголовок `X-Frame-Options: SAMEORIGIN` – тогда браузеры запретят отображать вашу страницу во фрейме, если внешний сайт имеет другой origin. Или `DENY` – запретить вообще любому сайту, включая ваш, грузить эту страницу во фрейме. (Полезно для страниц с критическими действиями).
- **Content-Security-Policy frame-ancestors:** современный способ. В заголовке Content-Security-Policy можно указать, с каких источников разрешены **родительские** фреймы. Например, `Content-Security-Policy: frame-ancestors 'self'` полностью аналогично `SAMEORIGIN` – разрешает встраивание только на страницах того же домена. Можно перечислить конкретные доверенные домены, если нужно разрешить embedding только для них.

Вышеуказанные заголовки нужно настроить на уровне веб-сервера или приложения. Для интернет-магазина обычно **нет причин позволять встраивать ваши страницы куда-либо**,

поэтому стоит выставить, как минимум, `X-Frame-Options: SAMEORIGIN` или даже `DENY` глобально (за исключением, если у вас сами используются iframe, например, виджет отзывов – но это редкость). Браузеры тогда просто не отобразят ваш контент во внешнем сайте, не дав обмануть пользователя [38].

Также, если интерфейс вашего сайта предполагает критичные действия по клику (например, «Оплатить»), можно добавить явные подтверждения действий (модальные окна «Вы уверены?») – это не прямая защита, но иногда может предотвратить незаметное выполнение. Однако основная защита – именно технический запрет фреймов.

Мы рассмотрели ключевые понятия. Теперь перейдём к практическим рекомендациям по каждой области безопасности интернет-магазина: от регистрации пользователя до оформления заказа и администрирования. Будем применять вышеописанные концепции на практике.

Безопасная регистрация и аутентификация

Этапы **регистрации** и **входа в аккаунт** – критически важны для безопасности, ведь здесь происходит выдача «ключей» от личного кабинета пользователя. Рассмотрим, как реализовать эти функции правильно.

Хранение паролей

Ни в коем случае не храните пароли в открытом виде. Пароли пользователей должны храниться только в виде **криптографических хешей** с солью. Это означает, что при регистрации (или смене пароля) вы применяете к введённому паролю специальную хэш-функцию, и сохраняете в базе **только результат**, а не сам пароль. При последующем логине пользователь вводит пароль, вы снова хэшируете его и сравниваете с сохранённым хешем. Если совпадают – пароль верный. Таким образом, ни у кого (даже у вас как разработчика, или при утечке базы) не будет возможности узнать исходные пароли пользователей из базы.

Выбор алгоритма хеширования: Используйте **специальные хэш-функции для паролей** – устойчивые к взлому перебором. Подходящие варианты: **bcrypt**, **Argon2id**, **scrypt**, или, как минимум, **PBKDF2**. В PHP проще всего – встроенная функция `password_hash()`, которая по умолчанию использует bcrypt (алгоритм Blowfish) с солью [39]. Bcrypt автоматически генерирует соль и позволяет задать «стоимость» (cost factor), влияющую на скорость. Пример:

```
$password = $_POST['pass'];
$hash = password_hash($password, PASSWORD_DEFAULT);  // сохраняем $hash в БД
```

Позже для проверки:

```
if (password_verify($inputPass, $hashFromDB)) {
    // пароль верный
} else {
    // пароль неверный
}
```

`password_hash` сам выбирает безопасный алгоритм (на текущий момент bcrypt) и генерирует соль, а `password_verify` – проверяет соответствие. Это **правильный и безопасный способ хранения паролей** [39].

Не используйте устаревшие хэши вроде MD5, SHA1 без соли – они слишком быстрые и уязвимы для перебора. Хэш-функции для паролей специально **медленные** и солёные, чтобы сильно осложнить жизнь атакующим [40] [41]. Даже если хэш пароля утёк, злоумышленнику придётся тратить огромное время на подбор оригинального пароля (брутфорс/словарь), особенно если пароль не тривиальный.

Пример неправильного хранения: Пароль в БД: `qwerty123`. Это катастрофа – в случае утечки все аккаунты с этим паролем скомпрометированы мгновенно. Даже хранение `SHA1(qwerty123)` плохо – в интернете есть радужные таблицы, позволяющие быстро обратимо найти пароль по его хэшу, если он без соли.

Дополнительно: Можно использовать так называемый «pepper» – секретный ключ приложения, который добавляется ко всем паролям перед хешированием. Он хранится отдельно (например, в коде или конфигурации, но **не в базе**). Это даёт дополнительную защиту: даже если хэши и соли утекут, без pepper-а атакующий не сможет валидировать догадки паролей [42]. Но pepper нужно надёжно защищать (как секрет). Это опционально, сильный алгоритм с солью уже даёт высокую стойкость.

Сброс/восстановление пароля: Тут тоже важный момент безопасности. Процедура обычно такая: пользователь запрашивает сброс, вы генерируете уникальный случайный токен, сохраняете его (с привязкой к пользователю и временем) и отправляете ссылку на email с этим токеном. При переходе пользователь может ввести новый пароль. **Нельзя** отправлять пароли по email, или тем более хранить «подсказки» и т.п. Токен сброса пароля должен быть одноразовым, достаточно длинным (не менее 16-32 байт случайных), и быстро истекать (например, через час). После использования – сразу инвалидировать. Это предотвратит злоупотребления, если кто-то перехватит письмо или догадается токен.

Защита формы входа

Рассмотрим типичные уязвимости и методы защиты, касающиеся формы логина и регистрации:

- **SQL-инъекция в логине:** Если код аутентификации напрямую подставляет введённые логин/пароль в SQL (как в предыдущем разделе), злоумышленник может выполнить инъекцию и войти без пароля. Поэтому **строго используйте параметризованные запросы** при проверке пользователя. Например, вместо:

```
$res = mysqli_query($conn, "SELECT * FROM users WHERE email='$email' AND
password='$passhash'");
```

лучше:

```
$stmt = $conn->prepare("SELECT * FROM users WHERE email=?");
$stmt->bind_param("s", $email);
$stmt->execute();
```

а затем сверить хэш пароля (вытаскивать хэш и делать `password_verify`). Тогда инъекция невозможна.

- **Передача пароля по сети:** Обязательно осуществляйте логин по HTTPS. Никогда не отправляйте пароль через незащищённый канал. Если вы используете AJAX вход, убедитесь, что URL начинается с `https://`. При использовании HTTPS, даже если кто-то прослушивает трафик, он не увидит данные формы.

