DIPARTIMENTO DI INGEGNERIA INFORMATICA
AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA
UNIVERSITÀ DI ROMA

Automatic and Context-Aware Cross-Site Scripting Filter Evasion

Fabrizio d'Amore
Mauro Gentile

Technical Report n. 4, 2012

Dept. of Computer, Control, and
Management Engineering Antonio Ruberti

Sapienza University of Rome

Technical Report

AUTOMATIC AND CONTEXT-AWARE CROSS-SITE SCRIPTING FILTER EVASION

October 2012

Fabrizio d'Amore
Dept. of Computer, Control, and
Management Engineering Antonio Ruberti
Sapienza University of Rome
damore@dis.uniroma1.it

Mauro Gentile
Dept. of Computer, Control, and
Management Engineering Antonio Ruberti
Sapienza University of Rome
gentile.mauro.mg@gmail.com

Abstract

Cross-Site Scripting (XSS) is a pervasive vulnerability that involves a huge portion of modern web applications. Implementing a correct and complete XSS filter for user-generated content can really be a challenge for web developers. Many aspects have to be taken into account since the attackers may continuously show off a potentially unlimited armory.
This work proposes an approach and a tool – named snuck – for web application penetration testing, which can definitely help in finding hard-to-spot and advanced XSS vulnerabilities.
This methodology is based on the inspection of the injection's reflection context and relies on a set of specialized and obfuscated attack vectors for bypassing filter based protections, adopted against potentially harmful inputs. In addition, XSS testing is performed in-browser, this means that a web browser is driven in reproducing the attacker and possibly the victim behavior.
Results of several tests on many popular Content Management Systems proved the benefits of this approach: no other web vulnerability scanner would have been able to discover some advanced ways to bypass robust XSS filters.

Keywords: computer security, network security, web application security, browser security, vulnerability detection, cross-site scripting, XSS

Contents

1 Introduction **5**

1.1 Overview 5
1.2 XSS Classification and Threat Model 6
1.3 Scenario and terminology 8
1.3.1 Basic filter based XSS prevention techniques 9
1.3.2 Web Application Security Scanners 9
1.4 Motivation 10
1.4.1 Workflow-based approach 10
1.4.2 Context-aware paradigm 11
1.4.3 In-browser scanner 12
1.5 Objectives 13

2 Defense and offense concepts **14**

2.1 Protecting against XSS 14
2.2 Attack scenarios 15

3 Methodology and Tools **19**

3.1 Stateful approach for penetration testing 19
3.2 Selenium: automating web browsers 20
3.3 Reflection context 20
3.4 Sets of attack vectors 21
3.5 Multiple browsers approach 23

4 *snuck*'s Architecture **24**

4.1 *snuck*'s architecture 24
4.2 Designing Use Cases 24
4.3 Modular set of attack vectors 27
4.4 Reverse Engineering process 27
4.5 Victim's behavior reproduction 28
4.6 Run example 29

5 Experimental Results **30**
5.1 Designing and starting the test 30
5.2 Test results 30
5.3 Evaluation and comparison 35

6 Future Work **42**
6.1 Future improvements 42
6.1.1 Client side XSS filter testing 42
6.2 Main limitations 45
6.3 Conclusions 46

7 Bibliography **47**

8 Sitography **49**

A Appendix **51**
A.1 Appendix 1 51
A.2 Appendix 2 52
A.3 Appendix 3 53
A.4 Appendix 4 53
A.5 Appendix 5 55
A.6 Appendix 6 56
A.7 Appendix 7 57

About this document

Preamble. This technical report derives from the Master's Thesis *"Automatic and Context-Aware Cross-Site Scripting Filter Evasion"* by Mauro Gentile, submitted in October 2012 in partial fulfillment of the requirements for the Master of Engineering in Computer Science at the School of Information Engineering, Computer Science, and Statistics of Sapienza University of Rome under the supervision of the prof. Fabrizio d'Amore.

Work's goal. The goal of this work is to propose an approach for significantly testing an XSS filter by reproducing the behavior of the attacker who insistently tries to break it. In addition, since we want to reduce the false positive rate to zero, we propose a new method to simulate the victim's behavior with respect to the occurred injection. We will illustrate these concepts in detail in the next chapters.

Paper organization. This paper is organized as follows. Chapter 1 gives a wide overview about modern web application security scanners, trying to explain their limitations and which are the methods we want to employ to test XSS filters. Chapter 2 explains the basic concepts behind Cross-Site Scripting prevention and presents many common attack scenarios. Chapter 3 describes the methodologies we adopted and the techniques we employed for building up the tool *snuck*. Chapter 4 explains the tool's architecture by giving particular attention to the operations it performs against an XSS filter. In Chapter 5 we present the performed experiments against popular Content Management Systems, whereas in Chapter 6 we propose several future improvements.

Chapter 1

Introduction

1.1 Overview

Cross-Site Scripting is a web application vulnerability in which malicious scripts are injected into a trusted web site by an attacker. This type of vulnerability has emerged as one of the most serious threats on the Web[1] since it revealed to affect a very large number of web applications.
On the one side it is usually straightforward to exploit an XSS vulnerability, on the other side it may be really a challenge to build a completely XSS-safe web site.[2] Hence, several research has been performed in order to detect or prevent unauthorized scripts from being included in the server output. The impact of such vulnerabilities may really harm an authenticated user since many techniques can be adopted to make a convincing exploit; stealing session information is just the tip of the iceberg since much more sophisticated attacks may take place once an injection point is detected, leading for instance to escalating privileges with only one click from the victim's perspective.
Cross-Site Scripting attacks are commonly underestimated by many web developers, who naively assume that stopping payloads like `<script>alert(1)</script>` is the most suitable way to lay up an XSS-safe web site. Unfortunately modern web browsers offer a huge set of possibilities for the attackers to execute malicious JavaScript. We will show many examples in this work.

[1] CWE, *2011 CWE/SANS Top 25 Most Dangerous Software Errors*, `http://cwe.mitre.org/top25/index.html` (Sep 2011)
[2] OWASP, *OWASP Appsec Tutorial Series - Episode 3: Cross Site Scripting (XSS)*, `http://www.youtube.com/watch?v=_Z9RQSnf8-g` (July 2011)

1.2 XSS Classification and Threat Model

As any good work regarding XSS attacks, we distinguish three types: reflected, stored and DOM Based XSS. These threats are briefly described in the next lines.

Reflected XSS. An XSS vulnerability is reflected if the injection is echoed by the server in the immediate response to an HTTP request. It is usually required that the victim clicks on a crafted link to make the attack start; in addition, a cross domain request could be employed to trigger such kind of issues. These are also referred as first-order XSS.

Stored XSS. The injection is stored in a permanent data store and it is echoed every time a user visits the unsafe web site. Obviously the range of potential victims is greater than in the reflected XSS, since the payload is displayed to any visitor. These are also referred as second-order XSS.

DOM Based XSS. OWASP refers to this type of XSS as "an XSS attack wherein the attack payload is executed as a result of modifying the DOM "environment" in the victim's browser".[3] In practice the attacker could misuse the existent client side script in order to make it work maliciously. In fact, the categorization "reflected" and "stored" XSS is not sufficient since attacks "that do not rely on sending the malicious data to the server in the first place" may definitely happen in Web 2.0 [1] (where client side scripting is gaining more and more attention [2]).

Besides the typical circumstances, XSS attacks can take place through advanced scenarios, a common example is through Content Sniffing: web browsers tend to perform obscure operations and somersaults to detect and handle various file types and encoding schemes. This may lead to Cross-Site Scripting in the case for instance the browser renders as HTML what meant to be an image.
Basically "a clever attacker could manipulate the browser into interpreting seemingly harmless images or text documents as HTML, Java, or Flash – thus gaining the ability to execute malicious scripts in the security context of the application displaying these documents",[4] therefore serving uploaded documents properly becomes fundamental for preventing these attacks. Good practices, such as returning an explicit and well-know *Content-Type* value and a precise *charset*, are mandatory for realizing a robust application; nevertheless the safest approach consists of adopting separate, isolated web origins such that uploaded files cannot be a threat at all. Obviously this would lead to define robust policies for accessing files: using the users' cookies for the "sandbox" domain is mindless, while introducing random

[3]OWASP, *DOM Based XSS*, `https://www.owasp.org/index.php/DOM_Based_XSS`
[4]Zalewski, M., *Content hosting for the modern web*, `http://googleonlinesecurity.blogspot.it/2012/08/content-hosting-for-modern-web.html` (Aug 2012)

and temporary tokens in the URLs looks to be a winning approach.
For the sake of completeness, since Internet Explorer up to version 7 could in some cases report a MIME type different than the type specified by the web server, a precise HTTP response header was introduced in order to stop the so called MIME-sniffing, the *X-Content-Type-Options*; if setted to *nosniff* the content will not be sniffed, thus preventing for instance to render a *text/plain* document as HTML. Nevertheless omitting this header may sometimes lead IE to perform some obscure procedures in order to sniff the content: Hasegawa, a well known security researcher, showed how it is possible to force IE to sniff the content in order to trigger an XSS attack [3].
As you might guess, realizing a robust application able to serve users' uploaded content is not a trivial task at all, for further information refer to the excellent Zalewski's book [Zal12].

Open Redirect [4] is another severe vulnerability, in which the attacker could make the victim visit malicious web pages without realizing it. These attacks could take place from both the client and the server side: JavaScript and in particular the *location* object is involved in the client-side, while the HTTP response header *Location* comes into play at the server-side.
On the client-side, it is quite common to discover DOM Based XSS in which the attacker may trigger an XSS through pseudo-schemes, such as *javascript* and *data*, while at the server-side having access to a *Location*'s value might lead to redirect the victim to another unexpected domain. Note that having access to an HTTP header's response might lead to another severe vulnerability, called HTTP response splitting,[5] which is associated with many exploitation scenarios, such as session fixation and, again, XSS. Basically web browsers offer several approaches to execute a redirect, thus to initialize attacks and execute script code. Browsers' plugins, such as Flash, Java, PDF, have their own methods to produce a redirect, the interested reader can refer to the HTML5 Security Cheatsheet, Redirection Methods.[6]

XSS attacks may be triggered through Clickjacking[7] too, this technique allows attackers to alter a web site's visual display while preserving its functionality. In this case the attacker may put a target web site into an invisible iframe and ask the victim to extract or inject content through drag and drop operations giving possibly place to a Self-XSS or cross-domain content extraction.

Being able to inject malicious JavaScript into a trusted web site implies a really high risk for the users: XSS can achieve very sophisticated results in the case the attacker uses

[5]OWASP, *HTTP Response Splitting*, `https://www.owasp.org/index.php/HTTP_Response_Splitting`
[6]HTML5 Security Cheatsheet, *RedirectionMethods*, `http://code.google.com/p/html5security/wiki/RedirectionMethods`
[7]Kotowicz, K., *Exploiting the unexploitable XSS with clickjacking*, `http://blog.kotowicz.net/2011/03/exploiting-unexploitable-xss-with.html`

exploitation frameworks, such as BeEF[8] or XSSF[9]. Basically, the hackers could fingerprint the internal network via JavaScript in order to identify known connected devices and exploit them whenever a vulnerability exists. Furthermore XSS tunneling proxy permits to browse the hooked domain through the security context of the victim browser; this implies that the authenticated surface can be scanned to detect new vulnerabilities, which could not be accessible earlier.

Web application security is a fundamental component in the modern computing industry. In the last years many flaws allowed hackers to illegally access sensitive data; consequently huge costs are required to assess the impact of the attack, the application's weaknesses and especially the backwashes in terms of customers loss.

1.3 Scenario and terminology

Since the end user's browser has no way to know whether a script comes from an injection, it will have no reasons to not execute it; developers use to adopt several countermeasures in order to avoid these kind of flaws, but realizing robust XSS filters may result in a very difficult task since several aspects should be taken into account.
Web application vulnerabilities can be detected by making manual penetration testing and this reveals to be successful in the case the tester can also examine the code. Unfortunately, this is a very time consuming task and it might require expert skills. Web vulnerability scanners, instead, allow penetration testers and developers to automatically analyze web sites aiming at of detecting security issues in a relative small time window and in a fairly good detection rate.
In this work we talk about several attack scenarios, in which basically the attacker injects a web page (*injection page*) by making an HTTP request; we expect this injection being reflected into the same page or into another page and we refer such a page as a *reflection page*. In addition, we are interested in the context, that is the exact point within the injected web page's DOM in which the injection is reflected and we refer such a context as a *reflection context*.
For instance the attacker could inject a web page A by submitting a form, while the payload will be reflected in a web page B, that is actually related to the first one. The relation is based on the web application's intended workflow and it should be identified by the tester once got a basic understanding of the way it works. At this point the attacker should need to know which is the reflection context within the HTML code of B in order to identify whether an attack is possible and especially which is the right payload to be employed in order to trigger an XSS attack.

