GeekBrains

Курс «Веб-технологии: уязвимости и безопасность»

Браузеры и концепции безопасности

Политики безопасности браузеров: Same Origin Policy, Content Security Policy и другие стандарты безопасности браузеров

Оглавление

Введение

Урок будет посвящен браузерам, а именно концепциям безопасности, которые в них применяются.

Урок будет состоять из трёх пунктов:

1) Обзор Same Origin Policy (SOP).
2) Content Security Policy, для чего применяется.
3) Три разных поколения политик Content Security Policy: зачем каждая из них нужна, чем та или иная из них лучше или хуже, сравнение и применение.

К концу урока вы поймете, как устроены основные концепции безопасности браузера, почему они работают именно так, а не иначе, как они помогают защитить пользователей от взлома. Узнаете, как работают Same Origin Policy, Content Security Policy, какие виды политик CSP существуют и какую CSP нужно применять в каждом случае.

Видеоурок 1

Мы уже изучили основные технологии, которые применяются в вебе: посмотрели, как взаимодействуют клиент и сервер, подробно разобрали URL и его составные части, узнали, как устроен протокол HTTP, и научились им пользоваться. Также научились писать страницы на HTML, использовать стили CSS. В последнем уроке мы научились программировать на JavaScript и взаимодействовать с HTML-страницей и DOM-деревом.

Теперь настало время разобраться в концепции безопасности браузера: как в нем защищено то, что делают разработчики на основе HTTP, HTML, JavaScript и других протоколов и технологий.

Same Origin Policy

Представим, что JavaScript имеет доступ к документам, Cookie, HTML-страницам, сохраненным данным, кешу — то есть ко всему содержимому браузера. Например, вы зашли на https://geekbrains.ru и JavaScript сможет забрать Cookie с любого другого сайта — https://vk.com или https://mail.ru. Конечно GeekBrains не станет писать JavaScript-код, который смотрел бы в данные пользователя, которые не относятся к GeekBrains. Но существуют злоумышленники, которые хотели бы создать сайт, который соберет всю информацию о вас, любые авторизационные данные, чтобы использовать это в своих целях.

Представьте, что, чтобы быть взломанным в браузере, вам достаточно зайти на сайт. Это в целом реально, учитывая punny-код и другие подобные механизмы введения пользователя в заблуждение, фишинг-атак и приманок. Вспомните пример с punny-код, который мы разбирали — мы меняли в адресе geekbrains латинские буквы на русские, и сайт указывал на сервер злоумышленника. Так злоумышленник сможет практически бесплатно переводить на свой домен других пользователей

через подобный URL. Если пользователь переходит по такой ссылке-обманке, он купился на фишинг (от fishing — рыбалка). Как рыбак забрасывает наживку, так и злоумышленник кидает ссылку и ждет, пока пользователь перейдет по ней и засветит Cookie или другую личную информацию.

Если бы для фишинга просто было сделать так, чтобы пользователь перешел по ссылке и скомпрометировал данные, которые находятся в браузере, никто бы не пользовался интернетом и не хранил бы там данные. Но в определенный момент люди подумали, что JavaScript может получать доступ в другие документы, и придумали политики безопасности.

Первая политика — Same Origin Policy, она запрещает JavaScript одного документа взаимодействие с содержимым другого. Мы сможем сделать XHR-запрос с домена geekbrains.ru на домен mail.ru, он даже прийдет на сервер mail.ru и сайт даже выдаст ответ. Но когда ответ придет в браузер, который использует Same Origin Policy или, коротко, SOP, он не даст прочитать этот ответ документу с другого домена или, точнее, с другим Origin, например, geekbrains.ru.

То же самое правило действует, когда JavaScript хочет, например, получить Cookie или какие-то другие данные с другого домена или, точнее, с другим Origin. Подробнее с концепцией Origin мы познакомимся, когда будем разбирать Same Origin Policy в деталях. Сейчас достаточно понять, что по Origin политика Same Origin Policy определяет можно ли JavaScript, XHR или чему-то ещё получить доступ к данным или нет. Поэтому она и называется Same Origin Policy — Политика (безопасности) Одинакового Происхождения. Если происхождение документа или текущей страницы одинаковое, доступ можно получить, а если неодинаковое — нельзя.

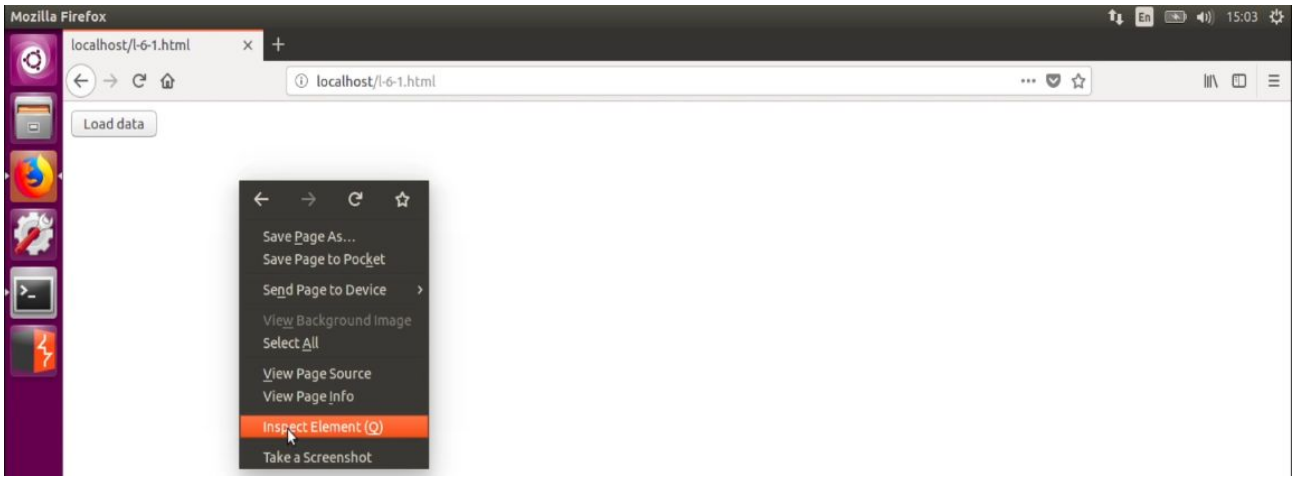
Практика

Посмотрим, как Same Origin Policy запретит XHR-запрос. Скопируем файл из предыдущего урока и используем его как основу:

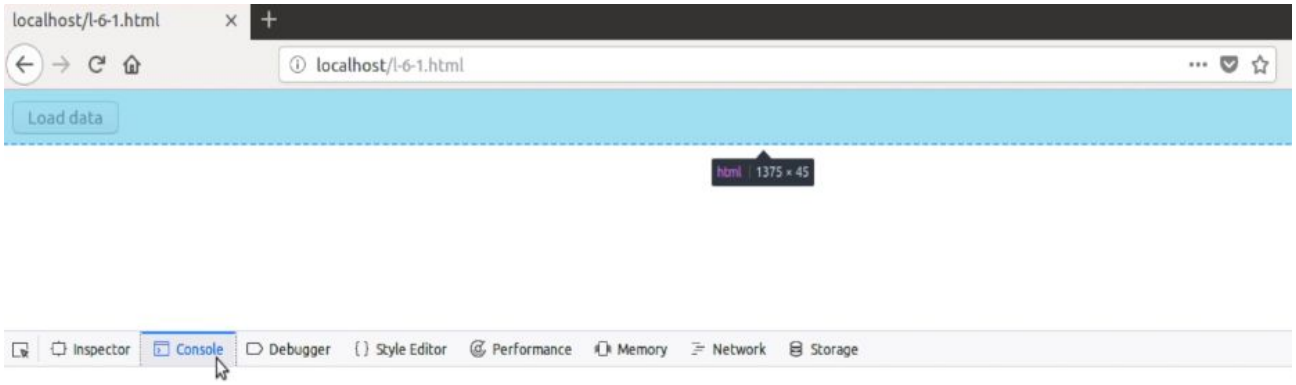
```
cp 1-5-4.html 1-6-1.html && nano 1-6-1.html
```

```
<script>
  function xhrTest() {
    var xhr = new XMLHttpRequest();
    xhr.open("GET", "http://victim.com/test.txt", false);
    xhr.send();

    if (xhr.status != 200) {
      alert(xhr.status + ': ' + xhr.statusText);
    } else {
      alert(xhr.responseText);
    }
  }
</script>
<button onclick="xhrTest()">Load data</button>
```

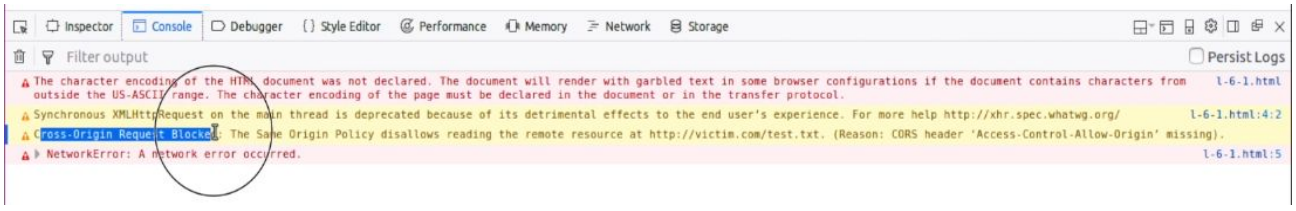
Попробуем сделать запрос на http://victim.com и посмотрим результат. Перезагрузим страницу: нажимаем на кнопку Load data, но ничего не происходит. Как мы знаем, в таких ситуациях нужно открывать инструменты разработчика, а именно консоль:



Нажимаем правую кнопку мыши, открываем инструменты разработчика и далее — консоль JavaScript:



Снова пробуем перезагрузить страницу — нажимаем Load data и видим ошибку Cross Origin Request blocked:



Cross-Origin значит, что с одного Origin был запрошен другой (именно другой, не тот же самый). Дальше идет подробное описание ошибки: она означает, что с домена localhost нельзя прочитать данные на victim.com, политика SOP это заблокировала. Дальше в скобках объясняется причина — отсутствует разрешающий такое поведение заголовок CORS. CORS 'Access-Control-Allow-Origin' нужен, чтобы ослабить некоторые моменты политики Same Origin Policy (именно ослабить, а не полностью отменить — для одних доменов разрешить, а для остальных запретить). И это один из правильных методов, как это сделать. Еще существует неправильный способ — метод jsonp(). Мы подробно разберем его позже.

Cross-Origin-взаимодействие распространяется не только на XHR-запросы, но и на Web Storage, Cookie и другие данные внутри страницы — в общем, на все данные в браузере.

Итоги

Подведем итоги — в этом видео вы узнали:

1) Что такое Same Origin Policy, для чего эта политика нужна и как она ограничивает взаимодействие и XHR.

2) Нашли, где в консоли JavaScript можно увидеть, почему Same Origin Policy блокирует данный запрос и понять, почему что-то не работает.

3) Разобрали, как Same Origin Policy помогает защитить данные пользователя в браузере и поняли, что если бы такой политики не было, браузеры были бы очень небезопасными по природе. Так как она есть и в большинстве браузеров реализуется корректно и практически одинаково, это позволяет нам работать в браузере безопасно.

Видеоурок 2

Поговорим про Content Security Policy. План видеоурока:

1) Что такое XSS и чем они опасны.

2) Что такое Content Security Policy (CSP), для чего она нужна и от чего защищает.

3) Примеры политик CSP — какие директивы CSP бывают, как их можно написать, примеры в браузере.

XSS

Мы уже видели примеры XSS или Cross-Site-Scripting ранее, они позволяют:

1. Работать с HTML-тегами: создавать новые, удалять существующие, менять стили элементов.

