Abusing Internet Explorer 8's XSS Filters

by Eduardo Vela Nava (http://twitter.com/sirdarckcat, sird@rckc.at)
David Lindsay (http://twitter.com/thornmaker, http://www.cigital.com)

Summary

Internet Explorer 8 implements an anti Cross-site Scripting (XSS) mechanism to detect certain types of XSS attacks. This feature can be abused by attackers in order to enable XSS on web sites and web pages that would otherwise be immune to XSS.

Background

Internet Explorer 8 introduced a new type of defense against Cross-site Scripting (XSS) attacks. The idea was to build filters into the browser which can detect and prevent certain types of malicious XSS attacks. Most filter based XSS approaches are implemented on the server side inside a web application or as part of a Web Application Firewall. This made the Microsoft approach a somewhat novel approach but one which other browser vendors have begun to follow. Although the filters do not protect against all types of XSS attacks, nor do they attempt to, they do attempt to raise the bar for a would-be attacker by making certain commonly attack scenarios non-exploitable.

The filters work by scanning outbound requests for potential malicious strings. When such a string is detected, IE8 will dynamically generate a regular expression matching the outbound string. The browser then looks for the same pattern in responses from the server. If a match is made anywhere in the server's response then the browser assumes that a reflected XSS attack is being conducted and the browser will automatically alter the response so that the XSS attack will be unsucessful.

The exact method used to alter a server's response is a crucial component in preventing XSS attacks. If the attack is not properly neutralized then a malicious script may still execute. On the other hand, it is also crucial that benign requests are not accidently detected. The Internet Explorer 8 team decided to use a 'neutering' technique to neutralize detected attacks. More specifically, when the the filters make a positive match against the server's response, the malicious part of the response will have a certain character (or characters) modified so that the attack will not execute (or not render properly). For example, if the string **`<script>alert(0)</script>`** is detected in an outgoing request's GET parameter and again in the response body, then the neutering mechanism is triggered and all occurances of this string will be altered to **`<sc#ipt>alert(0)</script>`**. Thus the injected script is not executed.

In most instances where a malicious injection is detected, Internet Explorer 8 will alter the malicious part of the response by replacing certain characters with the **#** character. The following list shows each of the filters used to detect malicious attacks. Highlighted is the character that will be neutered, i.e. changed to a **#** symbol when an attack is detected.

- (v|(&[#()=]x?0*((86)|(56)|(118)|(76));?))([\t]|(&[#()=]x?0*(9|(13)|(10)|A|D);?))*(b|(&[#()=]x?0*((66)|(42)|(98)|(62));?))([\t]|(&[#()=]x?0*(9|(13)|(10)|A|D);?))*(s|(&[#()=]x?0*((83)|(53)|(115)|(73));?))([\t]|(&[#()=]x?0*(9|(13)|(10)|A|D);?))*(c|(&[#()=]x?0*((67)|(43)|(99)|(63));?))([\t]|(&[#()=]x?0*(9|(13)|(10)|A|D);?))***{(r|(&[#()=]x?0*((82)|(52)|(114)|(72));?))}**([\t]|(&[#()=]x?0*(9|(13)|(10)|A|D);?))*(i|(&[#()=]x?0*((73)|(49)|(105)|(69));?))([\t]|(&[#()=]x?0*(9|(13)|(10)|A|D);?))*(p|(&[#()=]x?0*((80)|(50)|(112)|(70));?))([\t]|(&[#()=]x?0*(9|(13)|(10)|A|D);?))*(t|(&[#()=]x?0*((84)|(54)|(116)|(74));?))([\t]|(&[#()=]x?0*(9|(13)|(10)|A|D);?))*(:|(&[#()=]x?0*((58)|(3A));?)).

- (j|(&[#()=]x?0*((74)|(4A)|(106)|(6A));?))([\t]|(&[#()=]x?0*(9|(13)|(10)|A|D);?))*(a|(&[#()=]x?0*((65)|(41)|(97)|(61));?))([\t]|(&[#()=]x?0*(9|(13)|(10)|A|D);?))*(v|(&[#()=]x?0*((86)|(56)|(118)|(76));?))([\t]|(&[#()=]x?0*(9|(13)|(10)|A|D);?))*(a|(&[#()=]x?0*((65)|(41)|(97)|(61));?))([\t]|(&[#()=]x?0*(9|(13)|(10)|A|D);?))*(s|(&[#()=]x?0*((83)|(53)|(115)|(73));?))([\t]|(&[#()=]x?0*(9|(13)|(10)|A|D);?))*(c|(&[#()=]x?0*((67)|(43)|(99)|(63));?))([\t]|(&[#()=]x?0*(9|(13)|(10)|A|D);?))***{(r|(&[#()=]x?0*((82)|(52)|(114)|(72));?))}**([\t]|(&[#()=]x?0*(9|(13)|(10)|A|D);?))*(i|(&[#()=]x?0*((73)|(49)|(105)|(69));?))([\t]|(&[#()=]x?0*(9|(13)|(10)|A|D);?))*(p|(&[#()=]x?0*((80)|(50)|(112)|(70));?))([\t]|(&[#()=]x?0*(9|(13)|(10)|A|D);?))*(t|(&[#()=]x?0*((84)|(54)|(116)|(74));?))([\t]|(&[#()=]x?0*(9|(13)|(10)|A|D);?))*(:|(&[#()=]x?0*((58)|(3A));?)).

