

# Web Security

# Cross-site Scripting (XSS): атака

Внедрение в веб-сайт вредоносного кода на JavaScript.

Сохранение JS в базе данных сайта:

- Пост на форуме
- Личное сообщение
- Никнейм
- Всё что угодно...

У всех, кто увидит контент, JS выполнится.

```
<script>alert(1)</script>
```

Post

Внедрение JS в строку URL:

```
http://myapp.com?query=<script>alert(1)</script>
```

У всех, кто посетит ссылку, JS выполнится.

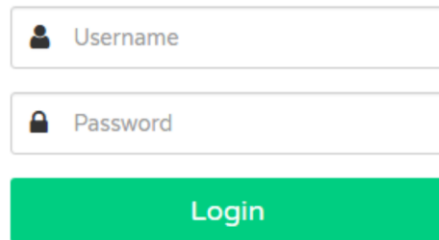
# Cross-site Scripting (XSS): защита

- Не верить пользователю
- Экранировать все значения от форм и строки запроса при сохранении в базу данных
- При выводе на страницу использовать `innerText` а не `innerHTML`
- Добавить в ответ сервера заголовок `Content-Security-Policy`.  
Пример:

```
Content-Security-Policy: default-src 'self'; img-src *; media-src  
media1.com media2.com; script-src userscripts.example.com
```

# Offtopic: Cookies

1. Пользователь открывает форму входа, вводит логин и пароль, отправляет форму на сервер



A login form consisting of two input fields and a button. The first field is labeled 'Username' with a user icon. The second field is labeled 'Password' with a lock icon. Below these fields is a green button labeled 'Login'.

2. Сервер проверяет логин и пароль и, если успешно, добавляет в ответ заголовок:

Set-Cookie: AuthToken=Abc123Secure

3. Браузер с каждым запросом на **тот же домен** отправляет заголовок **автоматически**:

Cookie: AuthToken=Abc123Secure

4. Сервер проверяет значение из cookie

# Offtopic: атрибуты Cookie

**Expires** — дата окончания срока действия

**Max-Age** — количество секунд до окончания срока действия

**Domain** — поддомен, для которого cookie будет отправляться

**Path** — путь в URL, для которого cookie будет отправляться

**Secure** — если true, то cookie отправляться только по HTTPS

**HttpOnly** — если true, то cookie не может быть прочитана из JS

**SameSite** — если Strict, то cookie не посылается при запросе с другого домена (см. следующий слайд)

# Cross-site request forgery (CSRF): атака

1. Форма на сайте bank.com, отправляется с авторизационной cookie в заголовке

```
<form method="POST" action="/transfer">  
  <input type="text" name="to_account" />  
  <input type="text" name="amount" />  
  <input type="submit" />  
</form>
```

2. Злоумышленник заманивает авторизованного на сайте банка пользователя на сайт evil.com со следующей формой (и сам её отправляет):

```
<form method="POST" action="https://bank.com/transfer">  
  <input type="text" name="to_account" />  
  <input type="text" name="amount" />  
  <input type="submit" />  
</form>
```

# Cross-site request forgery (CSRF): защита

- Проверять на сервере заголовки Referrer и Origin
- Выставлять на сервере CORS заголовки, которые запрещаются выполнять AJAX запросы с любых хостов
- Установить авторизационной cookie атрибут SameSite=Strict
- CSRF-token: генерация скрытого поля в форме и проверка его значения на сервере

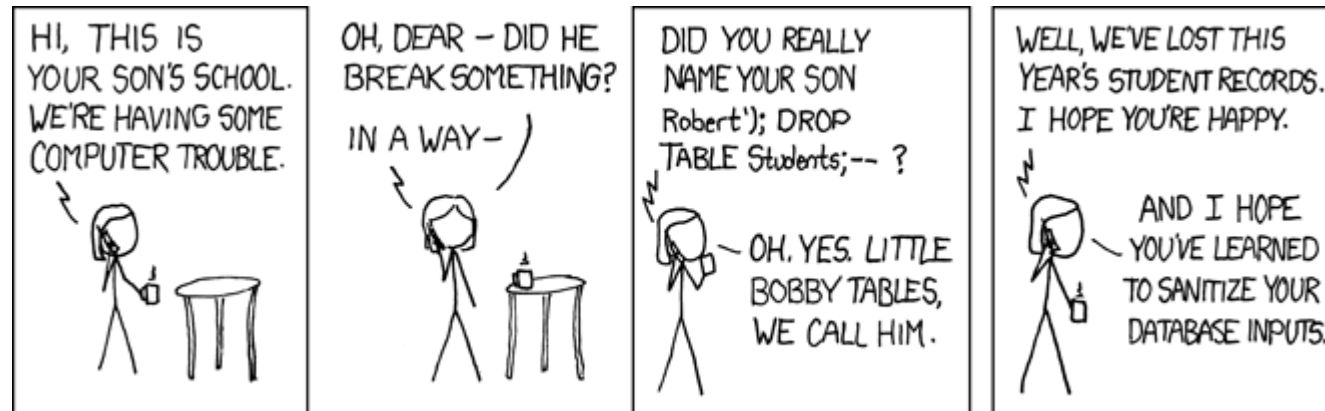
# SQL Injection: атака

1. Сервер генерирует имя пользователя, читая его из БД, фильтруя по ID из строки запроса или поля формы:

`http://myapp.com/users?id=123`

2. Злоумышленник, предполагая формат SQL запроса к БД, подставляет вместо ID часть зловредного SQL запроса:

`http://myapp.com/users?id=123;UPDATE  
Users SET IsAdmin = 1 WHERE ID == 777`





# SQL Injection: защита

- Не верить пользователю
- Использовать параметры при формировании SQL запроса (а не конкатенацию строк)
- Использовать Object Relational Mapper (ORM)

# Local File Inclusion: атака

1. На сервере существует способ скачать/посмотреть какой-то загруженный или сгенерированный файл:

`http://vulnerable_host/preview.php?file=example.html`

2. Злоумышленник пытается скачать другой файл с конфиденциальной информацией:

`http://vulnerable_host/preview.php?file=../../../../config.production.json`

# Local File Inclusion: защита

- Хранить реестр файлов с идентификаторами
- Проверять директорию, из которой файл скачивается
- Ограничить на уровне операционной системы доступ к файлам процессу веб-приложения

# Security Misconfiguration: атака

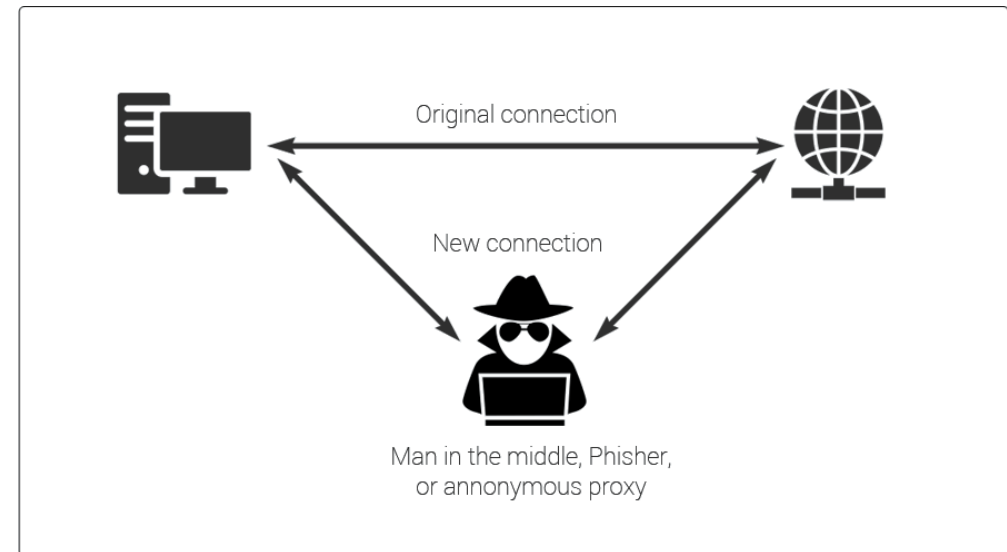
1. Злоумышленник вводит в форму данные, которые приводят к ошибке на сервере.
2. Сервер показывает developer page: stack trace, строки исходного кода.
3. Злоумышленник видит конфиденциальную информацию: строку подключения, секретную бизнес-логику

# Security Misconfiguration: защита

1. Не верить пользователю.
2. Отключить development режим в конфигурации сервера.
3. Не хранить конфиденциальные данные в исходном коде.
4. Для секретной бизнес-логики использовать отдельный сервис.

# Man in the middle (MITM): атака

1. Пользователь посылает запрос серверу веб-приложения.
2. Злоумышленник перехватывает запрос пользователя.
3. Злоумышленник самостоятельно посылает запрос серверу веб-приложения, от имени пользователя.
4. Злоумышленник посылает полученный от сервера веб-приложения ответ пользователю.
5. В какой-то момент важные данные будут перехвачены или изменены.



# Man in the middle (MITM): защита

- Включить HTTPS
- Добавить заголовок HSTS (запрет на HTTP):  
`Strict-Transport-Security: max-age=31536000`

# Race Condition: атака

1. Серверный код написан следующим образом:

```
let currentBalance1 = await getBalance(user1);  
let currentBalance2 = await getBalance(user2);  
  
if (currentBalance1 - amount > 0) {  
    currentBalance1 -= amount;  
    currentBalance2 += amount;  
}  
  
await setBalance(user1, currentBalance1);  
await setBalance(user2, currentBalance2);
```

2. Злоумышленник во много потоков запускает этот код и получает некорректное значение balance.



# Race Condition: защита

- Использовать транзакции на уровне базы данных
- Использовать распределённые блокировки
- Использовать однопоточный единственный веб-сервер

Open Web Application Security Project (OWASP) — это открытый проект обеспечения безопасности веб-приложений.

Каждый год публикуют доклад OWASP Top Ten, со списком самых опасных уязвимостей за год.

[https://www.owasp.org/index.php/OWASP\\_Top\\_Ten\\_Cheat\\_Sheet](https://www.owasp.org/index.php/OWASP_Top_Ten_Cheat_Sheet)