

Курс «Веб-технологии: уязвимости и безопасность»

# Client-Side-уязвимости

CORS, основы regexp, PostMessage, WebSocket, Content-Type и все что с ними связано. Дополнительные источники знаний и практики.

## Оглавление

[Введение](#)

[Видеоурок 1](#)

[Безопасность при CORS](#)

[Практика](#)

[Основы regexp](#)

[Origin=null](#)

[Итоги](#)

[Видеоурок 2](#)

[Post Message](#)

[Практика](#)

[Безопасность PostMessage](#)

[Итоги](#)

[Видеоурок 3](#)

[WebSocket](#)

[Практика](#)

[Безопасность WebSocket](#)

[Итоги](#)

[Видеоурок 4](#)

[Браузеры и Content Sniffing](#)

[Заголовок Content-Type](#)

[Заголовок Content-Disposition](#)

[Итоги](#)

[Видеоурок 5](#)

[Remote Code Execution \(RCE\)](#)

[SQL Injection](#)

[Buffer Overflow](#)

[Path Traversal](#)

[XSS Injection](#)

[Итоги](#)

[Что дальше?](#)

[BugBounty](#)

[Capture the Flag \(CTF\)](#)

[Конференции по ИБ](#)

[Исследования](#)

[Читать и учить новое](#)

[Ссылки к уроку](#)

# Введение

В этом уроке мы разберем современные технологии, применяемые на стороне пользователя, а если быть точнее, в браузере (так называемые **ClientSide**-технологии), и другие уязвимости веба.

Мы уже освоили основные веб-технологии и посмотрели на их безопасность, изучили безопасность в браузерах, подробно рассмотрели **Same Origin Policy**. Теперь нам предстоит узнать, какие **ClientSide**-технологии есть в вебе, какие уязвимости в них могут возникнуть при реализации.

Урок построен следующим образом:

- 1) **Cross Origin Resource Sharing (CORS)** и его уязвимости.
- 2) Что такое **Post Message**, зачем он нужен, какие уязвимости могут быть и как сделать его безопасным.
- 3) **Web Socket**.
- 4) Механизм распознавания содержимого страницы.
- 5) Основные уязвимости веба, общая картина и развитие в области управления безопасностью.

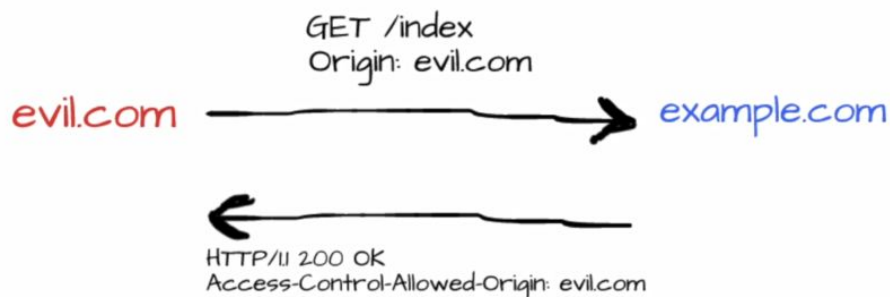
К концу урока вы поймёте, как работает **CORS**, **PostMessage** и **WebSocket**, будете знать основные уязвимости, связанные с этими технологиями, и уметь реализовывать эти технологии безопасно.

Также вы будете знать об основных, самых критичных веб-уязвимостях, и получите направление для дальнейшего развития.

# Видеоурок 1

**Same Origin Policy** запрещает взаимодействие между разными origin, но, если сайту необходимо междоменное взаимодействие, можно воспользоваться **Cross Origin Resource Sharing** и ослабить для некоторых доменов политику **Same Origin Policy**.

## Безопасность при CORS



Сайт <http://evil.com> посылает запрос на <http://example.com> — запрос кроссдоменный, т.е. с разных доменов. При этом он включает заголовок **Origin: http://evil.com**, в котором пишется **origin** текущей страницы. Этот заголовок браузер проставляет автоматически и ему можно доверять, но с некоторыми оговорками. Сайт <http://example.com> видит заголовок **origin**, опознаёт его как CORS-запрос, ставит заголовок **Access-Control-Allow-Origin**, добавляет туда <http://evil.com> и возвращает ответ. Браузер видит заголовок **Access-Control-Allow-Origin: http://evil.com** и сверяет **origin**, который там находится, с тем который есть. Если оба **origin** совпадают, браузер разрешает прочитать ответ, а если нет, не разрешает — так работает **Same Origin Policy**.

## Практика

Посмотрим на пример небезопасного **CORS**. Безопасность **CORS** зависит от того, насколько хорошо мы сделаем список разрешенных доменов, то есть тех, которым разрешено читать ответы при кроссдоменном запросе.

Откроем Ubuntu, конфигурацию **nginx**:

```
cd /etc/nginx/sites-enabled && sudo nano default
```

```
root /var/www/html;

# Add index.php to the list if you are using PHP
index index.html index.htm index.nginx-debian.html;

server_name _;

location / {
```

```
# First attempt to serve request as file, then
# as directory, then fall back to displaying a 404.
add_header Access-Control-Allow-Origin "$http_origin";
try_files $uri $uri/ =404;
}
```

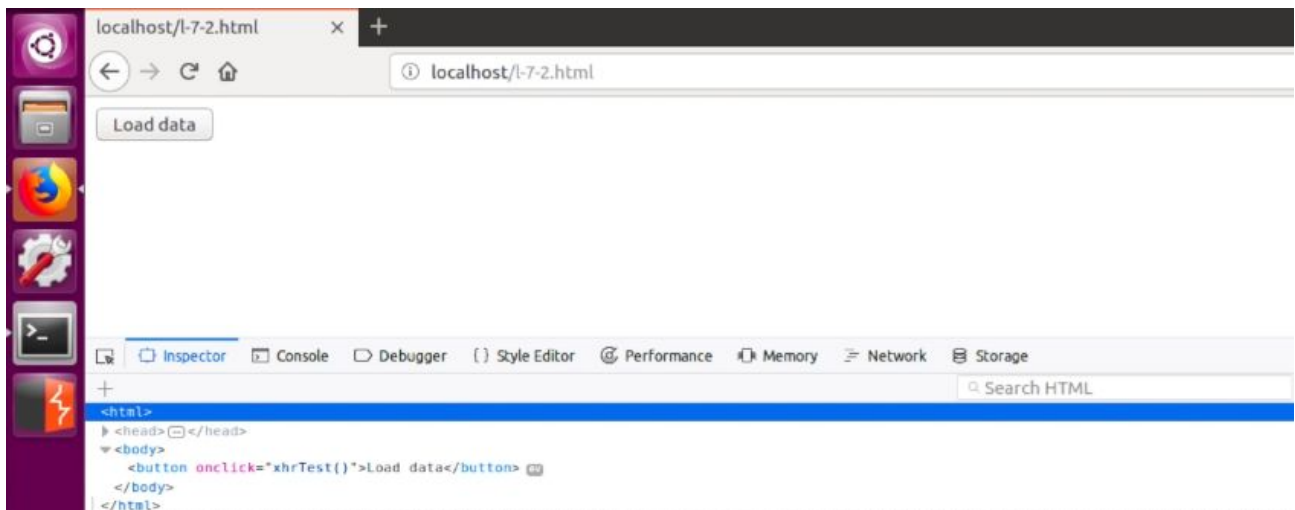
И добавим сюда небезопасно **"\$http\_origin"**. Это значит, что какой бы **origin** ни пришел за страницей, **nginx** его просто отобразит в заголовке **Access-Control-Allow-Origin** и позволит запросившему домену прочитать все данные, которые сервер отправит в ответе.

Перезагрузим **nginx**:

```
sudo nginx -s reload
```

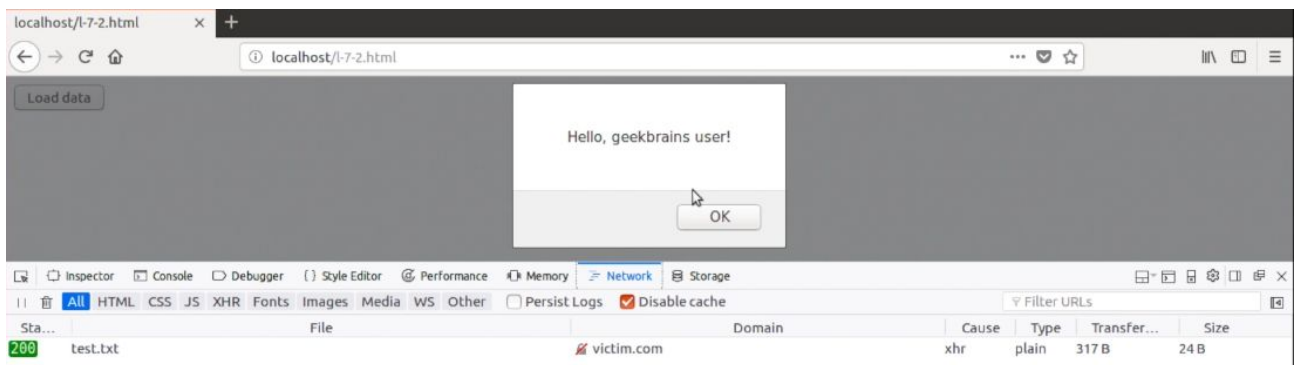
Попробуем делать кроссдоменные запросы с помощью **XHR**.

Воспользуемся файлом <http://localhost/l-7-2.html> из предыдущего урока:

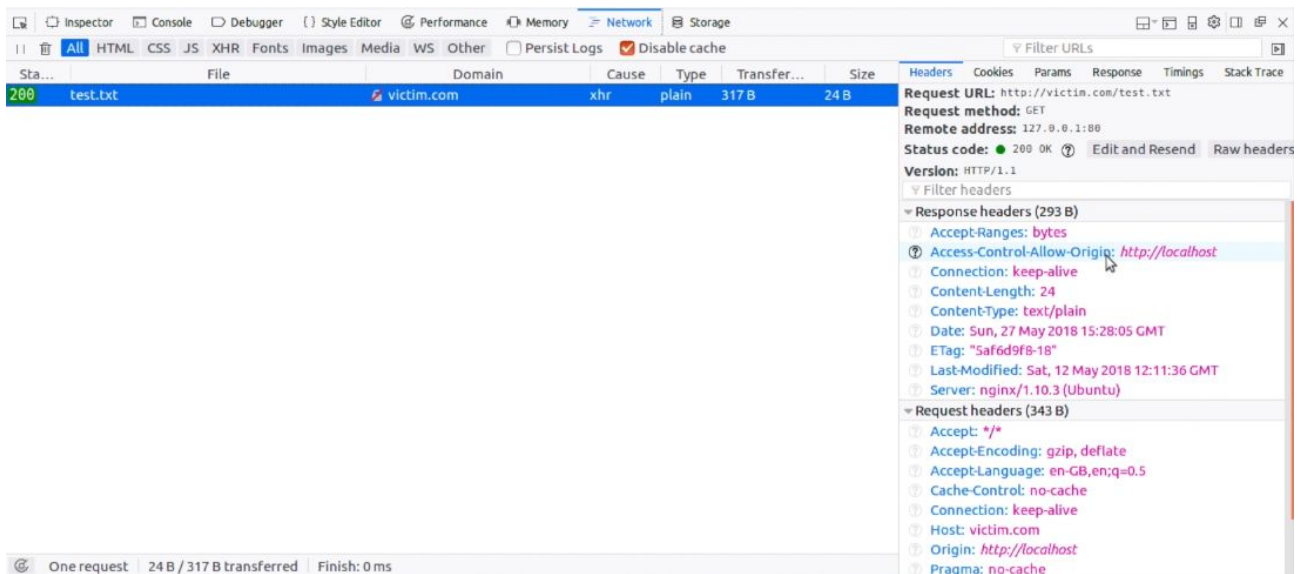


При нажатии на кнопку **Load data** идет запрос с исходного сайта на сервер [victim.com](http://victim.com), ответ на него высвечивается в **alert()**.

Откроем вкладку **Network**, чтобы видеть запросы, и нажмем **Load data**:

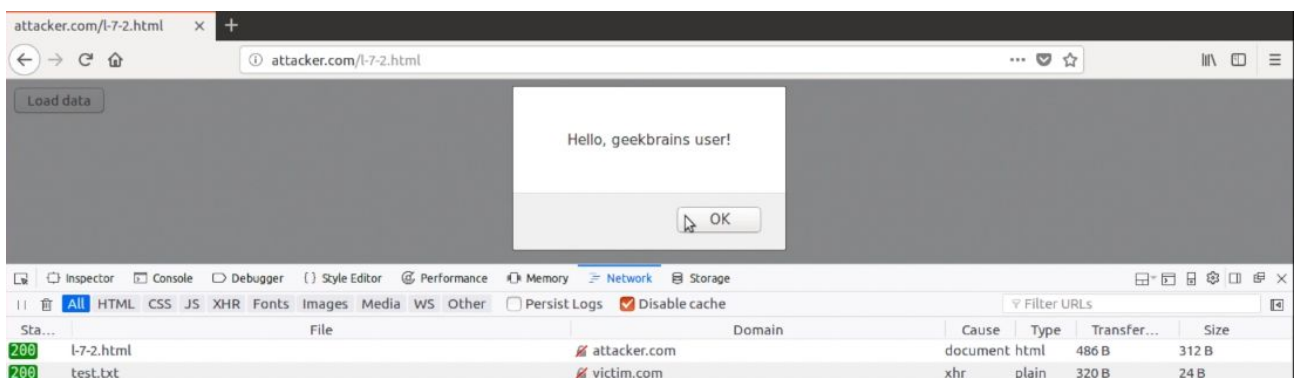


Данные успешно получены. Посмотрим на запрос:



Запрос был сделан с сайта <http://localhost> на домен [victim.com](http://victim.com) — это кроссдоменный запрос. Также мы видим заголовки: в **Request headers** стоит **Origin: <http://localhost>**, как и должно быть, и в ответе заголовок разрешает <http://localhost> прочитать содержимое ответа.

Допустим, <http://localhost> — доверенный домен, но что, если [attacker.com](http://attacker.com) захочет сделать то же самое? Проверим, переходим на <http://attacker.com/l-7-2.html> и нажимаем «Загрузить»:



Видим, что для [attacker.com](http://attacker.com) тоже разрешен кроссдоменный запрос, так как **nginx** на стороне запрашиваемого домена [victim.com](http://victim.com) проставит заголовок **Access-Control-Allow-Origin** для абсолютно любого домена, который придет за этим запросом.

Так делать нельзя, потому что никто не хочет нарушать **Same Origin Policy**, чтобы любые домены могли делать такие запросы и читать ответы на них. Рассмотрим правило **Same Origin Policy** — откроем конфиг **nginx** и добавим условие:

```
cd /etc/nginx/sites-enabled && sudo nano default

root /var/www/html;

# Add index.php to the list if you are using PHP
index index.html index.htm index.nginx-debian.html;

server_name _;

location / {
    # First attempt to serve request as file, then
    # as directory, then fall back to displaying a 404.
    if ($http_origin ~* (app.victim.com)) {
        add_header Access-Control-Allow-Origin "$http_origin";
    }
    try_files $uri $uri/ =404;
}
```

Значение переменной **\$http\_origin** будет сравниваться с **~\*** регулярным выражением справа (**app.victim.com**). Здесь **~** означает, что это регулярное выражение, а звездочка **\*** — что это **case insensitive**, то есть, что между большими и маленькими буквами нет разницы. Напишем ли мы **APP** или **app** — это будет восприниматься одинаково.

