Hunting Cross-Site Scripting Attacks in the Network

Elias Athanasopoulos, Antonis Krithinakis, and Evangelos P. Markatos
Institute of Computer Science
Foundation for Research and Technology - Hellas
N. Plastira 100, Vassilika Vouton, GR-700 13 Heraklion, Crete, Greece
{elathan, krithin, markatos}@ics.forth.gr

ABSTRACT

Cross-site Scripting (XSS) attacks in web applications are considered a major threat. In a yearly basis, large IT security vendors export statistics that highlight the need for designing and implementing more efficient countermeasures for securing modern web applications and web users. So far, all these studies are carried out by IT security vendors. The academic community lacks of the tools for performing similar studies for quantifying various properties of XSS attacks.

In this paper, we present xHunter, a tool that takes as input a web trace and scans it for identifying possible XSS exploits. xHunter does not provide any defenses against attacks in web applications and browsers. The tool is designed for processing thousands of URLs and isolating XSS exploits. Using xHunter one can see how real XSS exploits look like, what is the geographical distribution of web browsers that trigger XSS exploits, and other valuable properties, which if combined can draw a better picture of the XSS landscape today.

xHunter is based on two assumptions. The first one is that a significant fraction of XSS attacks is carried out using URLs and the second one is that these URLs contain parts that produce a valid JavaScript syntax tree with high depth. Thus, the basic operation of xHunter is to process URLs and identify parts that can be parsed in JavaScript. In this paper, we analyze all design choices and challenges for implementing xHunter. We evaluate a preliminary prototype of xHunter using about 11,000 URLs collected by a real-world XSS repository, XSSed.com, and 1,000 URLs collected from a monitoring point in an educational organization with about 1,000 users. The results suggest that xHunter has less than 3.2% of false negatives and about 2% of false positives.

1. INTRODUCTION

Cross-site scripting (XSS) attacks are considered as one of the major threats nowadays. Although XSS targets only web applications, the popularity of the exploitation method along with the prevalence of the web has made XSS a dominant threat in computer systems [26, 23]. Unfortunately, these statements can be officially supported and quantified only by large IT industry members involved in computer security. Undoubtedly, voluntary efforts from individuals exist, in order to keep track of the attack landscape of XSS. For example, XSSed.com [11] maintains a repository of XSS exploits. The amount of stored cases in XSSed.com is evidence that web sites are continuously threaten from XSS attacks. Although this particular XSS repository is invaluable to the research community, it can hardly assist in quantifying the real problem.

In this paper, we argue that the academic and research community lacks of the necessary tools for performing measurement studies and quantifying the threat constituted by XSS attacks. More precisely, we would like to have the tools for answering questions like the followings:

1. How often web sites are targeted with XSS attacks? Are XSS attacks a frequent phenomenon in every-day web traffic?
2. Which web sites are the targets?
3. Are there any orchestrated XSS campaigns in world-wide scale?
4. How are the *real* XSS exploits look like? XSSed.com maintains a large collection of vulnerable to XSS web sites along with their exploitation. However, many of the URLs listed contain proof-of-concept exploit code, like the use of the JavaScript `eval()` function to display a message, and not actual code that can cause harm.

To this end, in this paper, we propose xHunter, a tool that can passively monitor the network for identifying suspicious URLs. xHunter does not aim on providing any defenses against XSS, but rather collect statistics about them. The fundamental assumption behind xHunter is that XSS attacks based on JavaScript code injection (which is the most frequently encountered case of XSS) are carried out through URLs that contain a part that can produce a JavaScript-valid syntax tree. Unfortunately, JavaScript has a very relaxed syntax and is very context independent. Even simple text expressions that are usually encountered in URLs can produce a valid JavaScript syntax tree. In addition, web applications use their own encoding schemes. Attackers can take advantage of this by obfuscating the JavaScript exploit code.

In this paper, we present all challenges we encountered while designing xHunter along with a very preliminary prototype of the tool.

Organization. This paper is organized as follows. We present the architecture of xHunter in Section 2. We analyze in detail all major challenges xHunter has to deal with in Section 3. In Section 4 we present a preliminary evaluation of xHunter using a sample of about 11,000 malicious URLs collected from XSSed.com and a sample of 1,000 URLs collected from a monitoring point in an educational organization with about 1,000 users. We present related work in Section 5 and we, finally, conclude in Section 6.

