Протокол HTTP. Cookies. CORS. Атака CSRF.

Задание на закрепление базовых знаний по протоколу http. Проверка знаний по материалам занятий «протокол HTTP», «Cookie», «CSRF», «CORS».

Данная лабораторная нацелена на понимание механизмов атаки cross-site request forgery. Базовая информация по атаке приведена в разделе «CSRF».

Для выполнения вам нужно скачать и настроить учебную виртуальную машину (раздел «Виртуальная машина»).

В разделе «Учебный стенд» приводятся задачи для самоконтроля. В данном разделе предлагается проанализировать выполнение различных типов cross-origin запросов, которые разбирались на занятии (различные методы, ajax/не ajax и т.п.). Этот раздел направлен на самостоятельную проработку примеров, разобранных на занятиях, на закрепление/напоминание изученного ранее материала.

Само задание на лабораторную работу, а так же порядок его выполнения, приводится в разделе «Задание к лабораторной работе».

Оглавление

CSRF

Описание атаки

https://owasp.org/www-community/attacks/csrf

Cross-Site request forgery (межсайтовая подделка запросов) – атака, нацеленная на то, чтобы вынудить (атакуемого) пользователя выполнить нежелательные действия в другом приложении, в котором пользователь уже авторизован. Эта атака отличается тем, что злоумышленник, фактически не получая доступа к секретной информации, хранимой в браузере пользователя

(такой, как идентификатор сессии), тем не менее получает возможность выполнения запросов от лица данного пользователя. Зачастую, для проведения атаки могут использоваться технологии социальной инженерии (отправка сообщений со специально сконструированными ссылками, размещение специальных форм на сайте злоумышленника и т.п.).

Рассмотрим, какие механизмы используются при проведении атаки. Зачастую в cookie на стороне клиента хранится секретная информация (идентификатор сессии, секретный токен и т.п.), которая позволяет серверу идентифицировать пользователя. Поскольку браузер автоматически отправляет cookie при выполнении запроса к указанному домену, это облегчает процедуру идентификации пользователя на стороне сервера и при этом позволяет не включать дополнительные механизмы на стороне клиента.

Любые cookie связаны с конкретным доменом. Эти cookie добавляются к запросам на этот домен, при условии, что остальные параметры данного cookie не противоречат отправке.

Браузер может отправлять cookie при следующих вариантах выполнения запроса:

- При вводе адреса в URL строке браузера
- При загрузке ресурсов, для адресов в теге <script>, <img> или <link> на html странице – при этом выполняются GET запросы
- При переходе по ссылке <A href=’’> (это аналогично вводу адреса в URL)
- Отправка данных формы (form submit)
    - При этом отправляются cookie для домена, указанного в параметре action. Если action не указан, подставится текущий домен
    - Данные формы могут передаваться методом GET или POST

```
<p class="add-form-header">Добавить новую запись: </p>
<form class="add-form" method="post" action="?action=record">
  Название: <input type="text" name="name" value=""/><br/>
  <input type="submit" value="Добавить" />
</form><a href="?action=main">На главную</a>
```

- Выполнение Ajax запросов из скриптов
    - объект XMLHTTPRequest, в запросе будут отправлены куки для сайта, указанного в параметре функции open объекта xhr

```
let xhr = new XMLHttpRequest();
xhr.open('DELETE', '?action=record&id='+id);
xhr.send();
```

Важно понимать, что запрос к данному домену может быть инициирован не только приложением, которое установило cookie, но и любым другим приложением. Т.е. злоумышленник может заранее сконструировать данные формы для отправки (на своем сайте), и после того как сайт злоумышленника откроется в браузере пользователя, эти специально подготовленные данные будут отправлены на атакуемый сайт, и при этом браузер пользователя может автоматически отправить cookie, связанные с атакуемым сайтом, так что запрос на атакуемый сайт будет выполнен от лица пользователя.

Данная атака работает, если пользователь уже авторизован в другой системе.

Основной вектор атаки

1. Пользователь авторизуется на уязвимом сайте (атакуемый сайт).
    - У него в cookie сохраняется идентификатор сессии этого сайта.

2. Пользователь заходит на сайт злоумышленника. На этом сайте есть (возможно, скрытая) форма отправки запроса на атакуемый сайт.
    - При этом данные формы сконструированы заранее, и они не видны пользователю.
    - Отправка формы может происходить автоматически с использованием JS.
    - Либо запрос может уходить после клика на ложную кнопку («скачать реферат» и т.п.).
3. Подтверждение формы => отправка запроса в браузере пользователя на атакуемый сайт.
    - Браузер пользователя создает этот запрос и добавляет cookie, связанные с доменом атакуемого сайта. Поэтому идентификатор сессии будет передан браузером пользователя автоматически.
4. Поскольку в составе запроса на атакуемый сайт отправлен корректный идентификатор сессии, сервер «узнает» пользователя.
    - От лица пользователя выполняются нужные злоумышленнику действия на атакуемом сайте.