- <st**{y}**le.*?>.*?((@[i\\])|(([:=]|(&[#()=]x?0*((58)|(3A)|(61)|(3D));?)).*?([(\\]|(&[#()=]x?0*((40)|(28)|(92)|(5C));?))))

- [/+\t\"\'`]st**{y}**le[/+\t]*?=.*?([:=]|(&[#()=]x?0*((58)|(3A)|(61)|(3D));?)).*?([(\\]|(&[#()=]x?0*((40)|(28)|(92)|(5C));?))

- <OB**{J}**ECT[/+\t].*?((type)|(codetype)|(classid)|(code)|(data))[/+\t]*=

- <AP**{P}**LET[/+\t].*?code[/+\t]*=

- [/+\t\"\'`]data**{s}**rc[+\t]*?=.

- <BA**{S}**E[/+\t].*?href[/+\t]*=

- <LI**{N}**K[/+\t].*?href[/+\t]*=

- <ME**{T}**A[/+\t].*?http-equiv[/+\t]*=

- <\?im**{p}**ort[/+\t].*?implementation[/+\t]*=

- <EM**{B}**ED[/+\t].*?SRC.*?=

- [/+\t\"\'`]**{o}**n\c\c\c+?[+\t]*?=.

- <.*[:]vmlf**{r}**ame.*?[/+\t]*?src[/+\t]*=

- <[i]?f**{r}**ame.*?[/+\t]*?src[/+\t]*=

- `<is{i}ndex[ /+\t>]`
- `<fo{r}m.*?>`
- `sc{r}ipt.*?[ /+\t]*?src[ /+\t]*=`
- `<sc{r}ipt.*?>`
- `[\"\'][ ]*(([^a-z0-9~_:\'\"])|(in)).*?(((l|(\\u006C))(o|(\\u006F))(c|(\\u0063))(a|(\\u0061))(t|(\\u0074))(i|(\\u0069))(o|(\\u006F))(n|(\\u006E)))|((n|(\\u006E))(a|(\\u0061))(m|(\\u006D))(e|(\\u0065)))).*?{=}`
- `[\"\'][ ]*(([^a-z0-9~_:\'\" ])|(in)).+?(([.].+?)|([\[].*?[\]].*?)){=}`
- `[\"\'].*?{\)}[ ]*(([^a-z0-9~_:\'\" ])|(in)).+?{\(}`

More detailed information on the filters and how they work can be found at

- http://blogs.msdn.com/ie/archive/2008/07/02/ie8-security-part-iv-the-xss-filter.aspx
- http://blogs.msdn.com/dross/archive/2008/07/03/ie8-xss-filter-design-philosophy-in-depth.aspx
- http://blogs.technet.com/srd/archive/2008/08/19/ie-8-xss-filter-architecture-implementation.aspx

Simple Abuse Case

For the most part, this neutering mechanism is effective at blocking certain types of XSS attacks from occuring. However, altering a server's response before it gets rendered by the browser may have unintended consequences.

To begin with, the neutering mechanism can be abused by an attacker to block benign content on a page. For example, embedded JavaScript can be blocked from executing by "faking" a XSS attack. This is possible since the string **`<script`** will trigger one of the filters, if present in a request and in the response. This string naturally occurs on most webpages, so if an attacker appends a dummy GET parameter like **`&foo=<script`** then Internet Explorer 8 will see this on the outbound request and in the response body and thus trigger the neutering mechanism. An attacker may exploit this behavior in order to prevent client-side security functionality from working. For example, code to prevent framing (related to Clickjacking and UI Redressing attacks) can be disabled (though there are likely to be more direct ways to bypass such controls).

Another interesting consequence of blocking benign JavaScript embedded on a page is that the JavaScript code itself will be improperly interpreted by the browser as HTML. In most cases, this is not an issue but in certain cases can lead to XSS that wouldn't otherwise be possible. For example, consider a web page that persistently includes user-controlled content inside a JavaScript string while striping out single quote, double quote, and forward slash characters (but does not filter angle brackets). Then an attacker could inject the

string **<img src=x:x onerror=alert(0)>** . Normally, this would not be an exploitable scenario since the injected string cannot escape from the encapsulating JavaScript string quotes. However, forcefully neutering the opening script tag will force the script block to be parsed as HTML allowing the injected string to execute the alert.