2. XHUNTER ARCHITECTURE

In this section we present the basic architecture of xHunter. We first provide a short background of XSS attacks and then we phrase the two basic assumptions that are fundamental for the tool's design. We finally present the work flow of an xHunter run using a real example as input taken from XSSed.com.

2.1 Overview

An XSS attack involves the injection of some client-side code in the existing legitimate code of a web application. The malicious code is executed while the user is interacting with the web application. The user's web browser along with rendering the legitimate code of the web application, renders also the injected code. The user's web browser can be essentially compromised, since the injected code can steal cookies or force the web browser to perform various actions on behalf of the user.

There are basically two large categories of XSS attacks: (a) reflected and (b) stored. During a reflected XSS attack the injected code is placed in a URL. Upon the user clicks on the malicious URL the injected code executes. On the other hand, during a stored XSS attack, the adversary injects the malicious payload in some form of storage utilized by a web application. For example, consider a web application that handles a blog engine and stores all data associated with the blog in a database. An attacker can post an article which encapsulates the malicious code. This code is rendered in the user's browser upon the user's browser renders the blog article. Notice, that even in the case of stored XSS, the attack has been injected using a URL, since this is the only way the attacker can communicate with the web application.[1]

[1]XCS attacks that are mounted through another non-web channel are out the scope of this paper [9].

Assumption 1. The first fundamental assumption for xHunter is that XSS attacks are carried out through the transmission of URLs that contain the malicious code.

The malicious code during an XSS attack is usually expressed in JavaScript. There are plenty of other methods for exploitation, like injections of an `iframe`, redirection or leveraging of other client-side technologies such as Flash, injections in file uploads [5] or Phishing [10]. xHunter focuses only in cases where the web attack is triggered via a URL that contains JavaScript. These cases constitute a significant fraction of XSS attacks.

Assumption 2. The second fundamental assumption for xHunter is that an XSS attack is mounted using a URL which contains a part that can produce a valid JavaScript parse tree.

Based on the two assumptions above, xHunter works roughly as follows. It takes as an input a URL. It scans the input for parts that can produce a valid JavaScript parse tree. The tool embeds the JavaScript engine of Mozilla SpiderMonkey [1] for generating parse trees. A URL is considered suspicious if a part produces a JavaScript parse tree with a certain *depth*.

For example, consider the JavaScript snippet in Figure 1. xHunter can process the source and produce the syntax tree depicted in Figure 1. Notice the indentation of the parse tree. It is slightly different than the actual parse tree a JavaScript parser can export. The reason is that xHunter assigns to different parse nodes (i.e. tokens the parser consumes) a different *weight*. This weight is depicted in the figure as indentation level. We further analyze this later in Section 3. Notice also that xHunter does not evaluate the actual code. It is not aware of neither the DOM [15] structure nor the code of the web application the malicious code is trying to exploit. xHunter only checks if part of an input can produce a valid JavaScript syntax tree.

2.2 Operation

We now proceed and present a hypothetical xHunter run. xHunter takes as input a URL. First, the URL is HTML escaped (for occurrences of HTML encoded entities like `&apos;`) and is URL-decoded. Then, the part of the URL which follows the "?" character, usually named as the *query string*, is isolated from the rest of the URL. The query string contains all parameters which take part in an HTTP GET request. [2] All parameters are separated using the "&" character. Each parameter is in the form `key = value`. For each pair, xHunter tries to parse both the key and the value (if they both exist). Apparently, there are XSS attacks that host the exploit code in the key and not in the value, as it is more common [2].

[2]A similar approach is used for POST requests. In this case the query parameters are part of the HTTP request.