[8]BeEF, The Browser Exploitation Framework Project, `http://beefproject.com/`
[9]XSSF, Cross-Site Scripting Framework, `http://code.google.com/p/xssf/`

1.3.1 Basic filter based XSS prevention techniques

In the last years several approaches for protecting against XSS have been proposed. The complexity of such a task is obviously huge from the point of view of web developers, who should be trained by security experts about this type of threat. The adopted protection should also be exposed to a strong testing phase, performed by very experienced penetration testers.
The "real-world" XSS protections are widely discussed in Chapter 2; essentially web applications stop users to provide malicious inputs by inspecting them in order to detect whether an XSS injection attempt is occurring and this is the most common way of detecting and stopping XSS. Different approaches are also possible: for instance Web Application Firewalls are appliances that block common attacks by enforcing a set of rules to an HTTP conversation.
In this work XSS protection systems and sanitization functions that prevent malicious code to be supplied are referred as *filters*; in particular we consider that users can marshal content through a web application and that this latter adopts its own prevention techniques to stop harmful inputs or to modify them in an harmless form.
Input validation should be adopted when handling an input from a data entry point, while contextual output encoding should used when reflecting it. Although this practice looks to be easy, many applications validate inputs through poorly effective filters, that would allow attackers to manage an attack.
After all, filtering the user-generated content has critical consequences in terms of security: giving people the possibility to publish its own HTML is in many cases synonym of XSS. In addition, testing a filter is extremely compelling: discovering a successful bypass could require time and especially a very strong background in breaking web application protection systems.

1.3.2 Web Application Security Scanners

Once defined the basic terminology and showed a simple scenario, we remind the way web vulnerability scanners work. They can be divided into two wide categories: black-box and white-box scanners.
The first approach practically works by looking for data entry points (DEPs) through a crawler module – which are from the scanner's perspective toys to play with – and by injecting various patterns in order to identify security issues. Basically the most naive way to do that consists of trying to understand whether the injected payload is reflected within the reflection page without being somehow modified. Regular expressions and XPath queries may be really useful to achieve such a result. A good example could be SecuBat by Kals et al. [KKKJ06], whereas a wider discussion has been provided by Fong et al. [FO07].
By the way Petukhov et al. [PZ08] showed that this approach does not guarantee neither

accuracy nor completeness of the obtained results: false positives are actually possible and a poor coverage of DEPs may severely decrease the overall completeness of analysis. This problem becomes considerable in the case the tested application makes massive use of JavaScript code, possibly causing the crawler to miss relevant pages to explore whenever links are generated at run-time through client-side code. Classical scanning mixed with logged requests through a proxy[10] could address this problem, but obviously it might be required to interact with all the components constituting a web page to extract every "hidden" URL.
The second approach, white-box scanning, is based on web application analysis with the assumption that source code is available and can be reviewed; it is surely the most complete way to detect security vulnerabilities, but it requires experts and security minded people.

1.4 Motivation

Black-box web vulnerability scanners still struggle with the detection of XSS vulnerabilities, especially the second order ones. Actually the inability to catch the web application's intended workflow reveals to be the main limitation of the current scanners, moreover complex forms with aggressive input checking may block them, without giving any chance to go deeper in the web site structure.
Bau et al. [BBGM10] and Doup et al. [DCV10] are excellent evaluations of web vulnerability scanners. Both showed that almost all of them fail to properly detect second order XSS; in practice scanners are not able to understand what is the relationship among different web pages: an injection in a web page A may trigger an XSS in a completely different web page B.
Modern web application, in particular social networks, owe their popularity to people who publish their own contents; it is obvious that giving the users the possibility to share stuff, which is permanently stored and accessible to many visitors, may lead to XSS in the case no proper sanitization is performed. Hence, web developers might need to significantly test their sanitization systems in order to assess whether bugs are actually present. Obviously, since many steps could separate the injection page from the reflection page, the testing mechanism should need to know how precisely reach this latter and which are the operations needed to activate the XSS filter.
In other words, we need a tool which is able to break an XSS filter, given the path from the injection page to the reflection one.

1.4.1 Workflow-based approach

The classical data-driven approach for scanning web applications for security issues requires a crawler module, which is able to identify data entry points; basically it is straightforward

[10] Ivashchenko, T., *Automation of modern web application security testing*, `http://www.oxdef.info/talk/en/j4m2012/webapps/`

to infer that a poor crawling engine implies a very low detection rate [DCV10]. Hence, Korscheck [Kor10] introduced a smart way to realize a penetration test, that is practically based on the possibility to define a path to be tested, trying to overcome barriers in the workflow. The workflow-based approach can absolutely leverage the XSS detection rate and it seems to be a promising concept to solve the above mentioned limitations.
In this work we use this approach by giving the tester the possibility to define a sequence of operations the tool should replicate, but we specialize the testing with respect to one particular XSS filter; this means that we are not going to describe a tool which, given a workflow, tries to just discover XSS vulnerabilites along this path, but an approach for significantly testing XSS filters by giving the testers the chance to select a path within the application, that connects the injection page to the reflection page. Detailed information and concepts will be mentioned in the next chapters.

1.4.2 Context-aware paradigm

The way many scanners generate XSS injection patterns is often naive and consists of injecting a complete set of strings, such as `<script>alert(1)</script>` and similars, without considering the reflection context. For instance it does not make sense to inject an HTML element such as `<img src=xx:x onerror=alert(1) />` in the case the payload is reflected within an attribute of an HTML element, which cannot be broken.[11] This naive logic may considerable decrease the detection rate for simple to detect XSS vulnerabilities too.
Skipfish[12], web application security reconnaissance tool, and XSS Rays[13], XSS reversing/scanner tool, solve this issue by injecting a complex string, a multi context XSS vector, that works in multiple contexts and web browsers. However many different filtering techniques against user-generated content are possible, thus several muti-context XSS vectors should be injected in order to reliably assess whether a bug is present. For instance let us consider a form accepting a syntactically correct URL with a parsing engine just blocking strings containing `script`, `img` or `svg`; we might end up our test by reporting that no XSS is possible because every vector we adopted contained at least one disallowed tag, whereas something like `javascript:alert(1)` would have triggered an XSS.
Taking decisions during the test process would be an obvious way of proceeding in order to identify which are the best payloads to be employed for a certain reflection context. This approach can be easily achieved by first making a non-malicious injection, then by inspecting the page for the reflection context and eventually by specializing the malicious vectors to be supplied.
In fact generating attacks on the basis of the output in a feedback fashion was similarly used

[11] It means that quotes are escaped, therefore the attacker cannot break the attribute
[12] Zalewski, M., *Skipfish, web application security scanner*, `http://code.google.com/p/skipfish/`
[13] Heyes, G., *XSS Rays extension*, `http://www.thespanner.co.uk/2011/01/21/xss-rays-extension/` (Jan 2011)

by Ciampa et al. [CVDP10] in a heuristic-based approach for SQL-Injection vulnerabilities [5, 6] detection. Obviously pattern matching of error messages cannot be applied for XSS injections, but the idea of having a set of particular and targeted vectors is absolutely reasonable for increasing the chances of catching vulnerabilities.
The aforementioned approach is currently adopted by many valid scanners, that's the case for instance of IronWasp [7], Acunetix Web Vulnerability Scanner[14] and OWASP ZAP[15], whose XSS detection engines make targeted injection on the basis of the point in which the injection falls.

1.4.3 In-browser scanner

Modern web browsers have somehow different capabilities and offer different ways to trigger a Cross-Site Scripting attack. HTML5 Security Cheatsheet[16] is an excellent resource for identifying which vectors work on major web browsers. Actually a very good XSS scanner would need to really search for XSS vulnerabilities within the web browser context, therefore different browsers should be taken into account and different results may be generated on this basis.
The most trivial example consists of a web page which accepts a parameter, whose content is reflected within an attribute `style` of a DIV element; older versions[17] of Internet Explorer allow to trigger an XSS by employing the CSS extension expression().[18,19] By running a scanner within Firefox, it would report that no XSS is possible, while it would return positive results in Internet Explorer.
Scanners and web browsers need to be somehow strictly related in order to achieve a complete and reliable scan.
Obviously the same could be performed through basic scanners without web browser support, however this classical approach would result in many false positives; realizing whether an injection is successful can be done by catching alert dialog windows in the web browser. In this work the concept of grabbing an alert window will be adopted in order to reduce the false positive rate to zero. Obviously we make the assumption that no alert dialog windows are already present in the reflection page: we do not rate it as a strong assumption since modern web applications rarely disturb visitors through such alerts.

[14] Acunetix Web Vulnerability Scanner, `http://www.acunetix.com/vulnerability-scanner/`
[15] OWASP ZAP, `https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project`
[16] HTML5 Security Cheatsheet, `http://html5sec.org/`
[17] Internet Explorer 7, Internet Explorer 8 and 9 in compatibility mode or if no doctype defined. Also Opera allows to execute JS within the CSS context via the -o-link property - `http://html5sec.org/#9`
[18] MSDN, *About Dynamic Properties*, `http://msdn.microsoft.com/en-us/library/ms537634(v=vs.85).aspx`
[19] Hasegawa, Y., *Cause of XSS by excessive detection of "expression" in IE*, `http://openmya.hacker.jp/hasegawa/security/expression.txt` (Nov 2006)

1.5 Objectives

The aim of this work is to present a security tool, which is able to identify and exploit very difficult to spot cross-site scripting vulnerabilities within a web application, or more precisely to automatically reproduce the behavior of an expert while trying to identify a bypass for an XSS filter.
Since it is widely focused on second order cross-site scripting vulnerabilities, it uses a workflow-based approach as introduced by Korscheck [Kor10] with the tool iSTAR; it differentiates itself since it is able to automatically detect which is the reflection context and to specialize the attack vectors on this basis. Moreover it allows the tester to reverse engineer the XSS filter in order to understand what is allowed to be supplied; it gives him also the possibility to add new malicious payload to be injected in a simple and modular way and it is designed with attack vectors' obfuscation in mind. Another key point, that distinguishes our tool from other XSS scanners, is the fact that false positive rate is equal to zero because each reported vulnerability is the result of a successful injection.
Let us imagine to have a social network where users can share information and their friends can read it; this target web site performs some kind of input validation and it just allows to share plain text posts or links under the form of HTML anchors, i.e. `<a href="http://evil.com">click me!</a>`. The smart attacker would obviously think that links may be involved to trigger an XSS attack, it would try to inject something like `<a href="javascript:alert(1)">click me!</a>` and it would click the resulting link in order to check whether an alert window is fired. From the web scanner's perspective it may be useful to perform two steps in the case an injection requires user interaction to be triggered, the first one consisting of injecting a malicous anchor, while the second one consisting of reproducing the behavior of a victim, who clicks that link. This is the basic idea we used for our tool in order to reproducing both sides of a real attack; more information about this point will be explained in the next chapters.
Classical web application scanners work quite differently with respect to our tool, since this latter is able to discover XSS vulnerabilities within the reflection page only; it practically requires the tester to have a basic understanding about the way the application works and to be able to define a sequence of operations in which a data entry point is injected and a reflection page is inspected for the presence of malicious scripts. In other words it can be properly considered as an automatic XSS tester which can definitely help in discovering hard-to-spot vulnerabilities, given a certain logic for the application.
In this work we present *snuck*, a cross-site scripting filters evasion tool, and we also describe some experiments on a variety of web applications in order to show its behavior with respect to robust XSS filters.

Chapter 2

Defense and offense concepts

This chapter presents a brief survey on the common methods that web applications use to protect themselves against XSS and explains basic, but very common, XSS attack scenarios.