- **CSRF-защита формы:** Как ни странно, форма входа тоже может стать целью CSRF-атаки. Сценарий: злоумышленник может попытаться **использовать жертву как «марионетку» для входа под учёткой атакующего**. Он шлёт ссылку жертве, та, сама не зная, отправляет форму входа с логином/паролем злоумышленника (через скрытую форму). В итоге – в браузере жертвы залогинен аккаунт злоумышленника. Зачем это? Чтобы затем, обманом, выманить у жертвы какие-то действия/данные, думая, что она под своим аккаунтом. Например, жертва заходит в «Мои заказы» – а это заказы злоумышленника. Или публикует отзыв, думая, что от себя, а на самом деле под аккаунтом атакующего. Такие атаки редки, но логично защитить и форму входа токеном (anti-CSRF токен). Это несложно, а покрывает все POST-формы. В целом, лучше перестраховаться: **добавьте CSRF-токен в форму логина**, тем более если сайт позволяет несколько одновременных аккаунтов (для интернет-магазина это не частый случай, но если сессия не привязана жёстко к одному логину). OWASP явно упоминает защиту от CSRF при логине, называя подход «pre-session токен» [43].

- **Безопасность регистрации:** На этапе регистрации тоже нужны проверки: валидация email (проверить формат, возможно отправить подтверждение на почту), требований к паролю (как упомянуто, длина, сложность). Также стоит защититься от массовых автоматических регистраций (ботов) – например, включив капчу или другие способы (спросить простую математическую задачу, использовать скрытое поле для отсеивания ботов). Массовая регистрация может использоваться для спама, накрутки, или дальнейшего подбора слабых паролей, поэтому лучше предотвратить.

- **Сообщения об ошибке:** Будьте осторожны с сообщениями «неправильный логин или пароль». Лучше объединять их, чтобы не выдавать наличие или отсутствие пользователя. Если при вводе несуществующего email писать «пользователь не найден», злоумышленник может перебирать адреса и узнать, кто зарегистрирован. Поэтому обычно пишут одно сообщение для всех: «Неверные учетные данные».

- **Двухфакторная аутентификация при входе:** Мы упоминали ранее, но повторим: если есть возможность, дайте пользователям опцию 2FA. Это сильно повышает безопасность учётной записи. По крайней мере, для административных учёток в вашем магазине **обязательно** включите 2FA (через приложение-аутентификатор или SMS).

Подведём итог: для безопасной аутентификации храните пароли исключительно в виде хешей (bcrypt через `password_hash`), используйте HTTPS, ограничивайте попытки входа, защищайте формы токенами и внедряйте 2FA. Так вы существенно снизите риски компрометации аккаунтов.

Защищённая корзина покупателя

Корзина – важнейший компонент интернет-магазина, связующая стадия между просмотром товаров и оформлением заказа. Корзина должна быть не только удобной, но и безопасной, иначе

злоумышленники могут воспользоваться уязвимостями для мошенничества (например, изменить цены, добавить товары бесплатно и т.д.). Рассмотрим, как правильно реализовать корзину как для **гостей**, так и для **авторизованных пользователей**, и защитить её содержимое.

Корзина для гостя и авторизованного пользователя

У неавторизованного посетителя тоже должна быть возможность складывать товары в корзину. Обычно варианты реализации:
- **Хранение корзины в сессии.** Это самый простой подход. Когда пользователь-**гость** добавляет товар, мы сохраняем его в массив `$_SESSION['cart']` (в PHP) – например, список товаров с ID и количеством. Пока сессия жива (пользователь активно на сайте), корзина хранится на сервере. Для длительного сохранения можно сессионные данные сереализовать и хранить в базе или Memcached (но главное – привязано к сессии). Если гость пропадёт надолго, сессия истечёт и корзина очистится – это нормально.

- **Хранение корзины в cookie.** Некоторые магазины сохраняют содержимое корзины прямо в cookie (например, JSON с товарами). Это позволяет «помнить» корзину даже после закрытия браузера без регистрации. Однако **опасно хранить в cookie информацию, которую нельзя доверять** – пользователь может отредактировать cookie. Если уж делать корзину в cookie, лучше хранить **только идентификатор корзины**, а все детали – на сервере по этому ID. Но это усложняет, поэтому чаще сессия предпочтительней.

- **Корзина в базе данных для авторизованных.** Когда пользователь входит в аккаунт, есть смысл корзину сохранять в БД, чтобы он мог вернуться в любой момент (даже с другого устройства) и его выбранные товары остались. Обычно делают так: у таблицы пользователей есть связанная таблица `carts` или `cart_items`. При авторизации, если в сессии у гостя уже была корзина, можно объединить её с корзиной из БД (например, добавить позиции гостя к уже сохранённым у пользователя, избегая дублирования). После этого основным хранилищем корзины становится база, и сессия используется лишь как кэш.

Рекомендуемая схема: Для простоты и безопасности – **сессия для гостя, база для авторизованного**. То есть: пока не вошёл – корзина хранится на сервере (идентифицируется по сессионному cookie). После входа – связываем корзину с аккаунтом: переносим в БД. Это даёт два плюса: 1. У авторизованных корзина **постоянная** (persisted), что удобно пользователю. 2. Мы можем **контролировать целостность данных на сервере** и легче защититься от манипуляций.

Защита данных корзины от подмены

Главный риск с корзиной – злоумышленник попробует **подделать данные**: например, цену товара, количество, ID товара (превратить один товар в другой с более низкой ценой). Если корзина или её часть хранится на клиенте (в cookie или hidden-полях формы), это особенно актуально. Да и даже с сессией, необходимо проверять цены на сервере, т.к. цены в интерфейсе – тоже данные, потенциально подлежащие атаке.

Пример атаки через hidden-поле: Представим, страница корзины выводит каждый товар строкой с ценой и количеством, а также включает скрытое поле `<input type="hidden" name="price_5" value="100.00">` – цену товара с ID 5 для отправки на сервер при оформлении. Злоумышленник может через инструменты разработчика изменить value этого поля на `1.00`. Если сервер **не перепроверит цену**, а просто суммирует присланные значения,

он закажет товар за 1 у.е. вместо 100 [44] . Это **Web Parameter Tampering** – атака на параметры приложения, меняющая важные данные [45] .

Защита от подмены цен:
- **Не доверять цене, пришедшей от клиента.** Цена товара должна определяться исключительно на сервере, исходя из данных в базе. То есть при оформлении заказа сервер должен для каждого товара в корзине заново извлечь актуальную цену из БД и пересчитать сумму. Клиент может передать лишь идентификаторы товаров и желаемое количество, но не стоимость. Даже если вы для отображения вывели стоимость и общую сумму, **игнорируйте эти значения при расчёте итогов** – всё пересчитывайте по своим данным.

- **Проверка ID и доступности товара.** Атакующий может попытаться добавить в корзину скрытый товар или изменить ID. Сервер должен убедиться, что все переданные ID товаров существуют, активны и доступны для заказа. Если товар скрыт/удалён/распродан – отклонить или убрать из корзины.