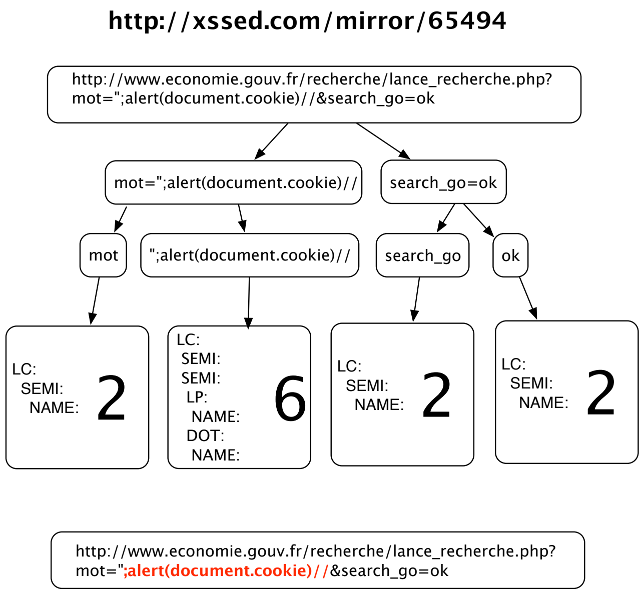
```
 1  if (user_logged()) {
 2    alert(document.cookie);
 3  }
 4
 5  LC:
 6   IF:
 7    LP:
 8     NAME:
 9    LC:
10     SEMI:
11      LP:
12       NAME:
13      DOT:
14       NAME:
```

Figure 1: Parse tree of a JavaScript snippet as produced by xHunter.

There are two possible cases when xHunter tries to parse a particular field. The field does not parse or the field produces a syntax tree with a certain depth. If the depth exceeds a certain value, the URL is considered suspicious. The depth of the syntax tree depends on the tokens that compose the field's text. If there is actual JavaScript code in the field then the syntax tree is expected to have a depth of a high value. Notice, for example, in Figure 1 that the code responsible for the `alert()` code contains six different tokens (lines 9-14). However, even simple text can produce a valid JavaScript syntax tree. Hopefully, real and possibly functional JavaScript code has higher probability to include certain tokens. Thus, each token is assigned with a weight and the overall depth is computed using each node's weight. We further discuss this in Section 3.

A theoretical xHunter run is depicted in Figure 2. The running example is a URL from XSSed.com that contains XSS attack code. Notice, how the query string and then the parameters are isolated from the original URL. Observe, also, that every field, in this particular case, produces a valid JavaScript syntax tree. Nevertheless, the attack code produces the syntax tree with the higher depth (6). Notice the final URL where the attack code has been highlighted. The first character (double quotes) is not highlighted, although it is part of the field that contains the attack. This character has been omitted, since it produces a syntax error if included. We further discuss how we deal with partial JavaScript expressions in Section 3.

There are plenty of cases xHunter, as presented here, fails to handle. First, web applications support URLs that do not fully conform to the specification [8]. Second, JavaScript has a relaxed syntax and thus simple text can produce a valid syntax tree. Three, the XSS exploit code can be partial or mixed up with other irrelevant text. We proceed and present all these issues in detail in the following section. For each problematic case we present how xHunter is enhanced to solve each issue.



http://xssed.com/mirror/65494
http://www.economie.gouv.fr/recherche/lance_recherche.php?mot=";alert(document.cookie)//&search_go=ok
mot=";alert(document.cookie)//
search_go=ok
mot
";alert(document.cookie)//
search_go
ok
LC: SEMI: NAME: 2
LC: SEMI: SEMI: LP: NAME: DOT: NAME: 6
LC: SEMI: NAME: 2
LC: SEMI: NAME: 2
http://www.economie.gouv.fr/recherche/lance_recherche.php?mot=";alert(document.cookie)//&search_go=ok

Figure 2: Example operation of a hypothetical xHunter run.

3. CHALLENGES

We now proceed and present various challenges that xHunter has to deal with. We focus in the three main problems we mentioned in Section 2: (a) web applications quirks, (b) JavaScript relaxed syntax and (c) exploit isolation. We discuss each of these in detail. We present URL examples taken from XSSed.com for each particular case.

3.1 Web Application Quirks

Web applications use URLs for communicating with web browsers. There is a well defined specification for URLs [8]. However, web applications often use their own encoding scheme or custom delimiters for separating the query string and the parameters from the rest of the URL. There are cases where xHunter can try alternative methods for parsing a URL. For example, a significant amount of URLs contained in the XSSed.com repository use "!", instead of "?" for separating the query string from the URL. In other cases there is no special delimiter used at all, thus xHunter, inevitably, scans the whole URL for JavaScript injections.