2.1 Protecting against XSS

With the advent of Web 2.0, user-generated content gained increasingly attention and started being the main content of modern web applications; on the one side applications need to be simple, usable and flexible, on the other side they need to be as secure as possible. This reveals to a be a kind of trade-off, the more the application is flexible and gives the users multiple levels of freedom in terms of allowed inputs, the more likely the attacker is able to discover a flaw to attack it.
Web applications use to prevent malicious users from injecting JavaScript or inserting malformed HTML within web pages with different techniques. The most common way to protect against XSS consists of adopting HTML filters, such as HTML Purifier,[1] that inspect input sources looking for XSS evidence and perform a sanitization process in order to clean any potentially harmful input.
XSS can be mitigated on the client-side too, this is the case of NoScript,[2] a Firefox extension whose goal is to detect and block malicious script from being executed. Actually modern web browsers started adopting built-in protections against reflected XSS by identifying attack evidences in the URL and modifying the injected web page in order to avoid the execution of the malicious code; Google Chrome and Internet Explorer are currently employing their own filters.
Another approach consists of adopting Web Application Firewalls (WAFs), which are appliances that basically detects and stops malicious requests to reach the web application in order to avoid security breaches. Since the underline application remains insecure, WAFs

[1]HTML Purifier, `http://htmlpurifier.org/`
[2]Maone, G., NoScript, `http://noscript.net/`

can be seen like plasters: they do not completely solve the issues, but they just try to stop them coming out.
A modern introduction for preventing XSS is the *Content Security Policy* (CSP): since new vulnerabilities may be introduced along the time, a secure site today can become vulnerable in the future. Content Security Policy[3] is an experimental security extension whose goal is to help mitigating and detecting types of attacks such as XSS and data injection by specifying the domains that the browser should consider to be valid sources of executable scripts. Thus inline scripts and event-handling HTML attributes are not executed by default in CSP enabled sites.
The CSP policy directives are delivered via HTTP headers, so the receiver browser should be able to understand which particular resource should be trusted and rendered.
For the sake of completeness, an interesting automatic tool aiming at retrieving CSP policies, called CSP AiDer [Jav11], has been introduced by Javed. In practice it works through a crawler which scans web pages to grab information needed in the policies.
Despite the potential of this further security layer, very few sites are currently adopting it and the qualitative effects of such an adoption may be not exciting as expected [8].

2.2 Attack scenarios

Since we need to understand how an attacker interacts with a target web application, we present three different attack scenarios; our main goal is to find out which is the most accurate and complete way an automated XSS filter "breaker" could work.

Scenario 1: Blog Comment I Let us consider a target application that allows users to leave comments: the accepting form consists of three fields, the first one need to be filled with the user's name, the second one is optional and it would contain the visitor's website, whereas the third one contains the comment (Figure 2.1).
From the penetration tester's perspective, it is useful to define a sequence of operations such that the form is populated with correct inputs, it is submitted and the reflection page is inspected for the presence of an XSS – i.e. by looking for the injection within the web page containing the comments.
For the sake of simplicity we are assuming that the first two fields are correctly sanitized, while the third one needs further investigation in order to state whether a stored XSS is possible – we are assuming that comments are stored in a database.

The attacker could adopt a black-box testing approach, basically based on two steps. The first step consists of reverse engineering the filter to identify which HTML tags and attributes are allowed to be supplied. The second step consists of injecting malicious HTML elements by looking for a flaw that would allow to execute arbitrary JavaScript.

[3]Mozilla, *Security/CSP/Specification*, `https://wiki.mozilla.org/Security/CSP/Specification`

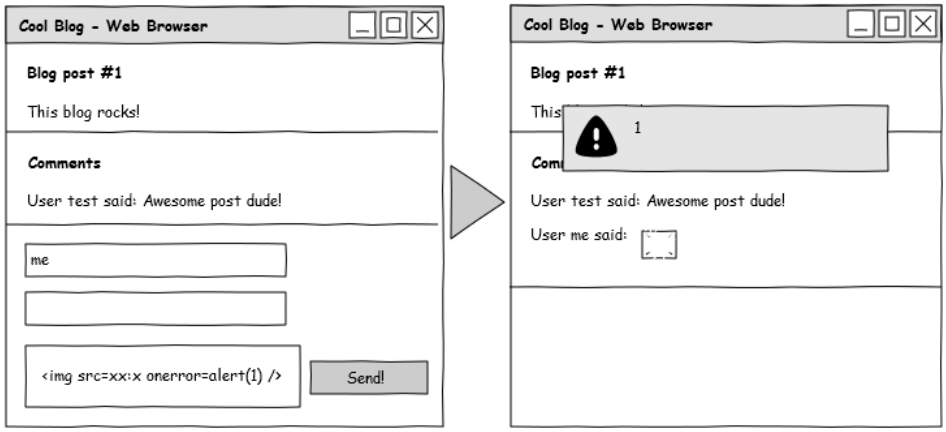

Figure 2.1: Scenario 1: Blog comment I

In addition, the attacker could detect the context in which the injection is reflected – for instance within a P element, i.e. UNTRUSTED_DATA – and it can take advantage of this information in order to identify a successful bypass.
This example is a classical scenario in which a stored XSS can be detected. The more perverse, creative and modern is the set of employed attack vectors, the more likely XSS vulnerabilities can be spotted. The set of attack vectors is a key point for XSS detection scanners; in particular HTML5 introduced a plethora of advanced ways for executing JavaScript code, therefore throwing an entire (outdated) XSS cheatsheet at the target is surely not a winning approach.
A much more complex scenario would be a modified one in which multiple steps have to performed before landing on the reflection page; however this case could be successfully addressed via a workflow-based approach as shown in iSTAR [Kor10].

Scenario 2: Blog Comment II By considering the previous example, we can assume that the username and the comment fields are correctly sanitized, while the one accepting the visitor's URL need to be significantly tested for detecting whether a stored XSS vulnerability can be triggered (Figure 2.2).

In this case the attacker would inject a random string in the URL field and look for its presence within the reflection page. Since the reflection context will obviously be the attribute *href* of an anchor, knowing which are the allowed schemes would give a precious information during the black-box testing.
Some web application scanners would treat this specific data entry point without making any distinction from the other ones; this implies that something like <script>alert(1)</script> is likely to be injected, but it will never trigger an XSS because of the context in which the injection will fall.
Smarter scanners would try to break the attribute *href* and inject an event attribute,

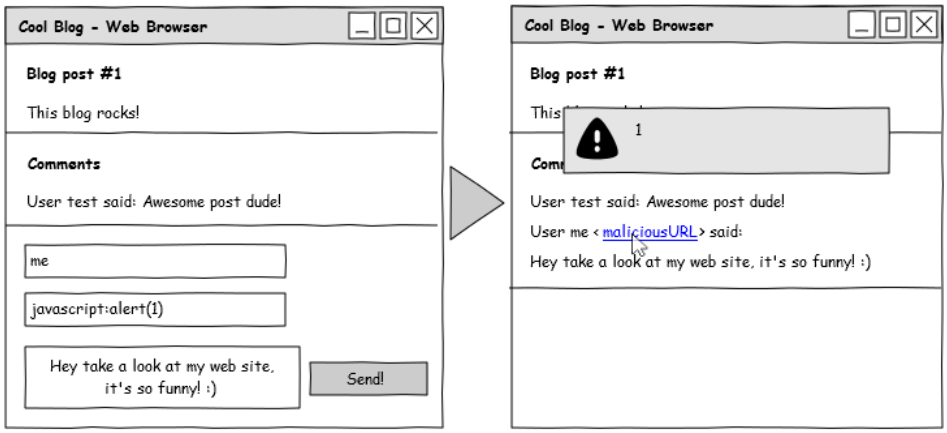

Figure 2.2: Scenario 2: Blog comment II

but this may not be accurate in terms of correctness. By assuming for instance that the reflection context is the attribute *value* of an INPUT element, whose *type* is set to hidden, then the unique chance to perform a successful injection consists of breaking the HTML element, an on*[4] attribute would not execute malicious JavaScript at all.
The most accurate approach consists of injecting malicious URIs, such as *javascript*, *feed*[5] and *data*[6] URIs, and reproducing the victim's behavior, who clicks the injected anchor.
At this point we can assume the target web application disallows malicious URIs by a naive filtering logic, which consists of splitting the supplied URL by the colon character and looking whether the scheme is disallowed (blacklist approach). It is straightforward to guess that no bypass can be realized with something like `data:text/html,%3Cscript%3Ealert(1)%3C/script%3E`. The attacker needs to somehow obfuscate the payload in order to bypass the XSS filter, a string like `data:text/html;base64,PHNjcmlwdD5hbGVydCgxKTwvc2NyaXB0Pg==` is very likely to be identified as an harmless input, whereby automated scanners have to take into account the possibility to use different encoding techniques too.
For the sake of completeness, since the adopted filtering logic is really trivial an injection such as `data:text/html,%3Cscript%3Ealert(1)%3C/script%3E` would

[4] on* is an abbreviation referring to the HTML event attributes, such as onclick, onmouseover, onkeyup and so on.

[5] `<a href=feed:javascript:alert(1)>click me</a>` executes Javascript if clicked in Firefox (up to version 13) – Soroush, D., *Drag and Drop XSS in Firefox by HTML5 (Cross Domain in frames)*, `http://soroush.secproject.com/blog/2011/12/drag-and-drop-xss-in-firefox-by-html5-cross-domain-in-frames/` (Dec 2011)

[6] *data* URIs inherit the domain of the opening page in Firefox and Opera – Kotowicz, K., *The sad state of DOM security (or how we all ruled Mario's challenge)*, `http://blog.kotowicz.net/2011/10/sad-state-of-dom-security-or-how-we-all.html` (Oct 2011)

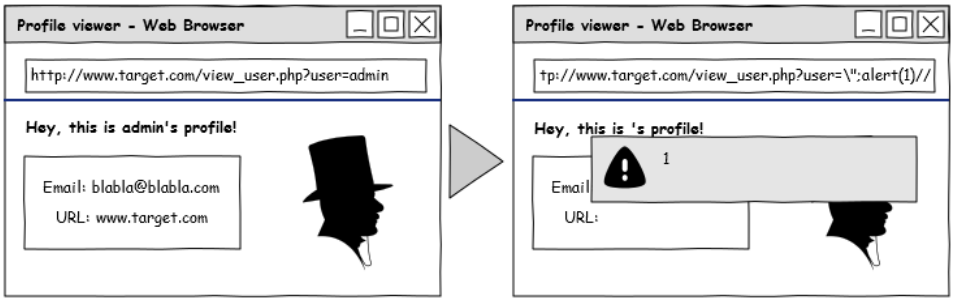

Figure 2.3: Scenario 3: Profile's details viewer

have resulted in a bypass too.
Basically we showed that the reflection context is particularly important in XSS testing, moreover the in-browser scanner concept is really effective in the case we want to reproduce both sides of an attack: the injection by the attacker and the interaction with the "malicious component" by the victim.

Scenario 3: Profile's details viewer Let us consider the target blog employes a web page which allows to show some details about registered users; it accepts an HTTP GET parameter whose content must be a username. We assume that the reflection context is a JavaScript variable such as `var a = "My name is UNTRUSTED_DATA !";` and we would like to know whether a reflected XSS is possible (Figure 2.3).
Since the injection falls within a JavaScript variable, it does not make sense to inject something like `<script>alert(1)</script>`, nevertheless the attacker could break the variable or directly close the SCRIPT element. However many details have to be taken into account in order to understand whether quotes are escaped and the way this has place.[7] Injecting `";alert(1)` would not be enough to trigger an XSS as a JavaScript error would appear: we would need to comment out the rest of the string. The most accurate way to automate this process consists of injecting several ad-hoc attack vectors, by taking into account the reflection context and the way special characters, such as double quotes, are escaped.

Our tool is able to handle reflected XSS testing too and in the shown scenario it basically breaks the JavaScript variable and it automatically comments out the rest. Since `alert(1)` is likely to be somehow blacklisted, it performs multiple injections with obfuscated payloads, such as `\u0061\u006c\u0065\u0072\u0074(1)`.[8]

[7] Barron, J., *Anatomy of an XSS Injection*, `https://blog.whitehatsec.com/anatomy-of-an-xss-injection/` (April 2011)
Barron, J., *Escaping Escapes*, `https://blog.whitehatsec.com/escaping-escapes/` (April 2011)
[8] \u0061\u006c\u0065\u0072\u0074(1) is equivalent to alert(1) unicode encoded

Chapter 3

Methodology and Tools

The main goal of this work is to show a promising approach for significantly testing XSS filters. Since the attacker might need to make many steps before having an XSS vulnerability triggered, it is useful to employ a stateful approach, which is based on the idea of recording a chain of user actions. In addition, having access to the web pages' DOM avoids useless injections and prevents data noises, i.e. false positives.

3.1 Stateful approach for penetration testing

The existing stateless penetration testing tools revealed to be often unsuccessful since the intended workflows may be hard to define in applications that use AJAX. Pavlosoglou [9] showed that given a login prompt, it is possible to automate a brute-force attack with a long list of passwords by using Selenium IDE[1] [10]; the advantage of such an approach consists of quickly assessing successful or failed logins, and, in the case of SQL-Injection testing, it allows to know all filter evasion characters and successful payloads. Obviously it is required to define a sequence of actions, that basically generate the login use case, and to find out a way to distinguish whether a login is successful or whether a SQL-injection is triggered.
We adopted these concepts to replicate the attacker's behavior through events within the web browser. The necessary steps to generate an injection and to look for its reflection are represented by *use cases* – as similarly shown in [Kor10] – which are actually XML files, whose content looks like a sequence of Selenium commands (refer to the next section for details).
This methodology puts a huge advantage since the test can be performed without deactivating CSRF[2] protection mechanisms as every request is naturally generated within the web browser.