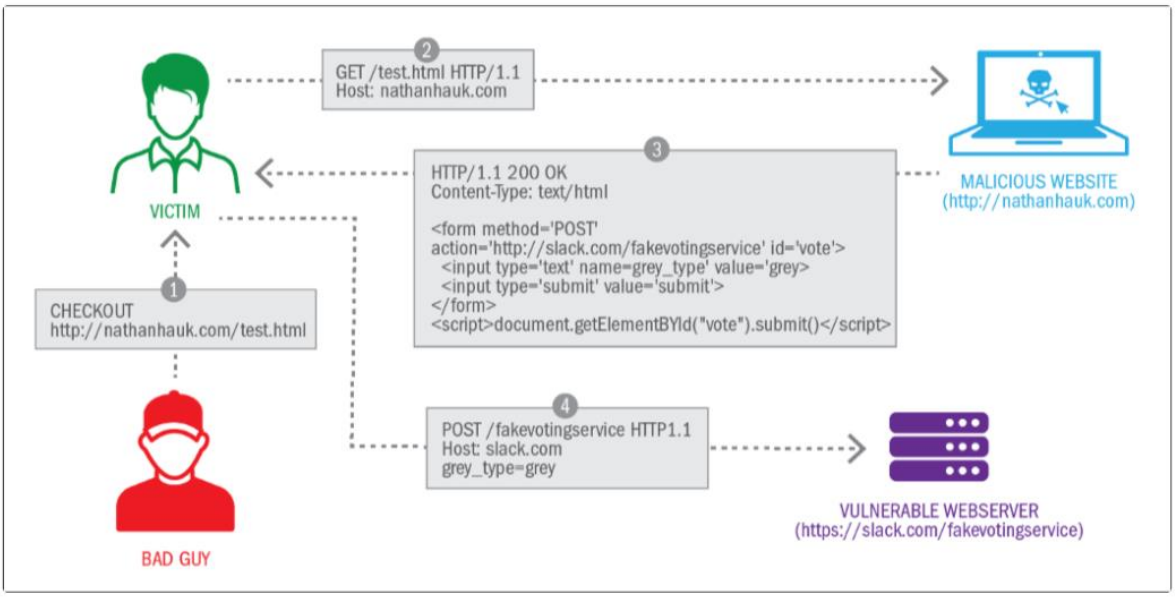


Рисунок 1 Пример CSRF атаки с автоотправкой данных формы.

Простота атаки заключается в том, что злоумышленнику не нужно получать доступ к секретной информации – она остается в браузере пользователя. Злоумышленник только должен понимать логику работы атакуемого сайта чтобы корректно сконструировать запрос.

Для того, чтобы скрыть сам факт атаки, можно избежать перенаправления в браузере пользователя. Поскольку на своем сайте злоумышленник волен выполнять любые действия, у него для этого есть различные способы:

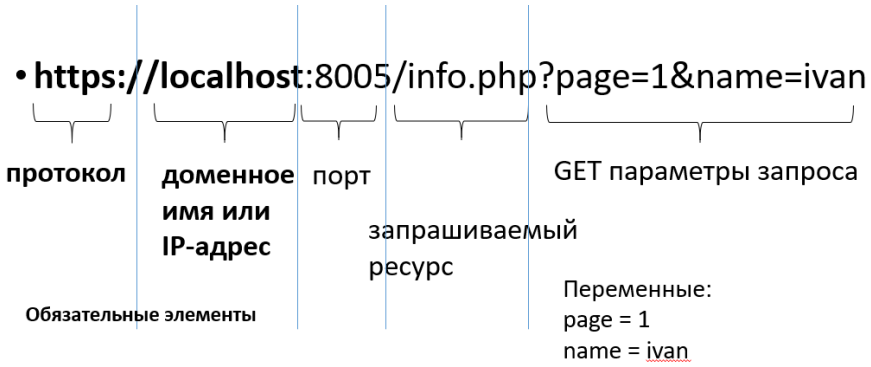
- Встраивание атакующей формы в скрытый iframe и ее автоотправка (работает для GET и POST запросов)
- Встраивание атакующего запроса в подгружаемые ресурсы на странице, например в теги <img> или <link> (только для GET запросов)
- Выполнение ajax запросов на атакуемый сайт (любые запросы – GET, POST, PUT, DELETE и т.п.)

Важно отметить, что злоумышленник не сможет получить и обработать ответ на cross-origin запрос (Same Origin Policy запретит это), однако сам запрос будет выполнен.

Cross-origin запросы

Для начала дадим определим, что такое cross-origin запрос. Для выполнения любого запроса необходимо указать как минимум три элемента:

- протокол взаимодействия (HTTP/HTTPS)
- доменное имя ресурса (или его ip адрес)
- удаленный порт подключения (если не указан, используется порты по умолчанию 80 для HTTP и 443 для HTTPS)



Origin (источник данных) – это и есть совокупность домена/протокола/порта.