Теперь немного усложним, чтобы поняли, что такое регулярное выражение:

```
root /var/www/html;

# Add index.php to the list if you are using PHP
index index.html index.htm index.nginx-debian.html;

server_name _;

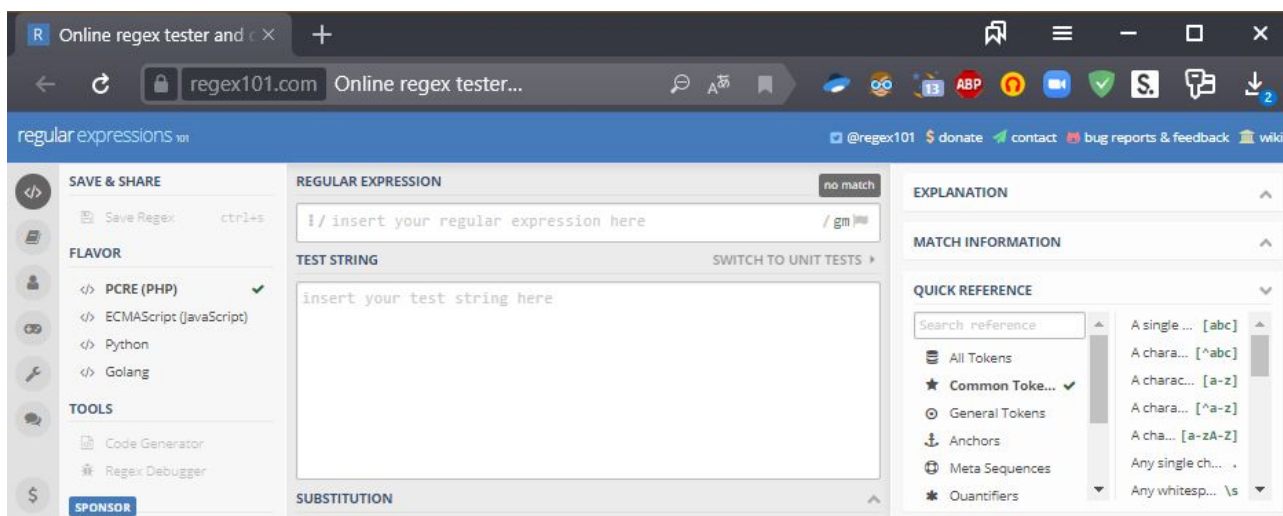
location / {
    # First attempt to serve request as file, then
    # as directory, then fall back to displaying a 404.
    # default_type application/javascript;
    if ($http_origin ~* (a+pp.victim.com)) {
        add_header 'Access-Control-Allow-Origin' "$http_origin";
    }
    try_files $uri $uri/ =404;
}
"/etc/nginx/sites-enabled/default" 98L, 2451C written
```

Это регулярное выражение. **a+** означает, что символ **a** может повторяться один или больше раз. Обычные символы (**a**, **b**) обозначают сами себя, точка (**.**) обозначает любой символ. Регулярные выражения в вебе встречаются очень часто.

## Основы regex

Рассмотрим регулярные выражения на примере.

Есть специальные сайты — <https://regex101.com>, <https://regexone.com> — там вы научитесь основам **regex**:



Регулярные выражения — очень полезный навык для IT- и ИБ-специалистов. Их нужно делать как можно более точными, чтобы не возникало проблем безопасности, про которые мы сейчас узнаем. Посмотрим на их работу.

Для поиска строки в файлах по регулярным выражениям в **Linux** есть программа **grep**. Первым аргументом программа принимает собственно регулярное выражение, то есть некий шаблон поиска, а вторым — файл, в котором нужно искать.

Перейдём в директории **/tmp** и создадим там файл:

```
cd /tmp && nano test
```

```
Hello, user! Amazing!
123
user Pavel
```

Попробуем найти выражение файле или в строке по регулярному выражению (grepнуть). Применим **grep**: допустим, мы хотим найти все слова **user**. Самое простое регулярное выражение — сама поисковая строка, т. к. здесь нет специальных метасимволов (токенов):

```
grep 'user' test
```

```
Hello, user! Amazing!
user Pavel
```

**Grep** нашел строчки, в которых есть **user**, и подсветил нам их. Подсветку можно выключить, это опционально.

Другой пример — используем мета-символы:

```
grep 'el{2}o' test
```

```
Hello, user! Amazing!
```

Выражение `l{2}` обозначает, что буква `l` встречается два раза, то есть он ищет все вхождения. где есть `ello`. Если написать `l{3}`, он ничего не нашёл бы:

```
grep 'el{3}o' test
```

Это можно было бы написать через метасимвол `+`, плюс, который значит один или больше символов:

```
grep 'el+o' test
```

```
Hello, user! Amazing!
```

Тут один или больше `l`, поэтому плюс сработал. Вернемся к примеру:

```
cd /etc/nginx/sites-enabled && sudo nano default
```

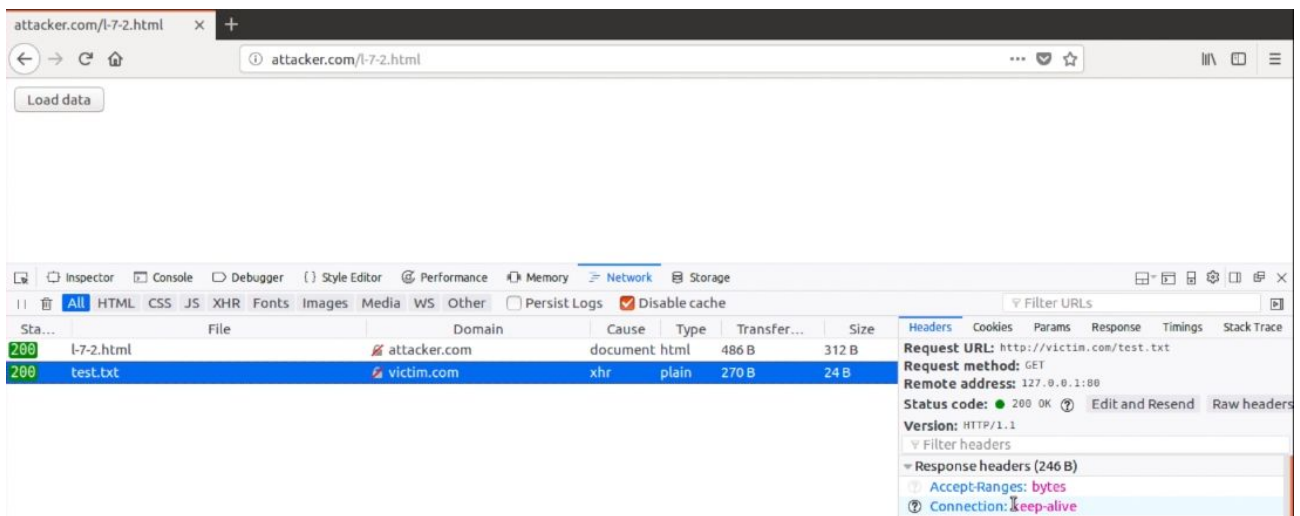
```
location / {  
    # First attempt to serve request as file, then  
    # as directory, then fall back to displaying a 404.  
    if ($http_origin ~* (a+app.victim.com)) {  
        add_header Access-Control-Allow-Origin "$http_origin";  
    }  
}
```

Правило такое: один или больше символов `a`, потом идёт `app`, потом любой символ, `victim`, потом любой символ, потом `com`. Выглядит, как будто это просто точка, но на самом деле это метасимвол `.`, обозначающий любой символ. Убедимся в этом. Сохраним и перезагрузим `nginx`:

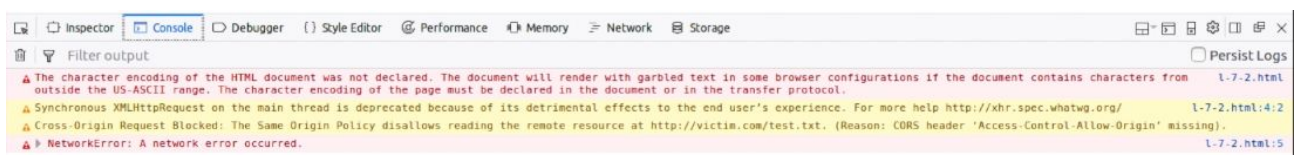
```
sudo nginx -s reload
```

Проверим через <http://attacker.com/l-7-2.html> — нажимаем «Загрузить»:





Ничего не загрузилось: заголовок **CORS** не проставлен, а в консоли информация о блокировке:



Теперь добавим хост **aapp.victim.com** в список известных:

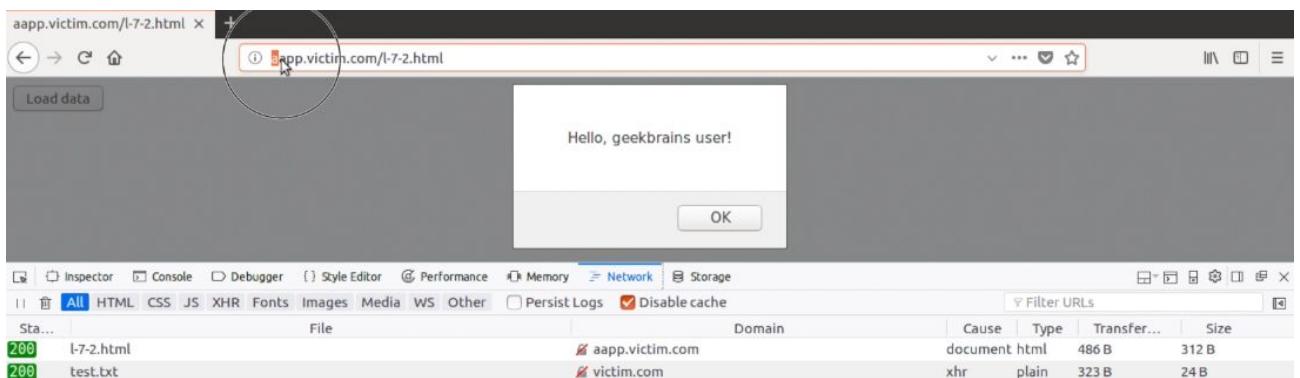
```
cd /etc && sudo nano hosts

127.0.0.1    localhost
127.0.0.1    aapp.victim.com aappqvictim.com victim.com
127.0.0.1    attacker.com
```

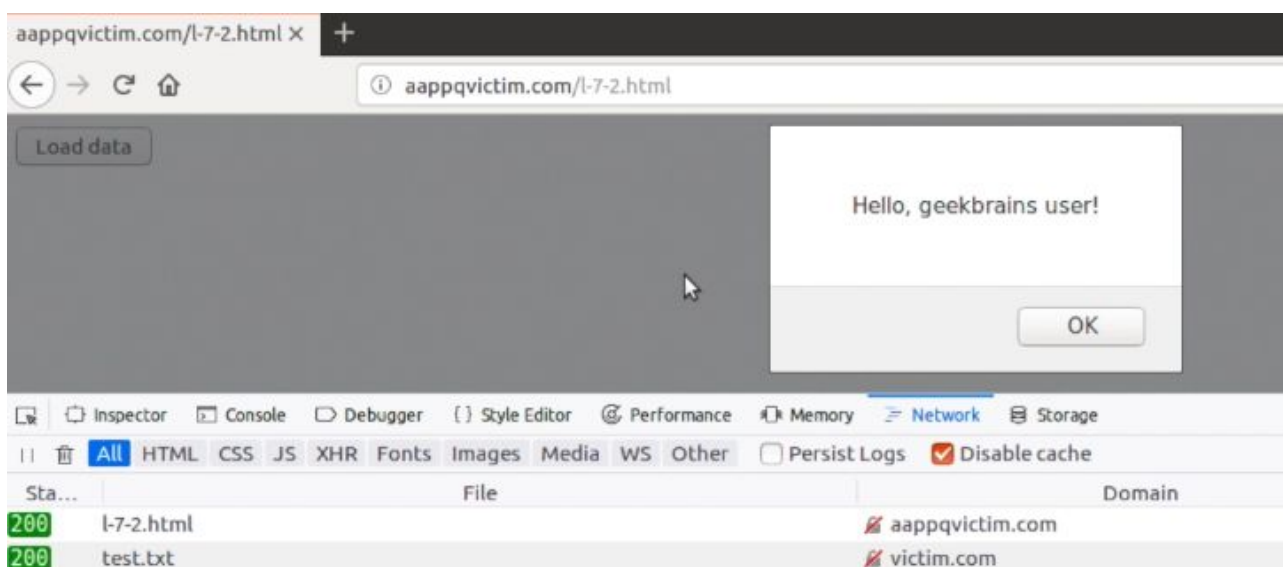
Попробуем загрузить сайт <http://aapp.victim.com/l-7-2.html>

Нажимаем

«Загрузить»:



[aapp.victim.com](http://aapp.victim.com) подходит под регулярное выражение, потому что здесь одна или больше **a**. потом идет **app.victim.com** — на первый взгляд всё безопасно. На самом деле нет: что, если злоумышленник зарегистрирует домен [aappqvictim.com](http://aappqvictim.com)?



С такого домена тоже всё загрузилось.

Этот домен может зарегистрировать кто угодно, в том числе и злоумышленник — это не поддомен [victim.com](http://victim.com), и за счет этого возникает уязвимость.

Это возможно благодаря тому, что в конфиге использована точка, а в регулярных выражениях она обозначает любой символ. Её можно заменить, например, на `q` или на что угодно. Это уязвимость; в регулярных выражениях очень легко ошибиться и нужно их писать аккуратно и как можно более точно.

Исправим, чтобы было безопасно — добавим обратный слеш. Тогда точки будут восприниматься как точки, а не метасимволы:

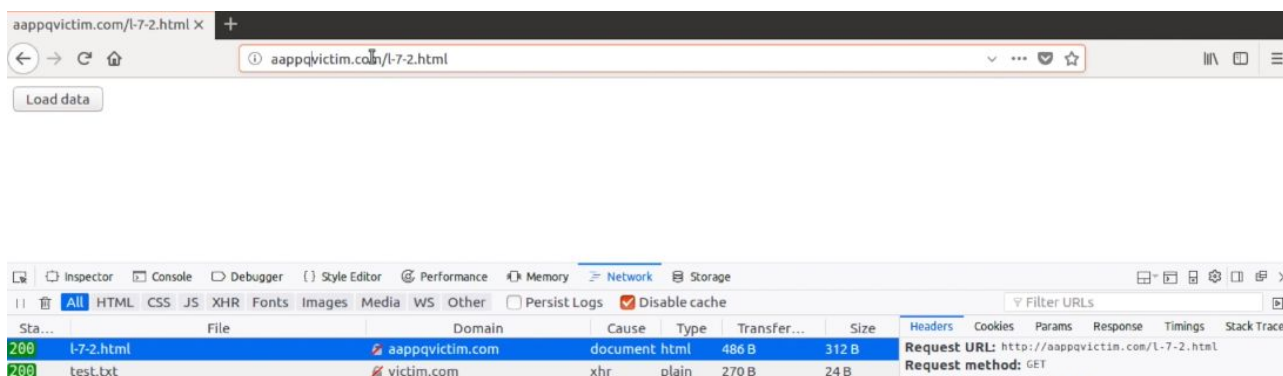
```
cd /etc/nginx/sites-enabled && sudo nano default

location / {
    # First attempt to serve request as file, then
    # as directory, then fall back to displaying a 404.
    if ($http_origin ~* (a+app\\.victim\\.com)) {
        add_header Access-Control-Allow-Origin "$http_origin";
    }
}
```

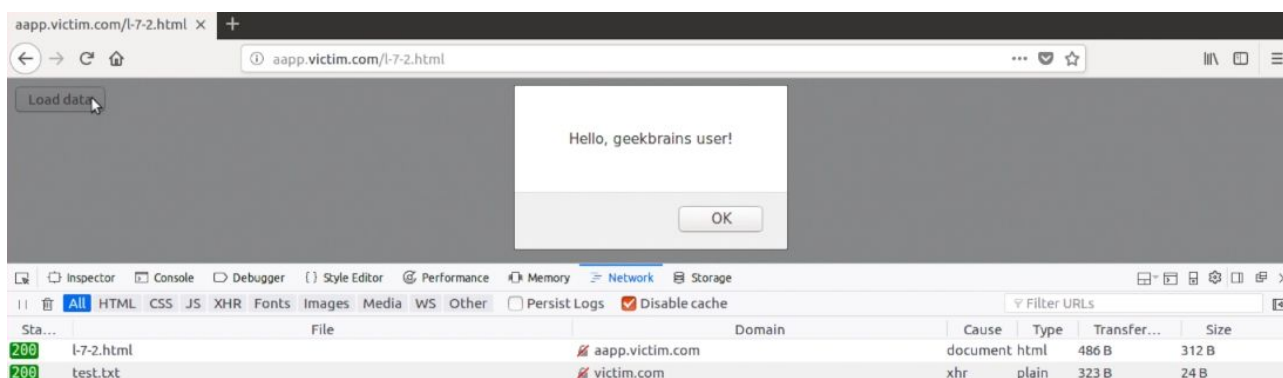
Сохраним и перезагрузим **nginx**:

```
sudo nginx -s reload
```

С домена [aappqvictim.com](http://aappqvictim.com) ничего не загружается:



Если мы поменяем q на точку, с домена [aapp.victim.com](http://aapp.victim.com) всё загрузится:



Теперь всё безопасно, потому что регулярное выражение разрешает именно те домены, которые должно.