However, there are cases where xHunter cannot handle. These are cases where the URL is encoded using a custom scheme by the web application. In Figure 3 we present two example URLs, which contain exploit code. In the first case a similar, but not identical, to

```
[http://xssed.com/mirror/55309]

http://www.metacrawler.com/
metacrawler/ws/results/Web/
!3Cscript!3Ealert(!2FXthe _
  miller!2F)!3C!2Fscript!3E/1/41

[http://xssed.com/mirror/64043]

http://www.turktelekom.com.tr/tt/
portal/!ut/p/c0/XYzBCoJAFEX_RQhq
9Z5a0oEI..RshwIQj/
```

Figure 3: Custom encoding used by web applications. In the second example some parts are omitted for better presentation.

URL encoding is used (the character `!` is used instead of `%`). This case can be handled if the scheme is used frequently (we found just a few cases in the sample collected from XSSed.com). The second case is impossible to handle, since the web application encodes all URLs in a scheme known only by itself. Hopefully, in more than 10,000 of URLs collected from XSSed.com there are only a few cases that follow this paradigm.

3.2 JavaScript Relaxed Syntax

JavaScript has a very relaxed syntax. It is quite possible for a text fragment to have a valid JavaScript syntax tree. For example, consider the two expressions in Figure 4. They are parts taken from a sample of benign URLs and both of them produce valid syntax trees with high depth. However, these expressions are not JavaScript exploits. xHunter employs two techniques in order to deal with such cases: (a) the reverse code heuristic and (b) weighted parse nodes. We proceed and analyze both techniques.

3.2.1 *Reverse Code Heuristic*

Expressions like the first one listed in Figure 4 have the following property. First, they can be parsed from left to right and from right to left. Second, they produce a syntax tree with the same depth, no matter the parsing direction. On the other hand, JavaScript exploit code is highly unlikely to produce even a valid syntax tree if parsed from right to left. Thus, xHunter whenever finds an expression that produces a valid syntax tree, attempts to parse the expression *reversed*. The expression is not considered suspicious if the reversed expression produces a syntax tree with the same depth. This heuristic solves many cases in the sample of URLs collected from XSSed.com, which otherwise would be considered as false positives.

```
foo;1,2,3,4,5

LC:
 SEMI:
  NAME:
 SEMI:
  COMMA:
   NUMBER:
   NUMBER:
   NUMBER:
   NUMBER:
   NUMBER:

id=331653;t=49;l=1

LC:
 SEMI:
  NUMBER:
 SEMI:
  ASSIGN:
   NAME:
   NUMBER:
 SEMI:
  ASSIGN:
   NAME:
   NUMBER:
```

Figure 4: JavaScript has a relaxed syntax. Even simple text fragments can produce a syntax tree with high depth.

3.2.2 *Weighted Parse Nodes*

Parse nodes which refer to tokens such as "." (`DOT`) and "+" (`PLUS`), for example, can be repeated several times in an expression and thus result to parse nodes that contribute to the final syntax tree's depth. These tokens occur frequently in URLs, without being part of a JavaScript code snippet. On the other hand, there are tokens that are more likely to be part of valid JavaScript code, such as the `LP` token, which denotes a left parentheses occurrence. These tokens occur less frequently in URLs and much more frequently in JavaScript code.

xHunter assigns a weight to each parse node. The overall depth of the parse tree is a weighted contribution of all parse nodes. There are nodes that have no contribution at all (such as the `DOT` token), nodes that have negative contribution (such as the `NAME` token) and nodes that have high contribution (an `LP` token has double contribution).

3.3 Exploit Isolation

It is highly likely that a JavaScript exploit is not isolated in a URL parameter. For example, consider the example in Figure 5. The value of the first query string

```
[http://xssed.com/mirror/65494]

http://www.economie.gouv.fr/
recherche/lance_recherche.php?
mot=";alert(document.cookie)//
&search_go=ok
```

Figure 5: XSS code can be found partial in URLs. In this example the "`"`" character (double quotes) must be omitted in order for the rest of the code to produce a valid syntax tree.