[1] Selenium IDE, `http://seleniumhq.org/projects/ide/`

[2] "CSRF" is the abbreviation for Cross-Site Request Forgery: a web application vulnerability where the attacker can force the victim into triggering actions that it did not want to perform

Furthermore, we decided to give the possibility to perform reflected XSS testing too. In this case the specific HTTP GET parameter to test and the target URL are required.

3.2 Selenium: automating web browsers

Selenium is a web application testing system allowing to reproduce a set of operations in the web browser context. It is able to run in many browser and operating systems and can be easily controlled through many programming languages. The aim of automating a web browser may vary according to the needs and developers could find really useful to record and playback a chain of operations for simulating the visitors' behavior through the application, whereas security professionals could exploit its capabilities for driving the browser into reproducing the behavior of an attacker.
The WebDriver[3] is a software giving a programming interface whose goal is to offer the possibility of making direct calls to the browser using the browsers' native support for automation. In practice for each supported browser there is a *driver*, which sends commands to it, and retrieves its results. Since many programming languages are supported, few lines of code could be used to automate the selected web browser for realizing a complete test against a web application.
Eventually, automating browsers through Selenium could deeply improve the test process since a complete set of automation functionalities are accessible in a very easy fashion. Furthermore, since quick reproduction of security issues is absolutely possible, security professionals could send the vendor a script which reproduces the attack procedure and makes him quickly aware of the threat, without possibly incurring in misunderstandings.

3.3 Reflection context

As shown in Chapter 1, the reflection context, that is the exact point in the HTML code of the injected page in which the injection falls, is a precious information for better understanding which are the most appropriate attack vectors to be employed. Many web application security scanners handle each data entry point in the same way, without considering the possibility to take decisions during the test in order to select the attack vectors more likely to trigger an XSS vulnerability.
Our tool uses XPath queries on the DOM tree to identify the exact reflection context and it specializes the attack on this basis. Let us assume a scenario in which the injection falls within a P element, then a XPath query like `//*[text[contains(.,'random_injection')]]` will inform us that the reflection context is the text of a tag P.
JavaScript functions are instead adopted when running in Internet Explorer as advanced

[3]Selenium WebDriver, `http://seleniumhq.org/docs/03_webdriver.html`

XPath queries do not return successfully, thereafter the DOM is inspected through client-side scripting code.[4]
As shown in Table 3.1, many contexts have been taken into account, starting from the *Abridged XSS Prevention Cheat Sheet*.[5] Note that many other possible contexts have been considered.
Basically *snuck* makes at first a harmless injection with a random string, then it detects the reflection context and eventually it starts the malicious injections with a set of targeted attack vectors. Since there are many cases in which XSS vulnerabilities need user interaction to be triggered – i.e. click a malicious link or an HTML element with an *onclick* attribute – the tool detects the injected "component" after each injection and interacts with it in order to reproduce the victim's behavior; XPath queries are employed for this latter task too.
We will show an exhaustive example in Chapter 4.

3.4 Sets of attack vectors

Attack vectors are categorized on the basis of the reflection context and four different classes have been selected:

HTML payloads. They are basically useful when the reflection context is the HTML body.

Malicious URIs. *data* and *javascript* URIs trying to execute JavaScript code. Useful when the injection falls in attributes such as *src* or *href*.

Javascript alerts. *alert(1)* and similar payloads. Useful when the attacker is able to break a JavaScript variable or to inject an *on** attribute.

Expression payloads. Payloads using the CSS extension *expression*. Useful when the injection falls in a *style* attribute in Internet Explorer.

Several attack vectors are considered for each of these classes; we assume that the bigger is the vectors' set, the more likely an XSS vulnerability can be spotted. In addition, obfuscation methods are taken into account for filter evasion: HTML entities and several encodings techniques are used within the attack vectors' set to significantly test robust defenses too.
As many different situations can arise, *snuck* performs some adjustments to the attack vectors before injecting them; for instance let us assume the reflection context is the textual content of a TITLE element, then it will inject the *HTML payloads* vectors, prepending them with `</title>`.
Detailed information about obfuscation and really fascinating web browsers' quirks are reported in an excellent book by Heiderich et al. [HVNEL10].

[4]Selenium Web Driver gives the possibility to execute JavaScript code in the context of the loaded web page
[5]OWASP, *Abridged XSS Prevention Cheat Sheet*, `https://www.owasp.org/index.php/Abridged_XSS_Prevention_Cheat_Sheet`

Reflection Context	Code Sample	Specialized Injection Sample
HTML Body	UNTRUSTED DATA	
marquee, style, xmp, title, etc. content	<title>UNTRUSTED DATA</title>	</title>
HTML Attributes	<input type="text" name="fname" value="UNTRUSTED DATA">	" onclick=alert(1)//
GET Parameter	clickme	" onclick=alert(1)//
Untrusted URL in a SRC or HREF attribute	clickme	javascript:alert(1)
	<iframe src="UNTRUSTED URL" />	
CSS value	X	expression(alert(1));
JavaScript Variable	<script>var currentValue="UNTRUSTED DATA";</script>	";alert(1)//

Table 3.1: Reflection contexts and specialized injections. For each point, in which the injection may fall in, we can identify a specialized attack payload; the table shows the most basic and common contexts only, many others have been omitted.

3.5 Multiple browsers approach

Since modern web browsers have different capabilities and offer several ways to trigger a Cross-Site Scripting attack – as widely shown in the HTML5 Security Cheatsheet – we decided to give the tester the possibility to choose which web browser he wants to run the test with; at the moment the choice is among Mozilla Firefox, Google Chrome and Internet Explorer.
The basic idea is that each attack vector may execute malicious code in a web browser A, but it may not in another web browser B, and/or C.
Classical web vulnerability scanners do not work in-browser, thereafter they use to inject very common and browser-independent attack vectors. Actually a complete and reliable scan would require to cover all the major web browsers in order to state that the considered XSS filter is not affected by any flaw in any "browser-context".

Chapter 4

snuck's Architecture

This chapter describes the tool's architecture by presenting its basic components, and shows some examples for better understanding the way it works.

4.1 *snuck*'s architecture

The *snuck* tool has been implemented in Java, since this language is supported through Selenium Remote Control drivers. As explained in the previous chapters, the core of the proposed approach consists of retrieving the reflection context to perform a specialized attack; thus XPath queries are used through the Selenium WebDriver, which allows to have complete access to the DOM tree.
The architecture of *snuck* is shown in Figure 4.1. The *XSS Injector* works as core of the injection process, it asks the *Use Case Parser* to parse the Selenium commands within the XML file given in input (Step I), and to translate them into browser events through the *Controller* (Step II). Once the first harmless injection has been made, the latest landed web page (reflection page) is inspected in order to retrieve the reflection context. The *XSS Injector* will use this information for selecting the specific *Set of Attack Vectors* (Step III) and starting the malicious injections. At the end of each injection the tool waits for alert windows and reproduces the victim's behavior, if needed (Step IV). Successful injections and reverse engineering information are treated by the injector component, which eventually composes a detailed HTML report about the discovered vulnerabilities (Step V).

4.2 Designing Use Cases

As shown in Figure 4.1, the penetration tester has to write the use case and give it as input to the tool. Login use cases can be also used in order to perform an authentication before making malicious injections.
We distinguish two types of XML configuration files, a first one in which a sequence of op-

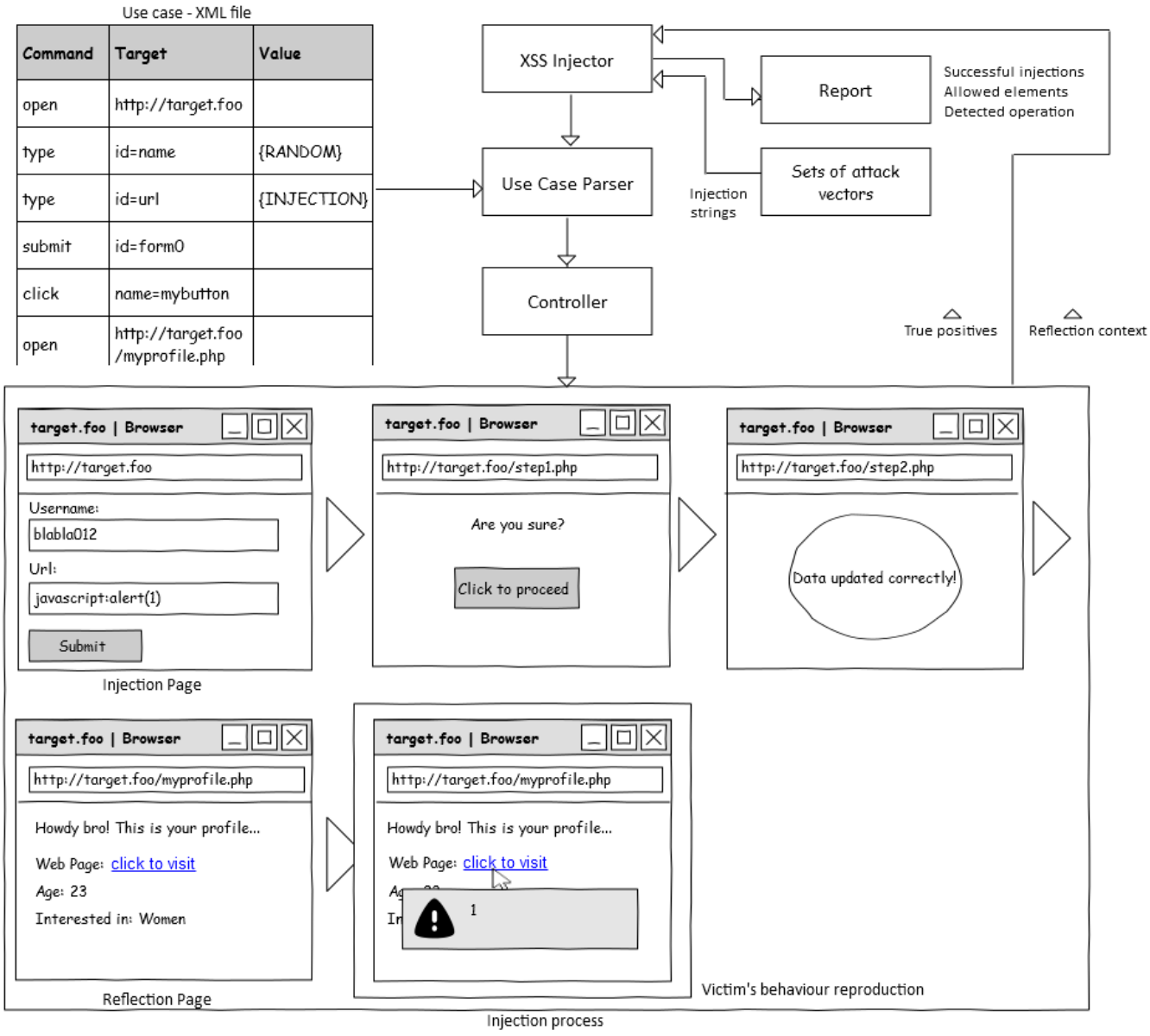

Figure 4.1: *snuck*'s architecture and injection process

erations is given, as reported in Appendix A.1, and a second one for reflected XSS testing. Appendix A.2 shows a very simple example, that basically contains the target URL, the fixed GET parameters and the one we need to test.
Let us assume to target an XSS filter which accepts content through a form, which asks the user to supply its own email, then we can easily design the use case as reported in Listing 4.1.

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
   <post>
      <parameters>
         <parameter>
            <name>email</name>
            <value>myemail_for_testing@test.org</value>
         </parameter>
      </parameters>
      <commands>
         <command>
            <name>open</name>
            <target>http://target.foo</target>
            <value></value>
         </command>
         <command>
            <name>type</name>
            <target>name=email</target>
            <value>${RANDOM_EMAIL}</value>
         </command>
         <command>
            <name>type</name>
            <target>name=content</target>
            <value>${INJECTION}</value>
         </command>
         <command>
            <name>submit</name>
            <target>id=comment</target>
            <value></value>
         </command>
      </commands>
   </post>
</root>
```

Listing 4.1: Use case sample in which a simple form is filled and submitted

The example shows that the tester could manage the definition of parameters, such as an email, through the element *parameter*, which needs two children, *name* and *value*; referring to these may be performed via placeholders in the form of $\$\{name\}$, whereas the tested field need to be populated with the placeholder $\$\{INJECTION\}$. Commands are supplied in the

form of Selenium tuples through the tag named *command*.
In the aforementioned web page we would have the following HTML code (Listing 4.2) – hosted at `http://target.foo` – as you can understand from the selectors used within the tags named *target*.