## Origin=null

Если домен разрешает читать для **origin=null**, например, если в ответе стоит заголовок **Access-Control-Allow-Origin: null**, это тоже небезопасно — так **iframe** с **sandbox** уже будут иметь **origin=null**.

Злоумышленники делают на своем сайте **iframe**, в нём — **sandbox** и отправляют запросы к вашему домену. Так как **origin** у **sandbox=null** и ваш домен разрешает читать для него, **iframe** злоумышленника сможет читать ответы вашего домена, а так быть не должно.

Следим, чтобы в **Access-Control-Allow-Origin** не попадало лишних доменов и **null** — тогда ваш **CORS** будет безопасным.

## Итоги

Подведём итоги:

- 1) Вы узнали, как работает **CORS**, какими заголовками он задается и управляется.

- 2) Вы узнали, какие уязвимости могут встречаться при реализации **CORS** и увидели на примере, как неправильная конфигурация **nginx** может привести к уязвимостям при кроссдоменных запросах (при **CORS**).

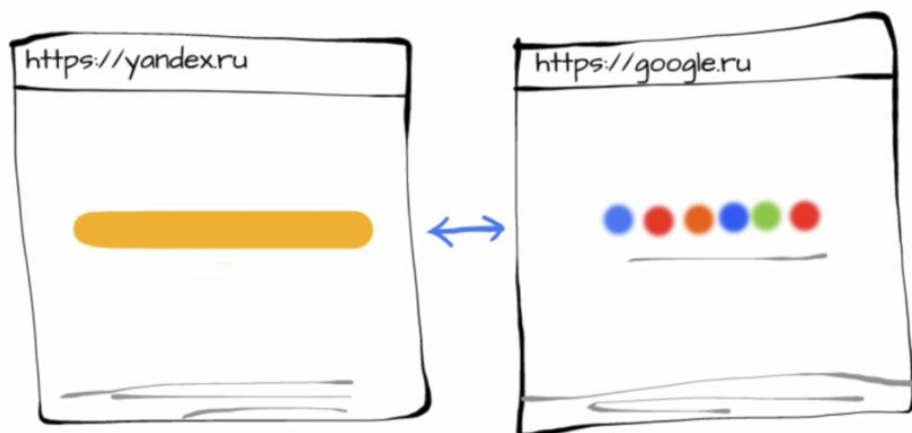
## Видеоурок 2

В этом видео мы рассмотрим:

- 1) Что такое **Post Message**, где он применяется и зачем он нужен.
- 2) Как работает **Post Message**, напишем пример кода.
- 3) Уязвимости, которые связаны с **Post Message**, и как их избежать. Как обезопасить свой сайт от уязвимостей **Post Message**.

### Post Message

Чтобы организовать взаимодействие между окнами и фреймами, существует **Post Message** — протокол, который позволяет одному окну общаться с другим. Он предоставляет специальный интерфейс, чтобы обойти **Same Origin Policy**, и разные **origin** могли общаться между собой, но только в пределах одного браузера.



**Post Message** не отправляет данные на сервер, т. к. это исключительно **Client-Side**-технология, которая полностью реализована на клиенте и там же остаются все данные.

### Практика

Разберем **Post Message** на примере. Создадим две страницы: одна — того, кто будет отправлять **Post Message**, вторая — того, кто будет принимать.

Сначала создадим файл для отправителя — назовём эту страницу **postmessage.html**:

```
cd /var/www/html && sudo nano postmessage.html
```

Создадим окно, которое будет принимать, им может быть как **iframe**, так и окно в соседней вкладке и даже отдельное окно браузера — подходят все три варианта.

Чтобы отправить **Post Message**, нам нужно получить объект окна, то есть тот, который будет храниться в переменной, к которой мы потом обратимся, чтобы отправить сообщение. Мы не можем отправить его в никуда, чтобы оно дошло куда нужно.

Используем **windows.open()** — это функция JavaScript, которая откроет окно в соседней вкладке:

```
<body>
<script>
    var targetWindow = window.open("http://victim.com/pm-receiver.html");
</script>
</body>
```

Откроем сайт <http://victim.com/pm-receiver.html>, эту страницу мы напишем чуть позже.

Добавим кнопку, чтобы открывать это окно, и обернём код в функцию. Это нужно, чтобы по событию **onclick()** при нажатии на кнопку мы смогли открыть окно, то есть выполнить код:

```
<body>
<script>
    function openTargetWindow() {
        var targetWindow = window.open("http://victim.com/pm-receiver.html");
    }
</script>
<button onclick="openTargetWindow()">Open Target Window</button>
</body>
```

Теперь нужно сохранить это окно в переменную, чтобы потом к ней обратиться. Создадим глобальную переменную **target** и в конце присвоим ей значение:

```
<body>
<script>
    var target = null;
    function openTargetWindow() {
        target = window.open("http://victim.com/pm-receiver.html");
    }
</script>
<button onclick="openTargetWindow()">Open Target Window</button>
</body>
```

Мы открыли окно, и его объект сохранили в переменной **target**

Приступим к написанию функции **sendMessage()** для отправки сообщений:

```

<body>
<script>
  var target = null;
  function openTargetWindow() {
    target = window.open("http://victim.com/pm-receiver.html");
  }

  function sendMessage() {

  }
</script>
<button onclick="openTargetWindow()">Open Target Window</button>
<input id="message"/>
<button onclick="sendMessage()">Send Message</button>
</body>

```

Поле ввода **input** будет содержать сообщение, которое мы хотим отправить, а кнопка будет вызывать функцию **sendMessage()**

Вызовем документ и функцию **getElementById()** — мы не просто так написали **id="message"** в **input**: это поможет удобно его достать:

```

<body>
<script>
  var target = null;
  function openTargetWindow() {
    target = window.open("http://victim.com/pm-receiver.html");
  }

  function sendMessage() {
    var message = document.getElementById("message").value;
  }
</script>
<button onclick="openTargetWindow()">Open Target Window</button>
<input id="message"/>
<button onclick="sendMessage()">Send Message</button>
</body>

```

Нам нужно значение, которое мы напишем в **input**, а не сам элемент, поэтому мы и пишем **.value** — это значение. У нас есть сообщение, отправим его в окно [victim.com](http://victim.com). Для этого напишем:

```

<body>
<script>
  var target = null;
  function openTargetWindow() {
    target = window.open("http://victim.com/pm-receiver.html");
  }

```

```
function sendMessage() {
    var message = document.getElementById("message").value;
    target.postMessage(message, "http://victim.com");
}
</script>
<button onclick="openTargetWindow()">Open Target Window</button>
<input id="message"/>
<button onclick="sendMessage()">Send Message</button>
</body>
```

**Target** — объект окна и мы вызвали для него функцию **postMessage()**. Эта функция первым аргументом принимает само сообщение (в нашем случае — **message**), а вторым — **origin**, которому разрешено получить это сообщение.

Важный момент, связанный с безопасностью: в **origin** необходимо указать конкретный **origin**. Там возможно указать, например, звездочку, но это будет значить, что любой домен сможет получить это сообщение. А если злоумышленник каким-то образом сможет повлиять на **window.open()**, на его сайт уйдут, например, пароли, учетные данные или другие важные данные с сайта.

Здесь мы указываем четко <http://victim.com> и не ставим звездочку, она практически никогда не нужна. Если вдруг мы указываем звездочку, нужно проверить, чтобы там не было чувствительных данных, которые могут нарушить конфиденциальность пользователя или компании.

Итак, отправляющая сторона на этом готова, теперь отредактируем получающую:

```
cd /var/www/html && sudo nano pm-receiver.html
```

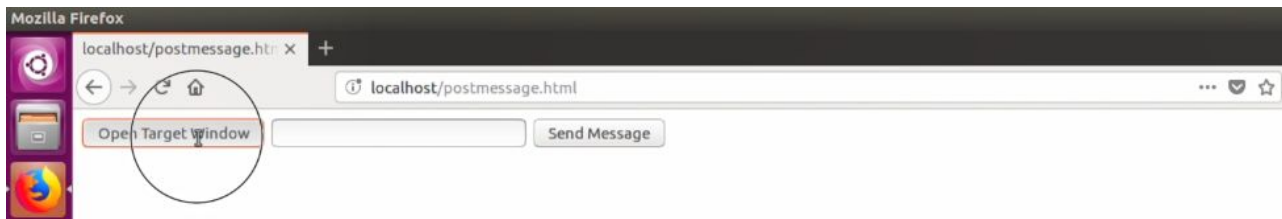
На получающей стороне (по умолчанию **postMessage**) сообщения приходят и с ними ничего не происходит, они, можно сказать, растворяются в воздухе. Чтобы получить сообщение, нужно добавить обработчик события. По сути, когда приходит сообщение в целевое окно, браузер оповещает его об этом, и окно может проигнорировать сообщение, если обработчика нет, либо обработать, если есть.

Добавим обработчик события — напомним **window.addEventListener()**:

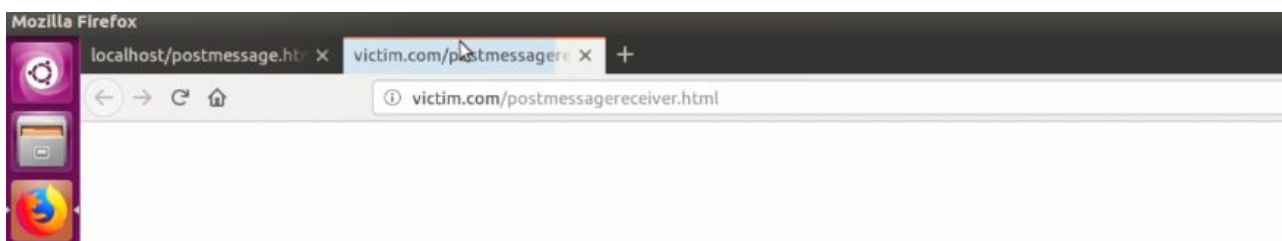
```
<body>
<script>
    function receiveMessage(event) {
        document.body.innerHTML = "<p>" + event.data + "</p>";
    }
    window.addEventListener("message", receiveMessage);
</script>
</body>
```

Первым аргументом идет тип сообщения — **message**, вторым — функция, которую необходимо вызвать — обработчик, и на этом получатель готов.

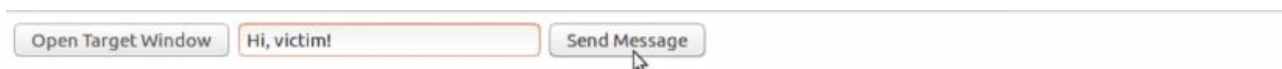
Проверим, что все работает — зайдём в **Firefox** на <http://localhost/postmessage.html> и нажмем кнопку **Open Target Window**:



Окно открылось:



Напишем и отправим сообщение:



Переходим на вторую закладку — видим, что сообщение успешно пришло:



Мы можем написать еще что-нибудь, отправить сообщение и отправка опять будет успешной. Если мы откроем новое окно и отправим сообщение, увидим, что оно отправляется только в новое окно. Это происходит из-за того, что объект окна теперь содержит новое окошко — хоть они и находятся по одинаковому адресу, но это разные объекты.

## Безопасность PostMessage

В данном случае **PostMessage** не совсем безопасен, потому что получатель не проверяет, от кого ему пришли данные.

При использовании **PostMessage**, необходимо проверять, куда уходят данные. Мы можем отправить их не туда, куда нужно, и чувствительные данные, например пароли, могут утечь злоумышленнику в руки. Также нужно принимать данные только оттуда, откуда мы их ожидаем.

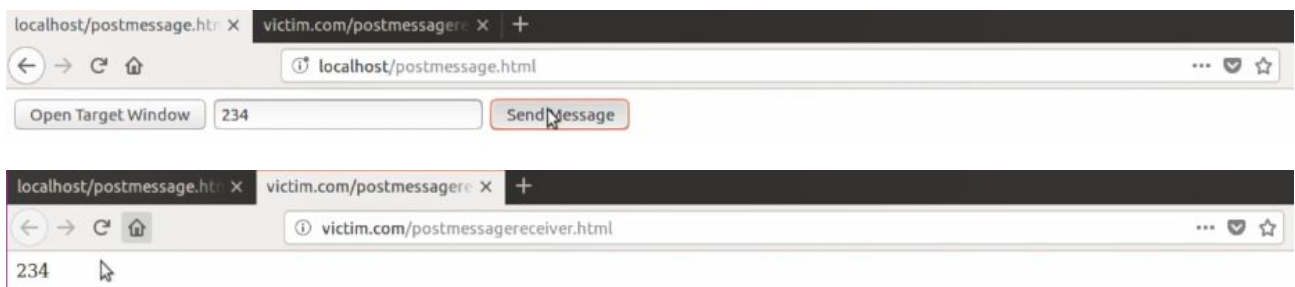


Для этого существует специальный объект в объекте **event**, который называется **origin**, и мы должны проверить, что **origin** именно такой, какой мы ожидали:

```
<body>
<script>
  function receiveMessage(event) {
    if (event.origin !== "http://localhost") {
      return;
    }
    document.body.innerHTML = "<p>" + event.data + "</p>";
  }
  window.addEventListener("message", receiveMessage);
</script>
</body>
```

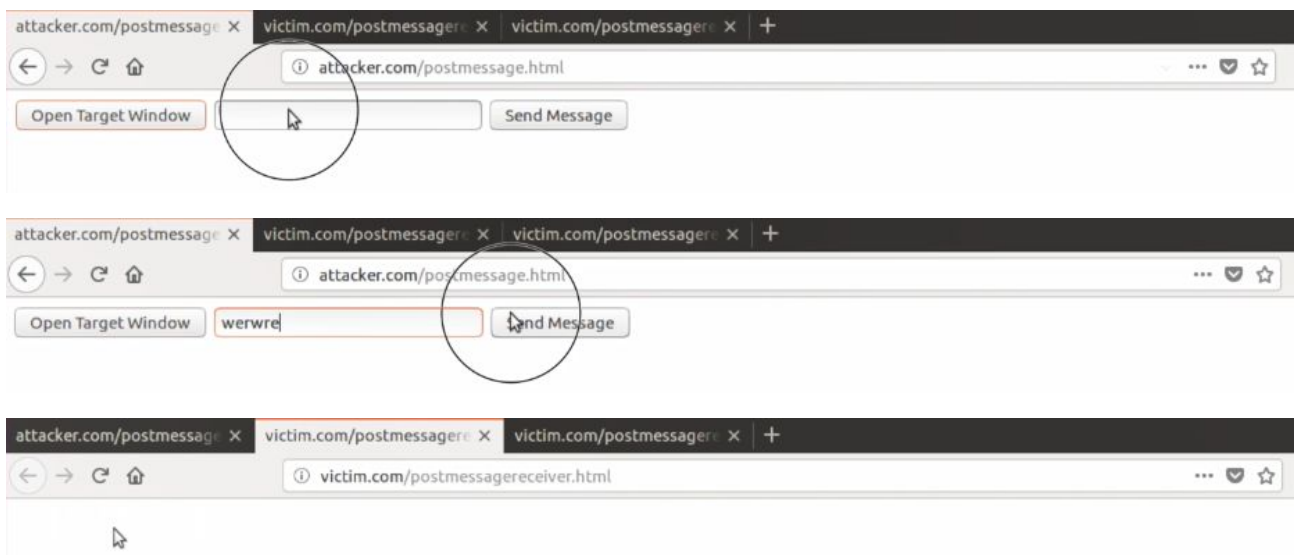
Если **origin** не равен <http://localhost>, выйдем из функции и ничего не будем делать. Проверим, что всё работает: закроем старые окна и перезагрузим первое.

