



The Interdisciplinary Center, Herzliya
Efi Arazi School of Computer Science

Enhancing JavaScript Engine to Detect and Prevent XSS

M.Sc. Thesis

Submitted by Tsvi Cherny-Shahar

Under the supervision of Dr. David Movshovitz

March 2014

Acknowledgments

I would like to express my deep sincere gratitude to my supervisor, Dr. David Movshovitz, for his continues support in my Thesis research. His expertise, vast knowledge, deep understanding, patience and guidance helped me throughout the whole research and writing of this thesis. As I came to acknowledge, research is far from being anything like a regular linear study, but a road that goes up and down, back and forth. David's guidance and insight have been invaluable throughout this journey, his persistence and understanding turned me into a better researcher and turned this research into a thesis.

I also want to express my deep gratitude to Prof. Gadi Taubefeld and Prof. Anat Bremler-Barr for their encouragement, support, help and advices in so many different aspects of my academic life throughout the years since my 1st degree. I consider myself lucky to get the chance to work and study from them. Their help support were part of the reasons which lead me to pursue my masters degree.

I want to thank Prof. Yael Mozes for her time, support and encouragement. Her help saved me quite a few precious nights.

I want to thank my parents for raising, helping and supporting my own path in life, even when the phone bill was huge and the line was always busy back in the dial-up days.

I want to thank my parents-in-law and brother-in-law for their tremendous on going love, help and support, their help is far from being obvious.

Last but most certainly not least I want to specially thank my incredible wife Nohar. She has a big part of this thesis for without her unconditional love, help, support, time and “problem-solving” chats when I’m stuck, I would have never achieved this goal, or many others in my academic, professional and personal life. Thank you!

Abstract

Cross Site Scripting (or XSS) attacks are one of the most popular web-application browser side attacks. This attack is using the vulnerability that the (user) input to the web application is not validated by the web application before being used in generating the response returned to the user. The XSS attacks use vulnerabilities in the web-application code to inject malicious code that runs within the victim browser and can cause various malicious activities such as theft of passwords, theft of user sessions (cookies), scan the end user network, send requests to the web application as if the victim made them (XSRF), etc. Although the vulnerability is known for a relatively long time it still considered as one of the top web-application threats [20].

Most of the current mitigation techniques were based on negative security logic, i.e. try to detect malicious input by detecting patterns that characterize XSS attacks in the input sent to the web application. These detection mechanisms are subjected to numerous evasion techniques, including JavaScript creating new JavaScript and several others, and as a result have limited effectiveness, and have been bypassed many times by attackers. Another mitigation technique requires the developer to encode/escape any input included in the web page. This encoding should be done according to the position of the input in the web page. Since it relies on the developer manual work it is subjected to human errors, and history has proven that these errors happen frequently.

Modern browsers allow developers to develop plug-ins and add-ons to their browsers. Some of these add-ons and plug-ins use the browser's JavaScript engine to execute JavaScript code. For example, Adobe flash animation can use JavaScript code that will be executed on the browser, or an add-on that uses JavaScript to block ads from websites. For example, in 2007

adobe reported [1] that SWF files (Flash animation) can perform XSS attack by embedding JavaScript commands inside the SWF itself. Current detection mechanisms and filters focus on HTML/XML responses to look for JavaScript, but not inside SWF files or other file types that might embed JavaScript.

Our solution detects and prevents automatically and accurately XSS attacks, by detecting the XSS attack in the JavaScript engine before the attack took place. By locating the XSS detection mechanism in the JavaScript engine there is no need for the solution to learn how to look for JavaScript. In addition, due to this location for the XSS detection and prevention mechanism, it is not subjected to all the evasion techniques used to bypass previous XSS filters and can handle all the XSS types of attacks: stored, reflected and DOM. If the embedded JavaScript is being executed it will reach the detection mechanism on its own, including JavaScript that has been generated by execution of another JavaScript.

Our XSS detection and prevention mechanism is primarily based on a positive security logic (white list), which is known to be more secure (allows only known and approved scripts to run). To be able to use positive security logic to detect and prevent XSS required to be able to validate that each script is legal and this can't be done using the script itself. Thus, we have design and implemented a generalizing mechanism on the compiled script, such that all variants of a legal script are generalized to one generalized assembly version. This enables us to create a whitelist of the signatures of the generalized assembly scripts. In some cases where the white list is not updated our mechanism falls back into using negative security logic. This negative logic mechanism is also using the generalized assembly scripts to detect malicious scripts. This is done by detecting during the generalization process certain operations that are usually used by malicious scripts. These operations are tagged, and in case the script is unknown, we check if these tagged operation are common in the web application. Thus, small changes to scripts in the web application, will not be detected as a malicious script, and unknown script that the only change using these tags define unknown scripts that doesn't contain any harmful operation will not be blocked as well. Thus, operation will be blocked, until the script gets into the white list. To summarize, our unique location for the XSS detection and prevention mechanism,

combined with the fact that it works on the generalized assembly version of the script, and the combination of both positive logic and negative logic enables us to block all XSS attack with very low risk of false positives. After 3 months of evaluations of our mechanism we have encountered only got only 4 false positives for all 33 Alexa top websites.

Table of Contents

Acknowledgments	i
Abstract	iii
Table of Contents	vi
1 Introduction	1
2 Known XSS Attack Vectors	6
2.1 Reflected (or Non-persistent)	6
2.2 Stored (or persistent)	8
2.3 DOM-Based XSS	9
3 Popular Industry Solutions	11
3.1 Encoding	11
3.1.1 Input encoding	11
3.1.2 Output encoding	12
3.2 Content-filtering / Input validation	13
3.2.1 Limitations of positive logic approach	15
3.2.2 Limitations of negative logic approach	15
4 Evasions	17
4.1 Hiding JavaScript by encoding	17
4.2 JavaScript extraction from HTML evasion	19
4.3 Browsers' parser and engines error tolerance	20

TABLE OF CONTENTS

- 4.4 JavaScript code generating JavaScript code evasion 21
- 4.5 JavaScript in plug-ins, add-ons and other filetypes 22
- 5 Previous & Related Work 24**
- 5.1 Automatic sanitization frameworks 24
- 5.2 Browser based detection and prevention mechanisms 25
- 6 Key Concepts 29**
- 6.1 Detection of XSS attack within the JavaScript engine 30
 - 6.1.1 Solution advantages 31
- 6.2 Positive approach 32
 - 6.2.1 Limitations of naïve solution 32
 - 6.2.2 Positive logic with GA 33
- 6.3 Negative logic fallback 38
- 6.4 Detection & Prevention 40
- 7 Learning & Publishing 43**
- 7.1 The generalization process of JSA 43
 - 7.1.1 Engine specific implementation 44
- 7.2 Learning LGAS of a web application 45
- 7.3 Solution flow charts 46
 - 7.3.1 Learning flow 46
 - 7.3.2 Detection & Prevention Flow 47
- 7.4 Publishing GAS to browsers 48
- 8 Firefox SpiderMonkey POC Implementation 50**
- 8.1 Components Overview 51
 - 8.1.1 mozjs.dll 51
 - 8.1.2 profmonkey.dll: 51
- 8.2 POC execution example 53

TABLE OF CONTENTS

8.2.1	Normal execution	54
8.2.2	Learning mode	56
8.2.3	Detection and prevention mode	56
9	Evaluation	59
9.1	Learning Process	59
9.2	Accuracy Measurements	61
9.2.1	Testing for false positives	61
9.2.2	Testing for false negatives	63
9.3	Performance measurements	63
10	Summary	65
	Appendices	67
A	Prevention Testing of JavaScript code	68
B	Attacks Prevented	70
C	Learned Websites List	72
	Bibliography	75

Chapter 1

Introduction

Back at 1995, at the days of the early web, when blink and marquee tags were still "cool", and the battle between Netscape and Microsoft was still raging. Netscape considered their client-server solution as a distributed OS, running a portable version of Sun's Java which was a competitor of C++ and aimed at

Anyway, I know only one programming language worse than C and that is JavaScript. [...] I was convinced that we needed to build-in a programming language, but the developers, Tim first, were very much opposed. It had to remain completely declarative. Maybe, but the net result is that the programming-vacuum filled itself with the most horrible kluge in the history of computing: JavaScript.

Robert Cailliau [29]

professional programmers. Netscape wanted a lightweight interpreted language that would complement Java by appealing to nonprofessional programmers, like Microsoft's VB. Brendan Eich designed the lightweight language under the codename Mocha, which officially released in the beta version of Netscape Navigator 2 under the name LiveScript, which finally announced and deployed at December 4th 1995 as JavaScript in Netscape Navigator 2.0B3. The effect of the JavaScript was huge, and of course Microsoft was quick to adopt the language at August 1996 in Internet Explorer 3.0 [30].

Since then JavaScript changed the face of the World Wide Web, but behind the scene the world

of web application security changed forever, when a new kind of code injection with countless attack vectors verged into the virtual world.

In December 1999 at Microsoft, 4 years since JavaScript released officially, David Ross worked on a code flaws found by Georgi Guninski. David demonstrated that he can inject code into the web page where the fault is on the server side, and not the client side (the internet explorer). David entitled the Microsoft-internal as "Script Injection". Finally Microsoft met with the Computer Emergency Response Team (CERT) which produced the first CERT advisory about cross-site scripting (or XSS) [9] [5]. The CERT advisory describes 2 types of XSS, reflected (a.k.a non-persistent) and stored (a.k.a persistent).

During July of 2005, Amit Klein published an article called "DOM Based Cross Site Scripting" or "XSS of the Third Kind" [14]. As the name suggests, DOM-Based is the 3rd type of XSS attack that has been uncovered, which its unique attack vector makes it harder to detect and prevent. We will discuss this type of XSS as well as the first two in chapter 2.

Up to 2005, although XSS has been around for a decade and known officially for half a decade, no real solution has been found to the problem. More than that, the main focus of the security experts and developers was on buffer overflows, viruses, worms, spywares etc. The approach of the industry regarding XSS was that it is a rather harmless, simply because XSS executes JavaScript which used to be considered harmless. All of this changed overnight In October 4th, 2005 when "Samy" started to run loose. [9]

Samy is the first XSS worm ever developed (and discovered). Samy propagated through the biggest social-network at the time - MySpace. Samy spread to over **1 million users within 20 hours** making it the fastest spreading virus of all times [31]! In other words, MySpace was shutdown overnight.

Samy used a simple XSS attack to propagate [13]. In order to block code injections MySpace blocked many tags that users can use in the posts they made on MySpace websites. In fact they allowed only ``, `<a>` and `<div>` to make sure no tags like `<script>` would be available. But MySpace developers missed that the style attribute of div tag allows placing a URL for the background, so the attacker can place a URL of JavaScript code (a.k.a inline JavaScript

code) and evade their posting security policy:

```
<div style='background:url('javascript:alert('xss'))''>
```

At this point the attacker can write a post on someone's page and include this tag that contains JavaScript code (this is a stored (or persistent) XSS attack).

The malicious code Samy injects has done 2 things:

1. Post the malicious message to all of my friends, causing all the friends to execute the code as soon they read the post.
2. Send a friend request to Samy's author, Samy Kamkar. Screenshot of his friend request count during the attack can be found at: <http://namb.1a/popular/> (last visited 22/3/2013).

Samy's XSS attack shook the security world and massive research on JavaScript-based attacks began and by early 2006 researchers were able to create JavaScript based port-scanners, intranet hacks, browser keylogger, session hijacking, trojan horses and even browser history stealer [9]. All of the above shows that XSS is a dangerous vulnerability with very high risk to the users and the website itself.

In 2007 XSS is considered the top web application vulnerability [19], while CVE (Common vulnerabilities and Exposures) report concludes that XSS is one of the most commonly reported types of vulnerability since 2005 [26]: "Buffer overflows were number 1 year after year, but that changed in 2005 with the rise of web application vulnerabilities, including cross-site scripting (XSS)...". CVE continues and explains several contributing factors for this change:

- "The most basic data manipulations for these vulnerabilities are very simple to perform, e.g. [...] `<script>alert('hi')</script>` for XSS. This makes it easy for beginning researchers to quickly test large amounts of software." In other words, it is easy to perform or test if a given web application is vulnerable to the attack, which means that XSS attackers do not have to be computer-experts or keen programmers.
- "With XSS, every input has the potential to be an attack vector, which does not occur

with other vulnerability types. This leaves more opportunity for a single mistake to occur in a program that otherwise protects against XSS.”

- ”Despite popular opinion that XSS is easily prevented, it has many subtleties and variants. Even solid applications can have flaws in them; consider non-standard browser behaviors that try to ”fix” malformed HTML, which might slip by a filter that uses regular expressions.”

Although XSS is a 17 years old security issue which 13 years of them officially known, it seems that instead of seeing less vulnerable applications, exactly the contrary happen [4] [23]. In the recent years, user experience of web applications improved dramatically due to the usage of JavaScript and JavaScript based technology developments (e.g. AJAX and ”Web 2.0”). The wide and rapid usage of these JavaScript based technologies exposes many web-applications to XSS. A quick look at xssed.com (XSS vulnerability news and archive) [6] shows XSS still is very common and new application-specific attack vectors are being submitted to the archive almost on a daily basis. Latest reports still show that XSS are still being widely exploited [23], which shows why XSS is still considered one of the top threats to web-applications [20].

We must point out that the XSS does not have to occur only in ”classic” web applications, but every plug-in or add-on that executes JavaScript. Modern browsers support plug-ins (e.g. Adobe flash animation, adobe acrobat PDF reader) and add-ons that use JavaScript thus vulnerable to XSS.

In chapter 2 we discuss and explain the different types of XSS. Chapter 3 discusses popular solutions being used in the industry, how they try to mitigate with the XSS vulnerability and their limitations. Chapter 4 shows different evasions that can bypass any popular solutions and discusses the crucial key points that cause this failure in detecting and preventing XSS. Chapter 5 presents previous and related work done in mitigating with XSS, their limitations in mitigating the different evasions and their contribution.

After understanding the XSS challenge, their limitations and reading previous work we raise

in chapter 6 key concepts that guided us toward the presented solution in the thesis, and how they are being used in the solution. Chapter 7 talks about the learning and publishing of the legal assembly sets we discuss in chapter 6.

We have implemented a POC of the solution in Firefox JavaScript engine. In chapter 8 we explain the design of the POC and present an execution example of the real system. We used the POC in order to perform evaluation of the solution, we present the results in chapter 9.

Chapter 2

Known XSS Attack Vectors

In this chapter we will present the different types of cross-site scripting vulnerabilities, their key differences and how they are being exploited.

2.1 Reflected (or Non-persistent)

Many websites return HTML pages that parts of the page are based on given input received in the HTTP request. If the web application does not verify and sanitize any given input that is included in the HTML page, a well-crafted request can contain JavaScript code that is placed by the web application in the returned HTML page, and the HTML and JavaScript code included in the returned web page can change the expected behavior of the web page and perform malicious actions within the browser. In other words the web application reflects the user's well-crafted input included in the request into the response, causing the requesting web client to perform new, unwanted, functionality.

This XSS attack is not persistent, meaning the attack is not stored in the website's server, and as a result the attack affects only the users being lured into requesting a well-crafted request (usually by social engineering).

Example: Alice is a user at the popular e-commerce website "ebazon.com". ebazon.com notifies members, such as Alice, about new exciting deals. Also, ebazon.com allows users to

save their credit card in the website, so they won't have to re-enter it every time they make a purchase. When Alice makes a search at ebazon.com, the browser passes the search query using the variable "query" to the search.php page, so when Alice searches for "myphone 5", the query URL is as follows: `http://ebazon.com/search.php?query=myphone 5` (ignoring URL encoding for easy reading).

ebazon.com has been designed to include in the search results page returned to the user, the text the user requested. In other words, it returns to the user the value of the "query" parameter. Therefore, the returned page of the HTTP request above contains the following HTML code: `Results for 'myphone 5'` which renders into Results for "myphone 5".

ebazon.com is vulnerable to reflected XSS, and the vulnerability can be exploited as follows: Assume that instead of searching for "myphone 5", Oscar, the evil hacker, lures Alice (by social engineering via e-mail) into searching a malicious text that creates the following HTTP Request: `http://ebazon.com/search.php?query=myphone 5 special offer<script>>window.location=''http://oscarevilsite.com/submitsession.aspx?cookieid='' +document.cookie</script>`

The request sent by Alice causes ebazon.com to return a search results page containing the value of the query variable. Therefore the return page HTML code contains the following: `Results for 'myphone5specialoffer<script>>window.location=''http://oscarevilsite.com/submitsession.aspx?cookieid='' +document.cookie</script>''`.