- **Фиксация цены на момент добавления (опционально).** Бывают случаи, когда цены меняются, или есть акции. Можно при добавлении товара гостем сохранить текущую цену в сессии. Но при оформлении заказа всё равно лучше проверить на сервере. Если цена изменилась – можно сообщить пользователю, что цена обновилась (это вопрос бизнес-логики). Главное – **не проводить старую цену, если она уже не актуальна** (злоумышленник мог воспользоваться устаревшей ценой).

- **Подпись данных (альтернатива).** В некоторых реализациях, если нужно передавать критичные данные через клиент (например, корзина в cookie), можно внедрить **криптографическую подпись**. Например, вместе с JSON корзины хранить HMAC (шифрованную контрольную сумму) от этих данных. Сервер проверяет HMAC – если хоть байт изменился, значит cookie подделан, и корзину отвергаем. Это сложнее настроить, но это способ удостовериться, что данные, сохранённые на клиенте, не были изменены. Опять же, проще не хранить важное на клиенте открыто.

Защита от подмены количества: Это тоже важный аспект. Например, если на складе товара осталось 5 единиц, а пользователь через хитрый запрос попытается заказать 500, или отрицательное число (чтобы выбить скидку? бывают баги). Сервер должен **валидировать количество**: минимум 1, максимум – разумный предел или остаток.

Дополнительные меры: - Если есть **промокоды или скидки**, их тоже нельзя доверять с клиента. Промокод должен проверяться по базе: действителен ли, на какую сумму, для каких условий. Нельзя принимать от клиента информацию вроде «применена скидка 100%» без серверной логики. - Если корзина хранится в сессии, убедитесь, что сессионные данные на сервере не уязвимы. Хорошо, если сессия хранится на сервере (файлы/БД) – тогда сложно подделать. Если же храните корзину в JWT (редко, но бывает) – тогда обязательно проверять подпись JWT. В общем, **любые данные, влияющие на денежные операции, должны на сервере считаться истиной**.

Реализация корзины: советы и пример

Простейшая реализация (PHP сессия):

```
session_start();
if (!isset($_SESSION['cart'])) {
    $_SESSION['cart'] = [];
}

// Добавление товара в корзину (пример):
function addToCart($productId, $quantity) {
    // Проверить валидность $productId, $quantity
    $quantity = max(1, intval($quantity)); // минимум 1
    // допустим, нет ограничения макс. кол-ва или проверяем наличие на складе

    // Если уже есть в корзине – увеличить количество, иначе добавить
    if (isset($_SESSION['cart'][$productId])) {
        $_SESSION['cart'][$productId] += $quantity;
    } else {
        $_SESSION['cart'][$productId] = $quantity;
    }
}

// Получение содержимого корзины:
function getCartItems() {
    $items = [];
    foreach ($_SESSION['cart'] as $prodId => $qty) {
        $product = loadProductFromDB($prodId); // функция получает товар из
базы
        if (!$product) {
            continue; // товар не найден (может был удалён) - пропустить
        }
        $price = $product['price'];
        $items[] = [
            'id' => $prodId,
            'name' => $product['name'],
            'price' => $price,
            'qty' => $qty,
            'total' => $price * $qty
        ];
    }
    return $items;
}
```

Здесь `$_SESSION['cart']` хранит пары productId => quantity. При добавлении проверяем количество. При выводе корзины загружаем каждый товар из базы и считаем сумму. Обратите внимание: **цена берётся из БД** (`$product['price']`), а не из сессии или от клиента. Это важно: даже если злоумышленник как-то занесёт неправильную цену в сессию (что маловероятно, но всё же), мы её не используем.

При оформлении заказа мы также будем обходить `$_SESSION['cart']` или данные корзины пользователя из базы и пересчитывать финальную сумму. Это гарантия от манипуляций.

В случае с авторизованными: можно реализовать сохранение корзины, например:

```sql
-- Таблица товаров корзины
CREATE TABLE cart_items (
    user_id INT,
    product_id INT,
    quantity INT,
    PRIMARY KEY(user_id, product_id)
);
```

При логине: взять все `$_SESSION['cart']` позиции, внести в таблицу `cart_items` для данного user_id (сложить с существующими). И очистить `$_SESSION['cart']`, чтобы не дублировать. Далее для пользователя используем только `cart_items`. При добавлении товара сразу пишем в БД (INSERT или UPDATE). При логауте – можно загрузить данные обратно в сессию, но не обязательно (гостю лучше новую корзину дать).

Такая схема сложнее в реализации, но необходима для **долговременной корзины**. Многие крупные магазины так и делают: вы можете залогиниться спустя месяц, и товары останутся в корзине.

Еще момент – привязка корзины к пользователю: Убедитесь, что пользователь **не получает доступ к чужой корзине**. Казалось бы, это очевидно – ведь корзина либо в сессии, либо в таблице с user_id. Но если, к примеру, API позволяет запрашивать содержимое корзины по какому-то идентификатору, нужно удостовериться, что **пользователь не может подменой параметра увидеть чужую корзину** (ID сессии или user_id). Это уже относится к контролю доступа, но упомянуть стоит.

Оформление заказа и безопасность платежей

Когда пользователь переходит к **оформлению заказа**, безопасность выходит на первый план: здесь обрабатываются персональные данные (адрес доставки, контакты) и финальная стоимость заказа, а также происходит оплата. Рассмотрим, как обезопасить этот процесс.

Проверка и обработка данных при оформлении заказа

При переходе на страницу оформления обычно пользователю предлагается заполнить форму: адрес доставки, способ оплаты, комментарий к заказу и т.д. Возможные уязвимости:
- **Некорректные или вредоносные данные в полях.** Пользователь (или злоумышленник) может ввести в адрес скрипт (`<script>`) или SQL-оператор, пытаясь вызвать XSS или SQL-инъекцию при сохранении заказа. Решение: **валидировать и санитизировать все поля заказа**. Адрес, имя, телефон – должны соответствовать ожидаемым форматом (можно проверять на допустимые символы, длину). Не допускайте в этих полях HTML-тегов вообще (они не нужны). Если нужно сохранить какие-то особые символы, экранируйте их при выводе в админке, чтобы при просмотре заказа менеджером не выполнился XSS.

- **Повторное или случайное подтверждение заказа.** Добавьте механизмы, чтобы один и тот же заказ не был оформлен дважды по ошибке (например, пользователь дважды нажал кнопку). Обычно это делают на стороне клиента (дизайбл кнопки после отправки) и на сервере (проверка, не пришёл ли дубликат запроса, например по какому-то номеру корзины). Это больше usability, но и безопасность – чтобы не провести случайно лишние транзакции.