Отправим сообщение:



Сообщение отправлено.

Попробуем отправить сообщение с другого **origin**:



Сообщение не отправляется — это происходит потому, что **JavaScript** сверяет **origin**, видит, что он не равен <http://localhost>, и возвращает выполнение функции, то есть прерывает.

Закрепим то, что касается безопасности **PostMessage** — отправитель обязательно должен проверять, куда он посылает данные, это делается в функции **postMessage()** в специальном аргументе. В нём ставится не звёздочка, а конкретный **origin**, куда должны уйти данные. Когда мы принимаем данные, обязательно следует проверить, от кого они пришли — сравниваем их с конкретным **origin**.

## Итоги

Подведём итоги:

- 1) Узнали, что такое **PostMessage**, для чего он применяется, где используются.
- 2) Разобрали пример **PostMessage** и узнали, как он работает.
- 3) Узнали об уязвимостях, которые связаны с **PostMessage**, поняли, как обезопасить реализацию **PostMessage** от этих уязвимостей.

## Видеоурок 3

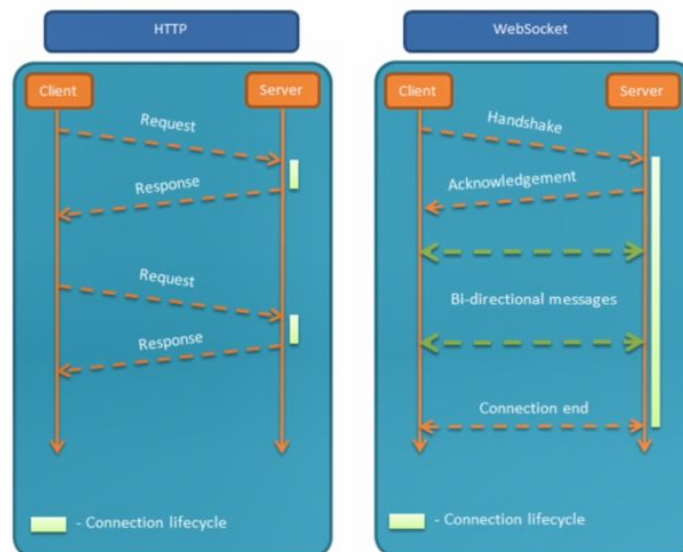
В этом видео мы рассмотрим:

- 1) Что такое **WebSocket**, для чего он нужен.
- 2) Как работает **WebSocket**.
- 3) Уязвимости, связанные с **WebSocket** на прикладном уровне.

## WebSocket

Представим, что клиенту и серверу необходимо часто передавать данные друг другу. Конечно, они могли бы отправлять **HTTP**-сообщения каждый раз, но для каждого такого сообщения нужно открывать новое соединение, а это дополнительное время. Если таких сообщений много, а данных в них — одно число или небольшой массив, открытие соединения занимает больше времени, чем передача данных.

Есть заголовок **keep-alive**, который открывает соединение и поддерживает его открытым, через него можно отправить несколько **HTTP**-запросов. Но у **HTTP** есть свои заголовки, которые тоже привносят дополнительную нагрузку, которая часто выше той, с которой мы отправляем. А **keep-alive** не может жить вечно, потому что на сервер чаще всего приходит много клиентов, и если все будут держать **keep-alive**-соединения, сервер не выдержит. **HTTP** изначально не был рассчитан на большое количество открытых соединений, поэтому придумали протокол **WebSocket**.



Основные отличия **WebSocket** и **HTTP** на верхнем уровне показаны на картинке сверху. **HTTP** делает запрос и получает ответ, чтобы получить данные ещё раз, он повторяет процедуру.

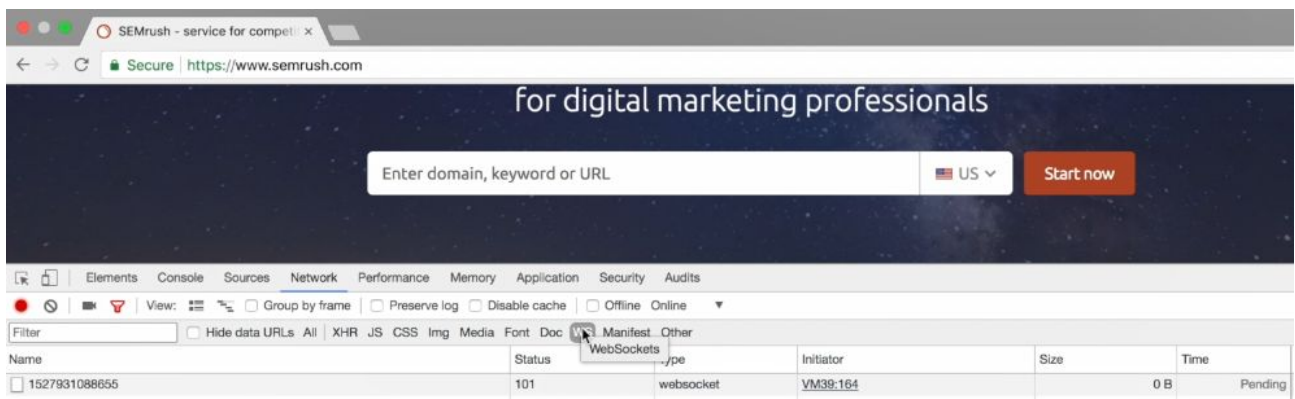
**WebSocket** работает немного по-другому, как мы видим на картинке. Сначала он договаривается с сервером о соединении, оно устанавливается и живет до завершения, то есть до того момента, пока клиент не отсоединится. Всё это время клиент и сервер могут обмениваться произвольными данными сколько угодно без дополнительной нагрузки.

**WebSocket** позволяет клиенту и серверу общаться без дополнительных затрат на установление **TCP/IP**-соединения. Подробнее о том, почему это затратно и лучше установить соединение один раз и держать его, вы узнаете в курсе по сетям. Также **WebSocket** позволяет передавать данные без **HTTP**-заголовка в принципе.

## Практика

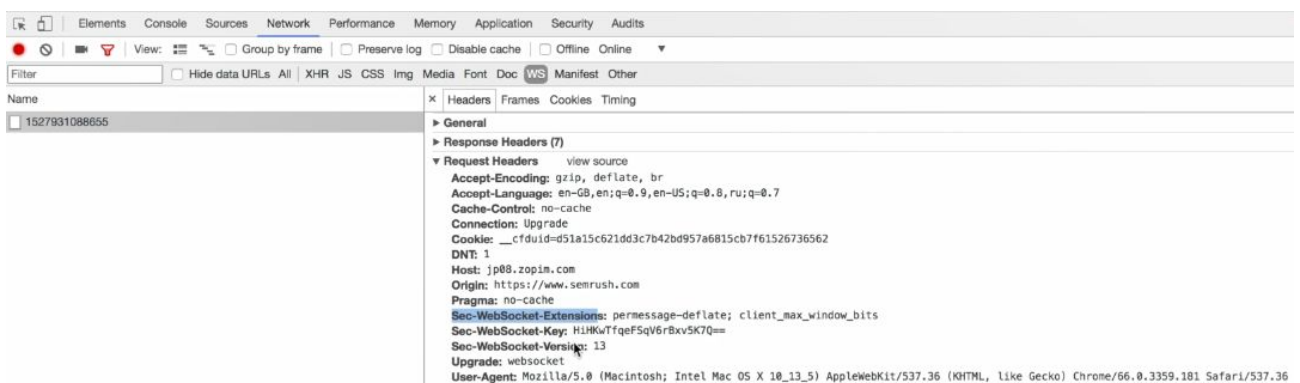
Разберёмся на примере, как **WebSocket** работает на уровне приложения, и какие уязвимости могут возникнуть. Чаще всего компании используют готовую библиотеку с **WebSocket**. Её писали сотни программистов, много раз тестировали на безопасность, поэтому больше вероятность, что там не будет уязвимостей на низком уровне.

Откроем **Google Chrome**, зайдём в инструменты разработчика и вкладку **Network**, затем на сайт (абсолютно любой), где есть **WebSocket**:



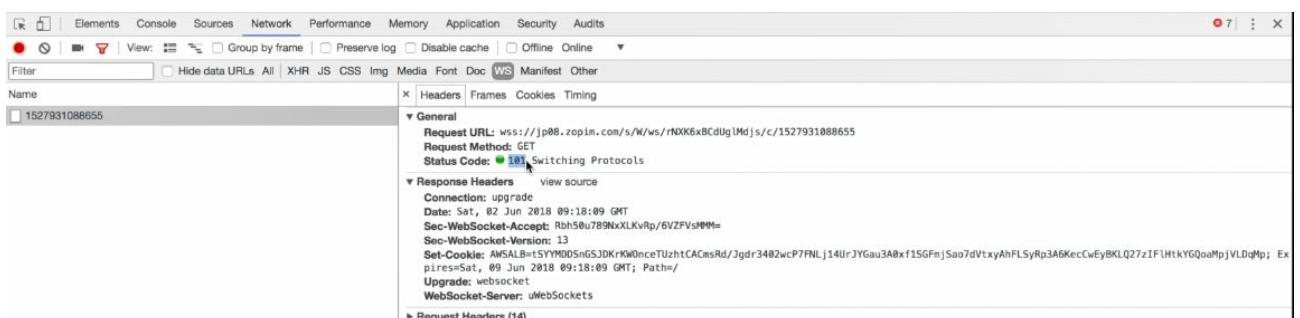
Видим, что есть специальный фильтр, чтобы показать **WebSocket** — **WS**. Тут есть запрос на открытие **WebSocket** — сначала посмотрим на заголовки.

При установке **WebSocket**-соединения на прикладном уровне самый первый запрос клиента — запрос на установление **WebSocket**. Это обычный **HTTP**-запрос, но здесь есть несколько необычных заголовков — например **Upgrade: websocket**:



Это сигнал для сервера, что мы хотим переключиться на протокол **WebSocket**. Здесь для нас важны **origin**, все **WebSocket**-заголовки и заголовок **Upgrade: WebSocket**.

Посмотрим на ответ:



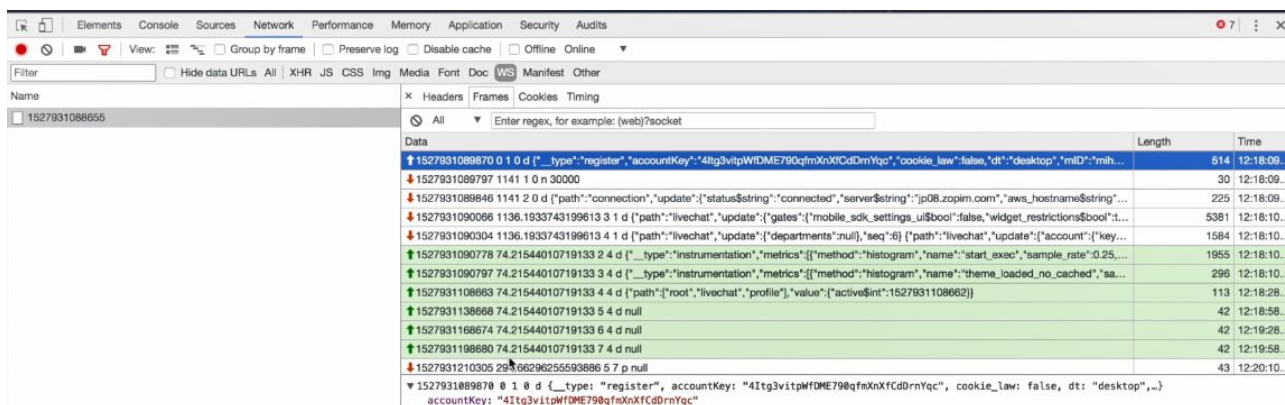
В ответе стоит необычный **Status Code=101**.

Это тип информационного кода 100, который означает смену протокола. **Switching Protocols** указывает на смену протокола; это бывает редко. В заголовках ответа есть все нужные нам заголовки

**WebSocket**, а значит, сервер согласился на **Upgrade**, т. к. именно он сообщил, что меняем протоколы.

После этого по **HTTP** клиент и сервер могут не общаться: открыт **WebSocket** и дальше они могут общаться фреймами (в **WebSocket** сообщения называются фреймами). Мы можем посмотреть сообщение в **Google Chrome** или в **Burp Suite**. В последнем можно менять эти сообщения, т. е. **Interception**. Единственное, что **Burp** не умеет относительно **WebSocket** — повторять сообщение, то есть мы не можем отправить в **Repeater** или в **Intruder** сообщение, но это умеет **OWASP ZAP** (OWASP Zed Attack Project).

Вернемся к фреймам — мы видим все, что отправляет **WebSocket** клиенту и все, что он принимает:



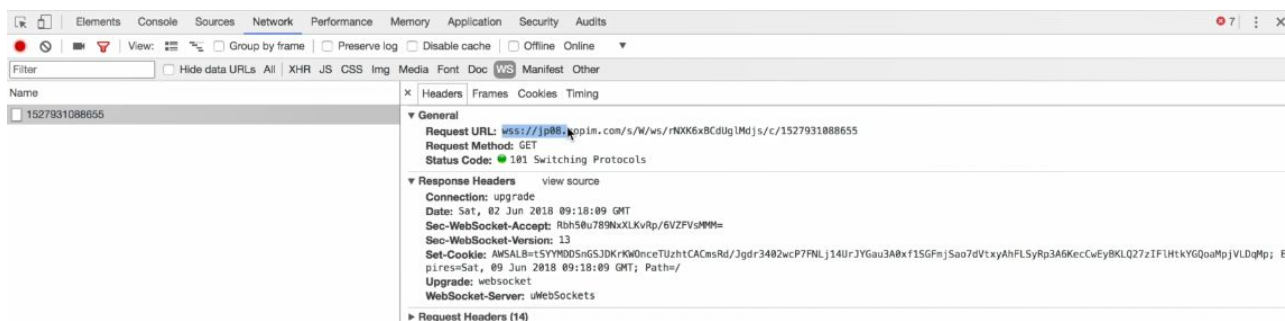
Name	Data	Length	Time
1527931088655	↑ 1527931088670 0 1 0 d {"_type":"register","accountKey":"4Itg3vitpWfDME790qfmXnXfCdDmYqc","cookie_law":false,"dt":"desktop","mID":"mih..."}	514	12:18:09...
	↓ 1527931088797 1141 1 0 n 30000	30	12:18:09...
	↓ 1527931088846 1141 2 0 d {"path":"connection","update":{"status\$string":"connected","server\$string":"jp08.zopim.com","aws_hostname\$string"...}}	225	12:18:09...
	↓ 1527931090066 1136.1933743199613 3 1 d {"path":"livechat","update":{"gates":{"mobile_sdk_settings_ui\$bool":false,"widget_restrictions\$bool":t...	5381	12:18:10...
	↓ 1527931090304 1136.1933743199613 2 4 d {"path":"livechat","update":{"departments":null},"seq":6} {"path":"livechat","update":{"account":{"key..."}	1584	12:18:10...
	↑ 1527931090778 74.21544010719133 2 4 d {"_type":"instrumentation","metrics":{"method":"histogram","name":"start_exec","sample_rate":0.25...	1955	12:18:10...
	↑ 1527931090797 74.21544010719133 3 4 d {"_type":"instrumentation","metrics":{"method":"histogram","name":"theme_loaded_no_cached","sa...	296	12:18:10...
	↑ 1527931108663 74.21544010719133 4 4 d {"path":["root","livechat","profile"],"value":{"active\$int":1527931108662}}	113	12:18:28...
	↑ 1527931138668 74.21544010719133 5 4 d null	42	12:18:58...
	↑ 1527931168674 74.21544010719133 6 4 d null	42	12:19:28...
	↑ 1527931198680 74.21544010719133 7 4 d null	42	12:19:58...
	↓ 1527931210305 29.66296255593886 5 7 p null	43	12:20:10...