Although such scenarios have been observed "in the wild", it is not very common that you run across persistent XSS into JavaScript strings that aren't exploitable through more traditional means. However, a couple of important facts can be stated about the filters at this point.

1. An attacker can abuse the filtering mechanism to alter how a page is rendered.
2. The filters can be abused to enable XSS in situations where it wouldn't otherwise be possible.

Universal XSS

With this knowledge, it is worthwhile to take another look at all the XSS filters to see if any of the other filters can potentially be abused like this. One will quickly note that the majority of the filters end up neutering opening tag names or event handlers so that a normal alphabetic character gets replaced with the **#** symbol. Note however that a few of the filters actually neuter an equal sign, **=**. Are there any scenarios where altering an equal sign can affect how the page is rendered? Most certainly!

Say an attacker is able to persistently inject content into quoted attributes on a webpage. Most websites allow this in one form or another and are safe from traditional XSS attacks since they filter out and/or encode certain characters such as quote characters and angle brackets. If an attacker were to neuter the equal sign immediately before such a persistent injection, then it becomes possible to have the injected attribute value be rendered not as an attribute value but as *a new attribute name-value pair*.

For example, consider an injection into the alt attribute of an image tag: **<img alt="[[injection here]]" src="x.png">**. Say the string **x onload=alert(0) x** is injected. Now consider the following two variations which differ by just one character:

- **<img alt="x onload=alert(0) x" src="x.png">** will not execute the alert.
- **<img alt#"x onload=alert(0) x" src="x.png">** will execute the alert! (Assuming the image is found.)

In this example, a space character (`%20`) is included in the injection to separate the **onload=alert(0)** portion of the injection. This allows it to be interpreted as a seperate attribute name-value pair. Note that other characters can be used as well such as new line characters and forward slashes (/). What is most important about such scenarios is that most websites allow such characters to be included in attributes since they pose no risk when accounting for traditional XSS attacks.

For this technique to work an attacker must trigger the neutering in the first place so that the **=** character can be replaced with the **#** character, like above. This can be done by identifying a **trigger string** that occurs naturally on the anywhere on the page before the persistent injection. Then the attacker can simply include a dummy GET parameter as part of the page request whose value is this trigger string (or a simplified and possibly URL-

encoded version that will still trigger the neutering) and submit the request.

Both of these filters can be used to neuter an equal sign:

- `[\"\'][ ]*(([^a-z0-9~_:\'\" ])|(in)).*?(((l|(\\u006C))(o|(\\u006F))(c|(\\u0063))(a|(\\u0061))(t|(\\u0074))(i|(\\u0069))(o|(\\u006F))(n|(\\u006E)))|((n|(\\u006E))(a|(\\u0061))(m|(\\u006D))(e|(\\u0065)))).*?{=}`

- `[\"\'][ ]*(([^a-z0-9~_:\'\" ])|(in)).+?(([.].+?)|([\[].*?[\]].*?)){=}`

Identifying a suitable trigger string (i.e. strings that will match one of these filters) requires detailed understanding of these regular expressions. In our field testing, most pages that allow persistent injections were exploitable by identifying a trigger string which matched against the second filter. To see why, consider the 5 main components of the second regular expression:

1. `[\"\'][ ]*`
2. `(([^a-z0-9~_:\'\" ])|(in))`
3. `.+?`
4. `(([.].+?)|([\[].*?[\]].*?))`
5. `{=}`

Part one of the regular expression matches a quote character followed by an arbitrary number of space characters (including no space characters at all).

Part two will match against the word "in" *or* any character that is not one of the following: alphanumeric, a quote character, ~, _, :, or a space. So characters like <, >, (, and) will each match (and many more).

Part three will match against *any* characters repeated *any number of times* (this is important). This match is non-greedy.

Part four has two components that can match. The first part is the more useful part for identifying trigger strings and will match against a . (a period, `%2e`) followed by *any* characters repeated any number of times (again, non-greedy).

Part five will match against the equal sign that we are neutering.

Putting this together, we can identify several common types of strings that would qualify as trigger strings. For example, something like **`">blah blah blah. blah blah blah <a href=`** or **`"image.png">blah blah<input value=`** suffice and will be found on most pages that have a suitable persistent injection. The key parts being that it starts with a quote character, includes a period surrounded by alphabetic characters somehow, and then ends with the equal sign that is being targeted for neutering.