User-agent (например браузер) делает запрос с другого источника (cross-origin http request), если источник текущего документа (url в верхней строке браузера) отличается от запрашиваемого доменом, протоколом или портом.

Например, http://myapp1.local и https://myapp1.local это разные origin.

CORS и preflight requests. «Простые» и «сложные» запросы

Для противодействия CSRF атакам добавлены следующие защитные механизмы – CORS (cross-origin resource sharing) заголовки и отправка предварительных запросов (preflight requests).

Решение об отправке cookie при выполнении cross-origin запросов принимается браузером по-разному в зависимости от дополнительных параметров. К примеру, переход по ссылке или отправка данных формы с переходом с одного сайта на другой будет обрабатываться одним образом, а отправка cross-origin ajax запроса – другим. Т.е. ajax и не ajax запросы обрабатываются по-разному. При переходе по ссылке или при отправке данных формы злоумышленнику доступны только два метода передачи – GET и POST.

Выполнение ajax запросов браузером так же происходит по-разному в зависимости от параметров запроса. При выполнении ajax (xhr) запроса через XMLHttpRequest объект, браузер делит все запросы на «простые» и «сложные». Для того, чтобы ajax запрос считался «простым», должны выполняться следующие условия:

- Допустимые методы: GET, HEAD, POST
- Использование только CORS-безопасных заголовков
    - https://fetch.spec.whatwg.org/#cors-safelisted-request-header
- С допустимым заголовком Content-Type
    - application/x-www-form-urlencoded, multipart/form-data, text/plain

- В запросе не используется ReadableStream объект

Если любое из данных условий не выполняется, запрос будет считаться сложным.

Заметим, что cross-origin ajax запросы методами GET и POST не будут считаться «сложными», если при этом не используются кастомизированные заголовки или специальные типы контента. Поэтому для таких запросов не будут выполнятся preflight запросы.

Обработка отправки cross-origin ajax запросов для «простых» и «сложных» запросов отличается. При выполнении «сложных» cross-origin запросов, браузер сначала пытается выяснить, разрешено ли выполнение cross-origin ajax запроса к данному ресурсу со стороны данного origin. Для того, чтобы выяснить права доступа и были добавлены следующие механизмы:

- Предварительные запросы (preflight requests)
- Разрешительные CORS-заголовки (Access-Control-*)

Последовательность обработки «сложных» запросов следующая

1. JavaScript приложение сайта https://hackerapp.local инициирует выполнение «сложного» запроса при помощи XMLHttpRequest (например, запрос методом DELETE к ресурсу https://mysilte1.local?record=1)
   Если при этом требуется отправка cookie, приложение выставляет у объекта XMLHttpRequest значение свойства withCredentials=true
2. User Agent (браузер) приостанавливает выполнение запроса

   DELETE https://mysilte1.local?record=1
   и вместо него инициирует предварительный (preflight) запрос
   OPTIONS https://mysilte1.local?record=1
   Цель этого запроса – проверить разрешения доступа методом DELETE к ресурсу https://mysilte1.local?record=1 от origin https://hackerapp.local
   Если при этом была запрошена отправка cookie (withCredentials=true), так же будет проверятся разрешение на отправку cookie.
3. Приложение https://mysilte1.local в ответ на OPTIONS запрос отправляет ответ, в котором могут быть указаны специальные разрешительные CORS-заголовки. Это должно быть сделано сознательно, т.е. если разработчики приложения https://mysilte1.local не отправят принудительно эти заголовки, то по умолчанию доступ к ресурсу будет запрещен.
4. Браузер анализирует ответ сервера https://mysilte1.local на OPTIONS запрос, ищет там необходимые CORS заголовки и анализирует их значение. Если полученные заголовки его устраивают, браузер выполняет исходный «сложный» запрос (в данном примере - DELETE https://mysilte1.local?record=1), иначе выполнение этого запроса блокируется
5. Если «сложный» запрос был выполнен, возможность доступа JavaScript приложения к обработке полученного ответа так же регулируется CORS-заголовками. В нашем примере, в ответ на DELETE запрос приложение так же должно добавить разрешительные CORS заголовки, чтобы JavaScript приложение сайта https://hackerapp.local могло получить и обработать ответ.

Общая схема обработки cross-origin ajax запросов приведена на рисунке.