- **Авторизация действий.** Убедитесь, что оформить заказ может только **владелец** корзины. То есть, если у вас есть некий идентификатор корзины в запросе (например, `POST /checkout?cart_id=XYZ`), **проверьте, что этот cart_id принадлежит текущему аутентифицированному пользователю** (или сессии гостя). Иначе есть риск, что злоумышленник попытается подставить чужой cart_id и оформить заказ от имени другого человека. Обычно, если вы правильно храните корзину в сессии или по user_id, такой проблемы нет – вы берёте корзину из текущей сессии/пользователя и оформляете её.

- **Пересчёт итоговой суммы на сервере.** Повторимся: финальную сумму заказа вычисляет сервер. Клиент может прислать свои расчёты, но доверять им нельзя. Алгоритм должен быть такой: получаем список товаров из корзины (с сервера), проходим по каждому, берём актуальную цену (с учётом скидок, акций, купонов – всё сервер проверяет), суммируем. Если у вас есть **доставка**, то стоимость доставки тоже определяйте на сервере (например, по индексу или выбранному способу). Никаких «скидка 100%» от клиента без проверки. В итоге сервер знает правильную сумму к оплате и может отобразить пользователю итог ещё раз с серверного расчёта (на странице подтверждения). Если вдруг есть расхождения с тем, что видел пользователь, нужно сообщить (в идеале, не допускать таких ситуаций, кроме изменения цен пока пользователь оформлял).

- **Сохранение данных заказа.** Когда все проверки прошли, создайте запись заказа в базе: обычно это таблицы `orders` (с основными данными: id заказа, пользователь, дата, статус, сумма и пр.) и `order_items` (позиции заказа: товар, цена, количество, сумма). Здесь важно: **сохранить именно ту цену, по которой товар продан**, даже если потом цена на сайте изменится. Чтобы в будущем правильно показывать детали заказа и сумму. Лучше всего в `order_items` хранить `price` и `quantity` на момент покупки, а не ссылаться только на товар – цены в товаре могут поменяться.

- **Статус заказа.** Присвойте вновь созданному заказу статус, например, «Новый» или «Ожидает оплаты» (если сразу онлайн оплата). Система статусов должна предотвращать «перепрыгивание» – например, обычный пользователь не может сразу установить заказу статус «Отправлен» или «Выполнен». Это уже вопрос авторизации: API изменения статуса должны быть доступны только админам или соответствующим сервисам.

Интеграция безопасной оплаты

Обработка платежей – наиболее ответственный этап. Рекомендация номер один: **не обрабатывайте сами данные платежных карт, если в этом нет крайней необходимости**. Вместо этого используйте проверенных платежных провайдеров (Stripe, PayPal, Яндекс.Касса, CloudPayments, Sberbank API и т.д.) и **перенаправляйте пользователя на их платёжную страницу или используйте виджет**, чтобы данные карты вводились **на стороне провайдера**. Таким образом, данные карты вообще не коснутся вашего сервера, и вам не нужно соответствовать строгим стандартам PCI DSS.

Есть два распространённых подхода:
- **Редирект на платёжный шлюз / внешнюю страницу.** Пользователь после оформления заказа перенаправляется, например, на защищённую страницу банка или платёжной системы, где вводит данные карты и подтверждает оплату. После этого его возвращают на ваш сайт с результатом (успех или неуспех). Ваш сайт при этом получает уведомление (callback) от системы оплаты или проверяет статус по API, и затем помечает заказ как оплаченный. Это безопасно, так как все чувствительные данные обрабатывает банк. Но UX может быть не самым гладким

(переход на внешний сайт). Тем не менее, многие доверяют именно такому подходу, поскольку видят знакомую страницу банка.

- **Встроенный платежный виджет (токенизация).** Многие провайдеры предлагают виджет (iframe или JS библиотеку), который отображает форму ввода карты прямо на вашей странице, но данные карты идут сразу на сервера провайдера, а вам возвращается только **токен** – специальный идентификатор, представляющий платёжные данные. Вы на своем сервере принимаете этот токен и отправляете его провайдеру (по секретному API-ключу) для проведения транзакции. Токен бесполезен сам по себе для злоумышленника, т.к. привязан к вашему аккаунту у провайдера и часто к сумме/получателю. Такой подход сочетает удобство (ввод без перехода) и безопасность (карта не проходит через ваш сервер).

Что важно при интеграции оплаты:
- **Используйте только HTTPS.** Платёжные системы обычно требуют это. Если у вас виджет, страница оплаты должна быть защищена, иначе браузер может заблокировать загрузку внешнего скрипта (мешаный контент) или, хуже, данные могут быть перехвачены.

- **Проверяйте суммы и валюты, передаваемые провайдеру.** Отправляя запрос на оплату, убедитесь, что сумма равна рассчитанной стоимости заказа, и что валюта правильная. Не давайте клиенту возможность подменить эти параметры.

- **Получение подтверждения оплаты.** Предусмотрите обработчик (endpoint) для уведомлений от платежного провайдера. Обычно провайдер может отправить уведомление на ваш сервер (webhook), когда платёж успешно проведён. Обработчик должен: проверить подпись/секрет уведомления (чтобы убедиться, что это именно от провайдера), найти соответствующий заказ по ID или иному метаданному, сверить сумму, статус и затем пометить заказ оплаченным. Не доверяйте уведомлениям без проверки – их могли бы подделать. Обычно провайдеры подписывают запросы или отправляют на заранее оговорённый URL.

- **Страница возвращения.** После оплаты пользователь возвращается к вам (на URL success или fail). На странице успеха **не принимайте решение об оплате, основанное только на наличии этой страницы**. Она для пользователя, но для вашей системы истинный источник – уведомление или проверка через API. Например, **не делайте так**: `if ($_GET['status']=='paid') { markOrderPaid(); }` – параметр `status` можно подделать. Лучше отметить заказ оплаченным только когда получили достоверное подтверждение от банка (в виде webhook или ответа на серверный запрос). Страницу успеха показывайте, если заказ уже в статусе «оплачен» по вашей базе. В противном случае – подождите подтверждения или предложите пользователю контакт, если деньги списались, а заказ не обновился.

- **Без сохранения лишних данных карты.** Никогда не сохраняйте полный номер карты, CVV или срок годности на своей стороне. В крайнем случае можно хранить **последние 4 цифры** и эмитента (Visa/MasterCard) для отображения пользователю и администратору, но не более [46]. Полные данные сохранять запрещено стандартами безопасности. Если нужна **повторная оплата** (один клик), используйте механизм токенов от провайдера (они обычно позволяют сохранить токен карты, привязанный к вашему клиенту). Этот токен безопасен для хранения, т.к. платёж провести по нему можно только через ваш аккаунт.

- **Соответствие PCI DSS.** Если вдруг ваш бизнес-сценарий требует действительно обрабатывать номера карт (например, передача в банк по телефону и ввод админом), вам необходимо соответствовать PCI DSS – стандарту безопасности индустрии платёжных карт. Это сложный и дорогой процесс, связанный с аудитами. Поэтому малому/среднему бизнесу проще этого избегать, используя готовые решения, как описано выше.