The browser cannot tell the returned `<script>` tag is not meant to be an HTML tag, but just text came from the user, so the browser treats the `<script>` tag as HTML code, executing the script inside. The script redirects Alice's browser to `http://oscarevilsite.com/submitsession.aspx?cookieid=[Alice cookie to ebazon.com]`, sending to Oscar Alice's session ID to ebazon.com. Now, that Oscar knows Alice's session ID, as long as Alice does not log out of ebazon.com and the session is active (and many, if not most, users never log out), Oscar can surf ebazon.com as if he is Alice, and purchase the new myphone 5 using Alice credit card she saved in the website.

2.2 Stored (or persistent)

A website vulnerable to stored XSS allows an attacker to store the malicious JavaScript code in the web application database, and this malicious code is included by the web application in web pages returned as a response to normal requests (no specially crafted requests). This XSS attack is persistent because the malicious code is stored in the server's database, and the malicious JavaScript code is returned to anyone that requests the web page.

Exmample: EyesBook.com is a popular social network, allowing users to create a profile page, mark other people as friends and share media between the different users. One important security policy the website enforces is that users cannot tell which users watched their profile page, to keep users privacy.

Oscar, a member of EyesBook.com, is very interested to see which users watch his profile. Oscar notices that his current place of work "the company" appears on his profile page. That means that every user that watches Oscar's profile page, EyesBook.com takes Oscar's current work place name, "the company" from database and embeds it in the returned page to the viewer. Also, Oscar knows the JavaScript function `send_private_message()` sends a message to the user one is currently viewing, therefore he decides to change his work place from "the company" to "the company<script>send_private_message('I am watching your profile');</script>". Alice, Oscar's Ex-girlfriend, who is not a friend of Oscar on EyesBook.com (didn't end well), wants to see his profile. Knowing that Oscar can never tell because she trusts EyesBook.com, she does not worry that Oscar might find out and clicks the button to receive his profile. EyesBook.com loads Oscar's profile to return it to Alice, and the company is part of this profile. The server reads from the database Oscar's current work place, and embeds that value in the HTML returned to Alice. Alice's browser receives Oscar's profile which contains the malicious work place. The browser displays the text "the company", but it cannot tell that the `<script>` block calling `send_private_message()` is just text inserted by Oscar and not code to be executed, therefore the browser executes the code causing Alice, unknowingly, send a private message to Oscar letting him know that she is watching his profile.

2.3 DOM-Based XSS

DOM-based attacks emerged the use of DOM objects to develop and execute client-side web applications in “web 2.0”. The attack is similar to reflected XSS in the sense that the victim request causes the attack, but the main difference is that unlike reflected XSS, in DOM-based XSS a DOM object in the browser generates the malicious code and not the server. In other words, in DOM-based XSS the request sent by the browser to the web application does not have to contain any malicious content, and the response from the server does not contain any malicious content. The malicious content appears in the request after the pragma and as a result kept in the browser and not sent to the web application. When the response is requested by the web application the JavaScript code included in the web page process the JavaScript code that was kept in the browser and as a result the web page perform undesired actions. Due to the fact that the malicious code is not included in the request(s) sent by the browser to the web application, any XSS prevention mechanism located outside the victim’s browser (for instance, web application firewall acting as a front end to the web application) cannot detect and prevent the attack.

Exmample: The following code for the example is taken from Amit Klein’s paper “Dom Based Cross Site Scripting or XSS of the Third Kind” [14]. Alice wrote a simple website that displays to the user the text “Hi [name]”, where [name] is the value of the name parameter in the URL of the website. Alice decided that instead of filling the name on the server using server-side web application, it will be easier to fill the name on the client-side using JavaScript. Therefore Alice added the following code to the HTML:

```
<HTML>
Hi
<SCRIPT>
var pos=document.URL.indexOf('name=')+5;
document.write(unescape(document.URL.substring(pos,document.URL.length)));
</SCRIPT></HTML>
```

When bob surfs to Alice's website using the URL `http://whereisalice.com/index.html?name=Bob`, the returned HTML page **does not** contain the name Bob, but only when the HTML and JavaScript code reaches Bob's browser, the JavaScript engine will extract the name "Bob" from the URL and place it inside the HTML.

Let's assume Oscar lured Bob into surfing Alice's website using the following link:

```
http://whereisalice.com/index.html?name=Bob<script>alert(xss');<script>.
```

When Bob clicks the link, unlike reflected XSS, the server does not embed the injected code (i.e. `alert(xss');`) into the HTML page, but the server returns the same (static) page it returned to the client when the value of name was "Bob". Only when the returned page is being rendered on Bob's machine the attack takes place and the injected JavaScript code `alert(xss');` is being embedded into the page and executed.

Chapter 3

Popular Industry Solutions

Since the XSS problem became popular, some solutions became popular in the web application security industry. These solutions, although they are popular are far from being complete. More than that, in the evasion chapter (chapter 4) we will show that they cannot be complete. In this chapter we are introducing the popular industry solutions, their limitations and problems.

3.1 Encoding

Encoding data transferred between the browser and the server is a popular method to mitigate with XSS. The idea of encoding the data is that by encoding we turn executable injected code into non-executable text. This way, theoretically, if we can encode correctly all the data sent from the browser to the server and from the server back to the browser, any injected code by an attacker would turn into a non-executable text, preventing the attack from being executed.

3.1.1 Input encoding

Input encoding encodes any input that comes from the client, turning any injected XSS code into non-executable text. Input encoding must be done by all the developers, throughout all the application's lifetime. In case of out-sourcing parts of the application and using 3rd party

code, the implementation of this approach gets more complicated and more prone to errors. Moreover, even if all the developers implement the web application while implementing input encoding without use of any 3rd party code, the developer needs to know what to encode, and how to apply the encoding according to the current context. If one developer is writing a form and later on another developer changes the form in any way, the input validation and encoding might change by a new parameter that might need to be encoded, in other words, the implementation is all about context. This task is very daunting and prone to errors [9], exposing this approach to many types of evasions.

Input encoding encodes only the input of the application which might allow us to detect reflected XSS which is being generated by the user's input, causing malicious text in the request to return a response with injected code, but input encoding is not effective against stored XSS and DOM-based. If for example, an attacker managed to store XSS in application's database (creating stored XSS attack), the input encoding solution will not detect anything wrong in a request that returns a web page with the injected code, simply because the request has no malicious text, but it is a regular, legal, request. In DOM-XSS the vulnerability "exposes itself" only after the webpage returned from the server and the client executes the JavaScript code in the returned web page, therefore the attack happens too late. In AJAX it can be even more complex where the malicious input might never even reach the server, and by that workaround all the input encoding solution.

3.1.2 Output encoding

Output encoding solutions encode any data coming out from the server back to the user, and by doing so it turns any injected code into non-executable string. One big advantage is that using this approach we can mitigate easily with stored XSS. Assume that stored XSS attack is placed in a database of a web application. Every time the data returns from the database back to the client, the server encodes the data. That means that every stored XSS that has already been injected in the past, becomes useless (as opposed to input filtering which will be discussed in the next section).

One of the main challenges in implementing output encoding is that the developer needs to understand what is the output the application is sending back to the client, how it will be used, and finally decide the encoding should be used. This gets extremely complicated in today's AJAX and JSON world and prone to errors. Also, all the developers of the application must be aware of the use of output encoding and must use it all the time (like in input encoding), meaning, you cannot write it once or give this job to a specific knowledgeable developer, but all the developers must write the application while keeping in mind the user of output encoding [9]. Of course bugs in the output encoding of a web page means that this page is vulnerable to XSS.

This kind of solution cannot address DOM Based XSS because the malicious code is being generated on the client. The same limitations exist with plug-ins and add-ons, the server that return the SWF or PDF (or any other file format that supports JavaScript) does not encode the script inside these files. Besides these issues, output encoding has no generic solution and the developer need to implement it correctly to every case encoding is being used, making this approach prone to errors which lead to evasions. Like every solution that requires code change, it cannot apply to some 3rd party libraries, and if the solution is being implemented after the development has begun, the developers must go through all the code again.

3.2 Content-filtering / Input validation

Input validation solutions validate, at the server, the input they expect is really what they received from the user. For example, if the server expects to get from a certain field in the application a phone number, the developer in the server side must implement code that verifies that the returned text is really a phone number. This approach becomes harder when it comes to free-text fields like in discussion groups, and more complex when it comes to AJAX with data that is being sent from the client. Just like in input/output encoding, there is no generic solution. The security of the application depends on all the developers to perform the right validation without any errors or mistakes that might lead to evasions. As we've seen in input

encoding, if an attacker manages to store XSS in the application database, input validation will not detect the attack, simply because there is nothing special or “illegal” in the request. Also, this approach cannot mitigate with DOM-Based because the malicious input might not even reach the server. Filter, as the name suggests, is filtering the data using positive or negative security logic:

- Negative logic means that if the inspected “text” is in a known **blacklist** - it is illegal, otherwise it is legal. In other words, we need to know all the possible **illegal** text (all attack vectors) upfront.
- Positive logic means that if the inspected “text” is in a known **whitelist** - it is legal, otherwise illegal. In other words, we need to know all the possible **legal** text (all non-attack code) upfront.

If we focus on XSS solutions, a XSS solution using negative logic means that we need to know all the possible attack vectors upfront. A XSS solution using positive logic means that we need to know what all the possible legal JavaScript executions upfront.

A popular way to implement content-filtering is on the network traffic, there is no need to install solutions on clients or servers, simply plug-in a new hardware in the entrance of the organization and “you’re safe”. This way an organization places the solution as a “man-in-the-middle”, allowing the solution to inspect the traffic and block suspected attacks. All web-application firewalls belong to this type of solution. The first issue of using any kind of network traffic monitoring, is that it cannot handle HTTPS traffic or encrypted traffic of plug-in or add-ons. There is no simple way to override this problem. A possible approach is to use a proxy inside the organization that will create the encrypted connection with the outside world, but then the traffic inside the organization is not encrypted, leaving sensitive data (like passwords) unencrypted. This issue becomes more relevant as many applications start to use HTTPS, especially on web-sites that require the user to log-in and maintain a session. Assuming the traffic is not encrypted, the next challenge is to extract and find all JavaScript

code in the network streams. Some cases might be easier like JS files, but parsing HTML with inline code or even more challenging scenarios like XML, SWF or PDF could be a much harder process in terms of time the analysis take and the extraction process. As experience shown us, the extraction is very prone to errors and many evasions, which we discuss the most important ones in the evasion chapter (chapter 4). Once the JavaScript has been extracted, the filter uses negative security logic. Content-filter of network traffic cannot mitigate with DOM-Based XSS because the vulnerability itself is being generated on the client side while the JavaScript engine is executing the JavaScript code. Therefore, it is not necessary that there will be anything suspicious on the traffic that the content-filter will be able to detect. Also, this approach suffers from browser error-tolerance evasion and JavaScript generating JavaScript evasion, which will be discussed in the next section.

3.2.1 Limitations of positive logic approach

The main limitation of positive logic approach is that the solution needs to have a predefined whitelist of all the possible legal JavaScript code. This limitation becomes even more problematic when a new JavaScript code can be generated during runtime of the application. In this work we will discuss how we workaround these limitations using “Generalized JavaScript Assembly” and build successfully a whitelist for a give web application.

3.2.2 Limitations of negative logic approach

Many popular solutions to XSS are usually based on negative security logic content-filtering of network traffic or HTTP Requests (i.e. they try to detect patterns typical to XSS attacks on the HTTP request content). This approach is quite popular, for example a popular work done by J.Shanmugam and M.Ponnaivaikko [11] tries to match between user’s input at the server with a predefined blacklist made out of keywords and special characters. However, the effectiveness of these solutions is limited due to the following reasons:

- Negative logic can’t detect and prevent 0-Day attacks because the blacklist doesn’t contain the patterns of new attacks that have just been developed by the attacker.

- These solutions need constant updating of the XSS attack patterns (inherent limitation of negative security logic). They are exposed to creative evasion techniques that hide the XSS attack content in order to bypass the negative security logic filter. See for example OWASP’s cheat sheet [21] that provides many evasion patterns, and xssed.com [6] keep on posting new ones all the time. (As an example an evasion might occur because the JS code is located in a place that the detection mechanism didn’t expect, for example in OWASP’s XSS cheat sheet [21]: `<BODYONLOAD=alert(String.fromCharCode(88,83,83))>`, and if the XSS filter does not look for JS code inside “ONLOAD” attribute of BODY tag, the malicious JS code will not be detected.)
- They can’t address DOM-Based XSS attacks or plug-in attacks since these filters usually are implemented on the server side or as “man-in-the-middle” while DOM based attacks does not necessarily produce any traffic.
- User input can contain blacklisted JavaScript but it can be constructed only by actually executing the response page. In other words, they can’t detect and prevent XSS attacks where the malicious JavaScript code is generated by another JavaScript code (we further discuss this issue in chapter 4).

Chapter 4

Evasions

Any negative logic approach, the detection mechanism should know where and how to detect and extract forbidden JS code. Whether it is by location of the JS code or a way the JS code masquerade itself, a negative logic approach must know about the existence of an attack upfront in order to insert that attack into a known blacklist. That means that any XSS attack vector or evasion technique that is not known and blacklisted, is not an attack. In other words, any 0-day attack will surely evade any negative logic approach until the blacklist will be updated.

In this chapter we will present a few evasions techniques that show that on one hand they are quite easy to craft, and on the other hand they are quite difficult to detect using negative logic. This situation is obviously very good for the attackers, and very bad for the defenders. We also discuss in this section the JavaScript generating new JavaScript evasion that does not have any solution in today's popular approaches; this is because the detection problem of this evasion is equivalent to the halting problem, which has no possible solution.

4.1 Hiding JavaScript by encoding

Many times filters assume that by blocking a number of characters, it is possible to block XSS attack. The main problem that by changing the encoding of a page, we can use different

symbols that equivalent to the blocked character. An example from OWASP XSS filter evasion cheat sheet [21], the ‘<’ character can be encoded, by changing the page encoding into anyone of the following (all the following symbols are equivalent to ‘<’ just under different encodings):
%3C,<t,<t;,<,<,<,<,<,<,<,<,�<
;,<,<,<,<,<,<,<,<,
c,<,<,<,<,<,<,<,<,&#
X03c,<,<,<,<,<,<,<,&
#X00003c;,<,<,<,<,<,<,<,<
;,<,<,<,<,<,<,<,<,�
03C,<,<,<,<,<,<,�
3C;,<,<,&#u003c,&#u003C.

By using the ability to represent one character in another way, it means that every entry in the blacklist that contains ‘<’ character must also be placed in the blacklist with each and every one of the encodings. Regular expressions might help us in detecting these different options for each character, but it turns the regular expressions into quite complex to create, manage and execute. Some web pages allow the client to choose the encoding being used, and this allows an attacker to completely change the attacking string, creating many new attack vectors that the filter need to detect.