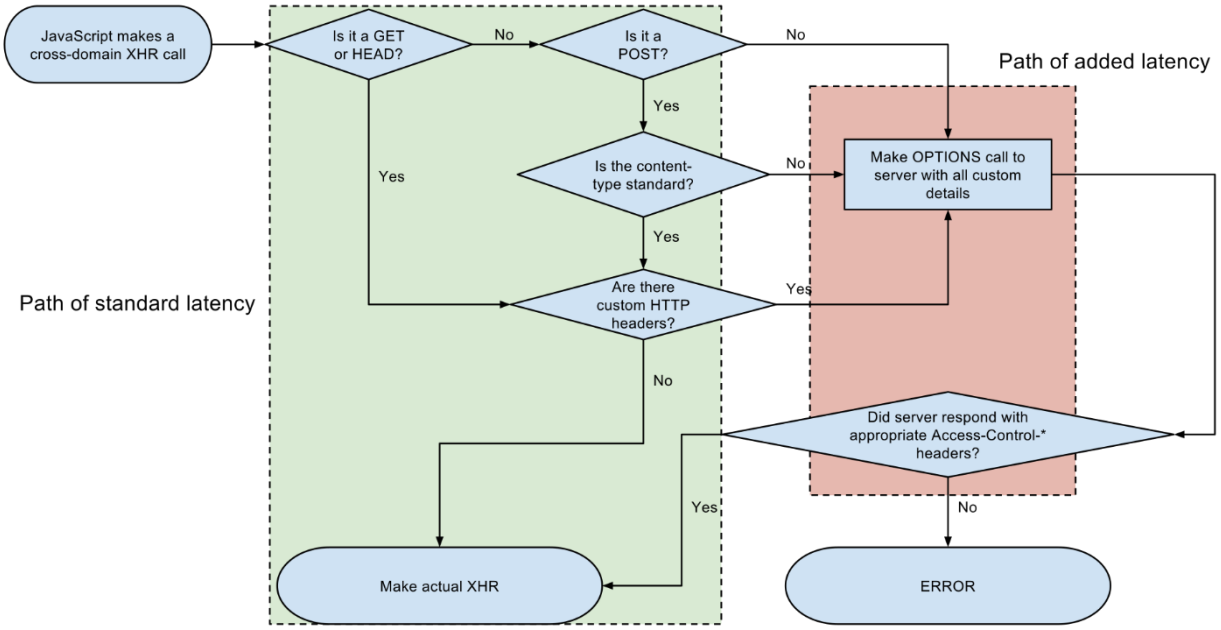


Рисунок 2. Порядок обработки cross-origin ajax запросов

Разрешительные CORS-заголовки могут добавляться на каждый ресурс независимо, т.е. cross-origin доступ к одним ресурсам может быть открыт, а к другим закрыт.

Дополнительная информация по CORS заголовкам приведена в [3],[4],[5].

В частности, для разрешения выполнения cross-origin запроса с другого домена заданным методом с передачей cookie, приложение в ответ на preflight запрос должно вернуть следующие разрешающие CORS заголовки

HTTP Response headers (CORS)	**Описание**
Access-Control-Allow-Origin	Разрешенные origin для данного ресурса
Access-Control-Allow-Methods	Разрешенные для выполнения к данному ресурсу методы запроса
Access-Control-Allow-Credentials	Разрешение браузеру отправлять Cookie при запросе к данному ресурсу

Примеры таких заголовков:

```
Access-Control-Allow-Credentials: true
Access-Control-Allow-Methods: DELETE, POST, PUT
Access-Control-Allow-Origin: https://hackerapp.local
```

Замечание насчет wildcard (*). В качестве значения заголовков Access-Control-Allow-Origin и Access-Control-Allow-Methods допускается указать * для указания любого разрешенного origin или метода. Однако в случае, если делается запрос withCredentials (с передачей cookie), указание * в ответ на эти заголовки не разрешено, т.е. CORS проверка в браузере вернет ошибку и исходный

запрос не будет выполнен (https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS/Errors/CORSNotSupportingCredentials).

Защитные механизмы

https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html

Среди отсновных защитных механизмов от CSRF атак можно выделить следующие:

- CSRF-токены – использование специальных случайных идентификаторов для предотвращения выполнения поддельного запроса.
- Использование свойств cookie - корректная настройка атрибута SameSite для cookie.
- Дополнительные механизмы защиты (глубокая защита)
    - Использование кастомизированных заголовков (для принудительного выполнения preflight запросов)
    - Проверка стандартных заголовков (Origin, Referer)
    - Double submit cookies
- Корректное использование методов передачи – не использовать GET запросы для изменения данных

CSRF-токен

Основная идея защиты состоит в использовании случайной последовательности, которая должна быть отправлена клиентом для выполнения запроса и при этом она не должна храниться в cookie.

Последовательность проверки следующая:

1. На сервере генерируется случайное криптографически стойкое значение – токен
    - Токен сохраняется на сервере с привязкой к данному пользователю (например, в данных сессии текущего пользователя на сервере)
2. Этот токен отдается в ответе сервера в ответ на запрос на изменение данных. Важно, что токен не должен сохраняться в cookie! Например, токен может отправляться в скрытых элементах формы редактирования информации.
3. При следующем запросе на изменение данных с клиента, токен передается в запросе на изменение в составе данных формы. При этом, поскольку токен не сохранен в cookie, он не будет передаваться в заголовке.
4. Сервер сравнивает значение полученного среди данных формы токена и значения, хранимого на сервере. Если они совпадают – запрос выполняется.