2. Реагировать на действия посетителя или пользователя, обрабатывать клики мыши. Каждый раз, как пользователь проявляет активность, мы отслеживаем ее и при желании можем отреагировать. Мы можем отслеживать перемещения курсора и даже нажатия на клавиатуру. Например, если пользователь будет набирать пароль или сообщение, мы сможем это отследить и даже отправить их себе.

3. Посылать запросы на сервер и читать от него ответы.

4. Получать и устанавливать Cookies. Например, на сайте на Cookie стоит флаг Http Only, то есть через JavaScript их не получить. Но злоумышленник совсем не обязательно будет пытаться напрямую украсть Cookies. То есть, если на сайте есть уязвимость XSS, даже при его защите на уровне SOP, злоумышленник всё равно сможет украсть Cookie, запросив

необходимые данные с помощью XHR, прочитав их и даже отправив себе на сервер. Конечно, это будет дольше и может быть не так незаметно. Но, тем не менее, XSS может спокойно эксплуатироваться, даже если Cookies и данные вроде бы защищены.

В XSS есть куча других возможностей для эксплуатации и бонус — получение Cookies. Но это сейчас не очень актуально, потому что крупные сайты, как правило, запрещают к сессионным Cookies доступ из JavaScript.

Можно сделать скриншоты, включить в браузере музыку, перенаправить пользователя — все, что можно в JavaScript (это будет выполнено в браузере пользователя) можно сделать, когда возникает XSS-уязвимость.

XSS-уязвимость — возможность на легитимном сайте создать ссылку или сохранить JavaScript-код так, что на этом сайте он потом выполнится у пользователя в его браузере. Получается, что мы сможем выполнить произвольный JavaScript-код на легитимном сайте и таким образом Same Origin Policy не сработает.

Это происходит потому, что JavaScript, который срабатывает во время XSS, на самом деле работает в контексте того же Origin или легитимного домена. Поэтому у него будет доступ ко всем документам, данным, запросам и так далее. Это важно понять, чтобы оценить критичность XSS и осознать, что это серьезная Client-Side-уязвимость, то есть уязвимость на стороне браузера пользователя.

От XSS можно защититься эскейпингом некоторых критичных символов — после этого они уже не имеют того значения, которые делают их HTML-тегами, и отображаются обычным текстом. Но так как XSS возникает довольно много и порой в неожиданных местах, то все их не отследить. Тем не менее, это правильная техника и неотъемлемая часть обучения программистов правилам безопасного кода. Но это — лечение симптомов, а не самой болезни, и помимо эскейпинга символов, нужно что-то посерьезней.

CSP

Для этого придумали Content Security Policy (и сокращенно CSP) — это глубокая основательная защита от XSS-инъекций, как Same Origin Policy — защита от того, чтобы JavaScript не получал доступ к документам и данным других доменов. CSP позволяет предотвратить XSS, даже если уязвимость уже есть на сайте.

Важно помнить что CSP — не прямая защита от XSS. И если уязвимость уже есть, её все равно нужно исправлять эскейпингом. Защита с помощью CSP, как правило, помогает уменьшить общий вред от такой уязвимости.

Если политика CSP хорошо и добротно написана, она позволяет полностью избежать эксплуатации XSS. Тогда вредоносный JavaScript-код не выполнится и единственное, что можно будет сделать с уязвимостью, — вставить HTML-теги и изменить поведение страницы. Будет нельзя, например,

украсть Cookies, добавить или изменить элементы, реагировать на действия пользователя, считывать нажатые им клавиши или сделать скриншот экрана.

Хорошую политику CSP сделать очень сложно, и чем больше компания, тем сложнее: появляется много зависимостей, разных скриптов и других вещей, которые нужно учитывать. Одно зависит от другого и тяжело внедрить строгую политику. На небольших сайтах строгую политику обычно очень легко внедрить, это можно сделать буквально за день-два.

Практика

Посмотрим, как выглядит CSP в браузере, разберём несколько директив и поймём, как CSP блокирует исполнение JavaScript-кода.

Зайдём в конфиг **nginx**, воспользуемся уже известной нам директивой **add_header**:

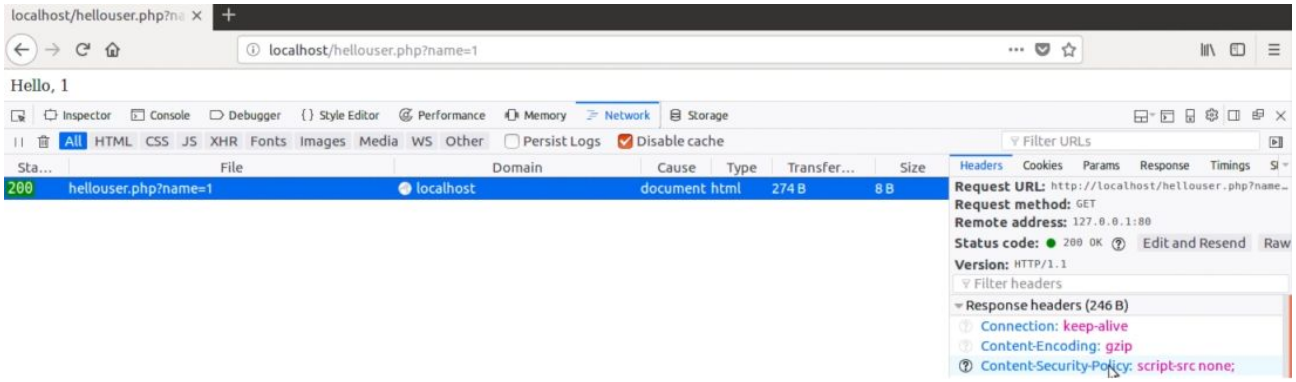
```
cd /etc/nginx/sites-enabled && sudo nano default
```

```
        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 Content-Security-Policy "script-src none;";
                try_files $uri $uri/ =404;
        }

        # pass PHP scripts to FastCGI server
        #
        #location ~ \.php$ {
        #       include snippets/fastcgi-php.conf;
        #
        #       # With php-fpm (or other unix sockets):
        #       fastcgi_pass unix:/var/run/php/php7.0-fpm.sock;
        #       # With php-cgi (or other tcp sockets):
        #       fastcgi_pass 127.0.0.1:9000;
        #}
```