Example: Let’s assume the following page allows the request to determine the encoding used, when the default is UTF-8, and the value of “q” is being embedded into the response page: `http://www.example.com/q.php?enc=UTF-8&q=Test`

Once the request is made, the filter checks if the value of “q” contains possible injected code, so it I would use “`<script>alert(‘xss’)</script>`” the filter would detect our attack and block it. Now, instead of using UTF-8 let’s use an evasion found by Kurt Huwig and set US-ASCII encoding instead. US-ASCII is a 7 bit encoding, which means that the browser and server ignores the 8th bit, so by flipping it, the client and server can look at ‘<’ and the other symbols in the correct way, but if the filter does not handle US-ASCII encoding correctly it

will miss the attack. Let's take a deeper look:

'<' = 0011 1100 = 0x3C → url encoded → %3C

'¼' = 1011 1100 = 0xBC → url encoded → %BC

So in US-ASCII encoding both character '<' and '¼' are the '<' character (ignore the MSB), but a filter that does not ignore the 7th bit does not know that, and thinks the text is not malicious. This way the following request evades the filter and performs successful XSS attack:

```
%BCscript%BEalert(%A2XSS%A2)%bC/script%BE
```

4.2 JavaScript extraction from HTML evasion

JavaScript solutions need to extract the JavaScript code they need to verify. Whether it is from network traffic, the client HTML or the server, there must be an extraction point that the solution receives the data to analyze and extract the JavaScript code. Therefore a possible evasion is to place JavaScript code in a place that the filter doesn't search. A few examples can be seen in OWASP filter evasion cheat sheet [21]:

- `<META HTTP-EQUIV="refresh" CONTENT="0;url=javascript:alert('XSS');">`
 - META tag with HTTP-EQUIV attribute as refresh can tell the browser to refresh the page.
 - CONTENT attribute tells the browser what is the time interval to refresh, while it is possible to place a URL to redirect the browser to. This URL can be used to execute JavaScript code.
 - Filter that fails to check the CONTENT attribute for "url" parameter, will miss this JavaScript code.
- `<META HTTP-EQUIV="refresh" CONTENT="0;url=data:text/html;base64,PHNjcmlwdD5hbGVydCgnWFNTJyk8L3NjcmlwdD4K">`
 - This evasion uses an abnormal place to inject scripts like the previous bullet, but

it also combines it with encoding evasion described in the previous section (section 4.1).

- By adding type of data and encoding to base64, instead of actually writing the injected script as it is “`<script>alert('XSS')</script>`”, by encoding to base64 we masquerade possible characters that the filter is looking for.
- Filter that fails to check the “url” parameter and take into account any possible encoding will miss this JavaScript injected code.
- The “base64” trick is available in more tags like “embed”.

As we’ve seen in these examples, just extracting the JavaScript code to be analyzed is a hard task, with a huge number of possibilities. The only location that a solution does not need to perform an extra-work to extract JavaScript is inside the JavaScript engine. When a solution is located inside the JavaScript engine, the JavaScript code reaches to the solution on its own. Any code that is being compiled and executed must go through the JavaScript engine, therefore just by placing the solution inside the JavaScript engine, extraction becomes useless.

4.3 Browsers’ parser and engines error tolerance

JavaScript engines agree to tolerate different JS syntax errors by trying to guess “what the developer meant”. Any solution that works outside the JavaScript engine will have to know all browsers’ error tolerance up front (and the behavior of different engine versions). For example, a typical negative XSS detection mechanism misses the following attack taken from OWASP cheat sheet [21]: `<SCRIPT>alert("XSS")</SCRIPT>>`. From pure HTML syntax perspective there is no script element and a string. However, certain browsers do interpret this script element and will execute it. A solution that is located inside the JavaScript engine does not need to detect these errors, because only tolerated scripts arrive to the engine, and all none-tolerated scripts are not detected or tolerated by the browser and does not reach the engine. JavaScript code error tolerance occurs during the compilation of the code. If the JavaScript code compiles successfully, than it means that any errors inside the JavaScript code

has been tolerated by the engine and on the other hand, if compilation fails than errors in the JavaScript has not been tolerated and the script will not be executed.

4.4 JavaScript code generating JavaScript code evasion

Even if all the above limitations can be solved, there is still a way to bypass XSS filters using JS functions like `eval()`, `document.write()` and `RegExp.replace()` that can generate, during runtime, new JavaScript code and new DOM objects. Thus, using these functions an attacker can construct the attack code during runtime by performing any kind of string manipulations (like building the malicious JS code from using string manipulations or decryption of encrypted malicious JS code). The implication of this capability is that a response returned to the client might contain many more JavaScript code that a simple parsing can't discover. Thus, any XSS detection and prevention mechanism will have to execute all `eval()` and `document.write()` functions just to be able to uncover all the possible JavaScript code in order to be able to check if they are legal JavaScript code or malicious. In other words, if a XSS filter wants to verify that the code generated by `eval()`, `document.write()` or `RegExp.replace()` functions doesn't contain any malicious/"illegal" JavaScript code, it must generate all the JavaScript code that they generate, which might contain more `eval()`, `document.write()` and `RegExp.Replace()` or any other form of encoding (or custom encoding) that might "hide" more `eval()` and `document.write()`.

Exmample: In order to explain why there is no generic way for a XSS filter to handle this evasion technique of natural JavaScript code that generate malicious JavaScript code using functions like `eval()`, `document.write()` and `RegExp.replace()` let's take a look at the following example generated by "javascriptobfuscator.com" [10]: `var _0xa8a1=["\x65\x76\x61\x6C\x28\x27\x61\x6C\x65\x72\x74\x28\x5C\x27\x61\x74\x74\x61\x63\x6B\x5C\x27\x29\x3B\x27\x29\x3B"];eval(_0xa8a1[0]);` The XSS filter that would like to detect the malicious code generated by this JS code will need to detect the JavaScript code in the response and to "understand" that the `eval()` function executed on the first cell of the "_0xa8a1" array, just like

a JS engine would do. Next it needs to “understand” there is more JS code to uncover and recheck, the JS code inside “_0xa8a1[0]”, just like a JavaScript engine, and to decode the string into its original form which is `eval("eval('alert('\`attack\`');');");`. Then it will need to re-execute the `eval()` recursively until the “attack” is being uncovered, just like a JS engine would do. Since, there is no way to analyze the code that can be generated statically since it is equivalent to solving the halting problem, in order to uncover all the JS code, a XSS filter need to emulate the full JavaScript engine behavior and execute the JavaScript code (while applying error tolerance of all different engines), and only then check the generated content to verify if it contains malicious JS code. To overcome all the above limitations we suggest to place the XSS detection and prevention mechanism inside the browser’s JavaScript engine (not inside the browser or the interface between the browser and the engine, but inside the engine module itself). This position enable the mechanism to get all runtime-generated code, including code that comes from plug-ins, add-ons and to handle JS code that generates more code as well as the JS engines’ errors tolerance (all non-tolerated errors, will not reach our mechanism because the engine will fail to compile them).

4.5 JavaScript in plug-ins, add-ons and other filetypes

Modern browsers allow developers to develop plug-ins and add-ons to their browsers. Some of these add-ons and plug-ins use the browser’s JavaScript engine to execute JavaScript code. For example, Adobe flash animation can use JavaScript code that will be executed on the browser, or an add-on that uses JavaScript to block ads from websites. For example, in 2007 adobe reported [1] that SWF files (Flash animation) can perform XSS attack by embedding JavaScript commands inside the SWF itself. Current detection mechanisms and filters focus on HTML/XML responses to look for JavaScript, but not inside SWF files or other file types that might embed JavaScript; therefore many detection mechanisms do not even check SWF or PDF file types. By locating the XSS detection mechanism in the JavaScript engine there is no need to the solution to “learn” how to look for JavaScript. If the embedded JavaScript is

being executed it will reach the detection mechanism on its own.

Chapter 5

Previous & Related Work

As far as we know, there is no attempt to mitigate XSS attacks from inside the JavaScript engine using compiled JavaScript, but there are several approaches in the literature that have common properties as we do. Besides that, there are several other approaches we argue against because of limitations they might endure.

5.1 Automatic sanitization frameworks

Many web-application frameworks claim that their sanitization abstractions can be used to make web applications secure against XSS. J. Weinberger, P. Saxena, D. Akhawe, M. Finifter, R. Shin and D. Song [28] analyzed 14 major commercially-used web application frameworks and reached the conclusion that these frameworks fail to secure the application from XSS.

One of the main limitations of this approach is that the sanitization happens on the server. In other words it cannot handle JavaScript generating JavaScript, DOM-Based XSS or browser error-tolerance evasions. The paper [28] also states that no framework supports any dynamic sanitization for dynamic evaluation in the client.

We argue that even a dynamic client side automatic sanitization framework wouldn't be good enough because it would still have to detect and sanitize JavaScript and HTML with errors, depending on the browser's error-tolerance. On top of that it would have to sanitize non-

standard JavaScript and HTML, JavaScript from plug-ins and add-ons, and do all that in runtime.

5.2 Browser based detection and prevention mechanisms

Browser based XSS detection and prevention mechanism called BEEP [12] allows the developers to mark in the code functions the developer wants to attach a security policy called “security hook”. The writers present 2 observations:

1. Browsers perform perfect script detection. If a browser does not parse content as a script while it renders a web page, that content will not be executed.
2. The web application developer knows exactly what scripts should be executed for the application to function properly.

With the 1st observation we agree, we also observed that fact so our solution also depend on it, but we disagree with the 2nd observation. Today’s web application can be huge and include 3rd party libraries. A developer in one of the development teams cannot know exactly where his modules are being used and will be used in a huge project, let alone deciding which security policy secure his script needs. The system architect might know but it is still a hard task and very prone to errors. Elias Athanopoulos, Vasilis Pappas, and Evangelos P. Markatos [3] managed to attack BEEP also share this conclusion regarding the 2nd observation. Another limitation of BEEP is the use of 3rd party libraries or frameworks, the developers cannot change the source code, therefore they wouldn’t be able to secure the libraries code, and even if they could, they need to go over the 3rd party library in order to decide which policy to attach.

ConScript [16], similar to BEEP, allows the developers to attach policies to a JavaScript code, but ConScript, by using “deep advice” mechanism, allows the developer to apply very complex and agile policies. ConScript allows the developer to “hook” functions, replace them and by that restrict their usage. Although ConScript is very powerful, it requires the developer to attach the right policy to the right functions, and writing a policy can be hard and prone to

errors as can be seen in a presentation done by Leo Meyerovich and Benjamin Livshits [15]. These errors might lead to other vulnerabilities or allow attackers to evade the policy. Another limitation is attaching policies to 3rd party libraries.

CSP [25] (by Mozilla) provides a large set of properties that describe a per-page policy. Using these properties CSP can define a policy for the content in the received page. The property “inline-scripts” tells CSP to allow or block JavaScript code running outside `<script>` blocks. Using this property CSP tries to mitigate XSS attacks. Because of the way CSP works it forces some web applications to change the way they’re built, also add-ons that use inline JavaScript might stop working. Also, CSP causes a performance hit, as shown in an analysis done by A. Barth and D. Song [27].

In BEEP, ConScript & CSP the responsibility on the security falls into the developer’s hands, which in many (if not most) cases is not security aware, which may lead to other vulnerabilities, bugs or even might end up in turning of the security mechanism.

Noxes [7], Unlike previous presented solutions, does not try to mitigate or prevent the XSS attack itself, but it tries to prevent its consequences (like stealing the session ID). Noxes acts as a web proxy and uses manual and automatically generated positive-logic rules to block outgoing connections to unauthorized sources. The user can manually define a set of web sites a web application can connect, or use automatic learning mode (called snapshot) that monitors the web application (while user surfs it) and generate the set of web sites allowed. Also, Noxes allow the user to decide in real-time what to do with a new unexpected website connection, to allow it or block it. This approach has severe limitations such as an attacker that uses whitelisted domains, such as BlogSpot or Facebook as part of the attack, therefore the attack will not be blocked simply because most users won’t block Facebook or BlogSpot. Also because it does only prevent sending information out of the browser it does not prevent malicious JS code running in the browser and attacks like XSS based defacement will not be prevented. We also argue that solutions that depend on user’s decision cannot work in the “real-world”. Most

users do not know or understand if a connection is legal or not, and if an attack will happen and Noxes will ask the user what to do, many users will consent the malicious connection.

Kudzu system [22] is detecting XSS vulnerabilities using a symbolic-execution based framework for client-side JS code analysis. The system explores the web application execution space and detects XSS vulnerabilities and mitigates DOM-based XSS. One of the main problems with this system is that it can only find vulnerabilities, but not block them. Users still depend on the developers to fix the problems in order to defend themselves. Moreover, many times the fix might be in a 3rd party code which make the fixing much more problematic. Also, Kudzu ignores the error tolerance problem of different JS-engines. Kudzu might assume that a specific given JavaScript is behaving in one way, or even declare the code as flawed while a specific popular browser might execute the code in a different way Kudzu anticipated and expose the user to a possible attack vector. This limitation is inherit by the fact that Kudzu is not engine specific.

BLUEPRINT [17] is using a trusted JavaScript parser that is being used instead of the browser's built-in JavaScript engine in order to build the page by a given security policy called "blueprint" generated by the server. The approach partially solves the issue of error tolerance vulnerability by not trusting the browser's JS engine but if it allows a non-standard script, then eventually the browser might react differently than expected. Also, besides performance issues the solution claims that it can trust the server to know how to deal with untrusted content, while this claim is arguable [27].

The most similar method to our approach has been proposed by Rotberg and Movshovitz [18], but unlike our solution, it is implemented as a client side proxy that extracts the JavaScript code. However, placing a XSS detection and prevention mechanism outside the JS engine exposes the solution to all limitations we described above. Rotberg and Movshovitz show that using a canonicalization function on the JavaScript code it is possible to create a set of "canonical forms" of all the "legal JavaScript variants" generated by the web application. This

database that holds all the canonical forms of the legal JavaScripts code elements used by the web-application, can be used to verify if a canonicalized form of a returned JavaScript from the web application is legal by looking it up in the database. However, performing canonicalization over JS code is very complex, time consuming task and prone to errors.

Chapter 6

Key Concepts

By looking at previous solutions and today's industry we realized that a solution cannot dictate a new approach to build a website from scratch by suggesting to change all the code in the application, simply because there are millions of applications (if not more), and some of them are with millions of lines of code which many of them belong to different companies and 3rd parties, and a fundamental change in the application code is simply not feasible because of the size of the web applications. Another problem with a solution that requires code changes in order to solve the XSS vulnerabilities is that developers make mistakes and bugs and just like any other bug in web application, giving the developers the responsibility to add or modify code in order to mitigate with XSS will lead to code that will end up with XSS or other vulnerabilities.

We also came to the conclusion that a solution cannot depend on an end-user decision in order to mitigate with the attack. The end-user is not security-oriented, and many times hardly understands what happens in the background. We think it is reasonable to assume the typical end-user will click whatever will load the page the most quickly, without any “annoying security questions”, especially if these questions happen frequently.

To sum up the key concepts that was leading us toward to the presented solution:

- The solution must not require any changes in the JavaScript code for 2 reasons:

1. Many applications use 3rd party libraries to build their websites. Many times changing the code in these libraries is impossible due to legal or technical issues.
 2. Requiring the developers to change the code might be prone to errors by the developer. Developers are people, and people make mistakes, these mistakes can turn into new vulnerabilities.
- The solution must not depend on any end-user action or knowledge. The user many times does not know or understand about what happens in the background. Involving the user in the protection cycle is useless, and might only annoy the user that will click any button that will get the web application running.

The presented solution does not require any code modifications by the developers that might expose new vulnerabilities, or 3rd party codes that we cannot modify. Also, the solution does not require any action from the end-user client. The solution teaches the JavaScript engine to detect on its own unexpected code, and if such code has been detected, it protects the user without any action taken by the user.

6.1 Detection of XSS attack within the JavaScript engine

All the different solutions that try to detect the XSS attack itself have limited effectiveness due to the problematic detection of the malicious JavaScript code due to the various evasion techniques and the inability to detect malicious JavaScript generated by another JavaScript. To overcome this fundamental problem, our solution detects the malicious JavaScript when the JavaScript engine is compiling the JavaScript code into a JavaScript Assembly (JSA) by the JavaScript engine. This is the best way to detect the malicious JavaScript code since if the browser and JavaScript engine won't recognize the malicious content as JavaScript code due to evasion techniques, then the attack wouldn't occur in the first place and as a result we don't need to worry about it. On the other hand if the browser and JavaScript engine did uncover the malicious content as JavaScript code and even if browser and JavaScript engine detect errors but they tolerate the errors and will execute the malicious content as JavaScript code,

then and only then the revealed JavaScript code, after successful compilation, will be presented to the detection and prevention mechanism.