- Создание и проверка CSRF токена

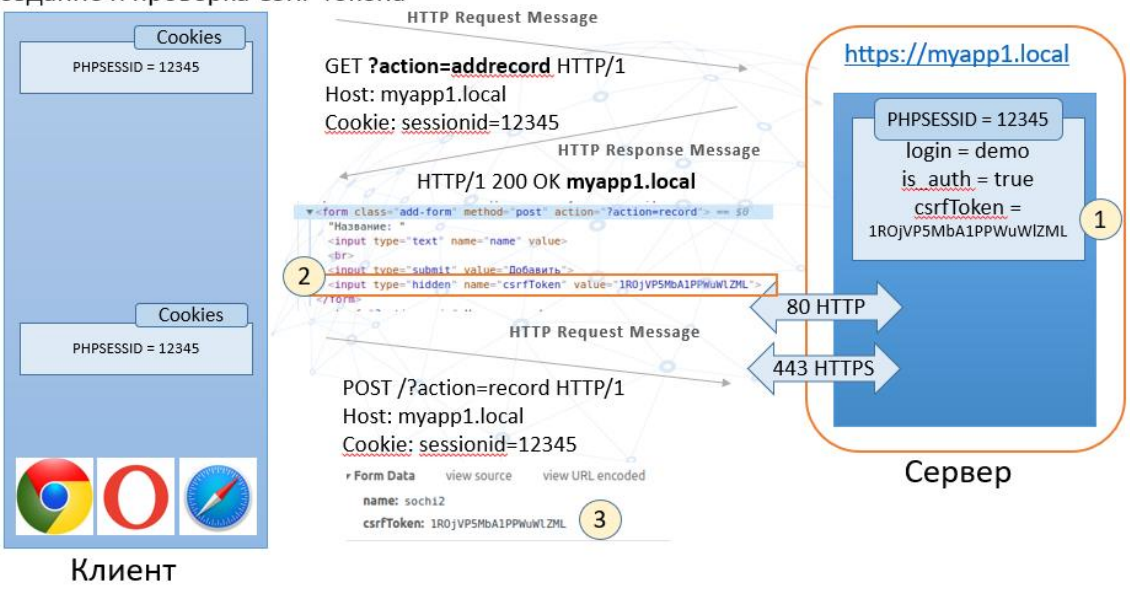


Рисунок 3 Последовательность проверки при использовании CSRF токена.

Генерация токена может происходить один раз на сессию пользователя, или в ответ на каждый запрос – это баланс простоты использования/защищенности.2

Важно, что значение токена связано с сессией конкретного пользователя, т.е. сервер будет проверять значение не CSRF-токена «вообще», а значение токена, выданного этому конкретному пользователю. Поэтому нельзя подставить значение «чужого» токена в запрос.

Поскольку (в соответствии с Same Origin Policy), JavaScript приложение не может обработать результат cross-origin запроса (если это специально не разрешено CORS заголовками), нет возможности провести атаку в два запроса (первым GET запросом получить код формы, обработать ответ, вытащить из него токен и выполнить второй POST запрос уже с токеном).

CSRF-токены рекомендуется добавлять ко всем запросам, которые меняют информацию на сервере. Обычно это любые POST, PUT, DELETE запросы.

Резюмируя обзор средств защиты, можно отметить следующее. PUT и DELETE запросы могут выполнятся только с использованием XMLHttpRequest (ajax), и при этом это «сложные» запросы, т.е. они защищены использованием CORS и preflight requests. А GET запросы обычно не приводят к модификации данных на сервере, и при этом результат cross-origin GET запроса не может быть обработан другим приложением (Same Origin Policy). Поэтому наиболее уязвимы к этой атаке POST запросы, и их надо закрывать CSRF-токенами.

Следующая комбинация защитных механизмов дает хорошую защиту от CSRF-атак:

1. Использование CSRF-токенов для POST запросов.
2. Корректная настройка CORS для GET, POST, PUT, DELETE запросов (либо настройка по умолчанию, когда cross-origin PUT и DELETE запросы будут запрещены политикой CORS вследствие отсутствия разрешающих заголовков, а обработка результатов cross-origin GET запроса будет запрещена).
3. Использование GET запросов только для получения данных, но не их модификации.

При этом могут использоваться дополнительные механизмы, которые рассмотрены ниже.