То, что обозначено зелёным — отправления клиента. То что начинается с красной стрелки на белом фоне — отправления сервера. Мы видим, что здесь находятся произвольные данные, так как мы можем в **WebSocket**-пакет засунуть все что угодно.

## Безопасность WebSocket

**WebSocket** не спасает от стандартных уязвимостей веба; кажется, что не должны возникать уязвимости вроде **XSS**, **SQL**-инъекций, **RCE**, — но на самом деле они могут появиться уже потому, что в **WebSocket** передаются произвольные данные. Их точно так же нужно эскейпить, фильтровать, обрабатывать и смотреть, чтобы в них не было ничего лишнего, как и в обычном HTTP-протоколе — это первый момент, относящийся к безопасности **WebSocket**.

Второй момент — как вы видите, в заголовках есть **Request URL**:



Name	Headers	Frames	Cookies	Timing
1527931088655	<div><div>General</div><div>Request URL: <a href="ws://jp08.zopim.com/s/W/ws/rnXK6xBcdigJhdjs/c/1527931088655">ws://jp08.zopim.com/s/W/ws/rnXK6xBcdigJhdjs/c/1527931088655</a></div><div>Request Method: GET</div><div>Status Code: 101 Switching Protocols</div><div>Response Headers</div><div>Connection: upgrade</div><div>Date: Sat, 02 Jun 2018 09:18:09 GMT</div><div>Sec-WebSocket-Accept: Rbh50u789NXLKvRp/6VZFVsMm=</div><div>Sec-WebSocket-Version: 13</div><div>Set-Cookie: AMSALBtSYMD0SnG5JDKrKw0nceTuzhtCACesRd/3gdr3402wcP7FNlj14UJ3YGau3A8xf15GfMjSao7dVtxyAhFLSyRp3A6KecCwEyBKLQ27zIF1HtkYQ0aMpjVLDqMp; Expires=Sat, 09 Jun 2018 09:18:09 GMT; Path=/</div><div>Upgrade: websocket</div><div>WebSocket-Server: uWebSockets</div><div>Request Headers (14)</div></div>			

Здесь протокол **wss://** значит **WebSocket Secure** — он шифрует данные **WebSocket**, как **HTTP** и **HTTPS**. Протокол **ws://** просто передает данные в открытом виде — любой, кто находится в той же сети, что клиент или жертва, сможет подслушать трафик и узнать, что передается в **WebSocket**. Если это чувствительные данные, например сообщение пользователя, пароль или токены, необходимо использовать **wss://**.

В принципе, лучше использовать защищенные каналы связи, потому что неизвестно, что придумает злоумышленник. Например, он сможет провести атаку **MITM** (Man-In-The-Middle или «Человек посередине»), то есть перехватить трафик клиента и отправить его на сервер. Старайтесь по возможности всегда использовать безопасные соединения — **HTTPS** или **WSS**, который защищает от **MITM**-атак.

В запросе на открытие **WebSocket** заголовок **origin** неспроста. Сервер должен проверять **origin** и открывать **WebSocket** только для легитимных клиентов.

Суммируем все по безопасности **WebSocket**:

- 1) Используем **WebSocket Secure**, то есть протокол **wss://**, а не **ws://**.
- 2) На принимающей стороне проверяем **origin**.
- 3) Не доверяем пользовательским данным, обрабатываем их так же, как и в обычных **HTTP**-запросах.
- 4) Если мы передаем внутри **WebSocket** чувствительные данные между сервером и клиентом, нужно делать собственную аутентификацию. Чаще всего используется **GET**-параметр: мы не полагаемся на **Cookie**, потому что они уходят с каждым запросом, что позволит злоумышленникам беспрепятственно открывать **WebSocket** со своего сайта и получать данные пользователей.

Подробнее об этом вы узнаете после того, как разберете работу атаки **CSRF**. Запомните, что для открытия **WebSocket** необходимо проверять аутентификацию пользователей — либо через **HTTP**-заголовок, токен в заголовке, либо через токен в **GET**-параметрах, то есть случайно сгенерированную строку, привязанную к конкретному пользователю и достаточно длинную, чтобы злоумышленник не смог ее угадать или подобрать.

## Итоги

Подведём итоги — в этом уроке вы узнали:

- 1) Что такое **WebSocket**, для чего он нужен.
- 2) Как **WebSocket** применяется, как он работает, в чем его отличие от **HTTP**.
- 3) Уязвимости, связанные с **WebSocket** на прикладном уровне.



# Видеоурок 4

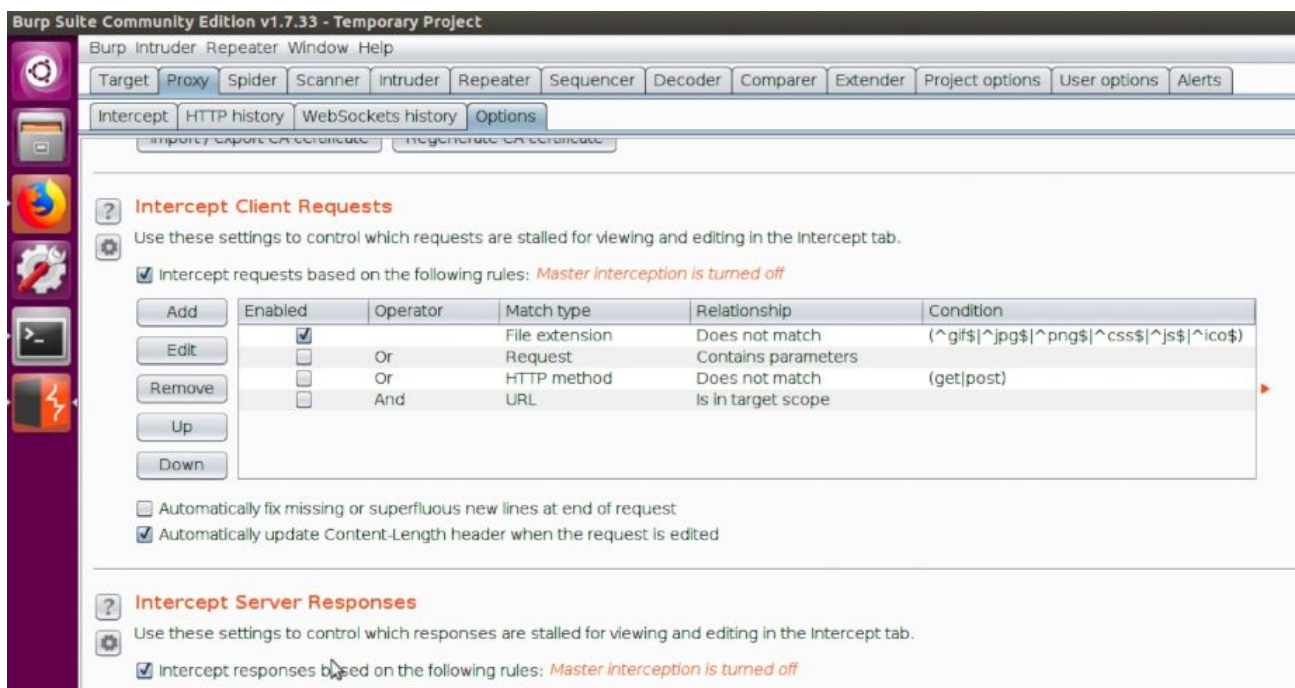
Тема урока — механизм распознавания содержимого страницы. План видео:

- 1) Что такое **Content Sniffing** и чем он опасен.
- 2) Как применяется заголовок **Content-Type**.
- 3) В чем может быть опасность, если мы определим заголовок **Content-Type** неверно.
- 4) Заголовок **Content-Disposition**.

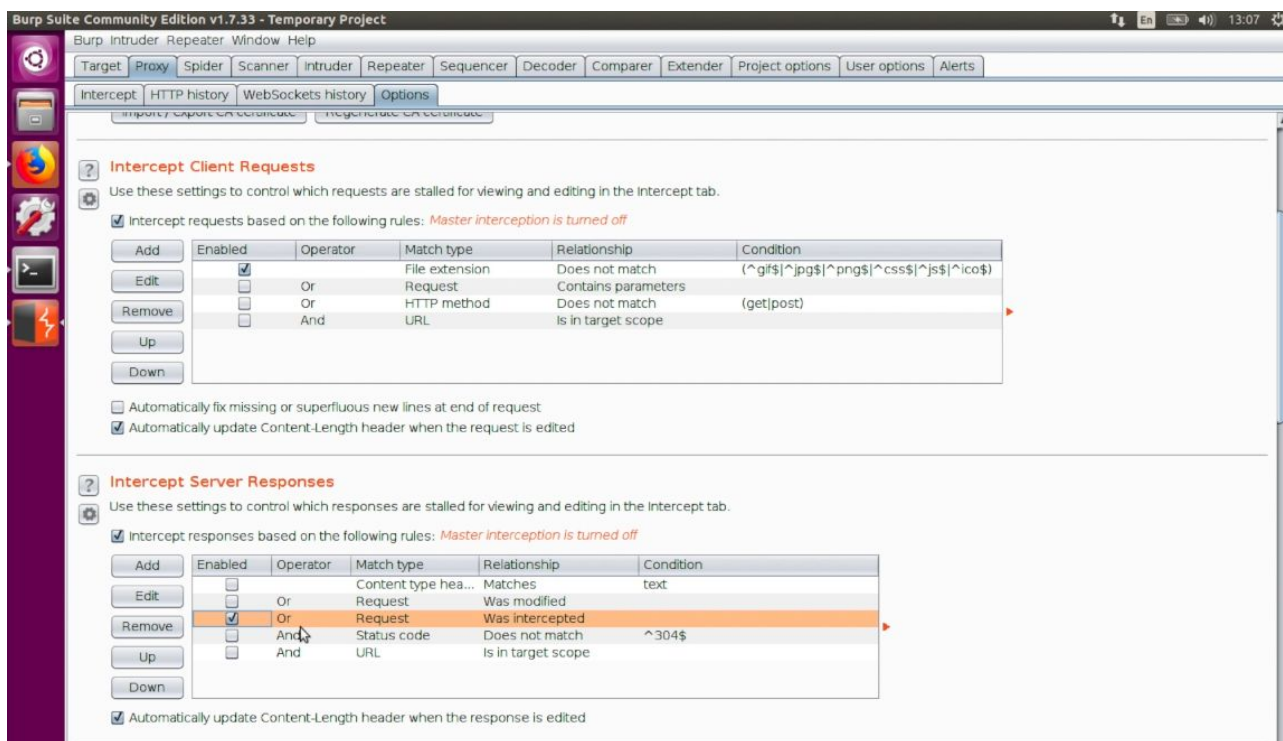
## Браузеры и Content Sniffing

Если сервер явно не указал тип содержимого страницы, браузер попытается угадать, что на ней расположено, и отобразить всё так, как будто там то, что он думает. Это называется **Content Sniffing**. Браузер смотрит внутрь документа, и если видит, что там находятся какие-либо теги, отображает это как **HTML**, а это неправильное поведение. Рассмотрим на примере.

Откроем **Burp**, зайдем в настройки и помимо **Intercept Client Request** — перехвата запросов от клиента, — включим **Intercept** ответов от сервера:

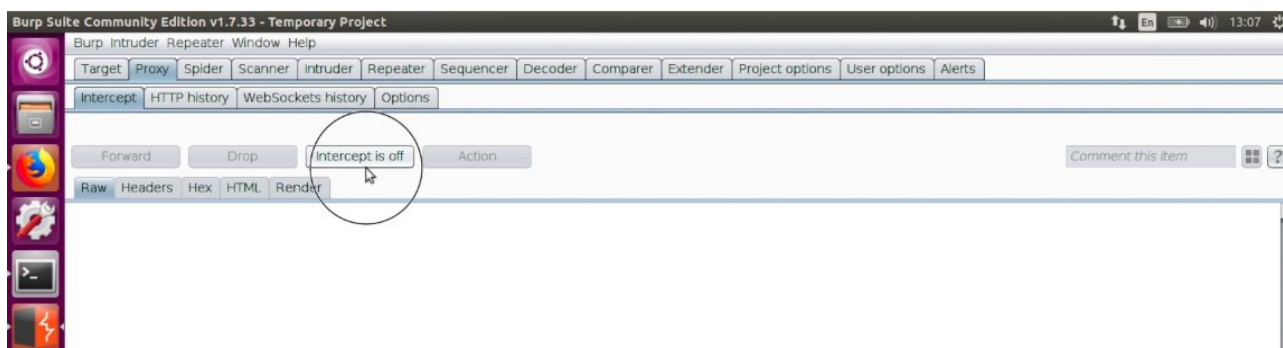


Нажмём галочку **Or Request Was intercepted**:



Это означает, что мы будем отслеживать все запросы, которые перехватили от клиентов, а также будем перехватывать ответы на них от сервера.

Далее включаем **Intercept is on**:

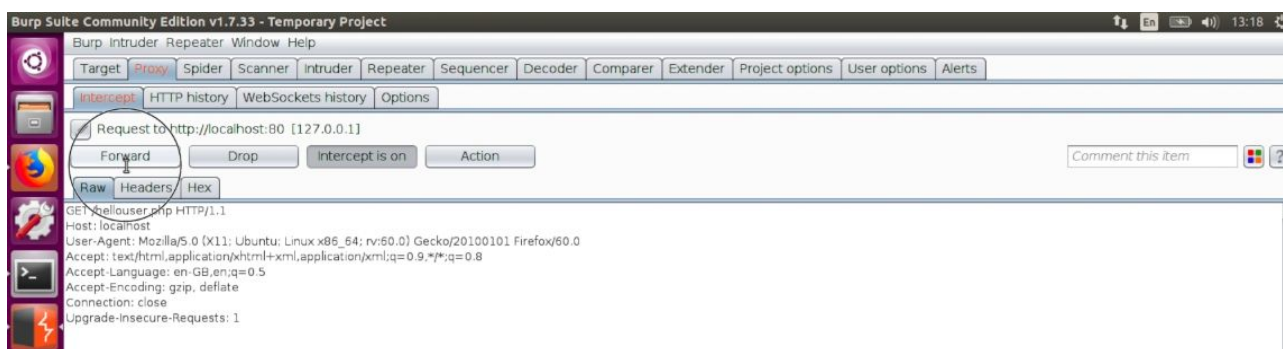


Зайдём на <http://localhost/hellouser.php>:

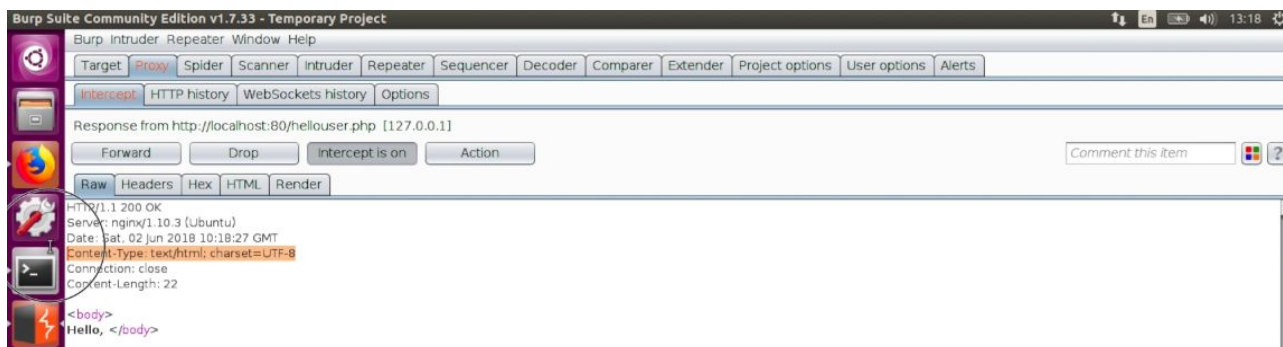




Как мы видим, **Burp** останавливает запрос. Переправим его:



Удалим заголовок ответа сервера **Content-Type** и посмотрим, что будет:



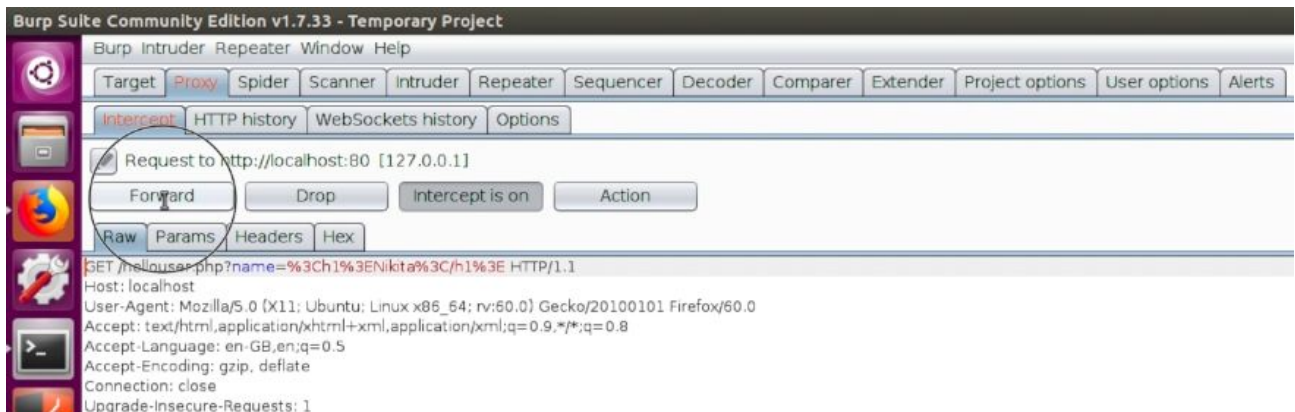
Всё отобразилось как **HTML**:



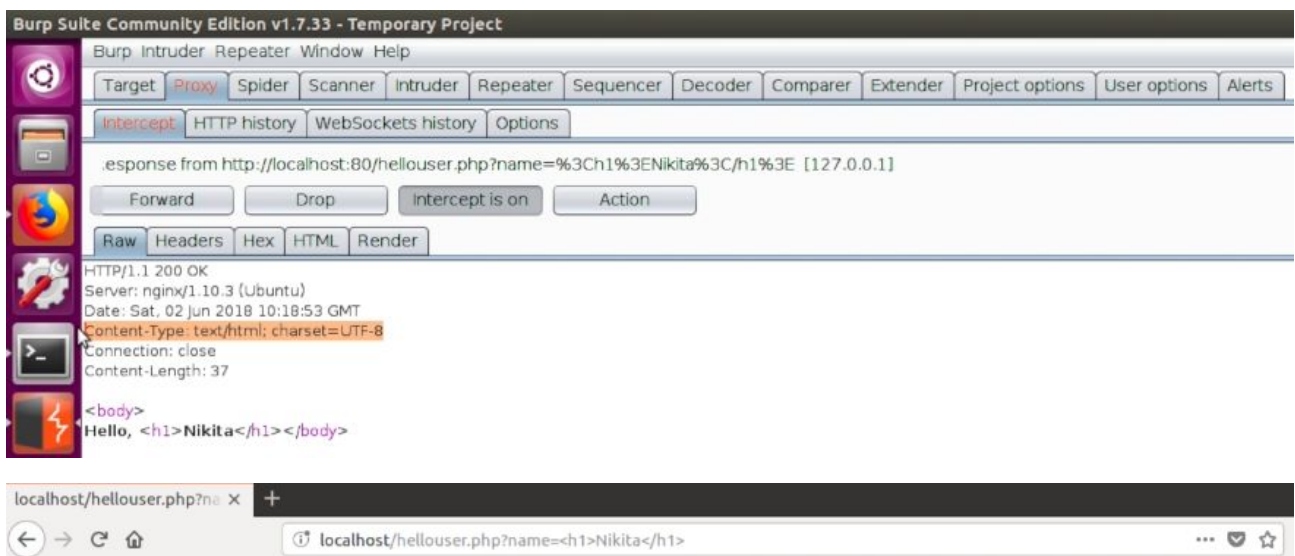
Мы убедимся в этом, если добавим **name** <http://localhost/hellouser.php?name=<h1>Nikita</h1>>:



Отправляем запрос:



Получаем ответ от сервера:



Браузер правильно распознал **HTML**. Это опасно: если злоумышленник сможет загрузить **HTML** там, где ожидается просто текст и мы явно не укажем, что это не код, браузер подумает, что это **HTML** и отрендерит его как код. И наоборот.

## Заголовок Content-Type

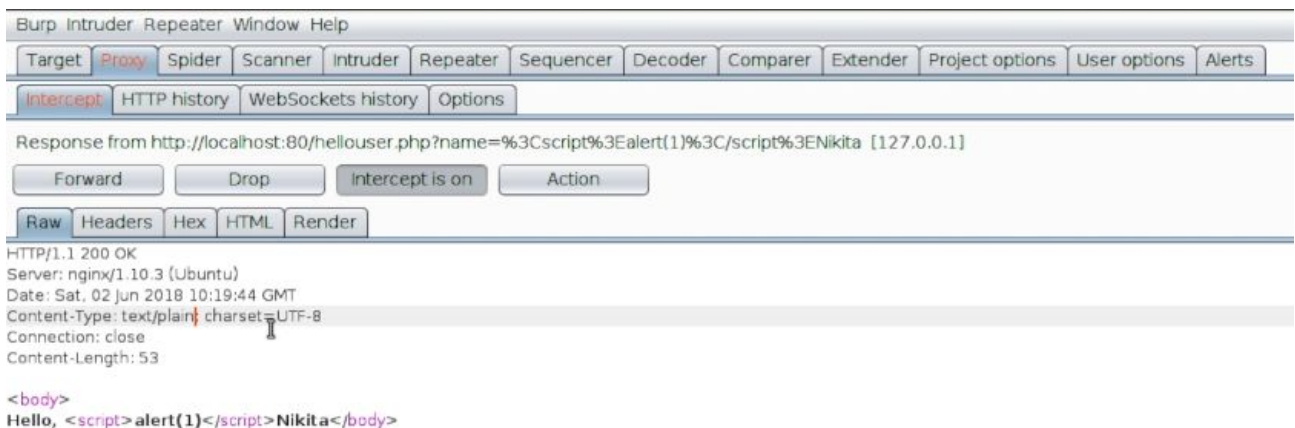
Попробуем сделать **alert()** с помощью inline-кода в **JavaScript**:



Переправим запрос:



Поменяем **text/html** на **text/plain** в ответе от сервера:



**MIME** тип **text/plain** означает, что в ответе обычный текст. Переправим этот ответ клиенту:



Всё отобразилось как текст: и **<body>**, и **<script>**, и **alert()**.

```
HTTP/1.1 200 OK
Server: nginx/1.13.10
Date: Wed, 09 May 2018 22:05:18 GMT
Content-Type: text/plain
Content-Length: 40
Last-Modified: Wed, 09 May 2018 22:04:55 GMT
Connection: close
ETag: "5af37087-28"
Accept-Ranges: bytes
```

```
<script>alert(document.domain)</script>
```

Если мы ставим заголовок **Content-Type**, браузер должен выбрать тип содержимого именно оттуда, но есть исключения, вроде старых версий **IE** (Internet Explorer). Они могут угадывать тип документа, который вернулся, несмотря на то, что в **Content-Type** явно указан, например **text/plain**. **Internet Explorer** старых версий всё равно включает **Content Sniffing** — пытается угадать, какой контент находится внутри. Он смотрит в документ, видит, что там находятся теги **<script>** и **alert()** и отображает это как **HTML** — а это неправильное поведение.

Такое поведение может быть совершенно неправильным, если мы в ответе от сервера вернем **JSON**, который не является **HTML**, но при этом в **Content-Type** укажем текст **HTML** — этим часто грешат разработчики и это может приводить к **XSS**.

```
HTTP/1.1 200 OK
Server: nginx/1.13.10
Date: Wed, 09 May 2018 22:08:21 GMT
Content-Type: text/html
Content-Length: 40
Last-Modified: Wed, 09 May 2018 22:04:55 GMT
Connection: close
ETag: "5af37087-28"
Accept-Ranges: bytes
```

```
{"first_name": "<script>alert(document.domain)</script>", "last_name": "test"}
```

Например, если сделать запрос имени и фамилии и в ответе вернется **JSON**, то, если в имени будет **<script>alert()</script>** и при этом в **Content-Type** указан **text/html**, скрипт сработает. Если **Content-Type** будет верный, то есть **application/json** — ничего не сработает, потому что **application/json** не выполняет **HTML**-код.

## Заголовок Content-Disposition

Чтобы браузер понял, нужно отобразить файл или дать пользователю скачать его, есть специальный заголовок **Content-Disposition**. Он предлагает открыть файл в браузере, будь то **HTML**-страница, обычный текст, **PDF** или другой формат, либо скачать его.

Если неправильно указать **Content-Disposition**, например, открывать файлы, которые должны быть скачаны, это может привести к уязвимости.

Откроем **Ubuntu** и разберем пример сайта, который обладает функционалом загрузки файла. Наверняка вы много раз видели подобные сайты, например можно загрузить аватарку, можно загрузить какой-то произвольный файл даже и разберем один из случаев когда можно загрузить произвольный файл.

Давайте посмотрим на код страницы:

```
cd /var/www/html && sudo nano upload.html

<form enctype="multipart/form-data" action="/upload.php" method="POST">
  Send file: <input name="userfile" type="file" />
  <input type="submit" value="Send File" />
</form>
```

Здесь простая форма **<form>**, есть поле **<input>** с типом **file** — значит, браузер сам создаст кнопку, на которую мы нажмем и выберем файл, который нужно загрузить; затем всё отправится на **upload.php**

Обратите внимание что тип кодирования — **multipart/form-data**: это обязательно нужно указывать, чтобы файл, который мы загрузим, отправился как нужно.

Посмотрим на файл **upload.php**:

```
sudo nano upload.php

<?php
$uploadaddir = '/var/www/html/uploads/';
$uploadfile = $uploadaddir . basename($_FILES['userfile']['name']);

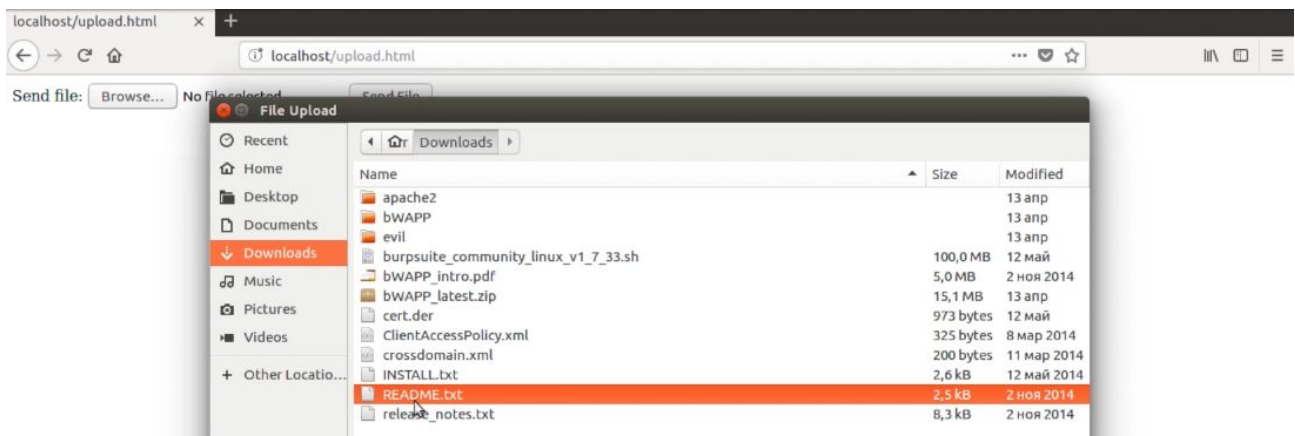
echo '<pre>';
if (move_uploaded_file($_FILES['userfile']['tmp_name'], $uploadfile)) {
    echo "File successfully uploaded!";
}
echo '</pre>';

?>
```

Выше написан **PHP**-код, задача которого — принять файл, который мы загружаем на стороне сервера, и переместить его в директорию **/var/www/html/uploads/** — в ней хранятся все файлы, которые мы загрузим.

Теперь попробуем загрузить — выберем файл и нажимаем **Send File**:





Как мы видим по сообщению, файл успешно загружен:



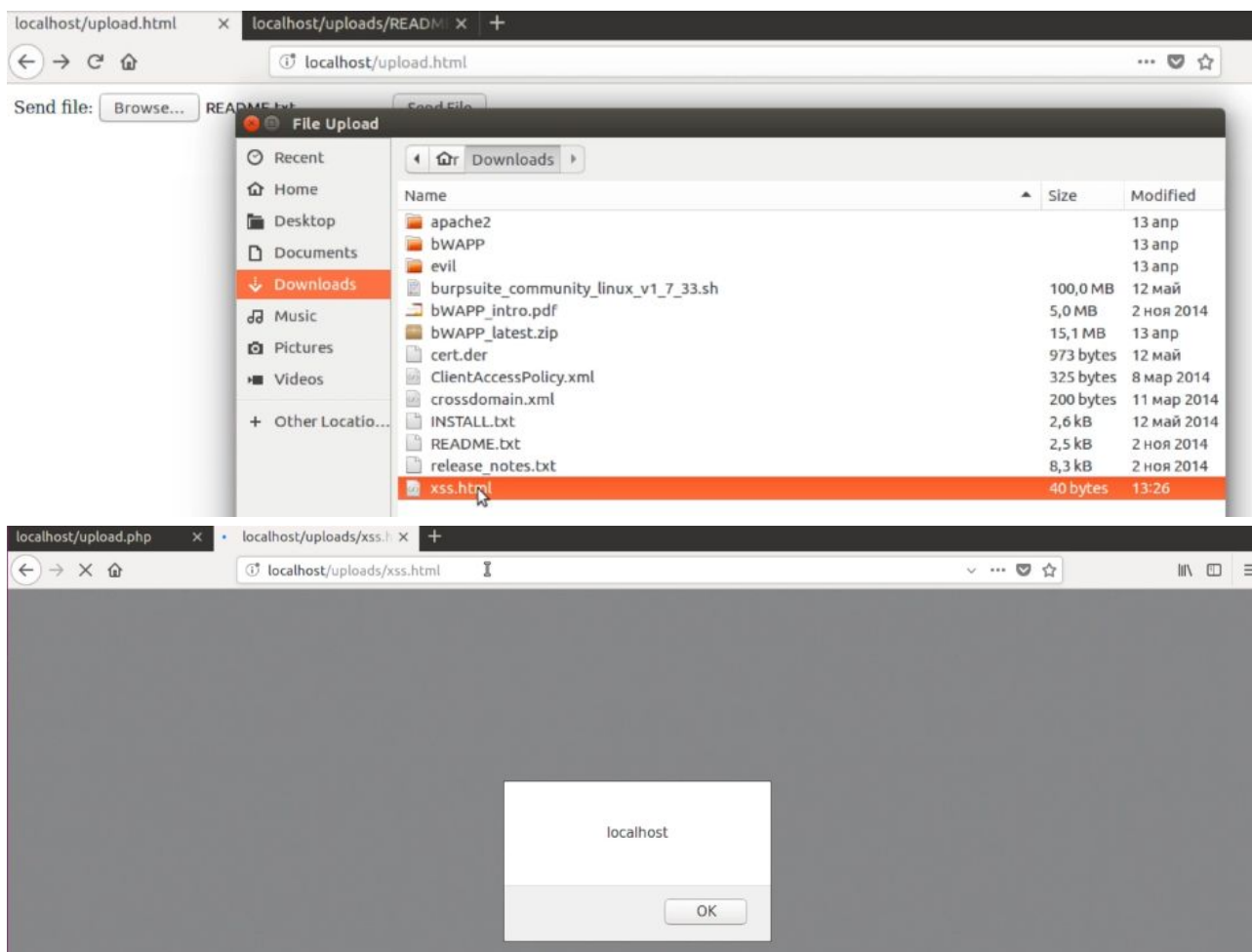
Попробуем открыть этот файл:



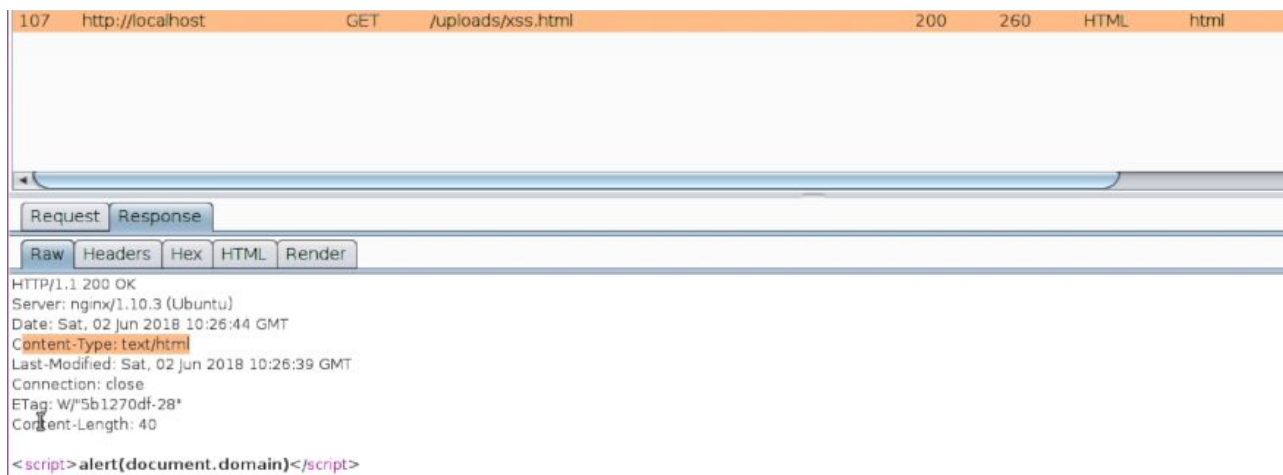
Как мы видим, файл успешно открывается.