The location of the XSS detection and prevention mechanism inside the browser's JavaScript engine eliminates the need to detect the JavaScript code within the HTTP response because the browser's JavaScript engine detects the actual JavaScript to be executed by the browser's JavaScript engine, and as a result the detection and prevention mechanism can't be bypassed by the various evasion techniques that try to hide the JavaScript code in the response.

It is important to point out that this position of the detection mechanism allows it to detect also DOM-Based XSS attacks (that out-of-browser XSS detection mechanism can't detect), this is due to the fact that the JavaScript engine compiles every JavaScript code that will be executed by the engine, and it doesn't matter if it comes from the server or was inserted into the DOM by the client-side script code. This is also true for Add-ins and Plug-ins that use the browser's JavaScript engine, so if a SWF file (which is a flash animation client side web application) tries to execute JavaScript code, since the compilation and the execution is done by the browser's JavaScript engine it will go through our detector and will be detected and prevented (as will be explained below).

6.1.1 Solution advantages

By placing the solution inside the JavaScript engine we are natively able to detect all the JavaScript code, even the code being generated by another JavaScript code.

The location of the solution is such that evasion techniques become non-relevant because it does not matter which evasion technique is being used, at the end it will have to uncover the real malicious JavaScript code in order to perform the attack. If an evasion technique is "so good" that it evades our mechanism, it must also evade the JavaScript engine itself, therefore it will not get executed.

Another advantage of this location is that the mechanism can work natively with plug-ins and add-ons. If a PDF or SWF are executing JavaScript code, it will go through the mechanism without any extra-work.

All of the above shows that by placing the solution in the JavaScript engine, the solution does not need to perform any extra-work in order to locate JavaScript code, it will reach the mechanism on its own.

6.2 Positive approach

The second major concept of our solution is to base the detection of malicious JavaScript code on positive logic (“white-list”) approach. We presented earlier the limitations of negative logic, which makes the solution vulnerable to 0-day vulnerabilities and require never-ending maintenance and update of the blacklist. All this and the fact that there is no reasonable method to define malicious code and as a result if we want to lower false positives we need to base our detection mechanism on positive security logic, and use negative only as last resort and time limited until the database is being updated.

6.2.1 Limitations of naïve solution

A naïve solution to try and create a whitelist of all the JavaScript code of a given web application would be at the beginning - to create a set of all the JavaScript code the application uses. This kind of solution fails immediately because an application can generate new JavaScript code during runtime, therefore we cannot pre-define all the possible JavaScript code. The use of a canonicalization process over the JavaScript code shown in the work done by Rotberg and Movshovitz [18] that we discussed in the related work section presents another naïve solution for creating a the positive solution.

One of the main limitations of the canonicalization process, even if done inside the JavaScript engine, is its complexity. The JavaScript code is very complex and permissive, all that causes JavaScript parsing and canonicalizing very complex and time consuming. The worst problem of the canonicalization is because of the complexity of the process, the implementation is prone to errors.

The complexity of the JavaScript language makes it hard to work with, therefore a naïve so-

lution that works directly on JavaScript is just not good enough for real-world use, from the hard implementation to the time consuming process.

6.2.2 Positive logic with GA

Modern JavaScript engines don't interpret JavaScript anymore, they compile it. For example, SpiderMonkey (Firefox's JS engine) compiles JavaScript to a series of opcodes called bytecode, V8 (Chrome's JS engine) compiles JavaScript to binary, which is also a series of opcodes. For simplicity, we will call the compiled JavaScript code "JavaScript assembly", and denote it by JSA for short.

Every JavaScript code that is being executed by the browser is compiled first by the browser's JavaScript engine into JSA before execution, and the JSA is the actual code that is being executed (executed directly in V8 case or by an interpreter as in SpiderMonkey's case). It is important to point out that JSA does not contain any comments, weird syntax or encodings we need to overcome. By using JSA to detect injections, and not by looking at the JavaScript code, any obfuscation or obscure evasion in the JavaScript code does not affect our detection mechanism since it doesn't affect the JSA that is being used for checking if the JavaScript is legal JSA.

In today's websites and web applications JavaScript can be generated dynamically based on user's input or application state. Thus, to be able to detect malicious JavaScript code using positive security logic we need to generate the legal set of compiled JavaScript code that a given web application can generate/execute. We cannot simply get the compiled JavaScript code and put it in the legal set, since the legal JavaScript code can change from one compiled JavaScript generation to another. Thus, we need to translate the JavaScript Assembly (JSA) into a Generic JSA (GA) that will enable us to check the compiled JavaScript Assembly (JSA) against the legal set of Generic JSA of the web application. The main challenge in generating a Generic JSA is to strip out all the elements in the code that change legally from one execution to another. Each time the JS engine compiles the JavaScript code, it compiles it with variables values that are unique to the current executions. Thus, the conversion of JSA

into a Generic JSA needs to strip or modify all the opcodes that contain data that regards the current execution, so the result Generic JSA code will contain only the opcodes that define the behavior of the JavaScript code.

The idea of the Generic JSA (GA) is similar to a template class in C++ or generic class in Microsoft.Net. When a programmer defines a generic (or template) class, the class does not have a pre-defined exact behavior or execution (we cannot compile a generic class), but it defines the general case for all the specific classes. GA is similar to that in the sense that it is a generic assembly form of a specific JavaScript assembly. Thus, an important element in our detection mechanism is the generalization function ($f_{generalize}()$) that converts a given JSA into Generic JSA (GA). It is important to point out that $f_{generalize}()$ has another task which is detailed in section 6.3. Let's assume JSA_1 and JSA_2 such that JSA_1 and JSA_2 are different executions of the same JavaScript code ($JSA_1 \neq JSA_2$), but the generalizing function is defined such that $f_{generalize}(JSA_1) = f_{generalize}(JSA_2)$ (The details of the generalization function are described below.)

Example: Let's take an example of JS based calculator.

A possible execution of the calculator is:

```
var x = 1; // some comments
y = 2; /* more comments */
result = x + y;
```

(Note: 1 and 2 are related to the specific user session and will have different values for different user sessions)

The above JavaScript code will be compiled into JSA with the numbers "1" and "2" inside it, because this is what the machine is going to execute. Thus, the following (pseudo) JS Assembly will be generated by the JavaScript compiler:

```
MOV REG1 1 ; x = 1
MOV REG2 2 ; y = 2
```

```
ADD REG1 REG2 ; x + y, put the result in RES_REG
MOV REG3 RES_REG ; result = 3
```

If a user chose to add 15 and 34, these numbers would appear in the JSA:

```
MOV REG1 15 ; x = 15
MOV REG2 34 ; y = 34
ADD REG1 REG2 ; x + y, put the result in RES_REG
MOV REG3 RES_REG ; result = 49
```

As a result, the JSA will be different from one execution to another. Therefore, we can't create a set of legal JSA that will be used to check the validity of the JSAs. As explained above, to overcome this problem our mechanism generates Generic JSA (GA) out of the given JSA using a generalizing function, $f_{generalize}()$. In the example above the GA (which is the output of $f_{generalize}()$ for add operation in the calculator) would look like this:

```
MOV REG1 literal ; literal is an OPCODE we added to symbolize a literal.
MOV REG2 literal
ADD REG1 REG2
MOV REG3 RES_REG
```

Due to this generalizing process it doesn't matter which values the user chooses to add, all the possible "add" executions in the calculator will generate the same GA. The ability to create such GA for all the legal execution of the above JavaScript is essential for our detection mechanism, and enables the "learning" process for each web application that constructs a "white-list" made of the set of legal GAs that represents the JavaScript code used by the web application. In this "learning" mode each generalized JSA is considered as a legal GA that we place in the web application legal generic JSA set (LGAS for short) that will be used in detection and prevention mode. In other words, the LGAS of a given web application is the web application white-list. Given this set of legal generic JSA, the detection mechanism apply the generalization function to each JSA it compiles and check if $f_{generalize}(JSA)$ appear in

the LGAS. If it appears the JavaScript is legal and executed, and if it doesn't the script is considered a potential attack and if there is no update to the whitelist, the system falls back to negative security over the JSA in order to determine if the script is an attack or an update. If detected as potential attack, it is blocked to prevent XSS.

As example to the ability to detect malicious code and block it, let's assume a web application that receives "myname" value from the client and places the string without proper validation or output encoding inside the following code: `eval("alert('"+myname_value+"');");`. First the `eval()` function will be executed which will generate the pseudo JS assembly:

```
MOV REG1 STR1
EVAL REG1
```

The execution of the opcode `EVAL REG1` will cause the JS engine to compile the code written inside `STR1`. In other words, the JS engine will now compile `alert('"myname_value"');`.

This causes the JS engine to generate the pseudo JS assembly:

```
MOV REG1 STR2
ALERT REG1
```

Thus, as we can see the given JS code generated 2 JSAs, the `eval()` execution and the `alert()` execution. Now let's look at the GAs.

The first GA is:

```
MOV REG1 literal
EVAL REG1
```

While the second GA is:

```
MOV REG1 literal
ALERT REG1
```

In the "learning" phase we will place both of these legal GAs (or LGA) into the LGA set of this web application (or LGAS for short). Now let's assume an attacker tries to perform a XSS attack on that webpage with the following input:

```
myname = ''; window.location='http://evilsite.com/'; //''
```

As a result of this input the vulnerable application will return to the user the following code:

```
eval("alert(''); window.location='http://evilsite.com/;'/');");
```

and the evaluated code will redirect the user to evilsite.com. Let's take a look at the generated

JSA: First JSA does not change:

```
MOV REG1 STR1
```

```
EVAL REG1
```

Thus, it's $f_{generalize}()$ output is the known LGA:

```
MOV REG1 literal
```

```
EVAL REG1
```

However, the second JSA that will be generated by the JavaScript compiler and it looks like:

```
MOV REG1 STR2
```

```
ALERT REG1
```

```
GPROP REG2 window
```

```
SPROP REG3 STR3; this code cause redirection.
```

And the generalization function will generate the following GA:

```
MOV REG1 literal
```

```
ALERT REG1
```

```
GPROP REG2 window
```

```
SPROP REG3 literal
```

As we can see this second GA is very different from the previous second GA and will not appear in the LGAS of the web application (as opposed to the first run). Therefore it will be considered as a XSS attack and will be blocked.

As can be seen in [18], performing canonicalization on the JavaScript code itself is a very hard

task, and can be vulnerable to evasions. The location of the XSS detection and prevention mechanism within the browser's JavaScript engine allows the mechanism to perform the conversion to generic form on the compiled JavaScript code instead of the JavaScript code itself. This conversion of JavaScript assembly into generic form is much simpler and not vulnerable to various evasion techniques. To summarize, the combination of the location of the detection component and performing the generalization on JavaScript Assembly makes our mechanism immune to evasion techniques.

6.3 Negative logic fallback

One limitation of positive logic approach is the inability to handle legitimate changes that are not reflected in the white list, for instance and updated script that was not entered into the whitelist. In that case our solution falls back into using negative security logic, but the solution location allows us to perform negative logic over the JSA, which natively uncovers evasion techniques. Our negative logic approach does not look for a pre-defined signatures of known attacks, but for certain actions done in the JavaScript code that are considered suspicious. We want to assess the threat of a JSA and determine if the JSA should be blocked or not. In order to assess the threat of a JSA, during the construction of the GA in `fgeneralize()` we keep tags of suspicious actions and the amount they are used and a histogram of the opcodes in the GA. A suspicious action can be calling `eval()` or `document.write()` for generating new code, but these actions are just raising suspicion, they cannot harm by themselves, but can be part of an attack. We consider action like `window.open()`, settings `document.location` property to redirect or even using XML objects `open()` function to start new HTTP request as dangerous actions because these actions create new HTTP requests that can send personal data.

These suspicious and dangerous actions can be detected quite easily by parsing the JSA, and hiding the use of these functions to evade detection is not possible because hiding these functions in the JSA will cause the JavaScript compiler not to execute the attack. Even manipulations like indirect use of an object can be detected easily in JSA, while can be quite

complicated in JavaScript Code.

For instance:

A **direct** use of document object to perform redirection:

```
document.location = 'http://google.com';
```

An **indirect** use of document object to perform redirection (one level of indirection):

```
var i = document;  
i.location = 'http://google.com';
```

Direct use can be detected quite easily in JavaScript code, but indirect can be much more complex, especially when using multiple levels on indirection.

Our system detects indirect use easily and nativity thanks to the use of JSA and not the JavaScript code. Analyzing a simple JSA is much simpler and faster than analyzing JavaScript code that can be very complicated.

In case a script is not known the URL, JS, GA and negative tags are being submitted to the web server to a module we call GA server and the following occurs:

- If there are no negative tags:
 - The server automatically adds the GA to the whitelist and logs the operation.
 - The script is allowed to be executed by the JavaScript engine.
- If there are negative tags:
 - In the server:
 - * If there are only suspicious actions in the negative tags then the server does not add the GA to the whitelist automatically, but logs and alerts the webmaster (or any other knowledgeable employee) to decide if the GA enters the whitelist. This information might also discover possible vulnerabilities in code that have not been uncovered yet.

- * If there is at least one dangerous action in the negative tags then the server does not add the GA to the whitelist automatically, also it logs and alerts the webmaster that a dangerous action has occurred. The webmaster will need to manually decide if the GA enters the whitelist.
- In the browser we want to check if these negative tags are not simple update of known page which causes the hash to change. In order to do that we look for the most similar scripts. We do that by calculating the distance between the unknown script opcode histogram and every other histogram in the LGA. The closest GAs are the ones that their distance is the smallest. Once we found the most similar scripts we find the difference between their negative tags:
 - * If the difference between the negative tags contain only suspicious actions we allow the execution.
 - * If the difference between the negative tags contain at least one dangerous action the execution is being blocked.
- The script is allowed to be executed by the JavaScript engine.

6.4 Detection & Prevention

In case a script is not known the URL, JS, GA and negative tags are being submitted to the web server to a module we call GA server and the following occurs:

- If there are no negative tags:
 - The server automatically adds the GA to the whitelist and logs the operation.
 - The script is allowed to be executed by the JavaScript engine.
- If there are negative tags:
 - In the server:

- * If there are only suspicious actions in the negative tags then the server does not add the GA to the whitelist automatically, but logs and alerts the webmaster (or any other knowledgeable employee) to decide if the GA enters the whitelist. This information might also discover possible vulnerabilities in code that have not been uncovered yet.
 - * If there is at least one dangerous action in the negative tags then the server does not add the GA to the whitelist automatically, also it logs and alerts the webmaster that a dangerous action has occurred. The webmaster will need to manually decide if the GA enters the whitelist.
- In the browser we want to check if these negative tags are not a simple update of a known page which causes the hash to change. In order to do that we look for the most similar scripts. We do that by calculating the distance between the unknown script opcode histogram and every other histogram in the LGA. The closest GAs are the ones that their distance is the smallest. Once we found the most similar scripts we find the difference between their negative tags:
 - If the difference between the negative tags contain only suspicious actions we allow the execution.
 - If the difference between the negative tags contain at least one dangerous action the execution is being blocked.

The GA server can also perform deeper analysis on the code with other tools once it is being submitted to allow the webmaster take a more knowledgeable action. If the webmaster chooses to allow the GA, it is now considered as a legal GA (LGA) and it gets into the legal generalized assembly set (LGAS), while if the webmaster chooses to block the GA it is considered now as illegal GA (IGA) and it gets into the Illegal generalized assembly set (IGAS). Both of these sets hold GA hashes, but some are marked as legal and the other marked as illegal.

So in case a client knows GA and recognizes it as LGA, it allows the execution without further

action, and if the GA is recognized as IGA it is being blocked without any further action.

Chapter 7

Learning & Publishing

In the following chapter we explain more deeply the generalization process of JavaScript assembly. Then, we discuss the learning mode of the system, which creates the whitelist that is being used during the detection & prevention modes. We talk about how the solution publishes the whitelist to the clients (browsers).