```
<form action="post.php" method="POST" id="comment">
   <input type="email" name="email" />
   <input type="content" />
   <input type="submit" value="OK" />
</form>
```

Listing 4.2: Injected sample form's HTML code

At this point we assume to target a web page, which accepts an HTTP GET parameter named *xss_testing*. Designing an XML configuration file is again straightforward, however we can avoid writing it manually by just writing two parameters in the command line, in particular `-reflected` followed by the targeted URL and `-p` followed by the parameter to inject, *xss_testing*.
Many other examples are reported in the tutorial, that is reachable from the *snuck*'s web site.

4.3 Modular set of attack vectors

Attack vectors are stored in textual files, categorized as shown in Section 3.4. For any of the four types there exists a corresponding textual file, fully describing, one per line, possible injections. The penetration tester is allowed to add new vectors by just appending further lines to the corresponding file; such repositories are stored in a directory named *payloads*.
Placeholders could be used inside the set of vectors categorized as *HTML Payloads*. For instance, if we add the line `<script src=data:,%alert%></script>` to the list of vectors, then the tool will consider the string `%alert%` as a placeholder and it will replace it by a JavaScript alert chosen at random among the vectors categorized as *JavaScript alerts*.
Something like `<svg onload=%uri%>` will be treated similarly, however the random choice will happen among the URIs vectors.
By moving to this direction, the tester could easily populate the set of injections without specifically selecting the payloads that will be supplied.

4.4 Reverse Engineering process

Reverse engineering an XSS filter [11] is a useful task for detecting what is allowed to be supplied in terms of HTML elements and attributes. This process consists of injecting several HTML elements with attributes and checking whether they are reported in the reflection

page; an alternative process is performed when the injection falls in attributes such as *href* and *src*, this is basically based on recognizing the allowed schemes.
The reverse engineering process is carried out through HtmlUnit,[1] which is the fastest and most lightweight implementation of Selenium WebDriver. In practice it is a browser without GUI, that allows to emulate the behavior of a real web browser supporting JavaScript, by reconstructing the web pages' DOM in main memory; thereafter no browser window is showed once the HTMLUnit driver is started.
We adopted this approach for the reverse engineering process since no particular web browser capabilities are required and, as said above, the aforementioned driver boasts of outstanding performance, so that a successful reverse process quickly gets to the end in few seconds.
The subsequent test – i.e. the malicious injections – is instead performed through the chosen web browser by using its own Selenium Web Driver. This obviously means that a web browser window is opened and the selected operations are automatically performed by the web driver; the web application's behavior is immediately showed to the tester during the injection process.
Furthermore, since the web driver runs very quickly, the tester might find quite difficult to understand the way the application responds to the injections, it may be worthy to introduce a delay between two consecutive injections by starting *snuck* with the argument `-delay` (see Appendix A.3 for the command line manual).

4.5 Victim's behavior reproduction

The victim's behavior reproduction is regulated by the XSS Injector component, which makes XPath queries in order to detect the specific point to interact with; obviously this task is performed for user interaction attack vectors only, such as malicious URIs and event handler attributes.
Let us assume the reflection context is the attribute *href* of an anchor, then malicious URIs will be retrieved through a XPath query, such as `//a[@href='INJECTION']`. By asking the driver to look for such an element in the reflection's page DOM, we can firstly understand whether the injection is reflected, and secondly retrieve the correspondent anchor to click on.
For the sake of completeness, if the attacker is able to break an HTML attribute in order to inject an event handler, then a XPath query such as `//*[contains(@onclick,'INJECTION')]` is trivially adopted.

In addition, the tool distinguishes two possible operations with respect to the web application's database, INSERT and UPDATE. The first one is detected when two (or more) sequential injections are both reported in the reflection page, so by making a first injection

[1]Selenium, *HtmlUnit Driver*, `http://code.google.com/p/selenium/wiki/HtmlUnitDriver`

A and a second injection B, then both are reported in the same reflection page; this is quite common in the case the target application runs a forum. The second case is detected when each injection is independent from the others, thus the second injection B will replace the previous injection A.
At the end of each injection the tool waits for one or more alert windows on the basis of the detected operation. Obviously the web browser might be flooded by alert windows whenever the operation is detected as an INSERT and the XSS filter is too weak, whereby a threshold has been set in order to stop the test after a certain number of successful injections.

4.6 Run example

We present in Table 4.1 the way the tool operates with respect to the Scenario 2 shown in Section 2.2. Many steps are involved to generate a complete and reliable scan, eventually an HTML report is returned for the inspection of the results.

Operation
Start *HtmlUnit*
Inject a random alphanumeric string to check whether the injection is reflected
The XSS filter won't probably reflect the injection as it is not a correct URL
Inject a random numeric string to check whether the injection is reflected
The XSS filter won't probably reflect the injection as is is not a correct URL
Inject a URL to check whether the injection is reflected
The XSS filter will accept it and reflect it in the reflection page
The reflection context is retrieved =>attribute *href* of an HTML element, A
The reverse engineering process is started, many correct URLs with different schemes are injected in order to understand which protocols are allowed
Detect the operation (INSERT OR UPDATE, see section 4.5 for information)
Quit *HtmlUnit*
The chosen web browser is started
Check whether the attribute *href* is breakable
Start injecting several malicious payloads, such as *javascript:alert(1)*, and click the resulting anchors
If *href* is breakable, then inject *on** attributes and eventually try to break the attribute for injecting new HTML elements
The web browser is closed
An HTML report is generated

Table 4.1: *snuck*'s run example with respect to the Scenario 2 shown in Section 2.2

Chapter 5

Experimental Results

One goal of this study is to evaluate the XSS injection approach and the tool *snuck* proposed in this paper. We selected many popular open source CMSs[1] and we performed several test against them.
The next sections illustrate how the tests were carried out and the experimental results we found out.

5.1 Designing and starting the test

As shown in the previous sections, the penetration tester is required to define a sequence of operations to be performed by filling in a use case, thereafter it needs to select which specific path, within the web application, connects the injection page to the reflection one. Obviously a basic understanding about the way the application works is required to discover the correlation among different web pages.
In our empirical study we selected some of the most common operations, that use to be tasty from the attacker's perspective. In other words we performed many tests against procedures that make the application accept user-generated content and reflect it.
We filled in the use cases manually as it is extremely easy and intuitive to write them.

5.2 Test results

This section will present point by point the results we found out in several CMSs and blogging platforms. All the shown vulnerabilities were responsibly reported to the vendors and part of these have been fixed. Much more detailed advisories are reported in some cases as well; please note that since *snuck* was under development and in a non-public phase at the time of making these experiments, you will not find any trace of it in the online advisories.

[1] CMS stands for Content Management System

Test Results
`<a href="feed:javascript:alert(1)">CLICK ME</a>`
`<a href="feed:data:text/html,<script>alert(1)</script>">CLICK ME</a>`

Table 5.1: Test results in *WordPress 3.3.1*: XSS filter for visitors' comments

WordPress. WordPress[2] is the most popular open source blogging platform, basically it allows to publish posts where visitors can leave comments. Since comments may contain a subset of HTML elements, we decided to significantly test the HTML filter that is used, *wp_kses*[3]. This latter employs a whitelist approach to sanitize the input, which makes sure that only the allowed HTML element names, attribute names, attribute values and only sane HTML entities can occur within the supplied comment.
We performed the test in *WordPress 3.3.1* by employing the use case in Appendix A.4; the basic idea was to give the tool some logically correct inputs to be supplied through the HTML form accepting the visitor's information and the comment. Actually the injection page is the same as the reflection page since supplied comments are reflected within the page with the "Leave a new comment" form. In addition, no other steps need to be done before landing in the reflection page: the injection process consists of two steps, populate and submit the form, and inspect the reflection page for detecting XSS issues.
Since WordPress adopts an anti-flooding mechanism in order to stop bots to continuously insert comments, we started *snuck* with the argument `-delay`.
The reverse engineering process gave us some information that we were already aware of since we previously inspected the filter's PHP code: basically neither harmful elements nor event handler attributes are allowed. Instead the malicious test reported a stored XSS vulnerability while running in Firefox (Table 5.1).

Obviously the reason why this is allowed is related to the fact that the *feed* scheme is considered harmless; unfortunately Firefox (up to version 13) executes JavaScript when encountering a sequence of *feed:* followed by a "malicious" protocol.
This issue was fixed in *WordPress 3.3.2*; detailed information about the fix and how to exploit this in order to manage a privilege escalation attack are accessible online.[4]

Habari. Habari[5] is a quite popular blogging platform, its functionalities are more or less the same as the WordPress ones. We focused on the filter used against the visitors' comments (use case in Appendix A.5). Unfortunately, the malicious test returned that

[2]WordPress, `http://wordpress.org/`
[3]WordPress, *Function Reference/wp kses*, `http://codex.wordpress.org/Function_Reference/wp_kses`
[4]Gentile, M., *Multiple vulnerabilities in Wordpress*, `http://www.sneaked.net/multiple-vulnerabilities-in-wordpress`
[5]Habari Project, `http://habariproject.org/en/`

Reflection Context
`<a href="UNTRUSTED_DATA/">[...]</a>`

Table 5.2: *Symphony 2.2.5*, URL field for visitors' comments – reflection context

no XSS is possible (*Habari 0.8*), while the reverse engineering process allowed us to infer that a white-list approach is used to sanitize the input; actually the *InputFilter* class makes sure that only sane input is reflected, however harmful HTML elements, such as *iframe*[6], are allowed in the case no attributes are supplied.
snuck found out this oddity when reverse engineering the filter: the attacker may supply a comment whose content is <`iframe`> to basically break the reflection page's output as the filter does not successfully handle unclosed tags.

Symphony CMS. Symphony[7] is a XSLT[8]-powered open source content management system, it is very popular among users who want a widely customizable CMS. We proceeded our survey on XSS filters against users' comments with the one employed by this web application, *xssfilter*.[9] Basically we conducted two different test (*Symphony 2.2.5*), in the first one we injected the comment field (similar to Scenario I in chapter 2), in the second one the URL field (similar to Scenario II in Chapter 2, with use case in Appendix A.6).

- *Testing the comment field.* In this case the injection page is the same as the reflection pages, so the main purpose of *snuck* is to populate the HTML form accepting the users' information with the right input and to inject the comment field. The detected reflection context is a P html element as expected, thereafter *HTML Payloads* are adopted for the attack.
  No positive results were reported since *HTMLPurifier* is adopted for sanitization purposes, while *xssfilter* works as a detection layer, stopping requests containing malicious inputs.
- *Testing the URL field.* Positive results were returned when testing the field accepting the visitors' web site. *snuck* successfully reversed the XSS filter by finding out the allowed schemes, as the reflection context is the attribute *href* an A element (Table 5.2), and it eventually discovered two different ways to bypass it (Table 5.3).

Actually the vulnerability was related to an incomplete regular expression in *xssfilter*: supplied URLs are marked as harmful and blocked through a black-list approach. At

[6]Habari, *inputfilter.php*, `http://doc.habariproject.org/inputfilter_8php_source.html#l00073`
[7]Symphony CMS, `http://getsymphony.com/`
[8]XSLT stands for Extensible Stylesheet Language Transformations, `http://en.wikipedia.org/wiki/XSLT`
[9]Symphony CMS, *xssfilter*, `https://github.com/symphonycms/xssfilter`

Test Results
`data:text/html;base64,PHNjcmlwdD5hbGVydCgxKTwvc2NyaXB0Pg==`
`data:text/html,%3csvg%20onload=alert%281%29%3e`
`feed:data:text/html,%3csvg%20onload=alert%281%29%3e`

Table 5.3: Test results in *Symphony 2.2.5* – detected XSS

the moment of this writing the vulnerabilities were not yet fixed; although we reported many other really severe security issues, just few reflected XSS were fixed in *Symphony 2.3*.

GetSimple. GetSimple[10] is an "XML based, stand-a-alone, fully independent and lite Content Management System"; we selected this one (*GetSimple 3.1*) among many others because it is quite trivial to understand how it works. In addition, it widely adopts HTTP GET parameters for its operations, thus it might be a good candidate for evaluating the way our tool works with respect to reflected XSS.
We mixed our approach with the typical one used by web application security scanners and we started crawling the application by extracting every HTTP GET parameter, and we eventually treated them as data entry points. We gave them as input to *snuck*. Although this approach revealed to be really time consuming, it returned many vulnerable parameters. In particular, no advanced payloads were needed to execute malicious code, `<iframe src=javascript:alert(document.cookie)//` would have been enough.
At the moment of this writing these vulnerabilities were not yet fixed.

Plone. Plone is an open source CMS written in Python that "has the best security track record of any major CMS".[11] We managed some experiments as shown in the previous cases, in particular against the XSS filter for the user-generated content, i.e. comments to public posts. Tests were conducted in *Plone 4.4.1* (use case in Appendix A.7), in which we setted the "comment text transform parameter" to Markdown.
Basically the XSS filter (*Safe HTML*[12]) is pretty gentle, besides no event handlers are allowed and no malicious schemes can be supplied, potentially harmful tags can be injected. Since random attributes are not filtered out, we could infer that a black-list approach is used with respect to HTML elements, while a white-list with respect to attributes.
Eventually our tool returned three attack vectors, they were strictly related to each other as they all exploited the same bug (Table 5.4).
Actually the HTML5 named character reference : (U+0003A COLON) is not decoded by the filter into a colon character, thus bypassing it. This latter looks for

[10] GetSimple, `http://get-simple.info/`
[11] Plone, `http://plone.org/`
[12] Plone, *HTML Filtering options*, `http://plone.org/documentation/kb/filteringhtml`

Test Results
`<a hrEf=data:text/html;base64,PHNjcmlwdD5hbGVydC gxKTwvc2NyaXB0Pg==>_DUMmY_9701`
`<A HrEf=javascript:alert(1)>_DummY_62502`
`<A hrEF=feed:javascript:alert(1)>_dUMMy_59371`
`<meta name="Description" content="0;url=data:text /html;base64,PHNjcmlwdD5hbGVydCgxKTwvc2NyaXB0Pg==" HTTP-EQUIV= "refresh">`
`<meta name="Description" content="0;url=javascript:alert(1)" HTTP-EQUIV="refresh">`

Table 5.4: Test results in *Plone 4.1.4* – detected XSS by combining the results from Firefox and Chrome

Opera(-only) Attack Vector
`<a x="dAta:image/svg+xml;charset=utf-8;base64,PHN2Z yB4bWxucz0iaHR0cDovL3d3dy53My5vcmcvMjAwMC9zdmciPjxzY3JpcHQ%2 BYWxlcnQoMSk8L3NjcmlwdD48L3N2Zz4NCg==" style="-o-link:attr(x); -o-link-source:current">click me</a>`

Table 5.5: Opera based attack vector in *Plone 4.1.4*

malicious schemes, such as *javascript:* and *data:* without taking into account this other opportunity of supplying a colon character.
By combining the reversing results with the aforementioned exploits we came out with two other injections (refer to the last two rows in Table 5.4), which take advantage of the *meta* element for generating a malicious redirect.
For the sake of completeness, since the *style* attribute is absolutely harmless from the filter's perspective, we came out with a fascinating Opera based attack vector (Table 5.5).
The shown vulnerabilities were not yet fixed at the moment of this writing, but a patch will be available very soon.

Breaking XSS filters is absolutely a creative task, which obviously cannot be completely automated, however the shown experiments prove that successful testing can be performed through the described approach. We do not present multi-step attacks, where the attacker would need to follow many steps before landing in the reflection page – i.e. in an e-commerce web site – but it would not be different from the aforementioned experiments.

5.3 Evaluation and comparison

This section presents the evaluation of nine modern web vulnerability scanners with respect to *snuck*. We crafted five XSS vulnerable web pages, which adopt some very naive filtering mechanisms and we asked the tools to break these XSS prevention "systems".
The test suite was composed by an index page (Listing 5.1) and the actual vulnerable pages, we will refer to these as *test A* (Listing 5.2), *test B* (Listing 5.3), *test C* (Listing 5.4), *test D* (Listing 5.5), *test E (Listing 5.6)*.