Мы добавили заголовок **Content-Security-Policy "script-src none;"** после этого все сохраним, выходим и перезагружаем сервер:

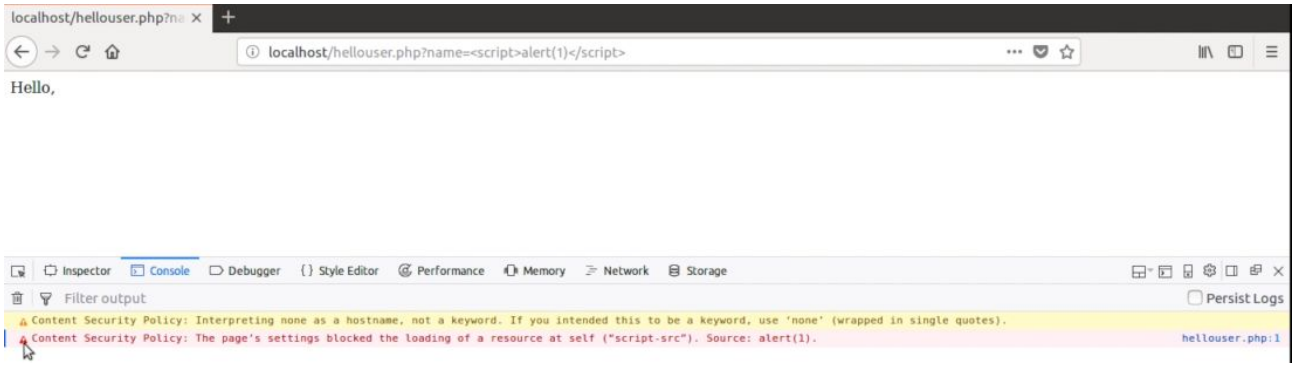
```
sudo nginx -s reload
```

Перезагрузим страницу и убедимся, что заголовок Content-Security-Policy действительно проставился:



Теперь у нас проставлено поле Content-Security-Policy.

Попробуем выполнить JavaScript-код:



JavaScript-код **<script>alert(1)</script>** не выполнился, несмотря на то, что мы не делали HTML-эскейпинг — алерт не отобразился.

Чтобы узнать, почему не работает JavaScript, необходимо зайти в консоль разработчика. Мы видим — здесь есть ошибка. Сообщение гласит, что content-security-policy заблокировала исполнение JavaScript, так как есть директива **script-src none**, которая значит, что скрипты на странице вообще нельзя выполнять. Это очень строгая политика и она редко применяется, потому что JavaScript сейчас везде.

Поменяем директиву, например так:

```
server_name _;

location / {
        # First attempt to serve request as file, then
        # as directory, then fall back to displaying a 404.
        add_header Content-Security-Policy "script-src unsafe-inline;";
        try_files $uri $uri/ =404;
}
```

Если мы сохраним конфиг, выйдем, перезагрузим веб-сервер Nginx и обновим страницу, то увидим, что JavaScript-код будет нормально выполняться. Это происходит потому, что unsafe-inline позволяет

выполнять inline-код на JavaScript. В HTML Inline — то, что определено прямо в HTML-коде. Скрипт **<script>alert(1)</script>**, который написан в самом HTML-коде, называется инлайн-кодом, то есть кодом, записанным прямо в строке.

Также существует CSP-директива **script-src** — она разрешает загружать код с другого домена. Чтобы ограничить домены, с которых можно загружать JavaScript, мы добавляем их URL в политику CSP.

Директива **unsafe-inline** — плохая идея, потому что, когда она есть, любой инлайн-JavaScript-код может быть выполнен: мы запишем его в код и не будем загружать извне.

Допустим, в доверенных у нас будет http://localhost: если мы загрузим скрипт с localhost, то он выполнится, а с других ресурсов— нет:

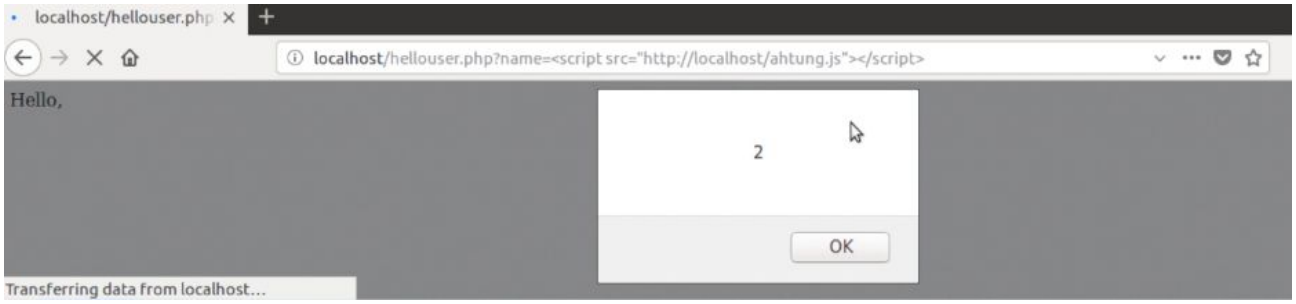
```
add_header Content-Security-Policy "script-src unsafe-inline http://localhost;";
```

Перезагрузим Nginx и создадим в локальной директории файл со скриптом:

```
cd /var/www/html && sudo nano ahtung.js
```