Итого: Держите максимальный уровень безопасности при оплате: пускай всю «грязную работу» сделают платёжные сервисы. Ваша задача – корректно передать сумму и ID заказа, получить подтверждение и обновить статус заказа.

Статусы и дальнейшая обработка заказа

После оформления заказ проходит стадии: оплачен, в обработке, отправлен, выполнен и т.д. Здесь несколько аспектов безопасности:
- **Разграничение прав:** Менять статус заказа (например, отмечать отправленным, отменять) должны только уполномоченные лица (менеджеры, админы) или автоматические процессы (после оплаты – автопереход в «Оплачен»). Обычный пользователь может, разве что, отменить свой заказ, и то чаще через запрос менеджеру, а не прямой переключатель статуса. **API изменения статуса** должно требовать аутентификации и нужной роли.

- **Защита от подделки запросов (CSRF) в админке:** Если у вас админ-панель для управления заказами, не забывайте и там ставить CSRF-токены на формы изменения статусов, создания товаров и т.п. Админы тоже могут случайно кликнуть по ссылке, поэтому CSRF защита важна для **всех** форм.

- **Логирование важных событий:** Изменение статуса заказа, возврат платежа, изменение суммы – всё это нужно логировать (как минимум с кем пользователем сделано и когда). Это поможет в расследовании инцидентов и в **мониторинге подозрительной активности**, о чём далее.

Мы увидели, что оформление заказа требует повышенной бдительности: проверять всё, что приходит от клиента, использовать внешние платёжные решения и тщательно контролировать права на дальнейшие действия с заказом.

Авторизация и контроль доступа

Помимо аутентификации (распознования пользователя) не менее важно правильно настроить **авторизацию** – систему проверки прав на те или иные действия. В интернет-магазине обычно есть разделение на **покупателей** (обычных пользователей) и **администраторов/менеджеров**. Администратор может, к примеру, управлять товарами, ценами, просматривать все заказы, менять статусы, а пользователь – только свои данные и заказы. Ошибки в авторизации могут привести к тому, что кто-то получит доступ к чужой информации или административным функциям.

Роли пользователей

Рекомендуется реализовать ролевую модель. Минимальный набор: - **Пользователь (Customer):** роль по умолчанию у всех зарегистрированных клиентов. Права: просмотр и редактирование своего профиля, создание заказов, просмотр своих заказов, отзывы и т.п. Не имеет доступа к административным страницам. - **Администратор (Admin) или Менеджер:** роль для персонала.

Права: управление каталогом товаров (CRUD товаров, категорий), просмотр **всех** заказов, изменение их статусов, управление пользователями (возможно), настройками сайта и т.д. Возможно, стоит разделить на подроли: *Контент-менеджер* (только товары), *Менеджер заказов*, *Администратор* (полные права). Это зависит от размера команды.

Как реализовать: - В базе у пользователя хранится поле `role` или отдельная сущность ролей. При входе в сессию (или JWT) можно сохранять роль. - На каждой защищённой странице или API-endpoint вы проверяете: `если (пользователь.роль != 'admin') { return 403 Forbidden; }`. Например, в PHP:

```
if ($_SESSION['user_role'] !== 'admin') {
    http_response_code(403);
    exit('Доступ запрещен');
}
```

Такой кусок нужно включить во все админ-скрипты. Если у вас MVC, то сделайте централизованную проверку в контроллере. Аналогично, для API на уровне middleware.

- **Не полагайтесь только на скрытие ссылок.** Даже если у обычного пользователя нет ссылки «Админ-панель», он может догадаться URL (`/admin`) и попробовать его открыть. Поэтому проверка роли на сервере обязательна.

Ограничение доступа к данным

Даже среди пользователей нужно проверять права на конкретные данные: - **Данные профиля:** пользователь A не должен суметь открыть или изменить профиль пользователя B. Если есть endpoint типа `/user/edit?id=42`, это уязвимость – злоумышленник поменяет id и отредактирует чужой профиль. Лучше идентификатор брать из сессии на сервере (текущий пользователь) и игнорировать, что пришлёт клиент. Если нужно редактировать не себя (например, админ редактирует пользователя) – проверять, что админ. - **Заказы:** аналогично, запрос вида `/order/view?order_id=1001` должен проверять, что заказ с ID 1001 принадлежит текущему пользователю, иначе выдавать ошибку. Только админ может просматривать заказы любого пользователя. Это называется защита от **IDOR – Insecure Direct Object References**, прямого обращения к чужим объектам. Лечение: проверки на каждом уровне. - **API ключи и скрытые разделы:** если у вас есть API, открытое в интернет (например, мобильное приложение дергает те же функции), то все приватные запросы должны требовать токен или сессию. Не забывайте об **OPTIONS запросах** (CORS preflight) – они должны либо быть неавторизованными, либо проверять минимум. Но основной GET/POST – обязательно. - **Файлы и изображения:** Админ может загружать картинки товаров, пользователь может загружать аватар. Нужно следить, чтобы один не мог удалить файл другого, либо читать несвоё. Часто медиа-контент делают общедоступным, но иногда бывают приватные файлы (например, PDF-счёт для конкретного заказа). В таких случаях или генерируйте уникальные труднопредугадываемые имена/пути, или проксируйте через скрипт с проверкой сессии.

Защита API и AJAX-запросов

Если ваш интернет-магазин имеет AJAX-функциональность (например, кнопка «Добавить в корзину» делает XHR) или полноценное JSON API, обеспечивающее функционал, применяются те же принципы: аутентифицировать запросы и проверять права. - **REST API аутентификация:** Здесь могут быть токены. Например, JWT токен, отправляемый в заголовке. Сервер должен

проверять подпись JWT и его срок, а затем на основе содержимого (клаймов) определить пользователя и его роль. Если токен невалиден – отказать. - **Rate limiting (ограничение частоты):** API может подвергаться скриптовым атакам (brute-force, перебор ID и т.п.). Вы можете внедрить ограничение: например, не более 100 запросов в минуту с одного IP для сложных операций. Или использовать готовые решения (лейеры типа nginx limit_req или прокси). Это защищает от некоторого рода DoS или массового сканирования. - **CORS настройки для API:** Если у вас публичное API для внешних сайтов, настроить CORS нужно аккуратно. Если же API только для вашего собственного фронтенда на том же домене, то вообще лучше его не разделять по доменам, чтобы избежать лишних головных болей. Если же, скажем, фронт на `shop.com`, а API на `api.shop.com`, то в CORS `Access-Control-Allow-Origin` укажите `shop.com` явно, а не `*`, и `Allow-Credentials: true` если используете cookie сессии. Тогда только ваш сайт сможет полноценно обращаться.