```
<!DOCTYPE html>
<html>
<meta charset='utf-8'>
<title>uhhhh?!</title>
<body>
<a href="a.php?x=<b>my_sane_html</b>">click</a>
<a href="b.php?x=localfile.html">click</a>
<a href="c.php?x=00001">click</a>
<a href="d.php?x=user_00001">click</a>
<a href="e.php?x=user_00001">click</a>
</body>
</html>
```

Listing 5.1: index.html - it points to the actual vulnerable web pages

```
<!DOCTYPE html>
<html>
<meta charset="utf-8">
<title>uhhhh?!</title>
<body>
<?php
function clean($var){
   return strip_tags($var, '<b><i><a>');
}

echo ( (isset($_GET['x']) && !empty($_GET['x']) ) ? clean($_GET['x'])
   : "0.o");
?>
</body>
</html>
```

Listing 5.2: a.php - reflected XSS vulnerable web page

```
<!DOCTYPE html>
<html>
<meta charset="utf-8">
<title>uhhhh?!</title>
<body>
<?php
function clean($var){
   return preg_replace(array("/:/", "/\"/"), array("", ""), $var);
}
?>
<a href="<?php echo ( ( isset($_GET['x']) && !empty($_GET['x']) ) ?
clean($_GET['x']) : "O.o" ); ?>">click me</a>
</body>
</html>
```

Listing 5.3: b.php - reflected XSS vulnerable web page

```
<!DOCTYPE html>
<html>
<meta charset="utf-8">
<title>uhhhh?!</title>
<body>
<script>
var z = "user_<?php
echo ( (isset($_GET['x']) && !empty($_GET['x']) ) ? clean($_GET['x'])
    : "O.o");
?>";
</script>
<?php
function clean($var){
   return preg_replace(array("/\"/", "/script/"), array("\\\"",""),
       $var);
}
?>
</body>
</html>
```

Listing 5.4: c.php - reflected XSS vulnerable web page

```
<!DOCTYPE html>
<html>
<meta charset="utf-8">
<title>uhhhh?!</title>
<body>
<script>
<?php
echo "// xxx ".( (isset($_GET['x']) && !empty($_GET['x']) ) ?
   clean($_GET['x']) : "0.o")." xxx";
?>
</script>
<?php
function clean($var){
   return preg_replace(array("/script/"), array(""), $var);
}
?>
</body>
</html>
```

Listing 5.5: d.php - reflected XSS vulnerable web page

```
<!DOCTYPE html>
<html>
<meta charset="utf-8">
<title>uhhhh?!</title>
<body>
<input type="hidden" value="<?php echo ( isset($_GET['x']) &&
!empty($_GET['x']) ) ? preg_replace("/>/", "", $_GET['x']) : "0.o";
?>" />
</body>
</html>
```

Listing 5.6: e.php - reflected XSS vulnerable web page

Practically speaking, the aforementioned pages strip out potential harmful characters from the HTTP GET parameters' value and reflect the "sanitized" content, therefore neither validation nor *contextual output encoding* are performed. This could obviously lead to XSS attacks in the case the attacker is smart enough to produce a successful attack vector.

Test A The PHP function `strip_tags()`[13] represents one of the most basic way to naively clean user-supplied HTML. Actually this function cannot supply a solid protection against XSS attacks as it does not validate attributes at all and it requires a series of regular expressions that strip out event handler attributes.[14]

[13] PHP: strip_tags, `http://php.net/manual/en/function.strip-tags.php`

[14] *Is strip_tags() horribly unsafe?*, `http://security.stackexchange.com/questions/10011/is-strip-tags-horribly-unsafe`

Test B The page expects to receive a local URL, by stripping out colon characters; moreover it makes sure that no quotes are reflected. This is a quite common scenario in which the web developer does not want to give users the chance to supply a scheme, such as `http:`, `https:`, `ftp:` and so on.

Test C The HTTP GET parameter is reflected in a double quoted JavaScript variable, the protection consists of putting a backslash before quotes; closing the script element is not allowed.

Test D The HTTP GET parameter is reflected in a single-line JavaScript comment, the basic protection is performed by stripping out the `script` keyword.

Test E This example is a little bit subtle, basically the reflection happens in the attribute `value` of an element `input`, but the `type` is set to `hidden`. Since modern web browsers do not allow to override the attribute `type`[15] and fixed any possibility to make the input visible through the attribute `style`, the unique chance to execute malicious code consists of breaking the attribute and injecting a new HTML element executing the payload.

We selected nine web application security scanners and *snuck* for our test, the selection was made on the basis of the results gained in "The Web Application Vulnerability Scanners Benchmark, 2012".[16]

Acunetix WVS Free Edition 8 (build 20120808) The free version of Acunetix WVS[17] allows to detect XSS vulnerabilities only, thereafter it is sufficient for our purposes.

OWASP ZAP 1.4.1 OWASP ZAP[18] acts as a proxy intercepting requests to web applications; by starting the active scan option against the visited web pages, it is able to discover vulnerabilities which are identified by alerts.

Ironwasp 0.9.1.4 We've already mentioned Ironwasp[19] in the previous chapters, it is a penetration testing suite, which covers the most common security vulnerabilities. Moreover, it boasts of a DOM based XSS detection engine, that is based on JavaScript static analysis.

[15] If override was possible, then injecting `" type=image src=xx:x onerror=alert(1) "` would trigger an XSS

[16] Chen, S., *The Web Application Vulnerability Scanners Benchmark, 2012*, `http://sectooladdict.blogspot.it/2012/07/2012-web-application-scanner-benchmark.html`

[17] Acunetix WVS, `http://www.acunetix.com/vulnerability-scanner/features.htm`

[18] OWASP ZAP, `https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project`

[19] Ironwasp, `http://ironwasp.org/`

ProxyStrike 2.2 ProxyStrike[20] acts similarly to OWASP ZED, it is actually a proxy that analyzes the exchanged parameters in background mode and possibly detects security issues.

SandCat Mini 4.4.3.0 Syhunt Mini[21] is a web application security scanner which performs several injections and attempts in order to discover issues.

ParosPro Desktop Edition 1.9.12 ParosPro[22] is a fully automated web application security scanner, which is basically composed by a crawler module, an attack component and an analysing module.

Arachni 0.4.0.2 Arachni[23] is a free web application security scanner framework, which boasts of a graphical interface through a web application which allows to start a scan and handle reports.

XSSSNIPER XSSSNIPER[24] is an automatic XSS discovery tool, it allows to perform mass scanning against web applications in a very easy fashion.

Xelenium Xelenium[25] is a security testing tool, that is able to discover XSS vulnerabilities by employing Selenium; it practically detects the web pages you want to test by acting as a proxy and performs injections with respect to the form fields which are present in these pages. Unfortunately running the aforementioned experiments would not generate any injection since the tool expects form fields only to identify data entry points.
Although we did not adopt this tool during our test, we decided to discover the way it practically works by making some basic experiments; we are quite fascinated by this tool as it seems to be very interesting and quite similar to the one we implemented.
Xelenium works in a very easy fashion, basically it extracts the form fields from the current web page and it injects them with several attack vectors; if the vector is perfectly reflected, then an XSS vulnerability is detected. The main drawback is that the reflection context is not taken into account and injections composed by multiple steps cannot be defined.

Table 5.6 shows the detection results, "NO" means that the vulnerability wasn't spotted, the light green "YES" indicates that the tool discovered the issue but the results need to be discussed, whereas the green "YES" stands for a successful spot.

Acunetix WVS Acunetix performs injections on the basis of the reflection context; in our test suite it discovered the issue in test D by injecting a carriage return in order to break

[20]ProxyStrike, `http://www.edge-security.com/proxystrike.php`
[21]SandCat Mini, `http://www.syhunt.com/?n=Sandcat.Mini`
[22]ParosPro,`http://www.milescan.com/`
[23]Arachni, `http://arachni-scanner.com/`
[24]XSSSNIPER, `https://bitbucket.org/gbrindisi/xsssniper`
[25]Xelenium, `https://www.owasp.org/index.php/OWASP_Xelenium_Project`

	test A	test B	test C	test D	test E
Acunetix WVS Free Edition	NO	NO	NO	YES	YES
OWASP ZAP	NO	YES	NO	YES	YES
IronWasp	NO	YES	NO	YES	YES
ProxyStrike	YES	YES	YES	YES	YES
SandCat Mini	YES	YES	YES	YES	YES
ParosPro	NO	NO	NO	YES	YES
Arachni	NO	NO	NO	NO	YES
XSSSNIPER	NO	NO	YES	YES	NO
snuck	YES	YES	YES	YES	NO

Table 5.6: Detection results with respect to five reflected XSS vulnerable web pages - read the paragraph above for the legend.

the single-line JavaScript comment, in particular it reported `922550%0a%28%29%3a%3b 942842` as a successful injection. Furthermore it discovered the XSS in test E and reported `" onmouseover=prompt(927905) bad="`, this was obviously not difficult to detect, the main problem is that the injection would not trigger an XSS in modern web browsers.

OWASP ZAP OWASP ZAP worked correctly with respect to test A by injecting `%3Cb+ onMouseOver%3Dalert%281%29%3B%3Etest%3C%2Fb%3E`, nevertheless it strangely did not report the XSS vulnerability. It detected the issue in test B, the problem here is that the returned injection `"><script>alert(1);</script>` is obviously improper with respect to that reflection context; although it is almost perfectly reflected, it cannot lead to XSS at least in that form. Eventually it worked correctly in test D and test E, very similarly to Acunetix WVS.

Ironwasp IronWasp worked similarly to Acunetix, but it was able to successfully detect a *Scriptless HTML Injection* - not an XSS - in test B by injecting `//olxizrk`. Furthermore it correctly addressed the issue in test D by reporting `%0adzkqivxy%3b%2f*` as a proper injection - note that random characters are injected instead of actual payloads, for instance it returned `" olqpir="vtikr(1)"` in test E, which is correct though.

ProxyStrike ProxyStrike correctly detected all the issues. Unfortunately information about possible attack vectors are not returned, it just reports for each HTTP GET parameter the potentially harmful characters which are perfectly reflected.

SandCat Mini Syhunt Mini worked similarly to ProxyStrike, by detecting all the issues but not returning any successful exploit.

ParosPro ParosPos spotted the last two test, D and E, by respectively returning *XSS in SCRIPT section* and *XSS w/o angle brackets*, which is perfectly legit. However no possible injection are returned.

Arachni Arachni discovered the XSS in test E. Unfortunately information about possible injections and the particular type of the issue are not given.

XSSSNIPER XSSSNIPER correctly discovered the XSS in test C and D, by respectively returning `%5B%2700001%27%5D` and `%5B%27user_00001%27%5D` as injections. This tool works by injecting complex random strings which contain potentially harmful characters and checking whether they are completely reflected; this approach is perfectly fair, but it could decrease the detection rate in the case the filtering mechanisms strips out at least one character.

snuck snuck worked correctly with respect to all the experiments, but the last one. Infact it returned many successful injections, it addressed test A with `%3Ca%20href%3Djavascript%3Aalert(1)%3Edummy_link`, test B with `javascript%26%2358;alert(1)`, test C with `%5C%22%3Balert(1)%2F%2F` and, eventually, test D with `%0D%0Aalert(1)%2F%2F`. The reason why it was not able to manage the issue in test E is related to the fact that it just reports successful injections, thereafter since no modern web browser is able to trigger an XSS in that situation, it cannot reproduce a correct exploit.
For the sake of completeness, the malicious payloads just presented are obviously url-encoded: *snuck* returns them url decoded, whereby the attacker just has to encode them before injecting.

The aforementioned experiments indicate that also straightforward XSS vulnerabilities might be difficult to detect. Although *snuck* seems to work properly in many situations, it is conceived to produce a test with respect to a given HTTP GET parameter, whereas web application security scanners need to discover the parameter before managing an attack. Actually it appears to be a good approach to adopt *snuck* for realizing whether a successful exploit exists and which particular payload would trigger an XSS exploit. In other words it could be used as a tool for assessing whether, given a possibly vulnerable parameter, it is really possible to inject it to perform an XSS attack.
Obviously a situation in which multiple steps need to be performed before landing in the reflection page would be a really different scenario; this might involve a stored XSS vulnerability in the case a weak filter is adopted. In this case *snuck* reveals to be an excellent tool as web application scanners would likely get in trouble, by having no idea about the way the application needs to be crossed.

Chapter 6

Future Work

6.1 Future improvements

The tool we presented in this work is still in an experimental phase. Even if quite stable, it is however worthy of a few improvements to widen its analysis capabilities. In this chapter we describe some further features whose implementation has just begun.
Since we were fascinated by the idea of creating an XSS filter breaker service, we came out with the idea in which users can remotely ask *snuck* to perform an XSS test against a certain web site, by just uploading an XML configuration file: the tool will be automatically started, will perform the required test and eventually will make the related report available. Nevertheless such an architecture would require that the applicant certifies he is the target web site's owner, and the complexity of such a system discouraged us to realize it.

6.1.1 Client side XSS filter testing

Instead of waiting for every web site to fix its XSS vulnerabilities, modern web browser vendors decided to supply a further security layer, that can mitigate some classes of XSS issues, i.e. the reflected ones. Bates et al. [BBJ10] widely analyzed the state of the art of client side XSS filters and proposed an interesting design which has been implemented in *XSSAuditor*, that is currently enabled by default in Google Chrome.
Basically these filters block injections by looking for content that is present in both the HTTP response and the HTTP request that generated the response. It is clear that building a filter with zero false negatives could really be a challenge, since several attack methodologies and encoding techniques have to be taken into account. In addition, false positives are actually possible and the common practice of "mangling" the injected payload by altering the HTTP response may give place to XSS on natively safe web pages, as shown by Nava et al. [12] with the Internet Explorer 8's XSS filter.[1]

[1]Ross, D., *IE 8 XSS Filter Architecture / Implementation*, `http://blogs.technet.com/b/srd/archive/2008/08/19/ie-8-xss-filter-architecture-implementation.aspx`

We slightly modified our XSS testing tool in order to use it as a tester for client side XSS filters. We crafted a vulnerable XSS web page, whose purpose was to simply reflect a certain GET parameter, as show in Listing 6.1.