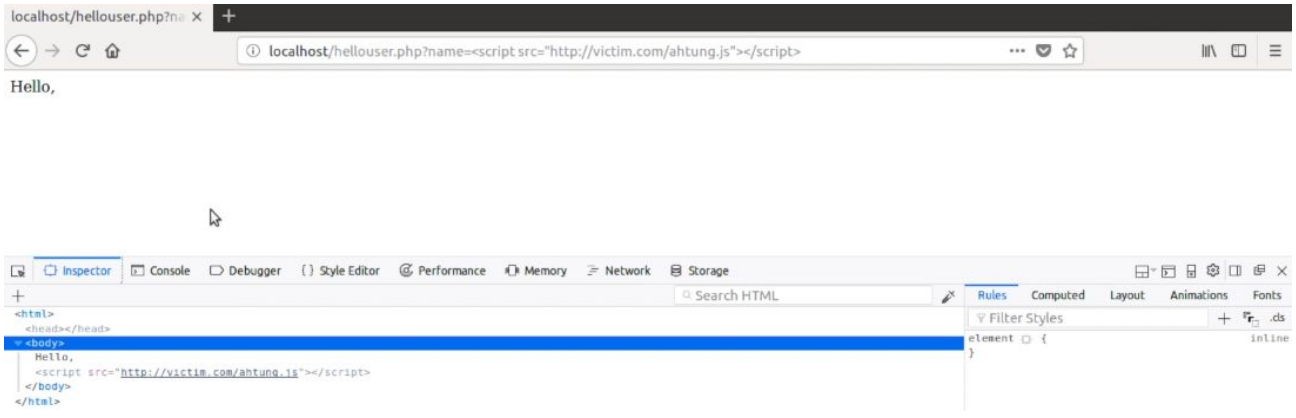
```
alert(2);
```

Зайдём в Firefox и напишем запрос http://localhost/hellouser.php?name=<script src="http://localhost/ahtung.js"></script>:



JavaScript-код успешно выполнился.

Попробуем подгрузить этот же самый скрипт, но уже с сайта http://victim.com:



Ничего не выполнилось, откроем консоль:

Мы увидим, что Content Security Policy заблокировал загрузку ресурса с сайта http://victim.com. Произошло это потому, что с помощью директивы CSP мы указали что с localhost можно загружать данные, а с victim.com — нельзя. Таким образом, действует правило, что запрещено все, что явно не разрешено. Нужно явно разрешить, откуда можно загружать скрипты.

Content Security Policy позволяет уменьшить вред от XSS, либо, если CSP очень хорошо настроен, исключить практически все XSS. CSP не защищает от XSS, которые возникают за счет того, что JavaScript-код плохо написан, то есть когда внедрение происходит внутри JavaScript-кода. Но это отдельный класс XSS — о них вы узнаете в курсе про Client-Side уязвимости. Важно понимать, что от большинства XSS строгая политика CSP спасет.

Итоги

Подведем итоги:

1) Вы углубили понимание XSS, узнали, какие опасности они представляют, поняли, что это серьёзная Client-Side-уязвимость.

2) Узнали про CSP, для чего она нужна и как защищает.

3) Увидели пример политик CSP, узнали про директивы default-src, script-src, поняли, как можно запрещать инлайн-скрипты и загрузку с других ресурсов. С помощью CSP можно запрещать или ограничивать загрузку картинок, стилей, шрифтов и так далее — директивы есть практически на каждый тип данных, который можно загрузить или вставить.

Видеоурок 3

Поговорим о трёх поколениях стандартов Content Security Policy. План урока:

1) Виды CSP.

2) Преимущества и недостатки политик CSP. Какую политику в каком случае нужно применять.

Три поколения CSP

Политику CSP можно описать тремя способами:

1. Whilelist-based.

2. Hash или Nonce-based.
3. Strict-Dynamic.

CSP v1: Whitelist

Первый метод, который мы рассмотрели, с белым списком доменов, так и называют — **Whitelist-based**. Когда мы все запрещаем по умолчанию и что-то разрешаем, это называется белый список (с английского — **Whitelist**). Когда мы все разрешаем и что-то запрещаем, это черный список (или **Blacklist**). Безопаснее делать **Whitelist**.

Иногда используют и **Blacklist**, потому что это может быть оправдано, проще сделать, или же белые списки использовать невозможно или очень сложно. Всегда, когда есть возможность, лучше использовать **Whitelist**.

Это было первое поколение CSP. Когда CSP только появилась, она была не такая разнообразная, как сейчас, в ней можно было лишь запрещать инлайн-скрипты, **eval** (это специальные функции в JavaScript-коде и о них мы узнаем позже), можно было добавлять белый список доменов — на этом тогда основывалась политика CSP.

На небольших сайтах белый список доменов в принципе хорош: там можно быстро перечислить все домены, с которых мы грузим. Но на крупных сайтах, где много что грузится с разных доменов, не всегда можно что-то запретить. Особенно это верно для порталов, где есть реклама: белый список доменов очень тяжело реализовать и для этого придумали директиву **Strict-Dynamic**.

CSP v2: Hash и Nonce

Прежде чем перейти к **Strict-Dynamic**, разберемся, зачем нужен **Nonce** или **Hash-based-**политика. Второе поколение политики CSP ввело такое понятие, как **Nonce-based**.

В JavaScript-коде есть инлайн-скрипты и разработчикам они очень нравятся, потому что их можно встроить на страницу и очень быстро и удобно менять. Но сейчас нет особого смысла так делать, потому что есть разные сборщики и bundler, которые после разработки скрипта собирают все в один файл, который можно подтягивать из поля **script src**. В интернете есть обратная совместимость, как и в любой другой технологии — она должна быть, потому что иначе все, что было сделано до нее, сломается и нужно будет многое переделывать. Нельзя запретить весь inline, потому что множество сайтов перестанет работать.

Можно было бы перевести все инлайн-скрипты в не-инлайн, но это иногда бывает дорого и долго. Поэтому в CSP придумали **Nonce**.

Практика

Nonce в целом расшифровывается как Number used Once — то есть число, которое используется один раз, должно быть уникальным, применяться только один раз для каждого запроса.