Рассмотрим, что будет, если загрузить не обычный текстовый файл, а файл с JavaScript:





Этот файл приводит к **XSS**. Откроем **Burp** и посмотрим на последний запрос:



Это произошло потому, что **Content-Type** равен **text/html**. Внутри действительно **HTML**, мы изначально файл таким создали и загрузили — в нём находится вредоносный **JavaScript**-код.

Это значит, что, если злоумышленник будет давать ссылку на файл, а пользователи — переходить по ней, у них в браузерах возникнет **XSS**-уязвимость.

Когда разрешена произвольная загрузка файла, необходимо отдавать его как **attachment** — то есть то, что должно скачаться на компьютер, а не открыться в браузере. Для этого мы должны добавить заголовки **Content-Disposition** и указать тип **attachment** — это можно сделать через **nginx**.

Откроем конфиг **nginx** и добавим отдельное расположение для директории **uploads**:

```
cd /etc/nginx/sites-enabled && sudo nano default

root /var/www/html;

# Add index.php to the list if you are using PHP
index index.html index.htm index.nginx-debian.html;

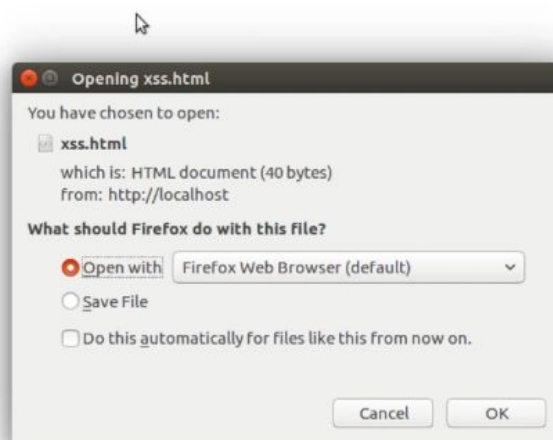
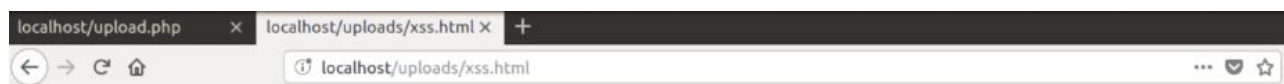
server_name _;

location / {
    # First attempt to serve request as file, then
    # as directory, then fall back to displaying a 404.
    try_files $uri $uri/ =404;
}

location /uploads/ {
    add_header "Content-Disposition" "attachment";
    try_files $uri $uri/ =404;
}
```

Здесь мы добавили заголовок **Content-Disposition** и указали тип **attachment**.

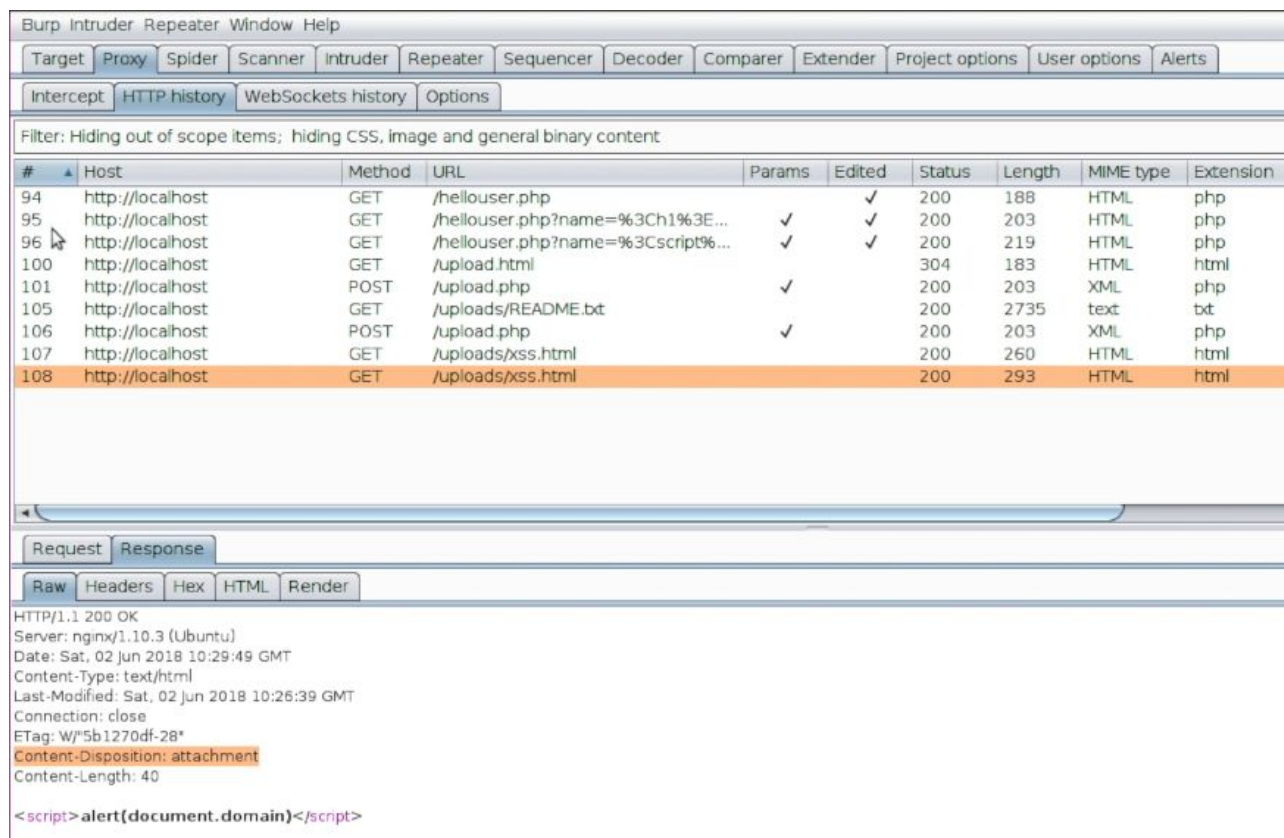
Теперь попробуем опять открыть этот файл:





Теперь браузер предлагает скачать файл, и после этого уязвимость **XSS** уже не срабатывает — это один из вариантов решения проблемы с **XSS**, **RCE** и другими уязвимостями, которые возникают при разрешенной произвольной загрузке файлов.

Откроем запрос и посмотрим ответ в **Burp**:



The screenshot shows the Burp Suite interface. The top menu bar includes: Burp, Intruder, Repeater, Window, Help. Below it are tabs for Target, Proxy, Spider, Scanner, Intruder, Repeater, Sequencer, Decoder, Comparer, Extender, Project options, User options, Alerts. Under the Proxy tab, there are sub-tabs: Intercept, HTTP history, WebSockets history, Options. A filter is applied: "Filter: Hiding out of scope items; hiding CSS, image and general binary content".

#	Host	Method	URL	Params	Edited	Status	Length	MIME type	Extension
94	http://localhost	GET	/hellouser.php		✓	200	188	HTML	php
95	http://localhost	GET	/hellouser.php?name=%3Ch1%3E...	✓	✓	200	203	HTML	php
96	http://localhost	GET	/hellouser.php?name=%3Cscript%...	✓	✓	200	219	HTML	php
100	http://localhost	GET	/upload.html			304	183	HTML	html
101	http://localhost	POST	/upload.php	✓		200	203	XML	php
105	http://localhost	GET	/uploads/README.txt			200	2735	text	txt
106	http://localhost	POST	/upload.php	✓		200	203	XML	php
107	http://localhost	GET	/uploads/xss.html			200	260	HTML	html
108	http://localhost	GET	/uploads/xss.html			200	293	HTML	html

Below the table, the "Request" and "Response" tabs are visible. The "Response" tab is selected, showing the raw response data:

```
HTTP/1.1 200 OK
Server: nginx/1.10.3 (Ubuntu)
Date: Sat, 02 Jun 2018 10:29:49 GMT
Content-Type: text/html
Last-Modified: Sat, 02 Jun 2018 10:26:39 GMT
Connection: close
ETag: W/"5b1270df-28"
Content-Disposition: attachment
Content-Length: 40

<script>alert(document.domain)</script>
```

Увидим, что заголовок **Content-Disposition: attachment** и браузер предлагает нам этот файл скачать, а не открывать — уязвимости нет.

## Итоги

Подведём итоги:

- 1) Узнали, что такое **Content Sniffing** и чем этот механизм опасен.
- 2) Посмотрели, какие виды заголовка **Content-Type** бывают и как они применяются.
- 3) Поняли, что очень важно проставлять заголовок **Content-Type** правильно. Иначе, если отдавать **JSON** и при этом проставлять **Content-Type: text/html**, то, когда в **JSON** есть **HTML**-код, возникает **XSS**-уязвимость. Нужно четко понимать, что мы возвращаем в ответе от сервера, и указывать именно такой **Content-Type**.
- 4) Посмотрели, зачем нужен заголовок **Content-Disposition** и какие уязвимости при загрузке файлов на сервер он позволяет предотвратить. Если не проставить его при загрузке **JavaScript**-кода на сервер, файл будет открываться как любая другая валидная

**HTML**-страница. Если в коде этой страницы будет содержаться вредоносный код, пострадают пользователи сайта. Но если проставить заголовок **Content-Disposition: attachment**, файл скачается и никому не навредит.

## Видеоурок 5

План урока:

- 1) Обзор нескольких наиболее опасных уязвимостей веба.
- 2) Перспективы: чему учиться дальше и что нужно делать, чтобы стать крутым хакером и ценным специалистом в области IT-безопасности.

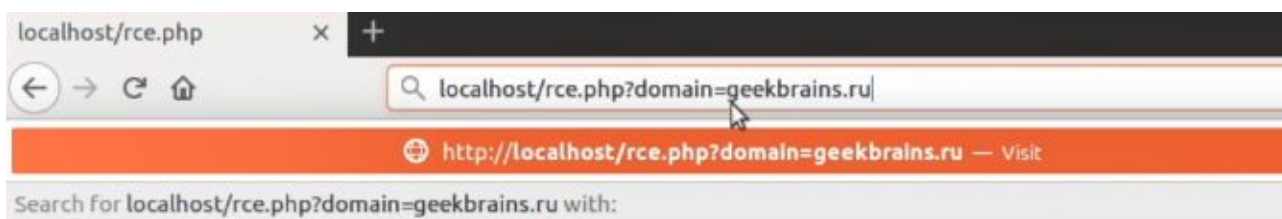
### Remote Code Execution (RCE)

Удаленное выполнение кода, или **RCE** — самая серьезная уязвимость, которая может возникнуть, потому что она позволяет злоумышленнику выполнить произвольный код на сервере. В этом случае злоумышленник получит полный контроль над сервером: сможет читать содержимое баз данных, делать запросы с этого сервера, устанавливать на него вредоносные программы, например так называемый бэкдор (**backdoor**), чтобы даже в случае, если уязвимость исправят, было бы возможно вернуться на сервер. Хакер сможет читать исходные коды программ, например коды разработки, и сделать с ними все что угодно.

Если указанный сервер в сети не один, то с него возможно получить доступ на другие и скомпрометировать абсолютно всю инфраструктуру компании.

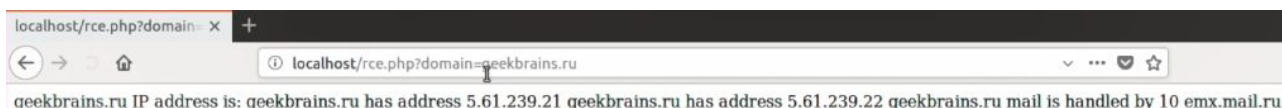
Существует много разных видов **RCE**.

Откроем **Firefox** и перейдем по адресу <http://localhost/rce.php>. Эта страница принимает на вход домен:

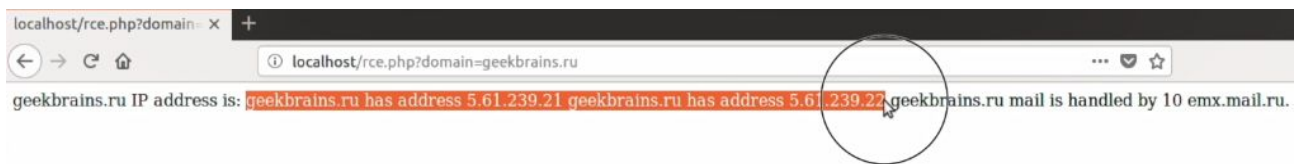


Укажем на вход <http://localhost/rce.php?domain=geekbrains.ru> и нажмём **Enter**.

Видим, как эта программа делает запрос на **geekbrains.ru**:



Она говорит, что у **geekbrains.ru** адрес **5.61.239.21** и **5.61.239.22**:



Но что эта программа делает на самом деле? Она выполняет команду **host** и выдает вывод обратно в браузер.

Добавим **ls /**, например: <http://localhost/rce.php?domain=geekbrains.ru;ls/>



Появился вывод команды **host**, затем идет листинг каталогов и файлов — все это список директорий, которые находятся в корневом каталоге.

На самом деле мы выполнили команду **bash** — поставили точку с запятой, после этого написали **ls /** и она выполнилась. Мы могли бы здесь написать **scp** — эта команда копирует файлы, в том числе **scp**, с одного компьютера на другой, или, наоборот, перенести файл с сервера злоумышленников на уязвимый сервер и залить бэкдор. Возможности ограничиваются только вашей фантазией.