Использование готовых решений RBAC

Для крупных проектов можно внедрить более сложные системы, например RBAC (Role-Based Access Control) или ACL (Access Control List), где права задаются на уровне операций. Но для начала достаточно чётко структурировать код: разделить публичные и админ-маршруты, и вставить проверку роли.

Пример (псевдокод контроллера):

```
function adminOrdersList() {
    if ($currentUser->role !== 'admin') {
        http_response_code(403);
        echo "Forbidden";
        return;
    }
    // иначе выполнить логику - достать из БД все заказы
}
```

А для заказа пользователя:

```
function viewOrder($orderId) {
    $order = loadOrder($orderId);
    if (!$order) {
        echo "Order not found";
        return;
    }
    if ($order->user_id !== $currentUser->id && $currentUser->role !==
'admin') {
        http_response_code(403);
        echo "Access denied to this order";
        return;
    }
    // показать детали заказа
}
```

Такой подход предотвратит простейшие попытки «угадать» ссылку или ID и получить лишний доступ.

Важно: как только появится административный функционал, вы должны относиться к нему с той же тщательностью, что и к публичному. Потому что если злоумышленник получит доступ к админ-панели (через XSS на админа, подбор пароля, уязвимость в авторизации), последствия обычно намного серьёзнее (полная компрометация приложения). Поэтому админка: - защищена паролями (желательно гораздо более сложными, 2FA), - ограничена по IP (если возможно, разрешить вход только из офиса, например), - **везде CSRF-токены**, - обновление софта (кстати, используйте последние версии фреймворков/CMS, чтобы не было известных дыр).

В совокупности, грамотно реализованная авторизация гарантирует, что пользователи видят только своё, а администраторы управляют всем – и никакой пересортицы. Ошибки же в этой области могут позволить атакующему, даже без взлома пароля, получить чужие данные или права.

Основы безопасного использования HTTPS и cookie

Мы уже частично покрыли HTTPS и атрибуты cookie ранее, но повторим основные моменты как чек-лист настроек на практике:

- **Установка Secure и HttpOnly для cookies сессии.** В PHP можно настроить это в `session_set_cookie_params` или php.ini. Например:

```
session_set_cookie_params([
    'lifetime' => 0,
    'secure' => true,
    'httponly' => true,
    'samesite' => 'Lax'
]);
session_start();
```

Либо, если выставляете cookie вручную:

```
setcookie('auth_token', $jwt, [
    'expires' => time()+3600,
    'secure' => true,
    'httponly' => true,
    'samesite' => 'Strict'
]);
```

Это автоматически добавит соответствующие флаги. **Проверьте, что после логина cookie помечен как Secure, HttpOnly (и SameSite желательно)**. Без этого вы снижаете уровень защиты (cookie можно похитить через XSS, или он утечёт в сеть). Использование HttpOnly, как отмечает OWASP, **помогает смягчить риск XSS-кражи сессионного идентификатора** [47].

- **Принудительный редирект на HTTPS.** Настройте веб-сервер (Apache, Nginx) так, чтобы все заходы по http:// автоматически редиректило на https:// аналогичного адреса. Обычно это 301 редирект. Например, в Nginx:

```
if ($scheme = http) {
    return 301 https://$host$request_uri;
}
```

или `listen 80; return 301 https://$host$request_uri;`. В Apache можно через модуль rewrite. Это исключит ситуацию, что кто-то случайно перейдёт по незащищенной ссылке (или будет активна старая).

- **HSTS (HTTP Strict Transport Security).** Добавьте заголовок:

```
Strict-Transport-Security: max-age=31536000; includeSubDomains; preload
```

Это означает, что браузер должен всегда использовать HTTPS для вашего домена и всех поддоменов, в течение указанного времени (31536000 секунд ~ 1 год). `preload` и `includeSubDomains` – опции: первая позволяет включить ваш сайт в список HSTS-прелод (браузеры уже знают при первой загрузке), вторая – распространяет правило на поддомены. *Внимание:* включайте HSTS только когда убеждены, что HTTP не нужен и вся инфраструктура обслуживает HTTPS, иначе можно случайно отрезать доступ по http. Но для магазина, как правило, нужно.

- **Совместимость с mixed content:** Убедитесь, что все ресурсы на страницах (скрипты, стили, изображения, AJAX) тоже грузятся через HTTPS. Иначе браузер заблокирует их или выдаст предупреждение. То есть, если у вас вставлены картинки с другого источника, используйте `https://` ссылки.

- **Cookie domain и path:** В настройках cookie убедитесь, что они не шире, чем должно быть. Например, если у вас www и без www – лучше использовать единый domain=.site.com, чтобы работало на обоих, но не делайте domain=.com (на уровне TLD). Path обычно по умолчанию `/` – так и оставьте, если нет причин ограничивать поддиректорией.

- **Не храните секреты в JavaScript или видимых местах.** Например, API-ключи платежей, секретные токены – держите на сервере. Фронтенд может иметь публичные ключи (например, ключ для reCAPTCHA, он публичный), но не секретный.

- **Регулярно обновляйте SSL-сертификат.** С Let's Encrypt и автоматикой это решается. Просроченный сертификат приведёт к недоступности сайта (браузеры будут ругаться).

Эти технические аспекты должны быть учтены при деплое. Без них можно случайно ухудшить все остальное: например, супер-защищённая логика, но сессия без Secure – и весь трафик утекает при первом же запросе по HTTP. Поэтому настройка HTTPS + cookie – первый шаг перед вообще приёмом реальных пользователей.

Журналирование и мониторинг подозрительной активности

Наконец, важная часть – **логирование** действий и мониторинг. Без логов вы можете даже не узнать, что кто-то пытался (или успешно) взломал систему. Логи же позволяют расследовать инциденты и в реальном времени отслеживать атаки.

Что стоит логировать в интернет-магазине:

- **Аутентикацию:** Успешные и особенно **неудачные попытки входа**. Фиксируйте время, логин (email), IP адрес. Множество неудачных входов подряд – повод насторожиться (возможно, brute-force). Можно при накоплении X ошибок писать в отдельный "security.log" предупреждение или слать алерт. OWASP рекомендует, чтобы все важные события, как логины и сбои логинов, были залогированы с достаточным контекстом (кто, откуда, что ввёл) [48] [49].

- **Регистрации и важные аккаунт-события:** Новая регистрация – можно логировать IP и email (для отладки проблем, и на случай ботов). Смена пароля, сброс пароля (отправлен токен на почту), смена email – всё это пишем в лог, с указанием пользователя и, может, старого/нового значений (особенно email – на случай, если аккаунт угнан, вы сможете восстановить контакт с владельцем).