Перейдём в пример, чтобы понять как работает **Nonce**. Откроем Ubuntu и поправим конфиг **nginx**.

Пожалуйста, обратите внимание — правки делаются уже в другом месте под секцией PHP. Убедитесь, что вы выполнили все шаги из первой методички, если нужно, пересмотрите видеоурок 2 — у вас должен быть настроен и корректно работать модуль php-fpm для 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.
                try_files $uri $uri/ =404;
        }

        # pass PHP scripts to FastCGI server
        #
        location ~ \.php$ {
                include snippets/fastcgi-php.conf;
        #
        #       # With php-cgi (or other tcp sockets):
        #       fastcgi_pass 127.0.0.1:9000;
        #       # With php-fpm :
                add_header Content-Security-Policy "script-src 'nonce-232323'
http://localhost;";
                fastcgi_pass unix:/var/run/php/php7.0-fpm.sock;

        #}
```

Nonce указывается в заголовке Content-Security-Policy. Мы добавили в конфигурационный файл директиву **nonce-232323**. Число 232323 здесь — пример, используем его, чтобы упрощенно показать, как это работает. На самом деле **Nonce** должен быть длинной строкой (256 бит) и число должно быть случайным: мы должны взять хороший генератор случайных чисел и получить оттуда эту строку (подробнее об этом вы узнаете в курсе криптографии). **Nonce** должен быть новым для каждого запроса (в данном случае у нас он будет одинаковый для всех запросов, это неправильно).

Сохраним конфиг и сразу перезагрузим **nginx**:

```
sudo nginx -s reload
```

Необходимо подписать с помощью **Nonce** все инлайн-скрипты:

```
cd /var/www/html && sudo nano hellouser.php
```

```
<body>
<?php
        echo 'Hello, ' . $_GET['name'];
?>
<script nonce='232323'>
    alert("Hi, mom!");
</script>
</body>
```

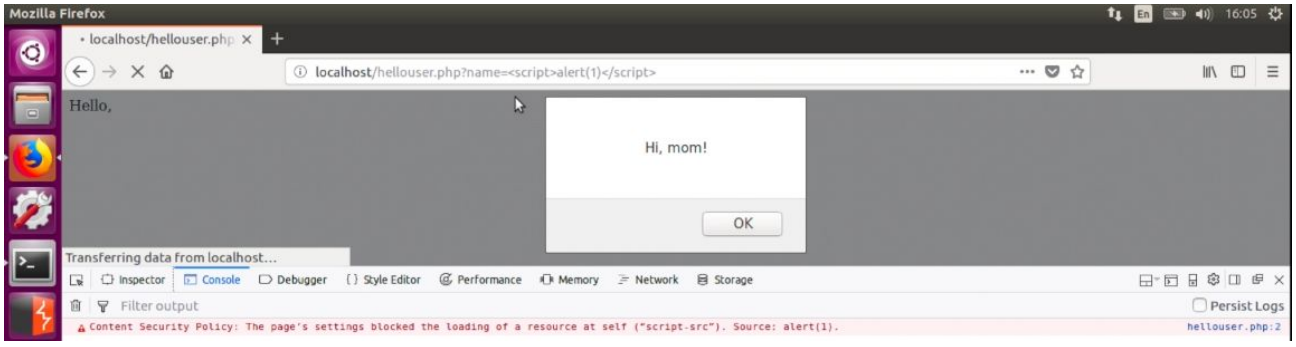
Мы добавили скрипт, который выводит **alert("Hi, mom!")** и атрибут **nonce='232323'**.

В жизни во все инлайн-скрипты проставляются уникальные значения **nonce,** это же значение будет в заголовке скрипта и его получит пользователь в ответ на запрос страницы. На каждый запрос страницы номер **nonce** будет новый — и для страницы, и для заголовков.

Дальше открываем браузер и перезагружаем:

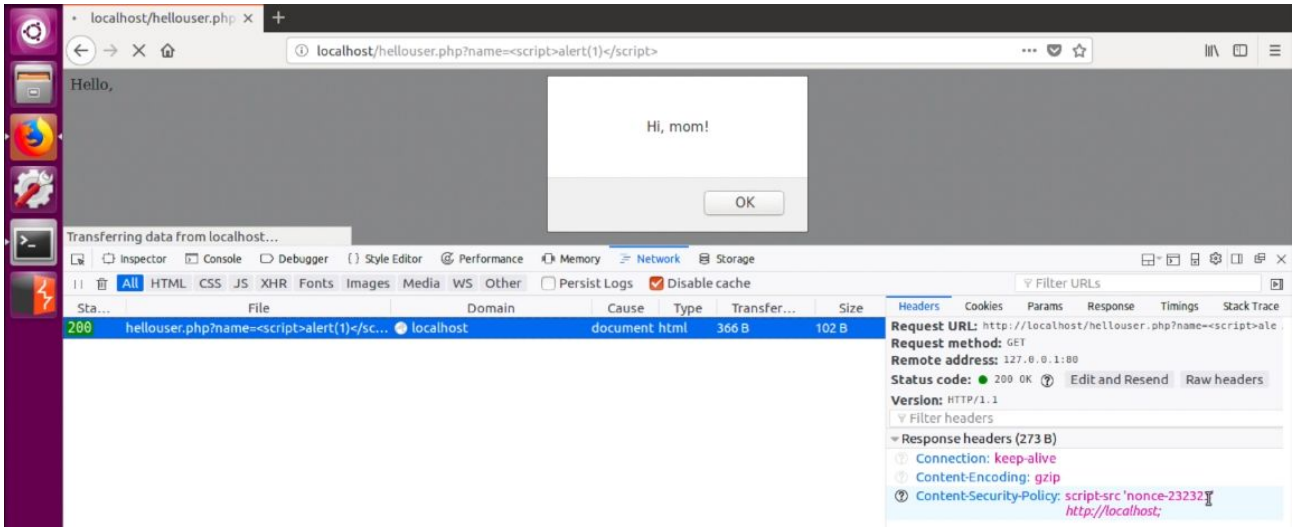
Как мы видим, **alert("Hi, mom!")** сработал.

Представим, что мы злоумышленники и хотим запустить такой JavaScript-код:

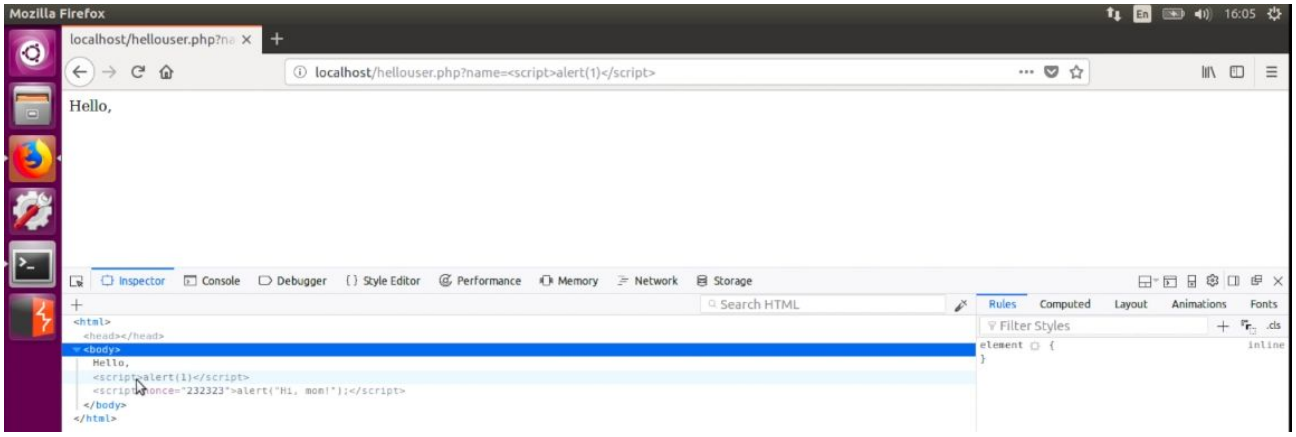


Нажимаем Enter: отработал только **alert("Hi, mom!")**, который был подписан **nonce**.

Браузер получает ответ, смотрит в заголовок политики Content-Security-Policy и видит, что **nonce** равно 232323:

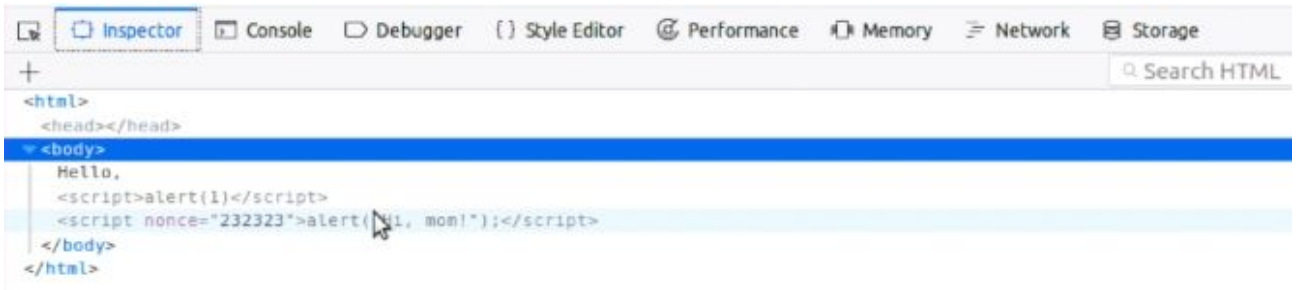


Дальше он идет по всем скриптам на странице, например, берет самый первый:



Браузер видит, что **nonce** нет или он неправильный, а значит не выполняет скрипт. Поэтому **alert(1)** и не было.

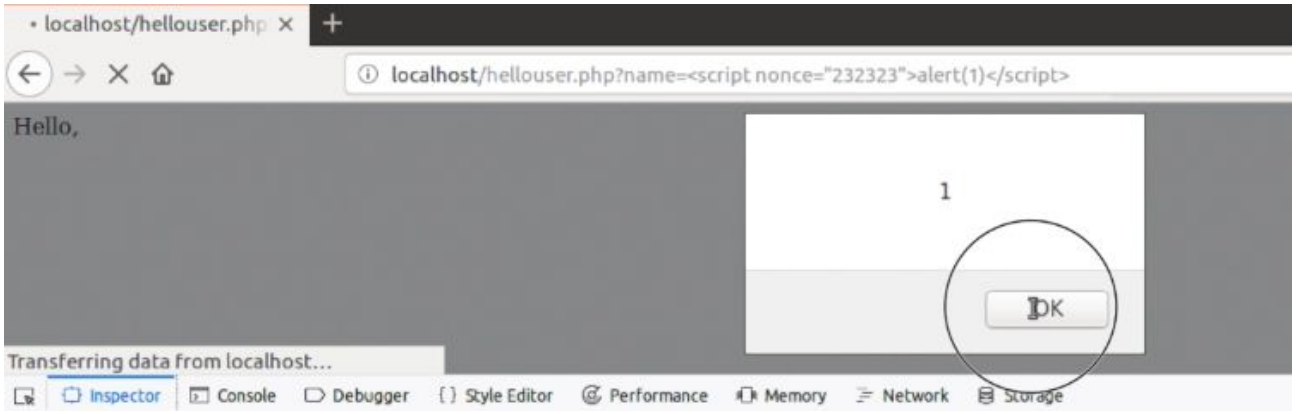
Дальше он видит скрипт с **nonce='232323'**:



Это совпадает с тем значением **nonce**, которое задано в заголовке, и выполняет его!

Так и работает эта политика, но что, если злоумышленник сможет угадать **nonce**? То есть вы вводите этот **nonce** в векторе атаки, например так — http://localhost/hellouser.php?name=<script nonce="232323">alert(1)</script>, и у злоумышленника есть правильный **nonce.**

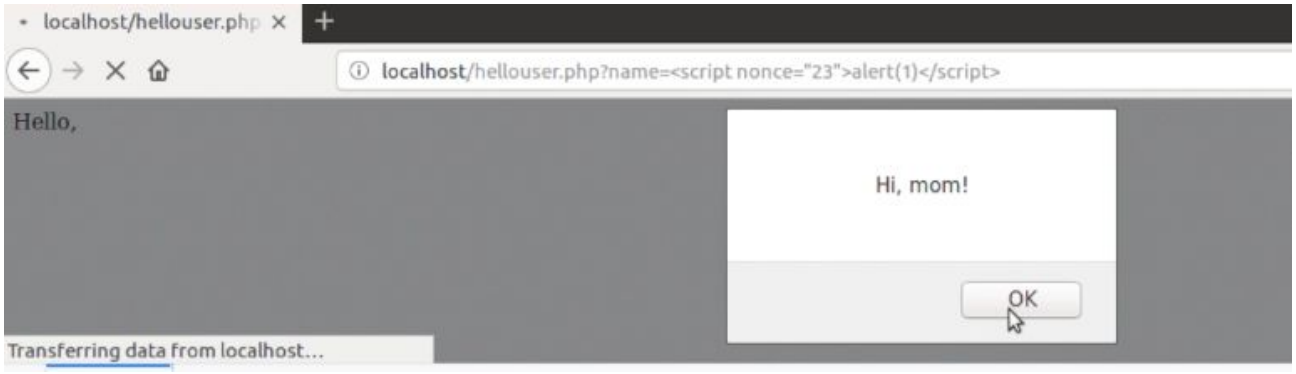
Посмотрим, что произойдет:



Код злоумышленника **alert(1)** успешно отработал, как и валидный код. Для этого и нужно требование, что **nonce** должен быть каждый раз новым, чтобы злоумышленник не смог его отгадать. Даже если злоумышленник сможет заполучить этот **nonce,** он сможет вставить его в запрос, но, например, не сможет угадать следующий **nonce**, а он будет нужен.

Более того, чтобы злоумышленник не смог угадать это число, его делают длиной 256 бит и берут из хорошего генератора случайных чисел — так его практически нереально подобрать. В курсе криптографии вы узнаете, почему это так и на чем это основано.

Давайте попробуем вставить неверный **nonce=”23”**:



С неверным **nonce** скрипт тоже не отрабатывает.

В методе есть альтернатива **nonce** — использование **hash**-функций, т. к. не всегда на сервере можно добавлять **nonce**. Для таких случаев придумали **hash-based.**

Hash — криптографические преобразование, функция, которая на вход принимает строку произвольной длины (миллион символов, миллиард, 100-200 и так далее), а на выходе выдаёт строку фиксированной длины, 256 бит. В этой функции, даже зная выходную строку, нельзя в реальном времени восстановить изначальную. Говоря математическим языком, у **hash**-функции нет обратной к ней функции.