parameter is `";alert(document.cookie)//`. This fragment does not parse as is. However, when the target web page is rendered the fragment is attached to an existing JavaScript expression and the exploit code runs. Multiple parsing attempts must be carried out, removing characters from left and right until a valid JavaScript expression is isolated, in order to detect this code. xHunter starts parsing from left to right. If the result does not produce a valid syntax tree, xHunter reduces the parsing window by removing a character from the left. If all characters are consumed, then xHunter reduces the whole expression by removing a character from the right, the window is set to the initial size (minus the removed character) and the whole process restarts. This strategy results in a high computational overhead. However, as we have already stated, xHunter is not meant to be a defense mechanism against XSS attacks that needs to run in real-time. xHunter is designed to process large web traces for exporting statistics related to XSS attacks.

4. CASE STUDY

In order to perform a preliminary evaluation of xHunter we test the tool with two samples of URLs. The first one is collected from XSSed.com, the largest XSS repository with public access. It contains about 11,000 URLs which contain XSS attacks and target real web sites. The second sample contains 1,000 URLs collected from a monitoring point in an educational organization with about 1,000 users.

4.1 XSSed.com

XSSed.com is a public XSS repository. It contains about 11,000 XSS attacks as of March 2010. The repository classifies all attacks in two categories. One named "XSS" and the other "Redirection" (or "Frame Redirection"). The latter involves URLs that, when clicked, perform a redirection from the target web site to a web site of the attacker's choice. However, we found many cases that were misclassified. Many URLs that perform a redirection are listed under the "XSS" category. xHunter has not been designed to detect redirection or `iframe` injection. Unfortunately, there is no way to filter out all these redirection URLs from the original set.

In Table 1 we list properties of the URLs marked as suspicious. Observe, that xHunter succeeds in identifying 8,204 (out of 10,535) URLs, which are known to be XSS exploits. Moreover, the most popular exploitation technique is by using the `alert()` function. However, as we have already stated, XSSed.com is a repository holding proof-of-concept attacks and not real ones. Nevertheless, there are also other popular JavaScript constructs used, like `document.cookie`, `document.write()` and `String.fromCharCode()`.[3] Finally, xHunter records 274 cases where none of the above constructs is used. For example, there are cases where the exploit code contains: `<img onload=xss()>`.

In Table 2 we list properties of the URLs marked as clean. Overall, xHunter marks 2,331 as not suspicious. Since, all URLs are collected from XSSed.com these are false negatives. However, this is not actually the case. The sample contains redirections, `iframes` and `<script>` elements that include JavaScript source code from a third party web site. xHunter has not been designed to deal with these cases. We also include in the trace 779 URLs that use HTTP POST instead of HTTP GET for attacking the web application. XSSed.com provides this information so we can distinguish all URLs that use POST from URLs that use GET. The operation of xHunter does not change in the case that has to process a POST request. The task of the tool is easier since the parameters (included in the POST part) are easily isolated from the rest of the URL. However, we leave on purpose all these URLs inside the trace to verify that none of them will be marked as suspicious. Indeed, xHunter successfully marks all 779 POST URLs as clean.

By subtracting all the above cases, 268 XSS exploits remain marked as clean. With manual examination we find out that these exploits are mixed. Some of them are redirection attacks (but classified as "XSS" and not "Redirection"), some other take advantage of web application quirks we discussed in Section 3. Thus, we believe that xHunter has less than 3.2% false negatives.

4.2 Benign URLs

We run xHunter with a second trace, which contains 1,000 URLs collected from a monitoring point in an educational organization with about 1,000 users. All URLs are considered benign. xHunter marks 20 URLs as suspicious. That is, xHunter has about 2% of false positives. We manually examine the 20 false positives. All 20 cases are from URLs that use the `‘‘;’’` character, which is significant in JavaScript, as a delimiter for

[3]These categories overlap. For example it is possible for a URL to contain the code: `alert(document.cookie)` or `document.write(document.cookie)`.

Occurrences	8,204
`alert()`	7,895
`document.cookie`	1013
`String.fromCharCode()`	550
`document.write()`	50
Other	274

Table 1: XSSed.com sample. URLs marked as suspicious.