Дополнительные механизмы

- Добавление специальных заголовков к любым типам запросов (Custom HTTP headers) и их проверка на сервере. Позволяет вынудить браузер делать preflight запросы при выполнении любых cross-origin ajax запросов, включая все запросы методами GET и POST.
- Проверка заголовков Origin и Referer. Поскольку они не могут быть модифицированы средствами javascript (из формирует User Agent, т.е. браузер), можно использовать их для выявления cross-origin запросов на стороне сервера.
- SameSite параметр cookies [7]. Появление этого параметра значительно затруднило проведение CSRF атак на приложения, которые не выключат это ограничение осознанно. Приложение может установить cookie параметр SameSite=None для того, чтобы была возможность настроить сложное межсайтовое взаимодействие. Но по умолчанию значение этого параметра будет lax, что серьезно ограничивает отправку cookie при cross-origin запросах.

Виртуальная машина

Для выполнения лабораторной работы вам необходима виртуальная машина Lunix с установленным учебным стендом. Скачать ее можно по адресу https://drive.google.com/file/d/10pOwQdHQfOWVC1F_r8dZ4TiRAKWPD0y7/view

Подключите скачанную виртуальную машину в VirtualBox (через импорт).

Для входа в операционную систему виртуальной машины используйте следующие данные:

Логин: dojo

Пароль: dojo

Учебный стенд

Задачи этого раздела даны для самоконтроля и закрепления материала. Мы рассмотрим учебный стенд, на котором рассматривались механизмы атаки и защиты. В данном разделе предлагается проанализировать выполнение различных типов cross-origin запросов, которые разбирались на занятии (различные методы, ajax/не ajax и т.п.). Этот раздел направлен на закрепление/напоминание изученного ранее материала – вы можете его использовать для самостоятельной проверки полученных знаний. Собственно задание на лабораторную приведено в следующем разделе.

В данном примере рассмотрены два сайта учебного стенда – myapp1.local и hackerapp.local

Исходные коды сайтов расположены соответственно:

myapp1.local: /var/www/myapp1

hackerapp.local: /var/www/hackerapp

Настройки разных режимов сайта myapp1.local находятся в файле /var/www/myapp1/settings.php

Настройки, которые можно менять в ходе выполнения работы (в файле settings.php):

	Значение	Описание	Установленная строка
1	CSRFDefence	Включение/выключение защиты с CSRF токеном	
	true	использовать CSRF токены	define("CSRFDefence", true);
	false	не использовать CSRF токены	define("CSRFDefence", false);
2	CORSheaders		

	false	Не отсылать CORS заголовки	define("CORSheaders", false);
	"ALL"	Посылать CORS заголовки, при этом заголовок Access-Control-Allow-Origin: *	define("CORSheaders", "ALL");
	"SITE"	Посылать CORS заголовки, при этом заголовок Access-Control-Allow-Origin: https://hackerapp.local	define("CORSheaders", "SITE");

1. Проверка настроек.
   Проверьте начальные настройки системы (в файле /var/www/myapp1/settings.php) (при необходимости, измените их, как указано ниже)

```
/**
 * CSRFDefence - настройка включения защиты CSRF
 * true - использовать CSRF токены
 * false - не использовать CSRF токены
 */
define("CSRFDefence", false); // возможные значения: true, false
define("CORSheaders", false); // возможные значения: false, "ALL", "SITE"
```

   Настройки cookie должны быть следующие

```
/**
 * настройки cookie для идентификатора сессии
 */
$cookie_settings = [
    'secure' => true,
    'samesite' => 'None',
    'httponly' => false
];
```

2. Анализ заголовков запроса и ответа.
   Запустите браузер (google). Откройте сайт https://myapp.local
   Вы увидите форму авторизации. Введите данные для авторизации (логин: demo пароль: pass). Проанализируйте результат ответа – какие данные вернулись в response headers, какое cookie сейчас сохранено (используйте панель разработчика браузера chrome, доступную по F12). Идентифицируйте cookie, которая используется для идентификации пользователя (хранит идентификатор сессии).
3. Понимание параметров Cookie.
   Откройте новую вкладку и введите там адрес http://myapp.local Проанализируйте заголовки запроса и ответа. Какие cookie были переданы в запросе? Какие cookie, связанные с данным доменом, не были переданы? Почему (какие параметры предотвратили передачу)? Почему вы снова видите форму авторизации?
4. Cross-Origin Post запрос (не ajax).
   Откройте систему по безопасному протоколу https://myapp1.local , авторизуйтесь.
   Откройте список записей, зафиксируйте, какая запись имеет максимальный идентификатор.
   Откройте сайт злоумышленника https://hackerapp.local Нажмите кнопку «проголосовать за котика» и посмотрите, какой запрос будет выполнен (используя панель разработчика браузера chrome). Проверьте список записей на странице https://myapp1.local/?action=table . Почему была добавлена новая запись?
5. Cross-Origin Delete запрос (ajax).
   Откройте сайт https://hackerapp.local нажмите кнопку для выполнения cross-origin запроса на удаление.

Проанализируйте выполненные запросы и ответы (их заголовки и результат выполнения). Какие запросы были выполнены? Почему на вкладке network появилось более одного запроса, чем они различаются? Почему не прошел запрос на удаление?