7.1 The generalization process of JSA

As explained before the main goal of the generalization function that generates GAs from the JS Assembly is to create a minimal series of Generic Assembly opcodes that represent the legal JavaScript code in the web application. The algorithm of the generalization function that generates Generic Assembly from a JSA iterates through all the opcodes in the JSA and modifies them as follows:

1. Ignore the all the values of the opcodes, since in the Generic Assembly we are only interested in the opcodes themselves that represents the behavior, not the data itself (If an injection of JavaScript code happened which changes the behavior of the web application, must add or change opcodes, not just data).
2. Ignore opcodes that their existence doesn't affect the outcome of the code. For example the following opcodes have been ignored in our Firefox implementation: JSOP_NOP,

JSOP_STOP, JSOP_THIS, JSOP_NEW.

3. Modify all opcodes that represent a data type to literal, while making sure the two sequential literals turn to one literal. For example in our Firefox implementation: JSOP_INT8, JSOP_UINT16, JSOP_STRING → literal, literal, literal → literal.
4. Modify all opcodes that have the same goal into one unique opcode. For example in our Firefox implementation: JSOP_CALL, JSOP_CALLGLOBAL, JSOP_CALLLOCAL, JSOP_CALLGNAME would all change into JSOP_CALLLITERAL.
5. Modify all opcodes that have performed a logical operator into one unique opcodes. For example in our Firefox implementation: JSOP_IFEQ, JSOP_IFNE, JSOP_AND, JSOP_OR would all change into JSOP_LOGICLITERAL.

Note: the exact details of the generalization function depend on the exact details of the JSA generated by each JavaScript engine. However, the concepts are the same and can be easily tailored to each compiled JavaScript engine. Following the above rules, any variant of JS Assembly that has been generated from a legal usage of the web application will be translated into the one of the LGAs that has been generated for that web application and placed in the LGAS. Any code injection that change the behavior of the JS code will ultimately lead to a different GA that will not appear in the LGAS and as a result will be detected as illegal GA (or IGA for short) and will be blocked by our mechanism.

7.1.1 Engine specific implementation

The proposal is implemented as part of the JavaScript engine itself and being tailored to its JS Assembly. Therefore the solution must be tailored made to the engine's JS Assembly, so does the LGAS database being provided to the clients. With that we should stress out that there aren't many popular JavaScript engines out there.

In the evaluation chapter (chapter 9) we will show that the generalization function we implemented in the POC does converge, which means that after a short time of surfing through the web application we collect a finite set of GAs that present all the JSAs the application creates.

7.2 Learning LGAS of a web application

In order to generate a LGAS for a new web application, or update an existing LGAS, our mechanism needs to “learn” the new JSAs in order to create GAs for them. To do that the solution is being executed in “learning mode”. “Learning mode” is being performed in a safe environment (and the activities are legal ones), and as a result we can assume that all JSAs are legal and every GA being created during learning mode is considered a LGA. Thus, in order to learn a new web application, a legal user(s) surfs through the web application, making sure all scripts are being executed at least once.

The update process of LGAS is similar to creating a new one. Once the website has been updated, we surf the website and replace the new LGAS with the old one. Another way to update is relying on the negative logic approach (section 6.3) to send the GA server unknown hashes and then add them to the whitelist. After all scripts have been compiled at least once, we have a set of LGAs describing the legal behavior of the application. During “detection and prevention” mode every known GA being generated that has been found in the LGAS is allowed. If GA has been found in the IGAS, is blocked, and if not found at all, we fall back to negative approach and cache the result.

Since generating a new LGAS requires surfing through the web application, we suggest that during the quality tests performed by the QA team responsible for checking the website, our solution can run in the background in “learning mode”, producing the LGAs. This way by the time the tests on the web application are over, the LGAS is ready to be published to the world.

We should point out that there is one type of web application that cannot be converged. A web application that its goal is to receives as an input JavaScript code from the user and returns in the response a web page containing that JavaScript code. This kind of web application, which its design and goal is to perform XSS, can create all possible JavaScript codes. Therefore, it is impossible to prevent XSS in this website, simply because preventing XSS means preventing the web application from doing its job.

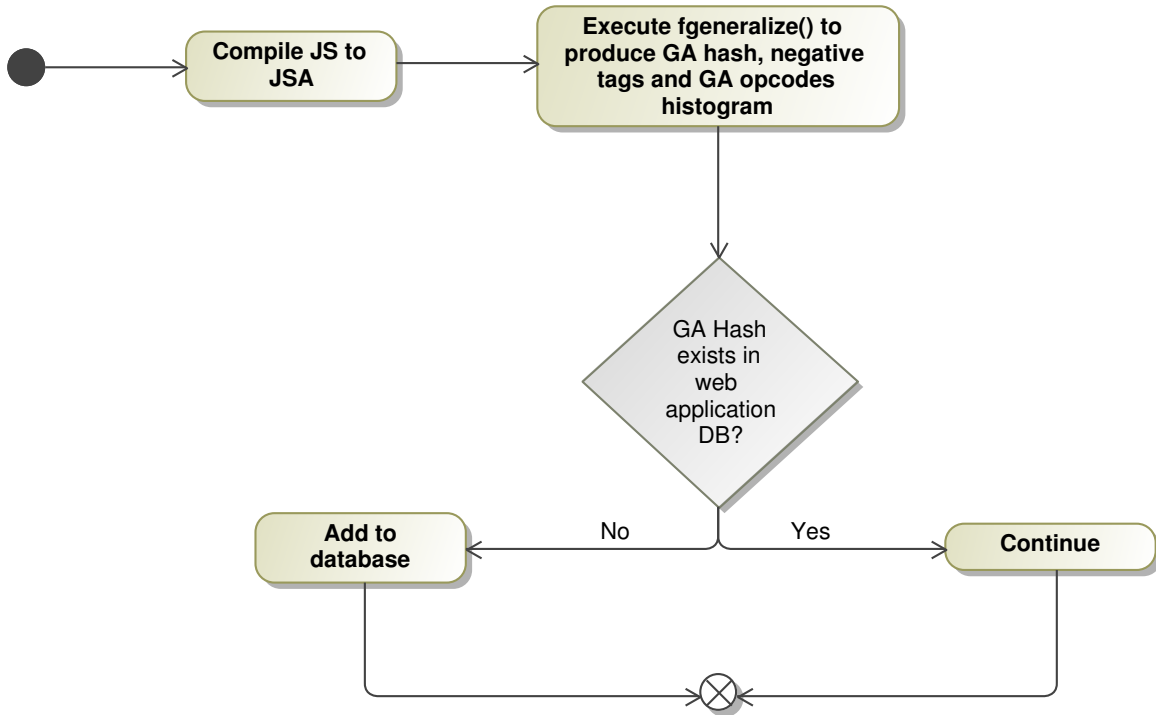


Figure 7.1: Learning mode activity diagram

7.3 Solution flow charts

7.3.1 Learning flow

In the following section we illustrate the flow of the learning process and detection & prevention process using activity diagrams. Every time JS code is being compiled into JSA, we call $f_{generalize}()$ to produce $hash(GA)$, negative tags which tag scripts with suspicious or dangerous actions and a histogram of the GA OPCODEs.

If the $hash(GA)$ is found in the database, we continue because we already know that GA, otherwise we add it to the database. We keep in the database the output of $f_{generalize}()$ by using $hash(GA)$ as the key.

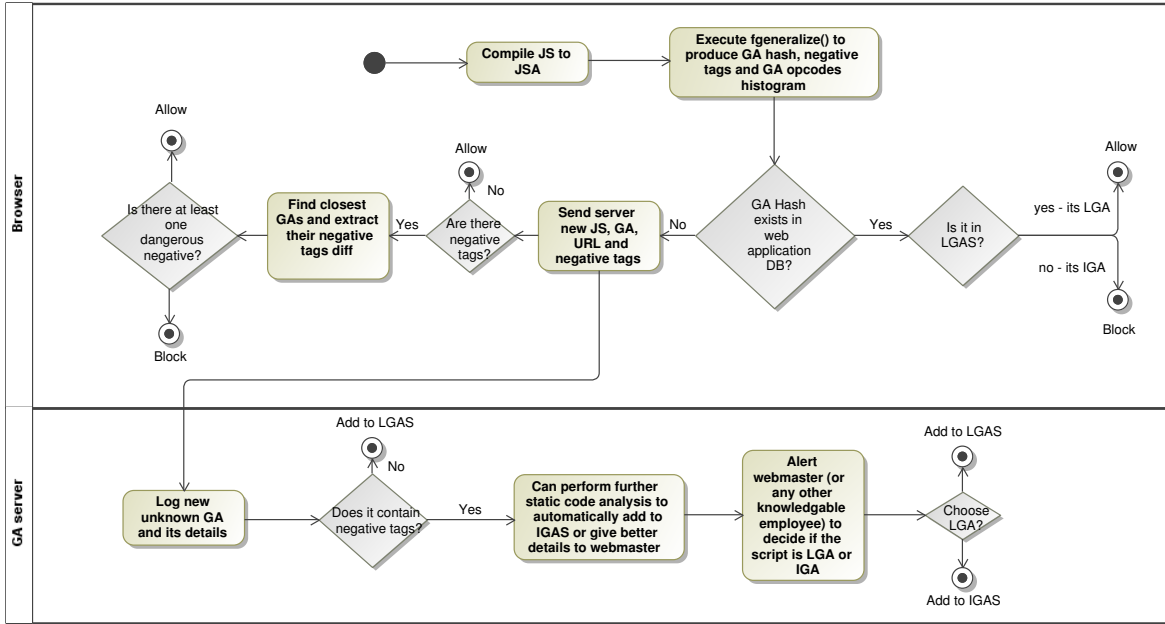


Figure 7.2: Detection & prevention activity diagram

7.3.2 Detection & Prevention Flow

Every time JS code is being compiled into JSA, we call $f_{generalize}()$ to produce $hash(GA)$, negative tags which tag scripts with suspicious or dangerous actions and a histogram of the GA OPCODEs.

In case $hash(GA)$ is found in the database, we check whether it is an LGA or IGA. In case of an LGA, the script is known and legal therefore we allow the script, on the other hand if the script is IGA we know the script and know it is illegal, therefore we block the script.

In case $hash(GA)$ is not found in the database, we face an unknown script. First of all we send the script to the GA server for further inspection. In the meantime the browser must decide how to react, whether to block or allow the script.

If there are not negative tags to the script we conclude that it contains no suspicious or dangerous scripts, therefore we allow the script and cache the result. If it does contain negative scripts we will look for the most similar scripts. There might be a website that uses dangerous scripts for its legal operation and our new script is a small update of the former script, so by

finding the most similar scripts we can try and see if there are any new negative tags. We determine the similarity by calculating the distance of the two scripts' histograms. The most similar scripts are the ones that have the smallest distance between the histograms, meaning they have the most common GA OPCODEs. We check if there are any differences between the negative tags of the current script and the most similar scripts. In case these differences contain dangerous negative tags, we block the script, otherwise we allow it.

7.4 Publishing GAS to browsers

Once GAS (Generalized Assembly Sets which contains LGAS and IGAS) of the web application is ready, we need to make sure the detection and prevention mechanism in the clients (browsers) receive the GAS of our application. We suggest publishing the GAS of a web application using the same mechanism being used to publish CRL (certificate revocation lists), i.e. a downloadable GAS file (based on CRL file publishing):

Each web application that has GAS, places at least 2 files in a predefined path:

- gas.dec - descriptor file
- gasX.dat - LGAS for engine X

A client checks the predefined path in the website, looking for the GAS file descriptor (for example: <http://www.somesite.com/gas.dec>). The descriptor contains the gasX.dat publish date (version), where X changes for each engine. For example:

- gasFF.dat is the GAS of the web application for Firefox
- gasChrome.dat is the GAS of the web application for Chrome

Notice that the descriptor may also contain more information like hash of gasX.dat and a signature on its data. The client first checks if it already downloaded the file, if not, it downloads the gasX.dat and replaces it with its own. The client should check gas.dec when it surfs to a web application that the client does not have LGAS for it (of course, the absence

of `gas.dec` means there is no GAS for this web application). If a client detects a JSA that produces GA that does not appear in either of the GAS (meaning, not in the LGAS nor IGAS) than before falling back to negative solution, the client should check if there is a new version of GAS. If so, the client downloads the new GAS and checks if the GA still does not appear in the GAS. If it appears in the new GAS, then the site has been updated, if not we fall back to negative logic. In order to minimize updating the GAS during runtime of JavaScript (and reduce performance), the client should check the `gas.dec` once in a while (for example, when surfing to the website for the first time in at least 24 hours).

Another possible design is instead of publishing the whole database, we can use a central server that clients can ask if a given GA hash is known or not, and keep the answer in a local database that acts as cache. Every time the client detects GA it doesn't recognize, it sends a request to the central server to ask if the hash is legal or not. We do not recommend using this design due to the impact this design might have over the browser performance. Requesting the server for each and every GA hash might produce thousands of requests if the user surfs to a web application for the first time or in case the web application has been updated. Furthermore, if we try to bootstrap a client with the whole database on the first time it surf to the web-application, an update can still generate many requests that might have a negative impact over the performance of the JavaScript compilation-execution process in the web application.

Chapter 8

Firefox SpiderMonkey Proof of Concept Implementation

To test our design we implemented the solution in Firefox’s SpiderMonkey on windows operating system under the codename Professor Monkey. SpiderMonkey compiles the JavaScript code into JSA called “bytecode”. The bytecode is then being executed by the JavaScript Engine to execute the code. The solution itself is placed in a DLL linked to SpiderMonkey named profmonkey.dll. The only change in mozjs.dll (SpiderMonkey) is that at the end of the compilation, instead of returning the bytecode, it calls professor monkey for inspection.

If a web application executes code using “javascript:” prefix, the browser does not have a URL, but an opaque URL (the URL is “javascript: ...”). In this case in order to get the URL of the page the solution we have implemented use the value of “document.URL”, which returns the URL of the current loaded page in that tab to extract the domain of the web application.

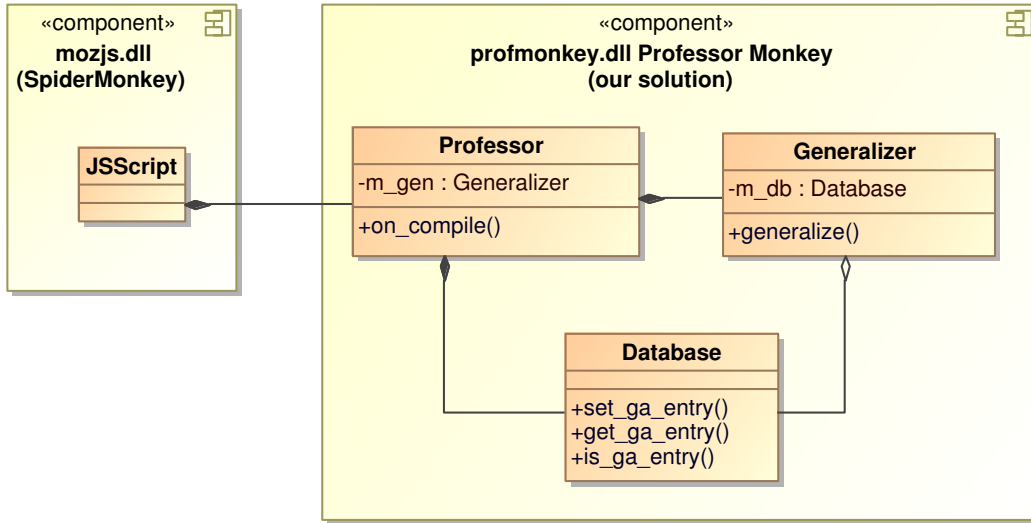


Figure 8.1: POC high-level structure

8.1 Components Overview

8.1.1 mozjs.dll

The component is Firefox’s SpiderMonkey, which is Firefox JavaScript engine. We have slightly modified it to call our DLL (section 8.1.2), and to allow or block the script by returning the original script, or execute alternate script and fail the compilation.

8.1.2 profmonkey.dll:

The component implements our solution. The component is linked with mozjs.dll and becomes part of the JavaScript engine. The design of this component is such that in order to support new JavaScript engine, the only method that it is needed to implement is the `generalize()`.

Professor:

Professor object is responsible for deciding whether to allow or block a given bytecode (JSA) during detection/prevention mode, or generating the LGA during learning mode. The Professor object executes the flows we explained in sections learning flow (section 7.3.1) and detection & prevention flow (section 7.3.2).