Occurrences	2,331
Redirection	611
`iframe`	292
Redirection and `iframe`	7
`<script src="">`	389
`<img src="">`	39
POST	779
Other	268

Table 2: XSSed.com sample. URLs marked as clean.

query parameters instead of ``&''.

xHunter can be easily enhanced to treat both ``;'' and ``&'' characters as delimiters that separate parameters in a query string. We modify the tool and we rerun all experiments. The 20 URLs in the benign trace are not marked as suspicious. The results for the trace collected from XSSed.com are not altered by this modification.

5. RELATED WORK

The field of XSS attacks has attracted the interest of the research community over the last few years. There are plenty of efforts towards the design of schemes that aim on protecting web browsers from injections [14, 12, 17, 27, 25, 21, 4]. xHunter differentiates from all these schemes, since it is not designed for providing any defenses, but statistics about XSS attacks. xHunter can be considered as a defense mechanism that resembles the behavior of an IDS, like Snort [22]. Snort has signatures for XSS attacks but these signatures can hardly cope with obfuscation techniques [13].

xHunter can monitor web attacks that are carried out in URLs using JavaScript. However, there are plenty of attacks that xHunter is not designed to deal with. First, xHunter cannot capture XSS attacks carried out through file uploads [5], since it can monitor only URLs. Second, xHunter cannot capture XCS attacks [9]; an XSS flavor which is carried out through a non-web channel. Third, exploitation through web client vulnerabilities [24] can be in principle captured, if the exploit is carried out through JavaScript embedded in a URL. The same holds also for CSRF [6] attacks. Finally, xHunter cannot capture web attacks carried out through SQL injection [3] or the well known drive-by download attacks [19].

Network monitors have been also developed for injections in native code. For example nemu [18] attempts to execute, using emulation, all network traffic in order to identify a shellcode. In a similar fashion, xHunter attempts to parse all parts of URLs, in order to identify XSS exploits.

Finally, Noscript [16] and noXSS [20] aim on identifying script injection in the web browser environment. These tools utilize similar techniques with xHunter. Especially, noXSS, like xHunter, implements a complete JavaScript parser for that purpose. It has been shown that these techniques are not efficient [7] due to high false positive and false negative rates. However, xHunter is not meant to be treated as a defense mechanism, but rather as an XSS network monitor. Thus, xHunter can be tolerant in false positives and false negatives (although as we show in Section 4, a preliminary prototype of xHunter does not exceed high false positive/negative rates).

6. CONCLUSION

In this paper we present xHunter. A tool that is designed for processing large web traces and isolating suspicious URLs that contain XSS attack exploits. xHunter tries to identify parts contained in a URL that produce a valid JavaScript parse tree. If a fragment produces a syntax tree of a certain depth, then the URL is considered suspicious.

xHunter is not meant for providing defenses against XSS attacks. It rather assists in collecting properties for performing studies related to XSS attacks. For example, xHunter can process network traces collected from various sensors distributed all over the world and give answers to questions like *How often web sites are targeted with XSS attacks?* or *Which web sites are the targets?*.

Throughout this paper, we analyze all technical challenges concerning the implementation of xHunter. Last but not least, we perform a short evaluation using a preliminary prototype of the tool. We process about 11,000 URLs that contain XSS exploits, collected from XSSed.com, and 1,000 benign URLs collected from a monitoring point in an educational organization with about 1,000 users. The results suggest that xHunter has less than 3.2% of false negatives and about 2% of false positives.

7. ACKNOWLEDGEMENTS

Elias Athanasopoulos, Antonis Krithinakis and Evangelos P. Markatos are also with the University of Crete. Elias Athanasopoulos is funded by the Microsoft Research PhD Scholarship project, which is provided by Microsoft Research Cambridge.

8. REFERENCES

[1] SpiderMonkey (JavaScript-C) Engine. `http://www.mozilla.org/js/spidermonkey/`.

[2] XSS exploit in key example. `http://xssed.com/mirror/33541/`.

[3] C. Anley. Advanced SQL injection in SQL server applications. *White paper, Next Generation Security Software Ltd*, 2002.

[4] E. Athanasopoulos, V. Pappas, A. Krithinakis, S. Ligouras, and E. P. Markatos. xJS: Practical XSS Prevention for Web Application Development. In *Proceedings of the 1st USENIX WebApps Conference*, Boston, US, June 2010.