6. Cross-Origin Post запрос (ajax).
   Откройте сайт https://hackerapp.local нажмите кнопку для выполнения cross-origin запроса на добавление.

   Проанализируйте выполненные запросы и ответы (их заголовки и результат выполнения). Какие запросы были выполнены? Почему получено сообщение об ошибке (что оно означает)? Была ли добавлена запись?
7. CSRF-Token. В настройках файла, включите защиту от CSRF. Откройте на редактирование файл /var/www/myapp1/settings.php
   Отредактируйте следующую настройку:

```
define("CSRFDefence", true); // возможные значения: true, false
```

   Откройте сайт https://myapp1.local . Выйдите из системы и авторизуйтесь повторно. Нажмите «добавить запись». Проанализируйте исходный код формы добавления и запрос, отправленный при добавлении. Идентифицируйте защитный токен.
   Откройте сайт https://hackerapp.local нажмите кнопку «проголосовать за котика». Проанализируйте выполненный запрос и полученный ответ. Сравните с запросом, выполненным, выполненным при отправке данных формы с сайта myapp1.local. В чем различие? Объясните механизм действия защитных токенов.

Задание к лабораторной работе

Работа выполняется в несколько этапов

1. Отработка CSRF атаки с использованием сайта multillidae (по примеру в документации данного учебного приложения)
2. Отработка CSRF атаки с использованием сайта DWVA

I. Обучение по примеру - CSRF атака на multillidae

Атакуемый сайт расположен по адресу http://nowasp.local/mutillidae/

Дополнительная информация по CSRF расположена по адресу

http://nowasp.local/mutillidae/hints-page-wrapper.php?level1HintIncludeFile=14
(внутри данной виртуальной машины)

Используйте данную документацию, чтобы создать страницу злоумышленника, которая будет атаковать форму добавления записи в блог: http://nowasp.local/mutillidae/index.php?page=add-to-your-blog.php

Для проведения атаки можно воспользоваться учебным примером

1. Для проведения автоматической отправки формы:

http://nowasp.local/mutillidae/hints-page-wrapper.php?level1HintIncludeFile=14
См. Example: Force someone to add a blog - HTML injection

2. Для проведения скрытого ajax запроса

   http://nowasp.local/mutillidae/hints-page-wrapper.php?level1HintIncludeFile=14
   См. Example: Force someone to add a blog - AJAX

Порядок проведения атаки.

1. Откройте сайт http://nowasp.local/mutillidae/ Авторизуйтесь, используя стандартный логин и пароль demo:pass (или можете зарегистрировать нового пользователя)
2. Найдите раздел OWASP 2017 -> A8. Cross Site Request Forgery -> Add to your blog. Попробуйте добавить запись в блог, проанализируйте выполненный запрос. Идентифицируйте cookie и другие параметры запроса (подтвердите скриншотами).
3. Поднимите на сайте https://hackerapp.local страницу для атаки на добавление новой записи пользователем на сайте http://nowasp.local/mutillidae/ с автоматической отправкой формы. Для этого достаточно разместить новый созданный файл в директории /var/www/hackerapp и вызвать его через https://hackerapp.local/%YOURFILE%
   Учитывайте установку корректного владельца и прав доступа на создаваемые в папке /var/www/hackerapp файлы.
   Для проведения атаки внимательно ознакомьтесь с примером, приведенным выше (Example: Force someone to add a blog - HTML injection).
   Откройте созданное приложение, проведите атаку.
   Подтвердите скриншотами выполняемые запросы, отправленные заголовки, заголовки ответа, результат атаки в атакуемом приложении (добавленная запись), исходные коды атакующей формы, настройки прав на файлы в директории /var/www/hackerapp.
4. Добавьте скрытую автоотправку запроса ajax при заходе на созданную страницу злоумышленника.
   (см. выше проведение скрытого ajax запроса - Example: Force someone to add a blog - AJAX)
   Откройте созданное приложение, проведите атаку. Подтвердите скриншотами выполняемые запросы, отправленные заголовки, заголовки ответа, результат атаки в атакуемом приложении (добавленная запись), исходные коды атакующей формы, настройки прав на файлы в директории /var/www/hackerapp.

II. Закрепление - CSRF атака на DWVA

Проведение CSRF атаки на тренировочный сайт DVWA (Damn Vulnerable Web Application)

Тренировочный сайт DVWA расположен по адресу http://dvwa.local

Для авторизации в системе можно использовать стандартный логин и пароль admin:password

На сайте предусмотрено несколько уровней защиты самого приложения, от минимального (Low) до максимального (Impossible). При этом уровень Low выключает использование любых механизмов безопасности, при уровне High используются хорошие механизмы защиты.