```
<?php
   echo $_GET["xss"];
?>
```

Listing 6.1: Sample web page vulnerable to XSS attacks (a)

By starting *snuck* against the previous vulnerable web page in reflected XSS "mode", we are able to detect whether an injection is blocked or not by the adopted web browser's XSS filter. In other words if we are able to grab an alert, then we are sure that the client side XSS filter is bypassed.
No source code changes were needed for performing the test in Internet Explorer 9,[2] while it required to start Google Chrome with *XSSAuditor* on.[3] In the case of Firefox, we just needed to import the *NoScript*[4] extension before starting the web driver – actually Firefox has no native client-side XSS filter, fortunately NoScript is a really valid and robust defense to most common web attacks, for instance clickjacking prevention is also accomplished.
Test were performed through Google Chrome 20.0.1132.57, Internet Explorer 9.0.8 and NoScript 2.4.4.
No interesting results came out by experimenting with these settings, every attack vector was correctly blocked by the adopted web browsers. We noticed that *XSSAuditor* does not prevent *data* URIs, nevertheless this cannot be considered as a security issue, as they do not inherit the privileges of their referrer in Chrome [BBJ10].
Interesting results can be achieved through *overflow*, that is the case of generating an XSS vulnerability by exploiting characters that already exist in the page (Listing 6.2).

```
<script>
   var x = "<?php  echo $_GET['x'];  ?>";
</script>
```

Listing 6.2: Sample web page vulnerable to XSS attacks (b)

snuck can be started against the previous web page, located for instance at `http://127.0.0.1/test/xss.php` in the following manner, Listing 6.3.

[2]Internet Explorer's XSS filter is enabled by default, you need to disable it before using *snuck* – we did not find a way to deactivate it through Selenium Web Driver.
[3]*snuck* starts Google Chrome with the flag: `--disable-xss-auditor`
[4]Firefox does not have a built-in client side XSS filter, *NoScript*, `http://noscript.net/`, is an extension which provides really robust protection against security vulnerabilities

```
> java --jar snuck.jar
   -report report.html
   -reflected "http://127.0.0.1/test/xss.php"
   -p x
```

Listing 6.3: How to start snuck against the page in Listing 6.2

XSSAuditor does not block any malicious payload in the shown example, it just works as an effective XSS protection if the injection does not take advantage of the reflection context. Internet Explorer's XSS filter stops malicious requests by replacing sensitive characters from the attack vectors with a # character.
IE and NoScript employ a set of regular expressions to detect a potential attack, thereafter the attacker should find a way to neuter these in order to execute malicious JavaScript.[5,6] Basically injections which follows the pattern *";a(b)* are immediately detected, the same happens when redirecting to javascript or data URIs, i.e. *";location=*.
(Un)fortunately our tool returned an attack vector which works in Internet Explorer, see Table 6.1. This latter is actually an awesome XSS technique without parentheses, discovered by Gareth Heyes[7], that we added into the list of possible injections for the JavaScript context.

Injection
`";onerror=eval;throw'alert\x281\x29';//`

Table 6.1: IE XSS filter bypass

For the sake of completeness, we decided to compare the client-side XSS filters' behavior with respect to another reflection context, in particular the *href* attribute of an anchor.

[5]Multiple excellent evasion techniques against IE and WebKit are showed in `http://xss.cx/examples/ie/internet-exploror-ie9-xss-filter-rules-example-regexp-mshtmldll.txt`
[6]Dalili, S., *SecProject Web AppSec Challenge Series 1 Results*, `http://soroush.secproject.com/blog/2012/06/challenge-series-1-result-and-conclusion/`
[7]Heyes, G., *XSS technique without parentheses*, `http://www.thespanner.co.uk/2012/05/01/xss-technique-without-parentheses/`