Generalizer:

Generalizer object receive bytecode-JSA and returns bytecode-GA. In our implementation the generalizer class has 1 function - generalize(). This class is the only class that needs to know SpiderMonkey's bytecode, therefore, in order to implement our solution in another JS Engine, the only change required is to implement a dedicated generalizer for that engine. The function receives bytecode from SpiderMonkey, and calculates the generic assembly. The bytecode in SpiderMonkey is a byte[] array. The generalizer receives byte[] as input, iterates all the bytecode OPCODEs copying them to the new generalized bytecode while applying the rules described in section 7.1, and returns a new byte[] that contains the GA. Also, during the iteration over the bytecode OPCODEs, the generalizer marks negative tags.

Database:

The database object manages the GA entries (both LGA and IGA). The object is the DAL between the professor and the database engine itself.

Database engine:

In our implementation we have expirimented with 2 types of databases:

- SQLite
- CRLF delimited files

We started with SQLite database, which was very convenient to work with but the INSERT operation into the database during learning mode is so slow that it actually slowed down the surfing in this mode to a point we couldn't use the browser.

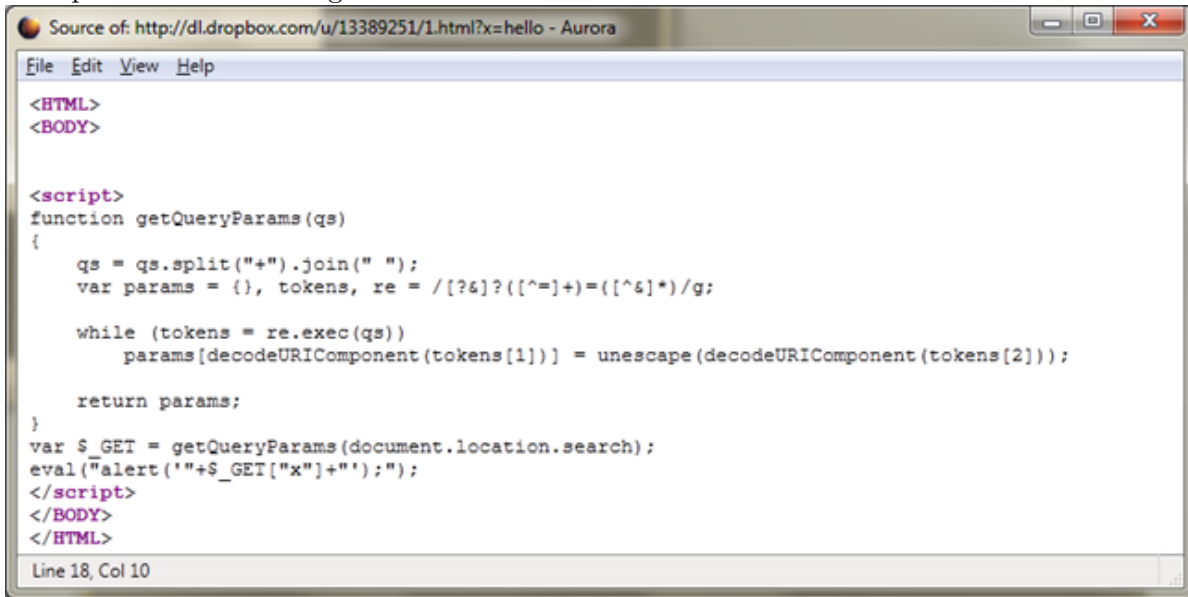
Next, we switched to simple text files using CRLF (CR=\r, LF=\n) as a delimiter between records, also every file contained GA database for a different web application. In this database the INSERT operation is very fast (by keeping a cursor to the file) and no performance hit was noticed.

Also, to lower the impact even more we use a cache system to keep the database of the opened web application in the memory.

8.2 POC execution example

In this section we will present an actual execution of our POC.

We uploaded the following HTML to the internet:



```
<HTML>
<BODY>

<script>
function getQueryParams(qs)
{
  qs = qs.split("+").join(" ");
  var params = {}, tokens, re = /[?&]?(?:^=+)=([^\&]*)/g;

  while (tokens = re.exec(qs))
    params[decodeURIComponent(tokens[1])] = unescape(decodeURIComponent(tokens[2]));

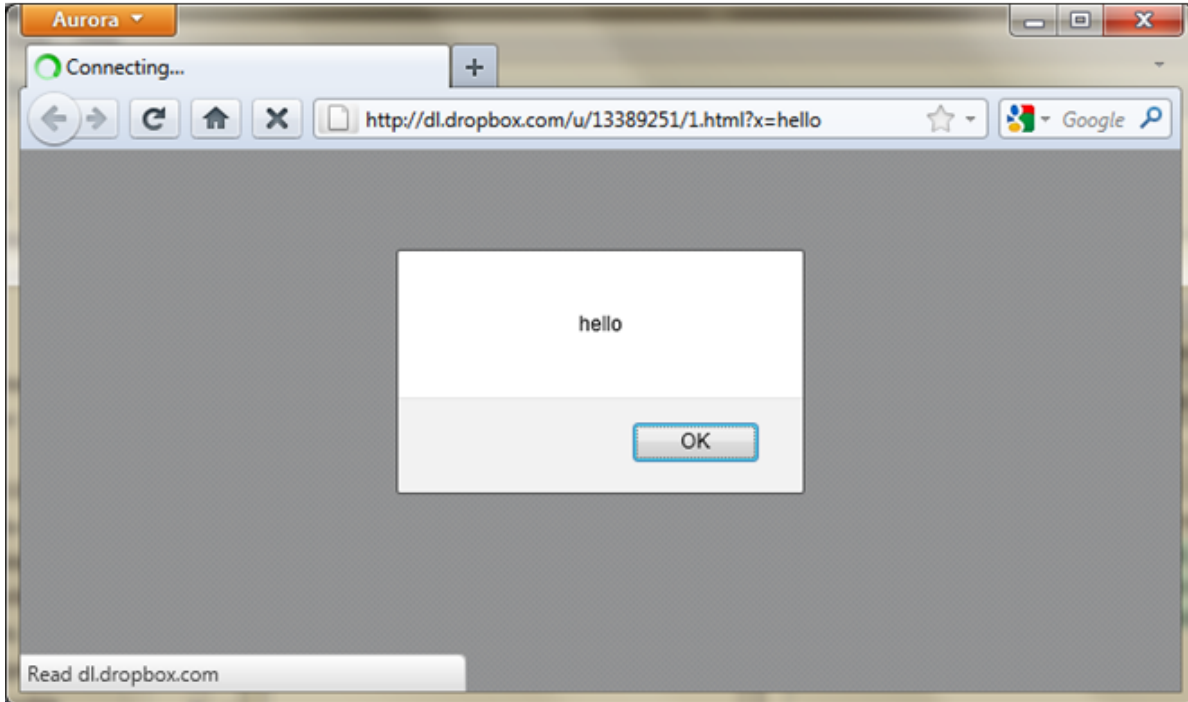
  return params;
}
var $_GET = getQueryParams(document.location.search);
eval("alert('"+$_GET["x"]+"');");
</script>
</BODY>
</HTML>
```

Line 18, Col 10

The code uses `eval()` to evaluate an `alert()` function that the text inside it comes from the variable “x” given to the application in the URL.

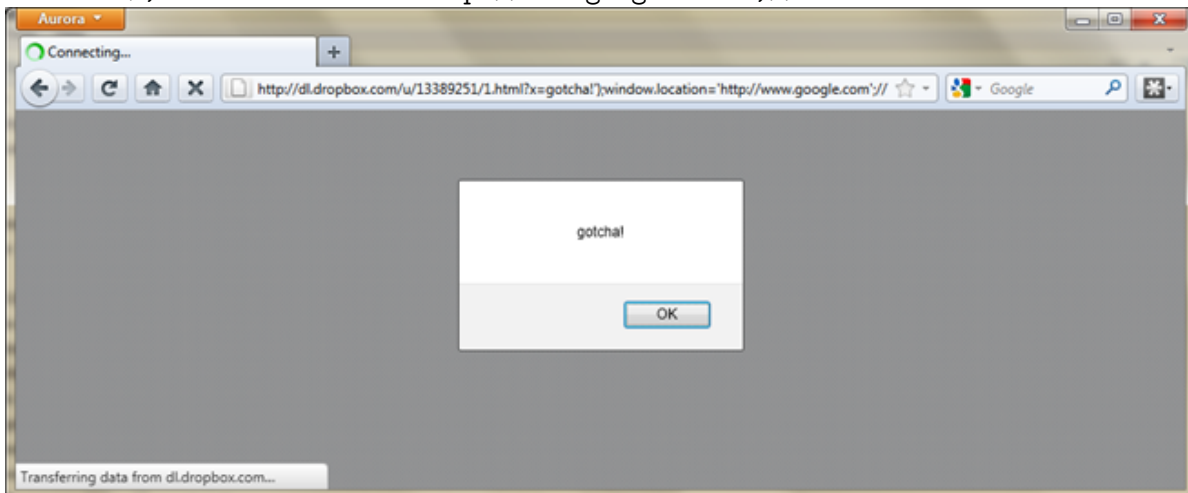
8.2.1 Normal execution

If we surf to the website we will get an `alert()` message box with the text given in “x”:



But if we place inside X the following string:

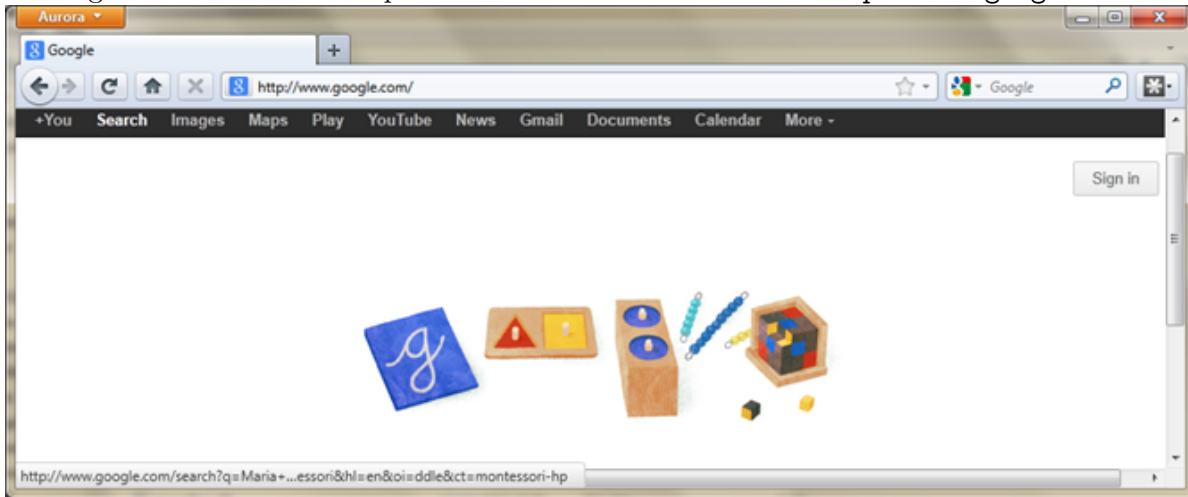
```
Gotcha!');window.location='http://www.google.com';//
```



The text “Gotcha!');” is being evaluated to the end the `alert()` function and allows the attacker to start a new JavaScript command.

Then the attacker inserts the text: “`window.location='http://www.google.com';`” which

is being evaluated to JavaScript code that redirects the user to <http://www.google.com>.



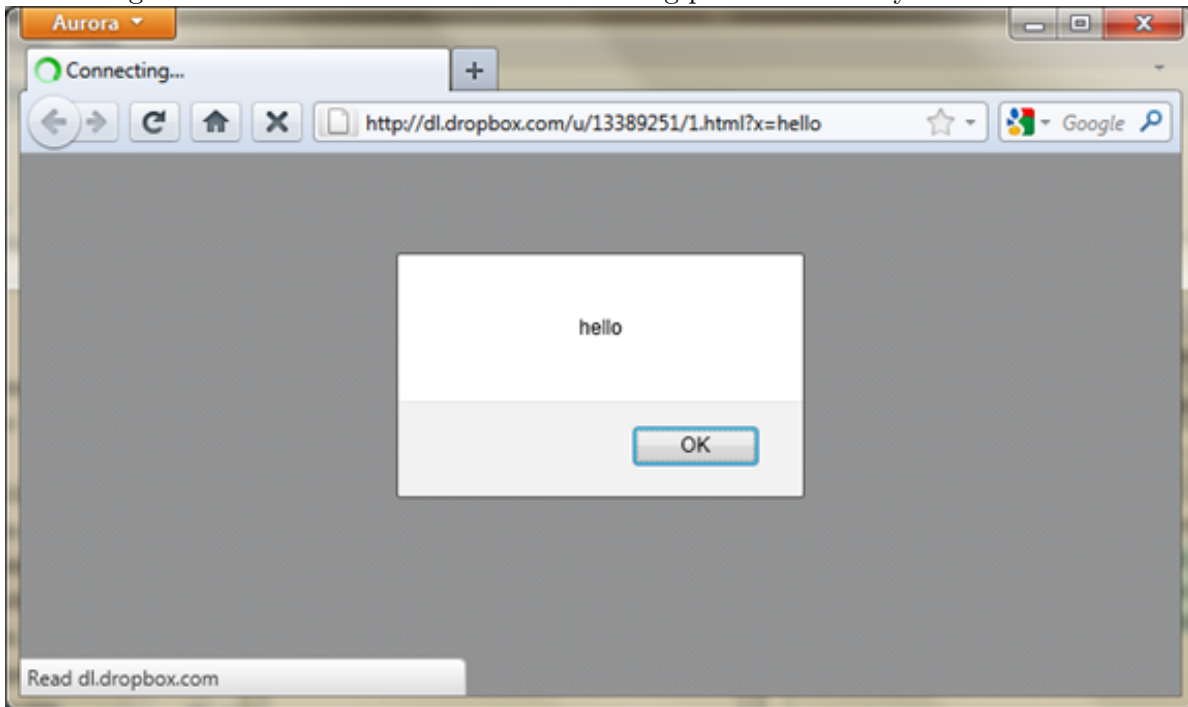
Finally the attacker uses the text “//” so it will be evaluated into a comment sign, therefore all the code after the redirection will be marked as a comment, preventing any errors that might prevent the attack from being executed.

So finally the code that is being evaluated is:

```
alert('gotcha');window.location='http://www.google.com';//'
```


8.2.2 Learning mode

In learning mode we will surf to the website allowing professor monkey to build the whitelist.