Переключение текущего уровня безопасности происходит в разделе DVWA Security
http://dvwa.local/security.php

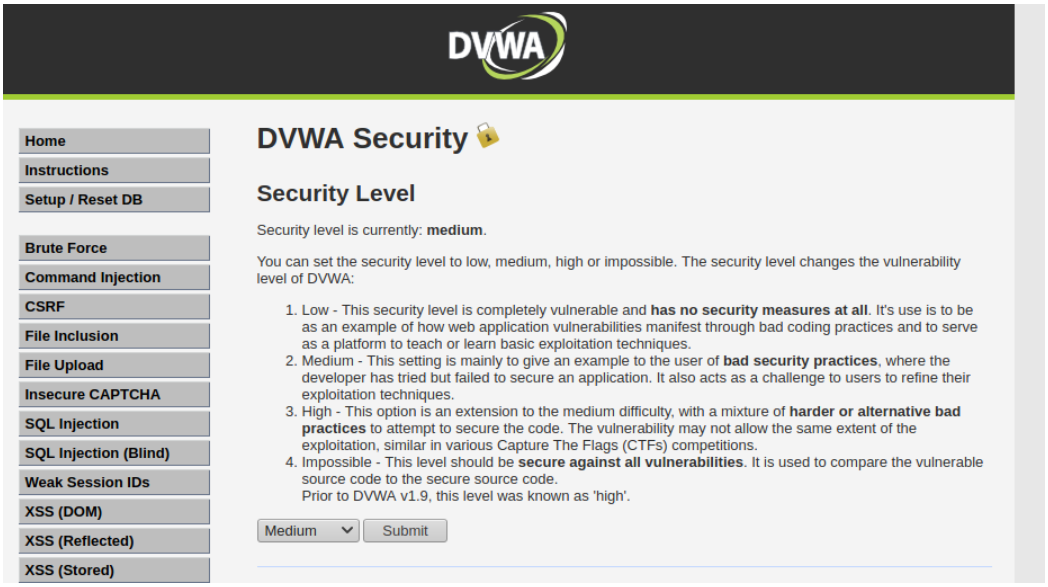


Рисунок 4 Настройка уровня безопасности DVWA

Порядок проведения атаки.

1. Откройте сайт http://dvwa.local Авторизуйтесь, используя стандартный логин и пароль admin:password. Проверьте, что установленный уровень безопасности Low (при необходимости, поменяйте).
2. (Уровень безопасности Low) Найдите ссылку CSRF (форма изменения пароля). Попробуйте поменять пароль (запомните новое значение), проанализируйте выполненный запрос. Идентифицируйте cookie и другие параметры запроса (подтвердите скриншотами).
3. Поднимите на сайте https://hackerapp.local страницу для атаки на смену пароля на сайт http://dvwa.local
   Для этого достаточно разместить созданный файл в директории /var/www/hackerapp. Учитывайте установку корректного владельца и прав доступа на создаваемые файлы.
   Откройте созданное приложение, проведите атаку.
   Подтвердите скриншотами выполняемые запросы, отправленные заголовки, заголовки ответа, исходные коды атакующей формы, настройки прав на файлы в директории /var/www/hackerapp.
4. Добавьте скрытую автоотправку запроса при заходе на созданную страницу злоумышленника.
   Для этого можно использовать, например, скрытый ajax запрос, встраивание запроса в <img> тэг (при условии, что сама картинка будет скрыта от отображения), скрытый iframe и т.п.
   Откройте созданное приложение, проведите атаку.
   Подтвердите скриншотами выполняемые запросы, отправленные заголовки, заголовки ответа, исходные коды атакующей формы, настройки прав на файлы в директории /var/www/hackerapp.
5. Поменяйте настройки безопасности – установите уровень безопасности High. Попробуйте заново провести атаку, используя созданную ранее страницу злоумышленника. Проанализируйте результат.
6. (Уровень безопасности High). Снова зайдите в DVWA страница CSRF (форма изменения пароля). Изучите исходный код формы изменения пароля. Попробуйте поменять пароль (запомните новое значение), проанализируйте выполненный запрос. Идентифицируйте cookie и другие параметры запроса (подтвердите скриншотами). Что изменилось в форме

изменения пароля и выполняемом запросе по сравнению с уровнем Low? Идентифицируйте защитные механизмы.

Дополнительная литература

1. Описание атаки CSRF (OWASP) https://owasp.org/www-community/attacks/csrf
2. Защитные механизмы CSRF (OWASP) https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html
3. CORS https://developer.mozilla.org/ru/docs/Web/HTTP/CORS
4. CORS-заголовки https://fetch.spec.whatwg.org/#access-control-allow-origin-response-header
5. “CORS in action” by Monsur Hossain
6. Ошибки CORS https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS/Errors
7. Свойство SameSite https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie/SameSite
8. Заголовки, запрещенные для модификации через JS https://developer.mozilla.org/en-US/docs/Glossary/Forbidden_header_name