- **Действия с заказами:** Создание заказа (ID, сумма, статус), отмена заказа, изменение статуса (кто перевёл, с какого на какой). Например: `2025-09-15 10:00:23 [admin:123] changed order 2021 status from "New" to "Shipped"`. Эти сведения помогут аудитору понять, не было ли странных манипуляций (вдруг кто-то неуполномоченный поменял статус на "Возвращён" и сделал refund).

- **Платёжные транзакции:** Получили уведомление об оплате – залогируйте "Order X marked as paid, transaction ID Y, amount Z". Если оплата неудачна – тоже с причиной (хотя бы "declined by bank").

- **Ошибки безопасности:** Попытки XSS (например, кто-то ввёл `<script>` – можно по шаблону отследить и логировать input), попытки SQL-инъекций (ловить `' OR` в полях – но это скорее на уровне WAF, а не приложения). Если вы используете какой-то firewall (модуль безопасности), он может в лог бросать такие события.

- **Подозрительные запросы:** Например, кто-то обращается к URL, которого не должно быть (`/etc/passwd`, `/admin/deleteAll` и прочие странности). Веб-сервер обычно пишет 404, но можно настроить свой обработчик, который дополнительно сигналит. Хотя базово достаточно просматривать логи веб-сервера.

Хранение логов: Собирайте логи в безопасном месте. Хорошо настроить ротацию (logrotate) – чтобы они не разрастались бесконечно. Для безопасности лучше, если логи **пишутся на сервере**, куда веб-приложение имеет доступ только на добавление (чтобы при взломе злоумышленник не мог их подчистить). Можно выводить логи в stdout и собирать их внешней системой (ELK, Graylog и т.п.). Для небольшого проекта хватит и текстовых файлов. Срок хранения – зависит от нагрузки, но обычно **не менее нескольких недель** или лучше месяцев, чтобы можно было откатить и проанализировать события, предшествующие инциденту.

Мониторинг: Просто писать логи мало – их надо просматривать. Периодически анализируйте: - Не происходят ли массовые 401/403 ошибок (может, кто-то брутфорсит URL). - Нет ли тысяч ошибок логина от одного IP (можно на уровне fail2ban блокировать такие IP). - Какие IP чаще всего лезут в админку – если вы видите попытки несанкционированного доступа, можно тоже блокировать или пересмотреть безопасность. - Отслеживайте **необычную активность аккаунтов**: например, вдруг один пользователь совершил 50 заказов за час – возможно, злоумышленник использует угнанный аккаунт или баг.

OWASP Top 10 2021 особо выделяет **недостатки журналирования и мониторинга** как уязвимость, потому что компании не замечают атак без логов [50] [51]. Поэтому настройте хотя бы простейшие уведомления: например, отправлять администратору email, если произошло более 5 неудачных входов подряд для одного аккаунта, или если возникла ошибка, указывающая на SQL-инъекцию. Есть готовые инструменты SIEM, но для начала вы можете реализовать "на коленке": когда ловите событие, пишете в лог и `mail()` высылаете на свою почту.

Защита логов: Учтите, что логи сами могут стать мишенью. Во-первых, не логируйте конфиденциальную информацию в открытом виде. Пароли – никогда не пишем (даже хеши нет смысла). CVV карты тоже не должно быть нигде. Если нужно залогировать номер карты – только маскируйте (***1234). Логи могут быть доступны при компрометации сервера, чтобы злоумышленник не нашёл там сразу все сокровища. Во-вторых, убедитесь, что вывод юзерских данных в лог** ***экранирован**, иначе возможна атака типа XSS в консоли или просмотрщике логов (редко, но кто знает). Например, если вы логируете строку с пользователем, а он ввёл что-то типа `</script><script>...` – лучше на всякий случай заменить опасные символы и там.

Подведём итог: грамотное журналирование делает ваш магазин **наблюдаемым**. Вы сможете обнаружить подозрительную активность и среагировать прежде, чем она приведёт к серьёзным последствиям (например, заблокировать атакующий IP, уведомить пользователей о попытке взлома их аккаунта и пр.). А если инцидент произошёл, логи дадут материал для расследования: что и как было сделано, какие данные затронуты.

Теперь, когда мы прошли все основные главы, ниже приводится краткий чек-лист, суммирующий меры по безопасности интернет-магазина. Вы можете использовать его для самопроверки и аудита вашего приложения.

Сводный чек-лист безопасности интернет-магазина

- **Безопасное соединение:**
- [ ] Включён HTTPS на всём сайте, действующий SSL-сертификат.
- [ ] Настроен редирект с HTTP на HTTPS.

- [ ] Установлен заголовок HSTS для принудительного HTTPS в браузерах.

- **Настройки cookie и сессий:**

- [ ] Сессионные и аутентификационные cookies помечены флагами Secure и HttpOnly [10] [47].
- [ ] Используется атрибут SameSite (Lax/Strict) для защиты от CSRF.

- [ ] Идентификаторы сессий генерируются надёжно; выполняется смена session ID после логина.

- **Хранение и обработка паролей:**

- [ ] Пароли хранятся только в виде хешей с солью (bcrypt/Argon2 через `password_hash()` или аналог) [39].
- [ ] Реализована проверка сложности пароля при регистрации; запрещены самые простые пароли.
- [ ] Ограничено число попыток входа; после нескольких неудачных вводов вводится задержка или капча.
- [ ] Опционально: настроена 2FA для усиления защиты аккаунтов (особенно админов) [36].

- [ ] Процедура сброса пароля безопасна: токены сброса одноразовые, достаточной длины, быстро истекают; пароль не пересылается по email.

- **Защита от CSRF:**

- [ ] Во всех HTML-формах, изменяющих данные (регистрация, вход, профиль, заказ и т.д.), используются CSRF-токены [18].
- [ ] Сервер отклоняет любые запросы без корректного CSRF-токена.
- [ ] Для AJAX/API запросов применяются меры против CSRF (например, проверка заголовков или токен в custom header).

- [ ] Установлен SameSite для сессионного cookie (дополнительная линия обороны).

- **Защита от XSS:**

- [ ] Все пользовательские данные, выводимые в HTML, проходят экранирование специальных символов (`htmlspecialchars` или аналог) [26] [27].
- [ ] Запрещено хранить и отображать необработанный HTML от пользователей; либо реализована фильтрация/санитизация (например, через HTML Purifier) [29].
- [ ] Критичные cookies помечены HttpOnly, чтобы JavaScript не мог их украсть [47].
- [ ] (Опционально) Настроен Content Security Policy (CSP) для ограничения выполнения скриптов.

- [ ] Пользовательский ввод в атрибутах, URL и т.п. тоже валидируется/экранируется (не только в теле HTML).