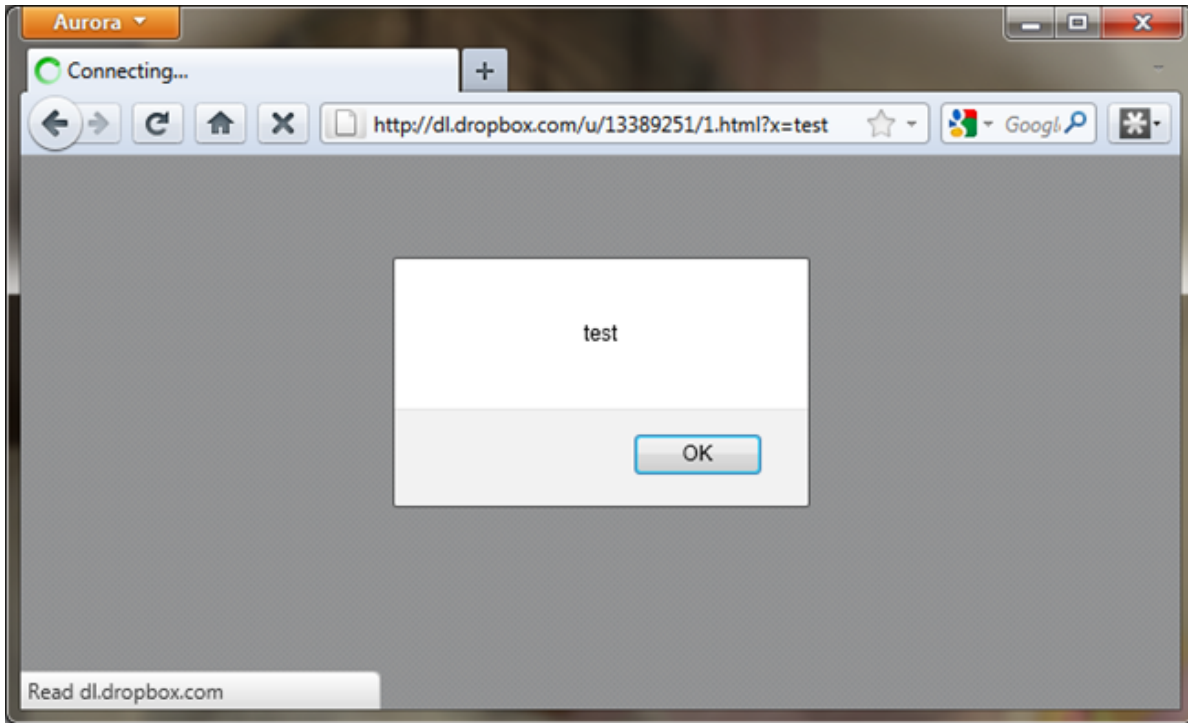


The surf in learning mode resulted in 3 new GA hashes inside the LGAS (eval() has been detected as a suspicious behavior):

```
e3273f62afe62b2921fa25e78706c7e1b06664ed ()  
1dfc9ba4faab38fa56dee74b96cce4d8605c3242 (eval|1;)  
c66be7210915f39e91456fc2eac9441012a0a3ea ()
```

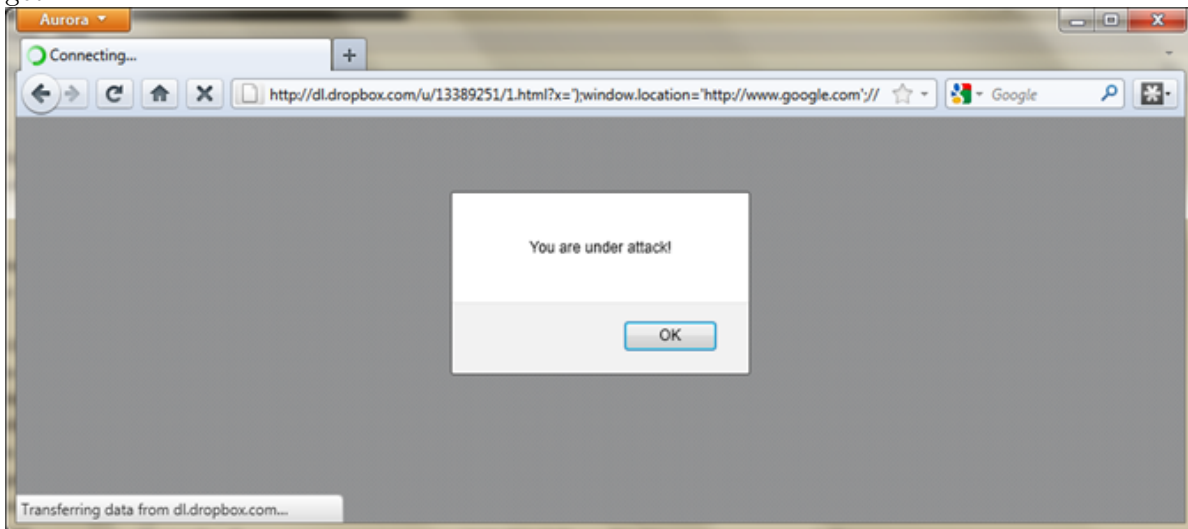
8.2.3 Detection and prevention mode

Now, we switch the browser into detection & prevention mode. Every JSA will be turned into hash(GA) and will be searched in the database. If the hash(GA) cannot be found inside the database it will be considered as an attack. First, let's run a legal page:



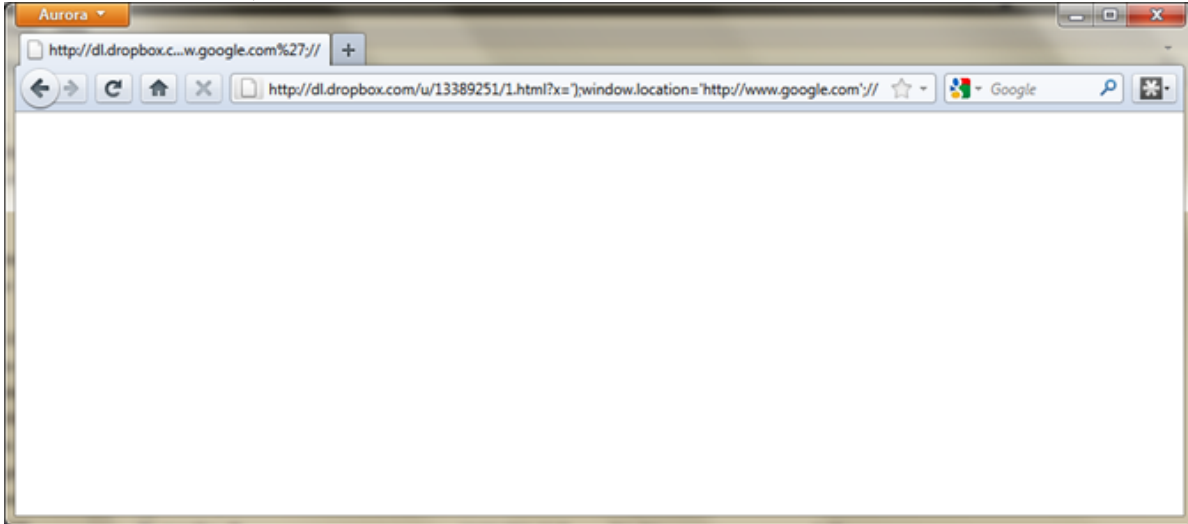
The legal execution created only hash(GA)s that exist in the LGAS, this is why we got the actual result we wanted.

Now let's try to inject new JavaScript code we used earlier which contains redirection, and we get:



This time the hash(GA) created does not appear in the database, that means new unexpected code has been compiled. Also, we detect that `window.location` is being set, which is consid-

ered a dangerous action, therefore in this case professor monkey displays an alert message box alerting the user he/she “under attack”, and fails the malicious code compilation:



Chapter 9

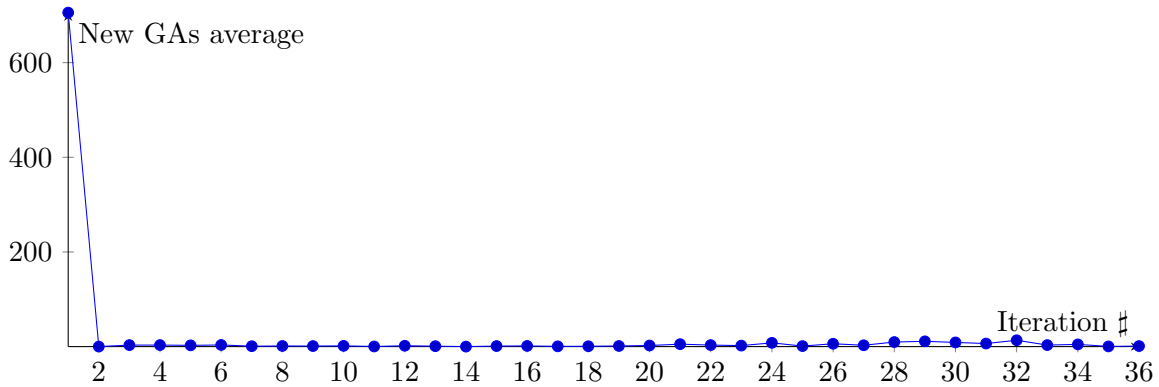
Evaluation

In this chapter we detail the evaluation process of the system using Firefox’s SpiderMonkey and present the results.

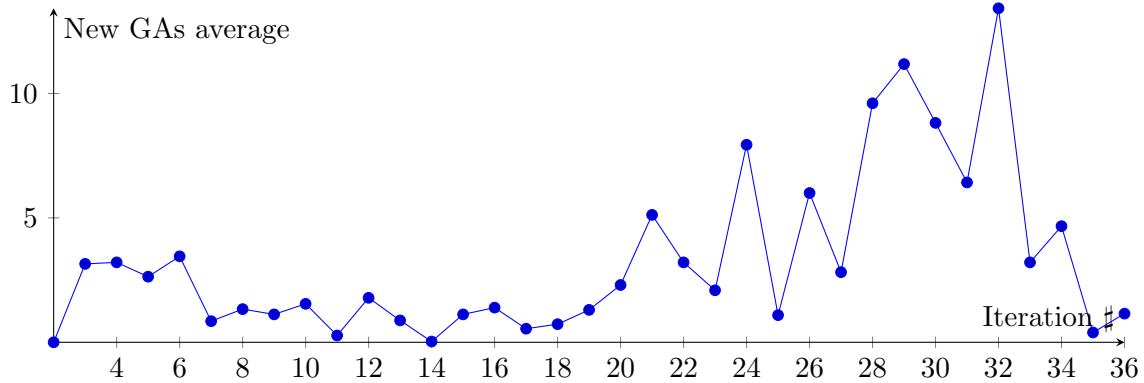
9.1 Learning Process

In order to make sure that we can generate a finite set of legal GAs, we learned using our system 33 sites (detailed in appendix C) out of Alexa top sites [2] for 72 days, executing the learning phase on the websites every other day.

In order to produce the GAs we crawled through all of these website, although due to the large scale of these websites we decided to do the evaluation only on the first 100 pages we came across. First, we used a web crawler to generate a list of these 100 pages per website. Next, the list was given to Selenium WebDriver [24] to automate the surfing of the Firefox POC through these 100 pages per website. Since we are in the learning mode during this surfing prof-monkey generated the GAs for these 100 pages per website. In order to verify that the learning mode is converging, we surf again through the 100 pages until we reach two full consecutive iteration without any new GA being detected and added to the LGA. The reason we need more than 1 iteration is because the web sites might add cookies or other data that



(a) Average of new GAs per iteration for 33 Alexa sites



(b) Average of new GAs per iteration for 33 Alexa sites (without initial iteration)

Figure 9.1

can change the behavior of the website.

In figure 9.1(a) we can see all the new GAs being detected as a function of learning execution iteration. In this figure we can easily see that most GAs are being detected in the initial iteration (1st iteration). In figure 9.1(b) we can see the same data but without the initial iteration. As we can see in this figure, every once in a while the site is being updated which leads to updating the GA set quite frequently. Also, we have noticed that some websites build their JavaScript code dynamically which makes it hard to learn all the possible GAs up front. Due to these reasons we cannot simply use the GAs we've learned as a whitelist to the application, but we also need to use a blacklist mechanism to deal with the new GAs (as explained in subsection 6.3). We present the results of the blacklist accuracy in the next section (specifically subsection 9.2).

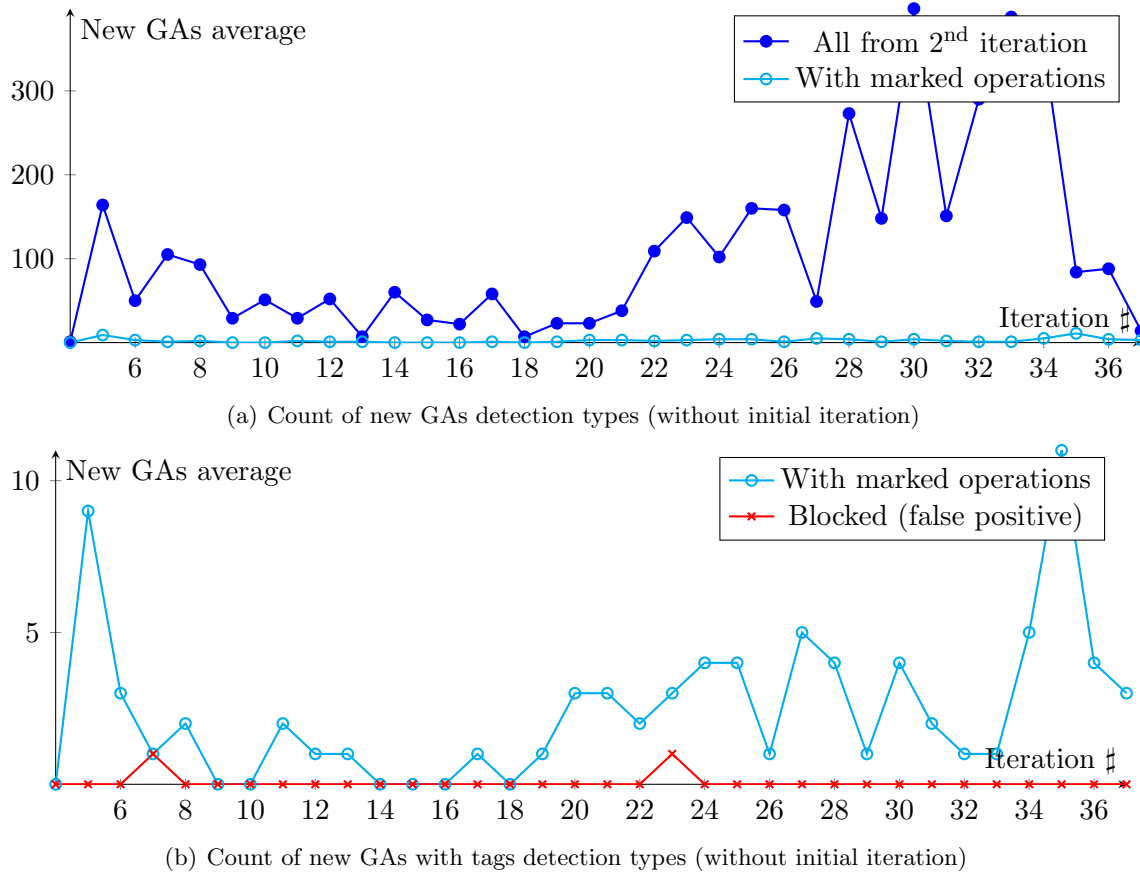


Figure 9.2

9.2 Accuracy Measurements

9.2.1 Testing for false positives

The blacklist mechanism detects suspicious and dangerous actions in JSAs that produce unknown GAs and allows the JavaScript engine to handle the new JSAs. Suspicious actions are operations that are considered as being used in an attack, but they are not dangerous by themselves, for instance, `eval()` or `document.write()` function. Dangerous actions are operations that can be used to attack the user and oppose a threat by themselves. For instance, setting the property `window.location` can be used to redirect the browser.

In our POC we have decided that these actions are suspicious:

- `eval()` function

- `document.write()` method
- Get of `document.cookie` property
- Get of `document.location` property

and these actions are dangerous:

- Set of `document.cookie` property
- Set of `document.location` property
- `window.open()` method
- `document.write()` method if writing text an *iframe*

As we pointed out in section 6.3, it is important to detect not only the direct use of the operations but also an indirect use.

With these actions defined we allow to execute JSAs that produce unknown GAs that have no dangerous operations in them. The data of the suspicious and dangerous actions is being sent to the website's server as extra data to the webmaster to allow him decide better if it is a legal JS or if there is higher potential that the web application is being attacked. JSAs with dangerous actions is being blocked until the GA is officially added to the legal set of GA (WhiteList).

Figure 9.2(a) shows the total number of new GAs in comparison to new GAS that has been marked as suspicious or dangerous. In fact, only 2.39% of all the GAs (excluding initial iteration) contain suspicious or dangerous GAs.

Figure 9.2(b) show the number of new GAs that has been marked as suspicious or dangerous to the actually blocked scripts. By the graph we can see that the solution false positives (blocked scripts) are 4 GAs with dangerous operations, which is 0.08% out of all the GAs excluding the initial iteration. That means that out of all the GAs marked with suspicious or dangerous

tags, 96.58% are allowed because of similarity to previously allowed GAs (no difference by their histograms as explained in section 6.3).

Notice that the graphs starts from the fourth iteration in order to produce better scaling in the presented graphs. The third iteration total new GAs (in all the 33 websites) is 1137 new GAs due to a website update. Using this iteration in the graph would change the scale dramatically, therefore it was removed.