```
<a href="<?php  echo $_GET['x'];  ?>">
   CLICK ME
</a>
```

Listing 6.4: Sample web page vulnerable to XSS attacks (c)

XSSAuditor does not block any injection in the form of *scheme:payload*, such as *javascript: alert(1)*, while Internet Explorer blocks malicious URIs – i.e. *javascript* and *vbscript* – by replacing the r character with a #. *NoScript* works similar to IE, it modifies the attack vectors in order to transform them into an harmless form, nevertheless *data* URIs are allowed and attackers could take advantage of this to trigger an XSS by using the base64 encoding. Further investigation and analysis would be obviously required for testing client-side XSS filters. Many security researchers are working on this point trying to continuously break them: wonderful examples are given by Kinugawa, who discovered several interesting bugs[8] in *NoScript*; the same holds for Heiderich, who found out several bypass techniques, some examples are reported in his excellent PhD thesis [Hei12].
Our idea was to employ a slightly modified version of *snuck* in order to use it as an evasion tool for testing client-side XSS filters. Since this task would require a kind of fuzzing approach, it would be useful to employ a set of fuzzed[9] attack vectors.
Moreover, multiple parameters injections are likely to result in a client-side XSS filter evasion, thus future works would require to extend the tool to managing multiple injection points.

6.2 Main limitations

Despite the potential of the presented approach for significantly testing XSS filters, some drawbacks should be taken into account.
Running a complete test against a filter requires time, which is proportional to the number of malicious vectors the tool will employ; basically using the Selenium Web Driver instead of writing a web browser extension is a good chance for covering the most common web browsers with very limited effort. A web browser extension should be rewritten for each browser you want to be compatible with, but obviously it would show off better performance – for instance Websecurify[10] is a browser extension which quickly perfoms web application security testing.
In addition, manually writing a use case for each test reveals to be quite cumbersome and annoying; however since our purpose was not to present a "point-and-click" scanner, but a much more customizable tool, realizing a software, which allows the tester to graphically

[8]Kinugawa, M., *NoScript Anti-XSS*, `http://masatokinugawa.l0.cm/2012/07/noscriptanti-xss18.html`
[9]Shazzer, `http://shazzer.co.uk/`, could be a valid repository to take the attack vectors from
[10]Websecurify, `http://www.websecurify.com/features`

select the components in a web page to interact with, could be the most accurate solution to this problem. We omitted this point in the previous section as further investigation is needed in order to assess whether it is achievable by slightly modifying the Selenium IDE, web browser extension.
Another problem might arise whenever the tested application uses CAPTCHAs to block robots to perform automatic operations, such as posting spam comments. In that case the automation would need to be stopped at any time a CAPTCHA appears, asking the human intervention.
Finally, we should not forget that *snuck* is at the moment restricted to a specific set of well known and predefined injections. This means that bypassing a filter would require to know in advance a possibly successful injection. This approach is quite fair with respect to common XSS filters, but it might be unsuccessful against much more complex protection systems; thereafter it would be really interesting to extend it with a *fuzzer* module that should be able to combine HTML tags and attributes, special characters and the available payloads in order to inject many fuzzed, but reasonable, vectors. This problem could be addressed by making *snuck* learn how different encoding techniques could be adopted in several contexts and by making it aware of how it is possible to combine HTML tags with rational event handler attributes. Future improvements are moving to this direction.
Eventually, evading filters requires a creative mind and new attack vectors should be continuously added into the pool of the employed payloads in order to expand the possibilities to spot an issue.

6.3 Conclusions

The primary goal of this work was to describe an approach for testing XSS filters in an uncommon way. This opportunity is related to the possibility to use Selenium WebDriver in order to drive the web browser into making a sequence of operations in which multiple injections are performed against a targeted web application. Furthermore we focused on the importance of the reflection context, by hoping that every web application security scanner vendor will move in this direction; basically no big effort would be required, since XPath queries or HTML parsers could be successfully used to retrieve the reflection context, however attack vectors would need to be categorized.
Eventually, approaching the XSS filters evasion problem by following the aforementioned methodologies might be quite cumbersome in the case the web application adopted multiple points in which users may marshal content; nevertheless it could be interesting to convert the tool into a plugin for common web application security scanners, by giving the tester the chance to perform a targeted test against a filter. Such an approach might significantly improve the detection of bugs in XSS filters and it would have the huge advantage of covering the most used web browsers, whereby a huge portion of potential victims.

Chapter 7

Bibliography

[BBGM10] Jason Bau, Elie Bursztein, Divij Gupta, and John Mitchell. State of the art: Automated black-box web application vulnerability testing. In *Proceedings of IEEE Symposium on Security and Privacy*, Los Alamitos, CA, USA, 2010. IEEE Computer Society.

[BBJ10] Daniel Bates, Adam Barth, and Collin Jackson. Regular expressions considered harmful in client-side XSS filters. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 91–100, New York, NY, USA, 2010. ACM.

[CVDP10] Angelo Ciampa, Corrado Aaron Visaggio, and Massimiliano Di Penta. A heuristic-based approach for detecting SQL-injection vulnerabilities in web applications. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems*, SESS '10, pages 43–49, New York, NY, USA, 2010. ACM.

[DCV10] Adam Doupé, Marco Cova, and Giovanni Vigna. Why Johnny can't pentest: an analysis of black-box web vulnerability scanners. In *Proceedings of the 7th international conference on Detection of intrusions and malware, and vulnerability assessment*, DIMVA'10, pages 111–131, Berlin, Heidelberg, 2010. Springer-Verlag.

[FO07] Elizabeth Fong and Vadim Okun. Web application scanners: Definitions and functions. In *Proceedings of the 40th Annual Hawaii International Conference on System Sciences*, HICSS '07, pages 280b–, Washington, DC, USA, 2007. IEEE Computer Society.

[Hei12] Mario Heiderich. Towards elimination of XSS attacks with a trusted and capability controlled DOM. `http://heideri.ch/thesis`, 2012. [PhD Thesis].

[HVNEL10] Mario Heiderich, Eduardo Alberto Vela Nava, Gareth Eyes, and David Lindsay. *Web Application Obfuscation: '-/WAFs..Evasion..Filters//alert(/Obfuscation/)-'*. Syngress, 2010.

[Jav11] Ashar Javed. Csp aider: An automated recommendation of content security policy for web applications. In *IEEE Symposium on Security and Privacy*, 2011.

[KKKJ06] Stefan Kals, Engin Kirda, Christopher Kruegel, and Nenad Jovanovic. Secubat: a web vulnerability scanner. In *Proceedings of the 15th international conference on World Wide Web*, WWW '06, pages 247–256, New York, NY, USA, 2006. ACM.

[Kor10] Christian Korscheck. Automatic detection of second-order cross-site scripting vulnerabilities. `http://www.korscheck.de/diploma-thesis.pdf`, 2010.

[PZ08] Andrey Petukhov and Dmitry Zozlov. Detecting security vulnerabilities in web applications using dinamic analysis with penetration testing. In *Proceedings of Application Security Conference*, 2008.

[Zal12] Michal Zalewski. *The Tangled Web: A Guide to Securing Modern Web Applications*. No Starch Press, 2012.

Chapter 8

Sitography

[1] A. Klein, DOM based cross site scripting or XSS of the third kind, http://www.webappsec.org/projects/articles/071105.shtml, 2005, [Last accessed in September 14, 2012].

[2] D. Wichers, Unraveling some of the mysteries around DOM-based XSS, ASDC12, https://www.owasp.org/images/f/f4/ASDC12-Unraveling_some_of_the_Mysteries_around_DOMbased_XSS.pdf, 2012, [Last accessed in September 14, 2012].

[3] Y. Hasegawa, Make a contract with IE and become a XSS girl!, Hitcon 2011, http://hitcon.org/hit2011/downloads/09_Make%20A%20Contract%20with%20IE%20and%20Become%20a%20XSS%20Girl.pdf, 2011, [Last accessed in September 14, 2012].

[4] E. Vela Nava and D. Lindsay, Cross site location jacking (XSLJ), OWASP AppSec Research, http://www.youtube.com/watch?v=P0YFwhYtrwM, 2010, [talk's video].

[5] V. Chapela, Advanced SQL injection, http://cs.unh.edu/~it666/reading_list/Web/advanced_sql_injection.pdf, 2005, [Last accessed in September 14, 2012].

[6] B. D. A. Guimares, Advanced SQL injection to operating system full control, http://www.blackhat.com/presentations/bh-europe-09/Guimaraes/Blackhat-europe-09-Damele-SQLInjection-whitepaper.pdf, 2009, [Last accessed in September 14, 2012].

[7] L. Kuppan, Ironwasp a web application security testing platform, Securitybyte 2011, http://securitybyte.org/resources/2011/presentations/ironwasp.pdf, [Last accessed in September 14, 2012].

[8] M. Zalewski, Postcards from the post-XSS world, http://lcamtuf.coredump.cx/postxss/, 2011, [Last accessed in September 14, 2012].

[9] Y. Pavlosoglou, Penetration testing with Selenium, https://www.owasp.org/images/3/37/OWASP_London_14-Jan-2009_Penetration_Testing_with_Selenium-Yiannis_Pavlosoglou_v2.pdf, 2009, [Last accessed in September 14, 2012].

[10] Z. Chao Chao and P. Cheng Yu, Automated web testing with Selenium, http://www.ibm.com/developerworks/opensource/library/os-webautoselenium/, 2010, [Last accessed in September 14, 2012].

[11] A. Sotirov, Blackbox reversing of XSS filters, Recon 2008, http://recon.cx/2008/a/alexander_sotirov/recon-08-sotirov.pdf, 2008, [Last accessed in September 14, 2012].

[12] E. Vela Nava and D. Lindsay, Abusing Internet Explorer 8's XSS filters, BlackHat Europe 2010, http://p42.us/ie8xss/Abusing_IE8s_XSS_Filters.pdf, 2010, [Last accessed in September 14, 2012].

Appendix A

Appendix

This appendix contains the use cases employed during the presented experiments and other details we omitted in the discussion.

A.1 Appendix 1

Sample use case in which a sequence of operation is defined. Note that data are described in the form of <`name, value`>.

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
   <post>
      <parameters>
         <parameter>
            <name>username</name>
            <value>new_test77</value>
         </parameter>
         <parameter>
            <name>email</name>
            <value>new_test77@test.org</value>
         </parameter>
      </parameters>
      <commands>
         <command>
            <name>open</name>
            <target>http://foo.foo</target>
            <value></value>
         </command>
         <command>
            <name>type</name>
            <target>name=author</target>
            <value></value>
```

```
         </command>
         <command>
            <name>type</name>
            <target>name=author</target>
            <value>${USERNAME}</value>
         </command>
         <command>
            <name>type</name>
            <target>name=comment</target>
            <value>${INJECTION}</value>
         </command>
         <command>
            <name>submit</name>
            <target>id=comment</target>
            <value></value>
         </command>
      </commands>
   </post>
</root>
```

A.2 Appendix 2

Sample XML configuration file for reflected XSS testing. Note that you also can avoid writing this file and start *snuck* with the arguments `-reflected` and `-p` set. Starting the tool in the following way will make it achieve the same task without the need of writing the configuration file.

```
>java -jar snuck.jar
   -report report.html
   -reflected "http://target.foo/xss.php?foo=bar&foo2=bar2"
   -p hidden_xss
```

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
   <get>
      <parameters>
         <targeturl>http://target.foo/xss.php</targeturl>
         <reflectionurl></reflectionurl>
         <paramtoinject>hidden_xss</paramtoinject>
         <parameter>foo=bar</parameter>
         <parameter>foo2=bar2</parameter>
      </parameters>
   </get>
</root>
```

A.3 Appendix 3

Available options in *snuck*.

```
>java -jar snuck.jar

Usage: snuck [-start xmlconfigfile] -config xmlconfigfile
-report htmlreportfile [-d #ms_delay] [-proxy IP:port]
[-chrome chromedriver]  [-ie iedriver]
[-remotevectors URL] [-stop-first]
[-reflected targetURL -p parameter_toTest] [-no-multi]

Options:

 -start         path to login use case (XML file)
 -config        path to injection use case (XML file)
 -report        report file name (html extension is required)
 -d             delay (ms) between each injection
 -proxy         proxy server (IP:port)
 -chrome        perform a test with Google Chrome, instead of Firefox
                It needs the path to the chromedriver
 -ie            perform a test with Internet Explorer, instead of Firefox
                Disable the built in XSS filter in advance
 -remotevectors   use an up-to-date online attack vectors' source
                instead of the local one
 -stop-first    stop the test upon a successful vector is detected
 -no-multi      deactivate multithreading for the reverse
                engineering process - a sequential approach
                will be adopted
 -reflected     perform a reflected XSS test
                (without writing the XML config file)
 -p             HTTP GET parameter to inject
                (useful if -reflected is set)
 -help          show this help menu
```

A.4 Appendix 4

Use case adopted in *WordPress 3.3.1* for testing the filter used against the visitors' comments.

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
   <post>
      <parameters>
         <parameter>
            <name>username</name>
```

```
            <value>new_test77</value>
         </parameter>
         <parameter>
            <name>email</name>
            <value>new_test77@test.org</value>
         </parameter>
      </parameters>
      <commands>
         <command>
            <name>open</name>
            <target>http://127.0.0.1/wordpress/?p=1</target>
            <value>noforce</value>
         </command>
         <command>
            <name>type</name>
            <target>name=author</target>
            <value></value>
         </command>
         <command>
            <name>type</name>
            <target>name=author</target>
            <value>${USERNAME}</value>
         </command>
         <command>
            <name>type</name>
            <target>name=email</target>
            <value></value>
         </command>
         <command>
            <name>type</name>
            <target>name=email</target>
            <value>${EMAIL}</value>
         </command>
         <command>
            <name>type</name>
            <target>name=comment</target>
            <value>${INJECTION}</value>
         </command>
         <command>
            <name>submit</name>
            <target>id=comment</target>
            <value></value>
         </command>
      </commands>
   </post>
</root>
```

A.5 Appendix 5

Use case adopted in *Habari 0.8* for testing the filter used against the visitors' comments.

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
   <post>
      <parameters>
         <parameter>
            <name>username</name>
            <value>test</value>
         </parameter>
         <parameter>
            <name>email</name>
            <value>test@test.org</value>
         </parameter>
      </parameters>
      <commands>
         <command>
            <name>open</name>
            <target>http://127.0.0.1/habari-0.8/habari</target>
            <value></value>
         </command>
         <command>
            <name>type</name>
            <target>id=comment_name</target>
            <value></value>
         </command>
         <command>
            <name>type</name>
            <target>id=comment_name</target>
            <value>${RANDOM}</value>
         </command>
         <command>
            <name>type</name>
            <target>id=comment_email</target>
            <value></value>
         </command>
         <command>
            <name>type</name>
            <target>id=comment_email</target>
            <value>${RANDOM_EMAIL}</value>
         </command>
         <command>
            <name>type</name>
            <target>id=comment_content</target>
```

```
            <value>${INJECTION}</value>
         </command>
         <command>
            <name>submit</name>
            <target>id=comment-public</target>
            <value></value>
         </command>
      </commands>
   </post>
</root>
```

A.6 Appendix 6

Use case adopted in *Symphony 2.2.5* for testing the filter used against the visitors' comments.

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
   <post>
      <commands>
         <command>
            <name>open</name>
            <target>http://127.0.0.1/symphony2.2.5/articles/
            a-primer-to-symphony-2s-default-theme/#comments</target>
            <value></value>
         </command>
         <command>
            <name>type</name>
            <target>name=fields[author]</target>
            <value></value>
         </command>
         <command>
            <name>type</name>
            <target>name=fields[author]</target>
            <value>${RANDOM}</value>
         </command>
         <command>
            <name>type</name>
            <target>name=fields[email]</target>
            <value></value>
         </command>
         <command>
            <name>type</name>
            <target>name=fields[email]</target>
            <value>test@test.com</value>
         </command>
```

```
         <command>
            <name>type</name>
            <target>name=fields[website]</target>
            <value></value>
         </command>
         <command>
            <name>type</name>
            <target>name=fields[website]</target>
            <value>${INJECTION}</value>
         </command>
         <command>
            <name>type</name>
            <target>name=fields[comment]</target>
            <value></value>
         </command>
         <command>
            <name>type</name>
            <target>name=fields[comment]</target>
            <value>${RANDOM}</value>
         </command>
         <command>
            <name>click</name>
            <target>name=action[save-comment]</target>
            <value></value>
         </command>
         <command>
            <name>type</name>
            <target>name=fields[website]</target>
            <value></value>
         </command>
      </commands>
   </post>
</root>
```

A.7 Appendix 7

Use case adopted in *Plone 4.4.1* for testing the filter used against the visitors' comments.

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
   <post>
      <commands>
         <command>
            <name>open</name>
            <target>http://localhost:8080/Test/news/op</target>
```

```
            <value></value>
         </command>
         <command>
            <name>type</name>
            <target>id=form-widgets-text</target>
            <value>${INJECTION}</value>
         </command>
         <command>
            <name>click</name>
            <target>id=form-buttons-comment</target>
            <value></value>
         </command>
         <command>
            <name>click</name>
            <target>name=form.button.DeleteComment</target>
            <value></value>
         </command>
      </commands>
   </post>
</root>
```