[5] A. Barth, J. Caballero, and D. Song. Secure Content Sniffing for Web Browsers or How to Stop Papers from Reviewing Themselves. In *Proceedings of the 30th IEEE Symposium on Security & Privacy*, Oakland, CA, May 2009.

[6] A. Barth, C. Jackson, and J. C. Mitchell. Robust Defenses for Cross-Site Request Forgery. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)*, 2008.

[7] D. Bates, A. Barth, and C. Jackson. Regular Expressions Considered Harmful in Client-Side XSS Filters. In *Proceedings of the 19th international conference on World Wide Web (WWW)*. ACM New York, NY, USA, 2010.

[8] T. Berners-Lee, L. Masinter, and M. McCahill. RFC 1738: Uniform Resource Locators (URL), 1994. `http://www.ietf.org/rfc/rfc1738.txt`.

[9] H. Bojinov, E. Bursztein, and D. Boneh. XCS: Cross Channel Scripting and Its Impact on Web Applications. In *CCS '09: Proceedings of the 16th ACM conference on Computer and communications security*, pages 420–431, New York, NY, USA, 2009. ACM.

[10] R. Dhamija, J. Tygar, and M. Hearst. Why Phishing Works. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 581–590. ACM New York, NY, USA, 2006.

[11] K. Fernandez and D. Pagkalos. XSSed.com. XSS (Cross-Site Scripting) information and vulnerable websites archive. `http://www.xssed.com`.

[12] M. V. Gundy and H. Chen. Noncespaces: Using Randomization to Enforce Information Flow Tracking and Thwart Cross-Site Scripting Attacks. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 8-11, 2009.

[13] R. Hansen. XSS (Cross-Site Scripting) Cheat Sheet. Esp: for filter evasion. `http://ha.ckers.org/xss.html`.

[14] T. Jim, N. Swamy, and M. Hicks. Defeating Script Injection Attacks with Browser-Enforced Embedded Policies. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 601–610, New York, NY, USA, 2007. ACM.

[15] A. Le Hors, P. Le Hegaret, L. Wood, G. Nicol, J. Robie, M. Champion, and S. Byrne. Document Object Model (DOM) Level 3 Core Specification. *World Wide Web Consortium, Recommendation REC-DOM-Level-3-Core-20040407*, 2004.

[16] G. Maone. Firefox add-ons: Noscript, 2006. `https://addons.mozilla.org/en-US/firefox/addon/722`.

[17] Y. Nadji, P. Saxena, and D. Song. Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 8-11, 2009.

[18] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. Emulation-based detection of non-self-contained polymorphic shellcode. In C. Krügel, R. Lippmann, and A. Clark, editors, *RAID*, volume 4637 of *Lecture Notes in Computer Science*, pages 87–106. Springer, 2007.

[19] N. Provos, P. Mavrommatis, M. Rajab, and F. Monrose. All your iFRAMES point to us. In *Proceedings of the 17th conference on Security symposium*, pages 1–15. USENIX Association, 2008.

[20] J. Reith. Firefox add-ons: noXSS, 2008. `https://addons.mozilla.org/en-US/firefox/addon/9136`.

[21] W. Robertson and G. Vigna. Static Enforcement of Web Application Integrity Through Strong Typing. In *Proceedings of the 18th USENIX Security Symposium*, Montreal, Quebec, August 2009.

[22] M. Roesch et al. Snort. *The Open Source Network Intrusion System. Web page at `http://www.snort.org`*, 1998.

[23] SANS Insitute. The Top Cyber Security Risks. September 2009. `http://www.sans.org/top-cyber-security-risks/`.

[24] P. Saxena, S. Hanna, P. Poosankam, and D. Song. FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS)*.

[25] R. Sekar. An Efficient Black-box Technique for Defeating Web Application Attacks. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS)*,

San Diego, CA, Feb. 8-11, 2009.
[26] Symantec Corp. April 2008. 1-3. Retrieved on 2008-05-11. Symantec Internet Security Threat Report: Trends for July-December 2007 (Executive Summary).
[27] M. Ter Louw and V. Venkatakrishnan. Blueprint: Precise Browser-neutral Prevention of Cross-site Scripting Attacks. In *Proceedings of the 30th IEEE Symposium on Security & Privacy*, Oakland, CA, May 2009.