These results show that the usage of the histogram similarity in our solution provides a very low rate of false positives.

9.2.2 Testing for false negatives

In order to test our solution for false negatives we created a small website (code can be found in appendix A) that uses both `eval()` and `document.write()` as part of its normal usage. The website is a calculator that receives number x , number y , binary operator op and calculates the result of x and y by applying op over them (using `eval()`). The result is being written on the page using `document.write()`. For example, if $x = 1$, $y = 2$ and $op = +$ then the returned page is the text: $1 + 2 = 3$.

After learning the website, we executed all the attack vectors in OWASP [21] that successfully got exploited in our Firefox browser, also we have tried a few more evasions of our own. The attack vectors are detailed in appendix B. In our tests all code injections we have tried have been blocked successfully by the solution, while the website worked correctly when the input was not JavaScript code.

9.3 Performance measurements

We have evaluated the performance hit detection & prevention mode in our (non-optimized) POC on Gmail. The reason we chose Gmail is because it is very rich in JavaScript compared to

other websites. The time added to the web page loading time by the browser about 1.078_{sec} per page in normal operation mode (i.e. when the mechanism is working in detection and prevention mode). This additional time is due to the fact that gmail web pages are heavily using JavaScripts and in average number of JSAs per page is about 4900. The generalization function itself takes about 0.0127_{ms} per JSA, and the average time of the whole professor monkey module is about 0.22_{ms} per JSA, and when multiples by 4900 we get the 1.078_{sec} . We should point out that most of the compilations occur during the first loading of Gmail web application, but it does not change the average. Web sites that use less JavaScript the additional time per web page loading will be significantly smaller, Thus, even these performance prove that our solution is a practical solution, especially that we strongly believe that our solution implementation can be optimized to reduce the time it takes to perform its JavaScript validation.

Chapter 10

Summary

In this work we presented a mechanism to detect and prevent XSS attacks by a modification of a JIT JavaScript engine that allows the JavaScript engine, using a generalization function, to convert each JavaScript assembly code generated by the JIT JavaScript engine to a generalized form and compare it to a whitelist of a given web application (that is created by a learning phase). We have implemented a POC of this mechanism and have shown that the learning phase is converging very fast to a finite list of generalized JavaScript assembly code, and as a result the learning phase is short and the mechanism will not generate false positives. The uniqueness of this mechanism due to its location within the JavaScript engine that it can detect and prevent XSS attacks that are generated by other JavaScript code, and it handles natively JavaScript generated by JavaScript, DOM-based attacks, non-HTML object executed by plug-in and addons. In addition, the solution we presented does not require any action from the developers to define a policy that could be incomplete or hard to craft.

The solution we presented discusses JavaScript assemblies being produced only in JIT-based JavaScript engine, but the solution can also be implemented in interpreter-based engines by replacing the generalization function to work on JS code instead of JSA code (use a canonicalization function like the one presented in Rotberg and Movshovitz paper [18]).

The naïve implementation of the generalized JSA database is vulnerable to return-to-libc attack (executing legal code of the application in another place in the application), but this can

be solved by adding more information to the hash placed in the LGAS, for example instead of hashing only the JSA, we can hash JSA+URL, thus by executing the code in another page will be detected as an attack. The more unique data we hash, the harder it is to use the attack (but it can affect the time of the learning phase).

It is important to mention that the presented approach can be extended to other JIT-based engine, not just JavaScript. Using the whitelist, at the end of the “compile” function, the engine can check if the generalized-version of the compiled code is known and legal or should be blocked.

Appendices

Appendix A

Prevention Testing of JavaScript code

We wanted to build a vulnerable website that uses both `eval()` and `document.write()`. Therefore we created a calculator that receives from the URL the parameters x , y and op and performs the operator op over x and y . For example, if $x = 1$, $y = 2$, $op = +$ then the web page looks like: $1 + 2 = 3$. The function `getQueryParams()` has been written by Ates Goral in stackoverflow.com [8].

```
<HTML>
<BODY>
<script>
function getQueryParams(qs)
{
    qs = qs.split("+").join(" ");
    var params = {}, tokens, re = /[?&]?(?:[=]?)=([~&]*)/g;
    while (tokens = re.exec(qs))
        params[decodeURIComponent(tokens[1])] = unescape(decodeURIComponent(tokens[2]));
```

```
    return params;
}
var \$_GET = getQueryParams(document.location.search);
document.write(\$_GET["x"]+\$_GET["op"]+\$_GET["y"]+" =
"+eval(\$_GET["x"]+\$_GET["op"]+\$_GET["y"]));
</script>
</BODY>
</HTML>
```

Appendix B

Attacks Prevented

The following list are all the XSS attack vectors that worked on our Firefox POC. We have tried all of them against the system, and all of them have been blocked successfully simply because it created new GA hashes that did not exist in the website detailed in appendix A. Most of them are from OWASP XSS filter evasion cheat sheet [21], and the rest has been created by us.

1. `?x=1&y=2;alert('xss');&op=+`
2. `?x=document.location&y=www.google.com/'&op=ation='http://w`
3. `;alert(String.fromCharCode(88,83,83))`
4. `<aonmouseover="document.location='http://www.google.com/'">xxslink`
5. `<IMG"""><SCRIPT>document.location='http://www.google.com/'</SCRIPT>">`
6. `<IMGSRC=#onmouseover="document.location='http://www.google.com/'">`
7. `<IMGonmouseover="document.location='http://www.google.com/'">`
8. `<IMGonmouseover=“document.location='htttin.com'">`

Appendix C

Learned Websites List

In the following table we present the results of the learning mode tests we conducted on the Alexa top-sites. The table is built as follows:

- WebSite - The website the test has been conducted upon
- Total GAs - The total number of GAs the website generated
- The second part of the table **excludes** the initial iteration and contains the following columns:
 - Total GAs - The total number of generalized assemblies the website generated
 - GAs marked - GAs marked with at least one suspicious or dangerous tags
 - Blocked GAs - GAs marked with at least one dangerous tags, therefore being blocked (false positive of the solution)

WebSite	Total GAs	Total GAs	GAs marked	Blocked GAs
163.com	1629	229	5	1
adobe.com	1370	233	13	1
alibaba.com	307	6	2	0
amazon.co.jp	145	33	1	1

amazon.com	891	124	2	0
aol.com	1738	143	8	0
ask.com	1012	46	0	0
babylon.com	290	9	0	0
bbc.co.uk	634	359	6	0
bing.com	202	18	0	0
craigslist.org	333	5	0	0
ebay.com	752	94	2	0
fc2.com	460	61	0	0
flicker.com	146	82	9	0
go.com	1307	365	13	0
google.com.hk	1430	252	3	0
google.com.tr	943	109	0	0
google.de	1385	295	3	0
google.es	1355	222	3	0
google.fr	1390	313	3	0
google.com	1231	292	9	0
imdb.com	458	8	0	0
microsoft.com	687	20	1	0
pinterest.com	710	255	6	0
piratebay.se	133	0	0	0
rakuten.com	1349	161	3	0
soso.com	333	11	0	0
stackoverflow.com	139	50	3	0
tumblr.com	988	149	1	0
wikipedia.com	523	54	1	0

yahoo.com	590	56	3	0
yandex.ru	546	43	1	0
youku.com	1118	277	16	1

Bibliography

- [1] Adobe. Vulnerabilities in some swf files could allow cross-site scripting. <http://www.adobe.com/support/security/advisories/apsa07-06.html>, 2007. Accessed:19.11.2012.
- [2] Alexa. Alexa top 500 global sites, 2013.
- [3] Elias Athanasopoulos and Evangelos P. Markatos. Code-injection attacks in browsers supporting policies. In *In Proceedings of the 2nd Workshop on Web 2.0 Security & Privacy (W2SP)*, 2009.
- [4] HP Security Laboratory Blog. Top ten web application vulnerabilities 1/31/2011 - 2/21/2011. <http://h30499.www3.hp.com/t5/The-HP-Security-Laboratory-Blog/Top-Ten-Web-Application-Vulnerabilities-1-31-2011-2-21-2011/ba-p/2408251>, 2011. Accessed:19.11.2012.
- [5] CERT. Cert advisory ca-2000-02 malicious html tags embedded in client web requests. <http://www.cert.org/advisories/CA-2000-02.html>, 2000 2000.
- [6] KF and DP. Xss attack information. <http://www.xssed.com/>, 2007. Accessed:19.11.2012.
- [7] Giovanni Vigna Engin Kirda, Christopher Kruegel and Nenad Jovanovic. Noxes: A client-side solution for mitigating cross-site scripting attacks, 2006.
- [8] Ates Goral. how to get get and post variables with jquery? - stackoverflow.com, 2009.

BIBLIOGRAPHY

- [9] Jeremiaiah Grossman, Robert "RSnake" Hansen, Petko "pdp" D. Petkob, Anton Ranger, and Seth Fogie. *XSS Attacks - Cross site scripting exploits and defences*. Syngress, 2007.
- [10] CuteSoft Components Inc. Free javascript obfuscator. <http://www.javascriptobfuscator.com/>. Accessed:19.11.2012.
- [11] S. Jayamsakthi and M. Ponnaivaikko. Risk mitigation for cross site scripting attacks using signature based model on the server side. In *Proceedings of the Second International Multi-Symposiums on Computer and Computational Sciences*, IMSCCS '07, pages 398–405, Washington, DC, USA, 2007. IEEE Computer Society.
- [12] Trevor Jim. Defeating script injection attacks with browser-enforced embedded policies. In *In WWW*, pages 601–610, 2007.
- [13] Samy Kamkar. Technical explanation of the myspace worm. <http://namb.la/popular/tech.html>, 2005. Accessed:19.11.2012.
- [14] Amit Klien. Dom based cross site scripting or xss of the third kind. <http://www.webappsec.org/projects/articles/071105.html>, 2005 2005. Accessed:2012.6.5.
- [15] Benjamin Livshits Leo Meyerovich. Enforcing end-point security with conscript. oakland10.cs.virginia.edu/slides/conscript.ppt, 2010. Accessed:17.6.2012.
- [16] Benjamin Livshits and Leo Meyerovich. Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser, 2009.
- [17] Mike Ter Louw and V. N. Venkatakrisnan. Blueprint: Robust prevention of cross-site scripting attacks for existing browsers, 2009.
- [18] Dr. David Movshovitz Ofer Rotberg. New approach to xss detection using javascript modeling, 2010.
- [19] OWASP. Top 10 2007 - owasp. https://www.owasp.org/index.php/Top_10_2007, 2007. Accessed:19.11.2012.

BIBLIOGRAPHY

- [20] OWASP. Top 10 project. https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project, 2010. Accessed:19.11.2012.
- [21] OWASP. Xss filter evasion cheat sheet - owasp. https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet, 2012 2012. Accessed:10.October.2012.
- [22] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for javascript.
- [23] WhiteHat Security. Winter 2011, 11th edition - measuring website security: Windows of exposure. https://www.whitehatsec.com/assets/WPstats_winter11_11th.pdf, 2011. Accessed:19.11.2011.
- [24] Selenium. Selenium - web browser automation, 2013.
- [25] 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, WWW '10*, pages 921–930, New York, NY, USA, 2010. ACM.
- [26] Robert A. Martin Steve Christey. Vulnerability type distributions in cve. http://cve.mitre.org/docs/vuln-trends/index.html#overall_trends, 2007 2007. Accessed:11.19.2012.
- [27] Joel Weinberger, Adam Barth, and Dawn Song. Towards client-side html security policies.
- [28] Joel Weinberger, Prateek Saxena, Devdatta Akhawe, Matthew Finifter, Richard Shin, and Dawn Song. A systematic analysis of xss sanitization in web application frameworks, 2011.
- [29] Wikinews. Wikinews interviews world wide web co-inventor robert cailliau. http://en.wikinews.org/wiki/Wikinews:Story_preparation/Interview_with_Robert_Cailliau, 2007 2007. Accessed:19.11.2012.
- [30] Wikipedia. Javascript -wikipedia, the free encyclopedia, 2013. [Online; accessed 22-March-2013].

BIBLIOGRAPHY

- [31] Wikipedia. Samy (computer worm) - wikipedia, the free encyclopedia, 2013. [Online; accessed 22-March-2013].

תקציר

התקפת "סקריפט חוצה אתרים" (Cross Site Scripting or XSS) היא אחת מהתקפות צד לקוח הפופולריות ביותר ברשת. ההתקפה משתמשת בטקסט שנשלח מהלקוח אל השרת והטקסט חוזר כקוד חזרה אל הלקוח. בדרך זו יכול תוקף להזריק קוד ענין לתוכנת הלקוח שיכולה לגרום לגניבת סיסמאות, מפתח-חיבור (session key) בין האתר ללקוח, סריקת הרשת הפנימית של הלקוח, בקשות שנעשות בשם הלקוח באתרים שונים וכו'. למרות שההתקפה ידוע שנים רבות, היא עדיין אחת ההתקפות הפופולריות ביותר.

רוב טכניקות ההתגוננות מפני ההתקפה משתמשות בלוגיקה שלילית (רשימה שחורה) על מנת לזהות את ההתקפות, אך טכניקות אלו חשופות לטכניקות התחמקויות רבות בהן קוד שמייצר קוד ואחרות שמגבילות את יעילות טכניקות ההגנה. בנוסף טכניקות הגנה רבות דורשות שינויי קוד או התערבות המשתמש, מה שהופך את הפתרונות לבעייתיים עקב דרישה לפיתוח נוסף שיכול לייצר פגיעות חדשה וכן אי תמיכה בתשתיות וספריות קוד קיימות או הסתמכות על ידע המפתחים או/וגם המשתמשים, מה שהופך פתרונות אלו לבעייתיים. מגבלה קשה נוספת היא מציאת הקוד לבדיקה אשר יכול להימצא בקבצי טקסט כמו HTML או js אך גם ב-Flash animation וקבצי pdf. בנוסף מציאת הקוד היא בעיה ששקולה לבעיית העצירה (בגלל היכולת של קוד לייצר עוד קוד) ועל כן, פתרון שצריך למצוא קוד בעצמו איננו מסוגל באמת לעשות זאת ותמיד יהיה מוגבל.

הפתרון שאנו מציגים מוצא בצורה יעילה ומדויקת התקפות XSS ע"י כך שאנו מציבים את המערכת שלנו בתוך מנוע הג'אווה-סקריפט, ובכך איננו צריכים להתמודד עם מציאת הג'אווה-סקריפט, אלא המנוע עושה זאת בשבילנו.

הפתרון משתמש בלוגיקה חיובית (רשימה לבנה) על מנת לאשר רק קוד מוכר, ובכך להפוך את הפתרון משמעותית לבטוח יותר. אנו עושים זאת ע"י שימוש בקוד ג'אווה-סקריפט מהודר, שהוא משמעותית יותר פשוט מקוד ג'אווה-סקריפט רגיל. ע"י שימוש בקוד המהודר אנחנו יכולים ליצר גרסה מוכללת של קוד הג'אווה-סקריפט המהודר ולהשתמש בו על מנת לייצר את הרשימה הלבנה ללוגיקה החיובית באנו משתמשים.

הבעיה בלוגיקה חיובית היא שאינה יודעת להתמודד עם עדכונים באפליקציית הווב לפני שמעדכנים את הרשימה הלבנה, ולכן במקרה שבא מגיע סקריפט חדש אנחנו משתמשים בלוגיקה שלילית מבוססת על היוריסטיקה עד אשר הרשימה הלבנה מתעדכנת.

לאחר 3 חודשי הערכה של הפתרון על גבי 33 מהאתרים הפופולריים ביותר ב-alexa.com מצאנו שהפתרון טעה רק ב-4 סקריפטים, וכן חסם את כל ההתקפות על אתר דמה שנלמד ונבדק.



המרכז הבינתחומי, הרצליה
בית ספר אפי ארזי למדעי המחשב

שיפור מנוע ג'אווה-סקריפט לזיהוי ומניעת "סקריפט חוצה אתרים"

תזה לתואר M.Sc.

מגיש : צבי צ'רני-שחר

מנחה : ד"ר דוד מובשוביץ

מרץ 2014