In the worst case scenario though, a web page will not contain any text which matches one of the above filters. Such pages can still be exploited, provided there is a second injection point on the page somewhere before the persistent one. This may seem like an unlikely

scenario however in practice, it turns out to be fairly common. There are several reasons for this. First, this second injection point need not be persistent. Second, the second injection point can be anywhere on the page before the persistent injection and into any context. Third, and most importantly, any traditional XSS filtering/encoding will be useless since all that is needed to be injected is a "benign" looking string like **`location`** or **`foo.bar`** since such a string will provide the neccessary charcters needed to identify a suitable trigger string on the target page.

So putting it all together, we have the two common scenarios that allow an attacker to execute XSS on otherwise safe pages:
1) A persistent injection into a quoted attribute with a string similar to **`x onerror=alert(0) x`** located on a page that also contains a (naturally occurring) trigger string before the persistent injection.
2) A persistent injection into a quoted attribute with a string similar to "**`x onerror=alert(0) x`**" located on a page that also displays user controlled data (reflected or persistent) before the persistent injection.

Demonstration

Consider the following, utlra-simplified webpage located at **`http://0x.lv/attr.php`** which contains the following HTML:

```
<form method="GET" action="attr.php">
  <input name="name" value="test" type="text">
  <input type=submit>
</form>
```

The page contains a simple form that allows arbitrary data to be submitted. The server will persist the submitted data and redisplay it on subsequent views of the page until a new submission is submitted. Double quote characters are encoded preventing traditional XSS attacks. This page can be exploited in unpatched versions of IE8 by following these steps:

1. Navigate to the vulnerable page at **`http://0x.lv/attr.php`** .
2. In the input box, enter this string **`x style=x:expression(document.write(alert(0))) x`** and then click submit.
3. Navigate to **`http://0x.lv/attr.php`** again and verify that the string is persisted.
4. Identify the appropriate trigger string. In this example, **`"attr.php"><input name="name" value=`** will be sufficient.
5. Navigate to **`http://0x.lv/attr.php?foo="attr.php"><input+name="name"+value=`** (`foo` is a non-existent GET variable) and the alert will fire.

The following strings would also work as a persistent injection in step 2:

- `/style=x:expression(document.write(alert(0)))//`
- `/onmouseover=alert(0)//`
- `x onmouseover=location=name x`
- `x onmouseover=eval(location.hash.slice(1)) x`

Scope
Which websites are/were actually vulnerable to this issue? Well, just about every one we tested. For example:

- bing.com: http://p42.us/ie8xss/bing.png
- google.com: IE8 filters were disabled before we were able to get a screen grab
- wikipedia.org: http://p42.us/ie8xss/wikipedia.png (most other wikis that allow anonymous edits are vulnerable too)
- almost every bbcode forum/blog
- web based email services
- digg.com: http://p42.us/ie8xss/digg.png
- twitter.com: http://p42.us/ie8xss/twitter.png (along with most all sites that let you create a profile)
- and many many more…

Mitigations

How can this vulnerability be mitigated? There are a few different techniques that can be used to prevent these kinds of XSS attacks.

From the server side, you can protect your users by:

- Filtering all user-generated content so that, even if it is interpreted in a different context, it cannot execute.
- Utilizing site-wide anti-CSRF tokens that prevent any sort of XSS from being exploited in the first place.
- Disabling IE8s filters using the response header opt-out mechanism. There are obvious pros and cons to doing this, so consider your options carefully. Despite the serious vulnerabilities discussed in this paper, the filters do go a long way towards protecting IE8 users from traditional XSS attacks. Obviously, once users have upgraded to the patched version we strongly suggest you keep the filters enabled.

From the browser side, you can avoid any security risks by:

- Stop using the internets :)
- Disable the filters from within the browser until you upgrade to the patched version of IE8. Again, the filters aren't evil - they do help block a lot of standard XSS attacks.

Once the necessary patch has been applied, we highly recommend enabling the filters again.

Jeremiah Grossman has provided some excellent information to consider when deciding whether or not to disable the filters from the server. Essentially, if you're not confident that you've already eliminated XSS across your domain, then disabling the filters is a step in the wrong direction. Please find his specific guidance at http://jeremiahgrossman.blogspot.com/2010/01/to-disable-ie8s-xss-filter-or-not.html.

For future reference, it is our understanding that Microsoft is considering supporting a response header that will allow more fine-grained control over the neutering mechanism. For example, it may be possible, in some future version of Internet Explorer, to specify a header so that rather than performing selective neutering, the entire web page is blocked. This would be very useful in allowing website owners to manage the filter's behavior so as to match the risks posed to their applications and domains.

Acknowledgements

We would like to thank Gareth Heyes, Mario Heiderich, Alex K (kuza55) and the sla.ckers.org community for many brilliant ideas on web obfuscation and evasion. We would also like to thank Jack Ramsdell (MSRC) along with David Ross and the IE8 development team for being great to work with in resolving this issue.

For further information, please contact Eduardo and/or David using the contact information above.