Разберём подробнее код, который выполняется, чтобы понять, почему происходит **RCE**:

```
sudo nano upload.php

<?php
    $command = "host " . $_GET["domain"];
    echo $_GET["domain"] . "IP address is: " . shell_exec($command);
?>
```

Здесь есть функция **shell\_exec()** — эта **PHP**-функция делает следующее: принимает строку (в данном случае это строка **host**, соединенная с **GET**-параметром домена) и выполняет ее в **bash** — это очень мощная функция и особенно уязвимая, если в нее попадает пользовательский ввод. **PHP** сперва берёт значение параметра домена и дописывает ее в команду:

```
host geekbrains.ru; ls /

geekbrains.ru has address 5.61.239.22
geekbrains.ru has address 5.61.239.21
geekbrains.ru mail is handled by 10 emx.mail.ru.
bin  boot  dev  etc  home  init  lib  lib64  media  mnt  opt  proc  root  run
sbin  snap  srv  sys  tmp  usr  var
```

Понятно, почему здесь могут выполняться команды: мы отделяем предыдущую команду точкой с запятой и дальше пишем свои, которые хотим выполнить на сервере. Тот же самый вывод, только не отформатированный, вы видели в браузере.

Видов **RCE** великое множество, это отдельная большая тема, которую мы изучим дальше.

## SQL Injection

Также одна из критических уязвимостей — **SQL**-инъекция. Это внедрение произвольного SQL-кода в запрос к базе данных. С её помощью мы сможем получить данные из базы: так можно красть базы данных пользователей, письма, переписки, персональные данные (если в базе данных хранятся паспорта).

Это очень серьезная уязвимость, нацеленная на кражу пользовательских данных. В некоторых случаях **SQL**-инъекцию можно развернуть до пределов **RCE**, то есть с помощью **SQL**-инъекции выполнять произвольный код, но это не всегда возможно.

Подробнее о том, как это сделать, и в целом об этой уязвимости мы расскажем в следующих курсах.

## Buffer Overflow

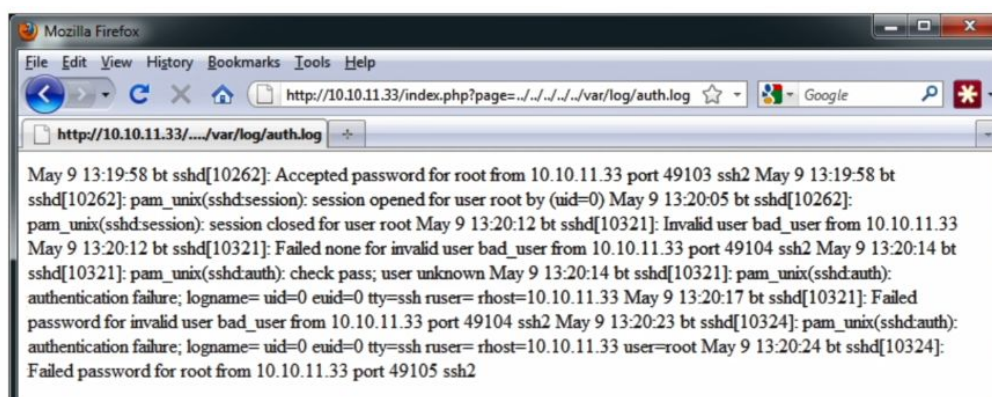
Инъекция кода возникает в приложениях, которые интерпретируются, а, в тех, что компилируются — нет инъекций, так как весь код, который должен быть в них выполнен, зашит на этапе компиляции.

Но зато в компилируемых приложениях есть атака на переполнение буфера, которая также приводит к **RCE**. Так что от **RCE** не защищены по умолчанию ни те и ни другие.

## Path Traversal

Есть такая уязвимость, как трассировка пути — она позволяет злоумышленнику выйти за пределы текущей директории.

Например, если сайт размещён в директории `/var/www/html`, то, используя символы `../` и так далее, мы будем подниматься выше по ней и сможем читать любые файлы с сервера:



Также можно прочитать, например, файл `/etc/shadow`, в котором находятся зашифрованные пароли пользователей, либо другие конфигурационные файлы, файлы с логинами и паролями от базы данных — их спектр очень велик и это тоже серьезная уязвимость.

## XSS Injection

Уже знакомый нам вид **XSS** — инъекция на **Client-Side** или, по-другому, на стороне клиента. С её помощью можно делать очень многое — все что позволяет **JavaScript**: менять **DOM**-дерево страницы, добавлять формы, прятать или показывать элементы, спрашивать логин и пароль у пользователя, украсть сессионный **Cookie**, делать скриншоты экрана, сделать клавиатурного шпиона или **KeyLogger**, ставить музыку.

## Итоги

Подведём итоги урока. В этом видео вы узнали:

- 1) Что такое **RCE**, посмотрели на пример **RCE**, разобрались, из-за чего они могут возникать.
- 2) Также познакомились с **SQLi**, или **SQL**-инъекциями.
- 3) Поговорили про бинарные уязвимости, например, атаку переполнения буфера — **Buffer Overflow**.
- 4) Увидели пример атаки **Path Traversal**.

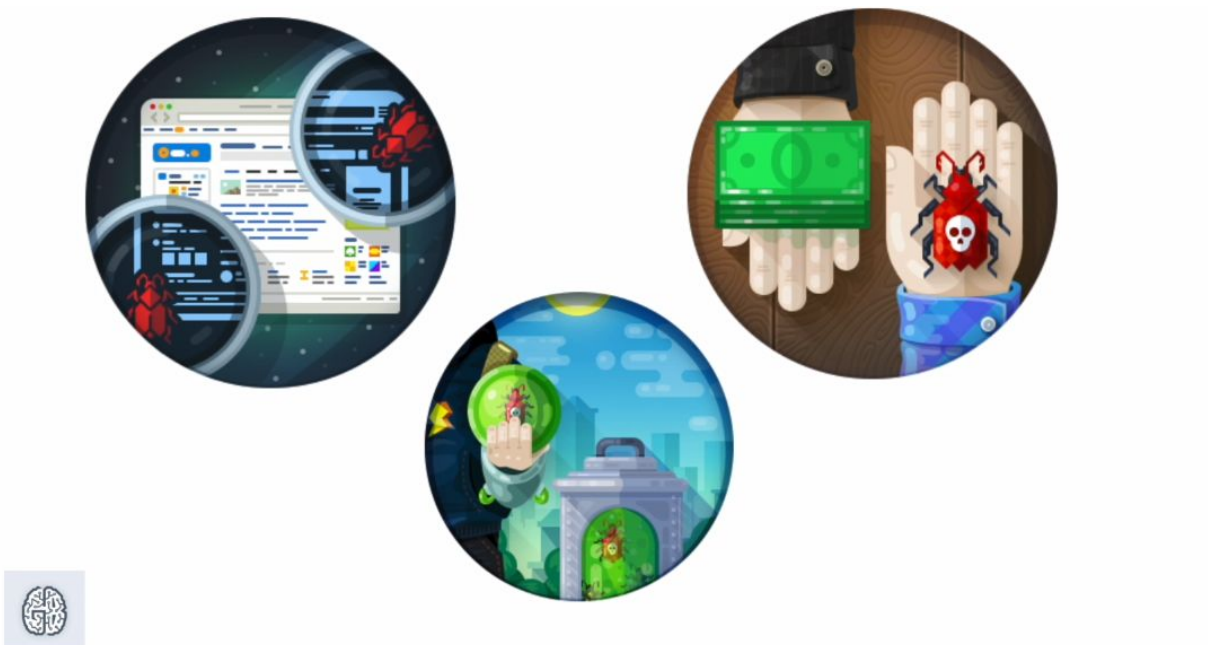
## Что дальше?

Классов уязвимостей гораздо больше, и о части из них вы узнаете в следующих курсах. Например:

- 1) Подробно познакомимся с уязвимостями клиентской части веб-приложений.
- 2) Изучим основы криптографии, рассмотрим основные криптографические алгоритмы.
- 3) Разберёмся с сетевой безопасностью.

## BugBounty

Хорошим подспорьем будет участие в **BugBounty**-программах, о них тоже подробно поговорим.



**BugBounty**-программы есть сейчас у всех крупных компаний. В общем они выглядят так: например, компания **Mail.ru** просит хакеров найти уязвимости, показать их и предоставить **Proof-of-Concept** (PoC), то есть шаги для повторения, чтобы можно было убедиться, что уязвимость существует. И если она достаточно критичная и входит в **Scope** компания за неё заплатит, а если не входит в **Scope**, то благодарность будет нематериальной.

В **BugBounty** много нюансов, но она позволяет хакерам найти уязвимости в крупных ресурсах, получить за это признание и заработать. А компании, в свою очередь, понимают, где в их процессах есть недостатки, получают возможность закрыть конкретные уязвимости и понять, что улучшить.

## Capture the Flag (CTF)

Также интересный опыт в безопасности вы можете получить, участвуя в **Capture the Flag** (CTF), то есть в соревнованиях по захвату флага.





Режим захвата флага — это просто. Есть две команды; в классическом варианте одна защищается, например у неё есть сайт, инфраструктура, несколько серверов и эти сервера изначально уязвимы. Вторая команда атакует этот сайт.

На сайте в разных местах — базах данных, исходных кодах, файлах спрятаны так называемые флаги — случайные строчки, которые атакующим нужно найти. Смысл игры в том, чтобы атакующие собрали как можно больше флагов: для этого нужно эксплуатировать уязвимости на сайте тех, кто защищается.

Защищающаяся сторона, в свою очередь, пытается отстоять флаги — ищет уязвимости, делает заплатки, закрывает дыры — это симуляция реальной работы отдела IT-безопасности в большой компании. Только в **CTF** это все происходит за гораздо меньший промежуток времени, и там все гораздо более динамично за счет того, что атакующие постоянно нападают и изначально уязвимостей на сайте довольно много.

## Конференции по ИБ

Также можно участвовать в различных конференциях по безопасности, готовить доклады, делать исследования и рассказывать об этом ИБ-комьюнити.



Можно приходить на конференции, чтобы получить новые знания, попробовать себя на практических воркшопах, освоить что-то новое для самостоятельных исследований — для этого совсем не обязательно выступать на конференции.

Ниже приведен список мероприятий по ИБ за 2019 год. [Полный список тут.](#)

## Мероприятия по информационной безопасности на 2019 год версия 1.3 (от 24.01.2019)

Мероприятие	Дата проведения	Место проведения	Сайт
Инфофорум	31 января - 1 февраля	Москва	<a href="http://www.infoforum.ru">www.infoforum.ru</a>
Код информационной безопасности	7 февраля	Уфа	<a href="http://conf.codeib.ru">conf.codeib.ru</a>
Актуальные вопросы защиты информации	13 февраля	Москва	<a href="http://www.tbforum.ru">www.tbforum.ru</a>
BIS Summit	13 февраля	Екатеринбург	<a href="http://bisummit.ru">bisummit.ru</a>
Код информационной безопасности	14 февраля	Воронеж	<a href="http://conf.codeib.ru">conf.codeib.ru</a>
E-Forensics	15 февраля	Москва	<a href="http://www.eforensics-conference.ru">www.eforensics-conference.ru</a>
Информационная безопасность финансовых организаций (Уральский форум)	18-22 февраля	Магнитогорск	<a href="http://ural.ib-bank.ru">ural.ib-bank.ru</a>
ИБ АСУ ТП КВО	27-28 февраля	Москва	<a href="http://www.ибкво.пф">www.ибкво.пф</a>
Код информационной безопасности	28 февраля	Ростов-на-Дону	<a href="http://conf.codeib.ru">conf.codeib.ru</a>
IDC IT Security Roadshow	14 марта	Москва	<a href="http://idc-cema.com">idc-cema.com</a>
RusCrypto 2019	19-22 марта	Москва	<a href="http://www.ruscrypto.ru">www.ruscrypto.ru</a>
Код ИБ. Профи	28-31 марта	Москва	<a href="http://msk.codeib.ru">msk.codeib.ru</a>
Код информационной безопасности	4 апреля	Новосибирск	<a href="http://conf.codeib.ru">conf.codeib.ru</a>
Код информационной безопасности	18 апреля	Краснодар	<a href="http://conf.codeib.ru">conf.codeib.ru</a>
Межотраслевой форум директоров по ИБ	22-23 апреля	Москва	<a href="http://infor-media.ru/events/43/838/">infor-media.ru/events/43/838/</a>
Код информационной безопасности	25 апреля	Санкт-Петербург	<a href="http://conf.codeib.ru">conf.codeib.ru</a>
Positive Hack Days (PHDays)	21-22 мая	Москва	<a href="http://www.phdays.ru">www.phdays.ru</a>



## Исследования

Можно выбрать тему в безопасности и прокачаться в ней, исследовав её вдоль и поперек. Исследовав тему, которую до вас никто не трогал, вы, возможно, найдёте новую уязвимость.

 **Mathy Vanhoef (vanhoefm)**

169

#286740

**Key Reinstallation Attacks: Breaking WPA2 by forcing nonce reuse**

Share:

State Resolved (Closed)

Severity Medium (6.8)

Disclosed publicly **November 3, 2017 3:37am +0300**

Participants 

Reported To **The Internet**

Visibility **Public (Full)**

CVE IDs  
CVE-2017-13077, CVE-2017-13078, CVE-2017-13079, CVE-2017-13080, CVE-2017-13081, CVE-2017-13082, CVE-2017-13084, CVE-2017-13086, CVE-2017-13087, CVE-2017-13088

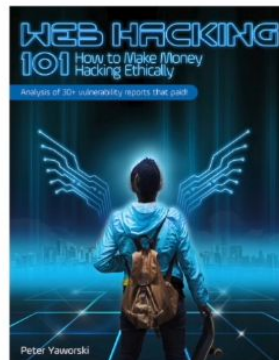
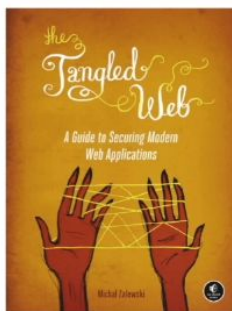
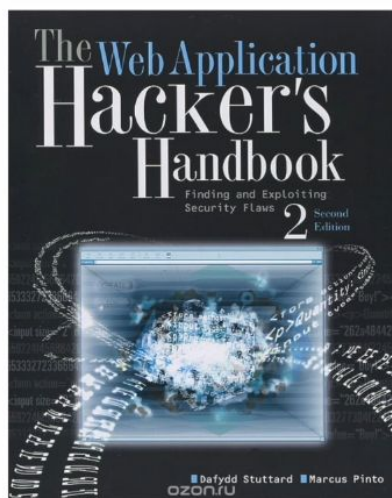
Weakness **Reusing a Nonce, Key Pair in Encryption**

Bounty **\$25,000**

А затем, если эта уязвимость существует и она широко распространена, пройдётесь по всем **BugBounty**-программам и везде соберёте кучу денег. Потом вы сможете написать по ней статью, выступить на конференциях с ней — в общем, здесь очень большой спектр того, что можно сделать.

## Читать и учить новое

Нужно постоянно читать статьи и книги на тему информационной безопасности, чтобы быть в курсе происходящего и не отставать от трендов.



Также можно читать открытые отчеты, так называемые **Bug-report** в программах по **BugBounty**.

Порой гораздо полезнее статей и книг может быть живое общение в кругу безопасников. В безопасности один человек, как правило, лучше разбирается в чем-то одном, а другой — в другом, и они друг с другом делятся знаниями. Если у вас есть друзья, коллеги, которые занимаются

безопасностью и хорошо разбираются в блоке определенных тем, вы можете в дружеской компании их обсудить.

## Ссылки к уроку

- 1) [Небезопасный cross-origin resource sharing.](#)
- 2) [\(Не\)безопасный frontend.](#)
- 3) [Кроссдоменный postMessage или как браузеры поддерживают стандарты.](#)
- 4) Безопасность HTML5: часть четвертая - <https://www.securitylab.ru/analytics/419889.php>
- 5) [OWASP Zed Attack Proxy Project.](#)
- 6) [Как использовать HTTP заголовки для предупреждения уязвимостей.](#)
- 7) [Статистика уязвимостей веб-приложений в 2018 году.](#)
- 8) [Типы уязвимостей сайтов.](#)
- 9) [BugBounty: заработай на чужих ошибках.](#)
- 10) [Всё о CTF в России.](#)
- 11) [Capture the Flag. Как взлом стал спортивным состязанием.](#)