- **Защита от SQL-инъекций:**

- [ ] Все запросы к базе с данными пользователя выполнены через подготовленные выражения (prepared statements) или ORM, **без** конкатенации SQL-строк [33].
- [ ] Данные фильтруются по типам (числа как числа, строки экранируются).
- [ ] У аккаунта БД минимальные необходимые права (например, нет DROP/DELETE прав для веб-пользователя, если не нужно).

- [ ] Обработчики ошибок БД не возвращают пользователю сырые сообщения SQL (во избежание утечки структуры).

- **Безопасность корзины и заказов:**

- [ ] Корзина гостей хранится в сессии на сервере; данные корзины авторизованных – в БД (привязано к user_id).
- [ ] При оформлении заказа цены и сумма пересчитываются на сервере по текущим данным в базе [44].
- [ ] Сервер игнорирует любые цены/скидки, пришедшие от клиента (эти поля не доверяются).
- [ ] Проверяются валидность и принадлежность всех товаров в заказе (не допустить несуществующих или чужих позиций).
- [ ] Количество товаров в заказе ограничивается (не отрицательное, <= остатка склада и т.д.).

- [ ] Заказ привязывается к пользователю; проверяется, что просмотреть или изменить заказ может только его владелец или админ.

- **Процесс оплаты:**

- [ ] Используется надёжный платёжный провайдер; ввод данных карты происходит **непосредственно на стороне провайдера** (через редирект или виджет).
- [ ] Сервер не сохраняет номер карты, CVV и прочие чувствительные платёжные данные (только токены или последние 4 цифры по необходимости) [46].
- [ ] Реализована валидация сумм и валюты перед отправкой запроса оплаты.
- [ ] Имеется обработка callback/уведомлений от платёжной системы с проверкой подписи/секрета; статус заказа обновляется только при достоверном подтверждении оплаты.
- [ ] Исключены сценарии, где пользователь сам сообщает об оплате (например, параметр `paid=1` в URL) – такие сигналы игнорируются без проверки со стороны сервера.

- [ ] Выплаты/возвраты можно инициировать только из защищённой админ-панели с соответствующими правами.

- **Авторизация и права доступа:**

- [ ] В системе реализованы роли (как минимум отделены обычные пользователи и администраторы).
- [ ] Административные разделы и действия защищены проверкой ролей на сервере (просто скрыть кнопку недостаточно) [5].
- [ ] Проверяется принадлежность данных: пользователь не может получить доступ к чужим заказам, профилям, файлам и пр. (проверки ID пользователя на сервере для каждой сущности).
- [ ] Админ-панель защищена от CSRF так же, как и пользовательские формы (чтобы злоумышленник не провёл нежелательное действие от лица администратора).
- [ ] Доступ к API ограничен токенами/ключами; все приватные запросы требуют авторизации. CORS настройки для API не позволяют открытый доступ для недоверенных источников.

- [ ] В случае, если у вас используются пользовательские файлы (например, изображения продуктов, аватары), проверьте что нет возможности скачать те, что не должны быть доступны (или ссылки достаточно непредсказуемы, или идёт проверка доступа при выдаче файлов).

- **Журналирование и мониторинг:**

- [ ] Логируются важные события: входы (успешные и проваленные) [48], ключевые изменения (пароль, email, права), действия админов (создание/удаление товара, изменение цен, статусов заказов), попытки возможных атак.
- [ ] Логи хранятся безопасно (недоступны через веб), данные в логах очищены от слишком чувствительного (пароли, полные карты не пишутся).
- [ ] Настроен мониторинг логов или уведомления об аномалиях (много неудачных логинов, неоднократные SQL ошибки и т.д.).

- [ ] Регулярно просматриваются логи на предмет подозрительной активности; есть процедура реагирования (блокировка IP, сброс паролей при подозрении на компрометацию и т.д.).

- **Обновление и общие меры:**

- [ ] Все используемые библиотеки, фреймворки и платформы своевременно обновляются до актуальных версий (исправление известных уязвимостей).
- [ ] Отключены или защищены демонстрационные и отладочные скрипты (phpinfo, debug панели) – чтобы злоумышленник не смог получить лишнюю информацию.
- [ ] Сервер настроен безопасно: закрыты неиспользуемые порты, стоят базовые фильтры (Web Application Firewall по возможности).
- [ ] Регулярно делаются бэкапы данных (на случай инцидента можно восстановить систему). Бэкапы хранятся отделно и проверяются.

Этот чек-лист не исчерпывает всех нюансов, но покрывает основные пункты, рассмотренные в нашем учебном материале. Соблюдая эти правила, вы существенно укрепите безопасность своего интернет-магазина и защитите пользователей от большинства распространённых угроз. Безопасность – процесс непрерывный: важно поддерживать бдительность, учиться на инцидентах и постоянно совершенствовать защиту. Удачной разработки и безопасных продаж! [52]

---

[1] [2] [3] Ecommerce Security for Beginners (2025 Edition) - Iconic
https://iconicwp.com/blog/ecommerce-security-for-beginners/

[4] [10] [12] [13] [19] [20] [21] [22] [23] [24] [25] [29] [30] [31] [32] [33] [34] [36] [37] [38] [46] [52] Веб-безопасность - Изучение веб-разработки | MDN
https://developer.mozilla.org/ru/docs/Learn_web_development/Extensions/Server-side/First_steps/Website_security

[5] [8] [9] Перестаньте использовать JWT для сессий / Хабр
https://habr.com/ru/articles/892556/

[6] Secure Cookie Attribute - OWASP Foundation
https://owasp.org/www-community/controls/SecureCookieAttribute

[7] [47] HttpOnly - OWASP Foundation
https://owasp.org/www-community/HttpOnly

[11] Testing for Cookies Attributes - WSTG - Latest | OWASP Foundation
https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/06-Session_Management_Testing/02-Testing_for_Cookies_Attributes

[14] [18] XSS- и CSRF-атаки — разбираем уязвимости вместе с экспертом
https://tproger.ru/articles/xss-i-csrf-ataki-razbiraem-ujazvimosti

15 16 17 Website security - Learn web development | MDN
https://developer.mozilla.org/en-US/docs/Learn_web_development/Extensions/Server-side/First_steps/Website_security

26 27 28 39 PHP Security Mini Guide Part 3: XSS and Password Storage
https://www.acunetix.com/websitesecurity/php-security-3/

35 CORS - Glossary | MDN
https://developer.mozilla.org/en-US/docs/Glossary/CORS

40 41 42 Password Storage - OWASP Cheat Sheet Series
https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html

43 Cross-Site Request Forgery Prevention - OWASP Cheat Sheet Series
https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html

44 45 Web Parameter Tampering | OWASP Foundation
https://owasp.org/www-community/attacks/Web_Parameter_Tampering

48 49 50 51 A09 Security Logging and Monitoring Failures - OWASP Top 10:2021
https://owasp.org/Top10/A09_2021-Security_Logging_and_Monitoring_Failures/