Это работает почти так же, как **nonce**, но тут мы хешируем скрипт — в данном случае **alert("Hi, mom!")** и добавляем хеш от него в политику script-src.

Дальше так же, как с **nonce** — браузер смотрит, какой хеш там есть. Для этого он заново хеширует каждый скрипт и сверяет, сходится ли хеш, заданный в заголовке CSP, с тем, который получился от скрипта. Если сходится, скрипт можно выполнять, а если не сходится — нельзя. Минус такого подхода в том, что если на странице много скриптов, хешей тоже будет много, и каждый раз, когда меняется скрипт, нужно заново пересчитать хеш. Конечно, это все можно автоматизировать, но, в любом случае, будут дополнительные расходы на вычисления, это может быть дорого и долго.

CSP v3: Strict-Dynamic

Strict-Dynamic нужен, чтобы не писать **Whitelist**. Мы игнорируем белый список, доверенным считается всё, что мы будем загружать скриптами. Есть только одно условие - первоначальный скрипт должен быть подписан nonce-ом или hash-ем. Плюсы такого подхода — не нужно поддерживать белый список, для больших сайтов это просто спасение. Минусы этого подхода — нужно больше смотреть свой код, доверять ему, потому что, если в нем есть ошибки, которые приведут к уязвимости, появится возможность обойти политику CSP — скрипт, который загружается нашими скриптами, будет считаться доверенным.

CSP от первого до третьего стандарта движется в сторону того, что они написаны буквально несколькими строчками и при этом помогают построить хорошую защиту от XSS или, по крайней мере, возвести препятствия, чтобы не было возможностей проэксплуатировать уязвимость.

Итоги

Подведем итоги:

1) Узнали, какие виды CSP существуют, подробно рассмотрели, как работает **Whitelist**, **Nonce-based** политика CSP. Обзорно посмотрели, как устроена директива **Strict-Dynamic**, что она нам даёт и какие плюсы и минусы у нее есть.

2) Поняли, в каких случаях какую политику CSP применять. Если у вас небольшой сайт, скорее всего, вам нужна **Whitelist-**политика, и, может быть, **Nonce,** если вы не готовы целиком отказаться от инлайн-кода. Если сайт огромный, нужно использовать **Nonce** и **Strict-Dynamic** — эти 2 подхода можно комбинировать и менять в зависимости от ситуации: когда вы будете разбираться с конкретным сайтом, вы поймёте, какая именно политика подходит.

Следующий урок будет полностью посвящен Same Origin Policy, или SOP, — это очень важный компонент, в котором нужно разобраться — это основы безопасности браузеров:

1) Подробно разберем SOP для разных компонентов браузера, рассмотрим SOP для XHR, для обычных документов, для Cookie, для Webstorage.

2) Узнаем, что такое Origin и какие механизмы его наследования существуют в браузере у различных страниц.

Ссылки к уроку

1. Небезопасный Cross-Origin Resource Sharing.
2. Пример эксплуатации CORS-уязвимости через XSS.