COMPARING WEB APPLICATION SCANNERS FOR XSS ATTACKS

by

DENGFENG XIA

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT ARLINGTON

May 2012

ACKNOWLEDGEMENTS

This project would not have been possible without the continuous guidance, encouragement, and support of many people.

First and foremost, I would like to express my most sincere gratitude to my supervising professor, Dr. Yu Lei, who has given me invaluable guidance and assistance all the time. His timely inspirations and willingness to motivate contributed tremendously to our project.

I would like to thank our supervising committee members, Dr. Christoph Csallner and Dr. Donggang Liu, for their interests in this project and assistance in the defense scheduling.

I would like to thank Dr. Matthew Wright for the help in his excellent Secure Programming course, which helped me learn related back ground knowledge which inspired this project.

I would like to thank all students in Software Engineering Lab who helped me during my experiments.

And also, I would like to express my gratitude to all my family members, who have given me unselfish support and encouragement for my academic work, which was the key factor of my concentration on this project. Their understanding and love provided me motivation for pursuing my interests.

May 8, 2012

ABSTRACT

COMPARING WEB APPLICATION SCANNERS FOR XSS ATTACKS

DENGFENG XIA, M.S.

The University of Texas at Arlington, 2012

Supervising Professor: Yu Lei

As the software industry pays increasing attention to web application security, various black box web security scanners have been developed. XSS attacks are one of the major attacks that can make severe damages to victim systems, and we focus on the XSS detection effectiveness for a number of scanners in this project. Most existing projects evaluate the scanning performance for various vulnerability types, and they do not give an adequate evaluation on XSS issues. Further, their evaluations either use vulnerable applications in real life, or use test applications created by themselves. As a result, their evaluation results might be biased due to the limited number of test applications. Finally, most projects only compare the final scanning results, without giving a deeper analysis of the inner scanning mechanisms of the different scanners.

In this project, we evaluate web application scanners that are widely used in practice. We not only compare their performance, but also explain reasons causing the differences. In our evaluation, we first use real life vulnerable web applications to evaluate the performance of the scanners in different scanning phases. Then we use JSP test applications created by ourselves to evaluate their abilities of sending fuzzed data and analyzing the scanners' responses. At last, we explain their performance differences by comparing their injection patterns. Our evaluation results indicate that their different scanning outputs in various phases have influenced their final scanning results. However, the performance of crawling does not seem to be the only key factor, which is different from conclusions of many related projects. Our deeper study about injection patterns suggests that all scanners have a certain variety of patterns we have compared, and their injection effectiveness may result from multiple factors.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iii

ABSTRACT ivv

TABLE OF CONTENTS vii

LIST OF ILLUSTRATIONS ixx

LIST OF TABLES x

Chapter Page

1. INTRODUCTION 1

2. RELATED WORK 5

3. CROSS-SITE SCRIPTING ATTACKS 11

3.1 Reflected XSS 11

3.2 Stored XSS 15

3.3 DOM-based XSS 16

3.4 XSS Preventions 18

4. A GENERAL DETECTION FRAMEWORK 22

4.1 Crawling 23

4.2 Deciding Injection Points 24

4.3 Making Injections 24

4.4 Reporting 25

4.5 Case Study 26

5. EVALUATION 29

5.1 Web Application Scanners 29

5.1.1 Overview 29

5.1.2 Netsparker Community Edition 30

5.1.3 Acunetix Web Vulnerability Scanner Free Edition 31

5.1.4 Skipfish 33

5.1.5 Wapiti 35

5.1.6 Summary 37

5.1.7 Other Tools 37

5.2 Case Studies 38

5.2.1 Subject Applications 38

5.2.2 Evaluation Method 41

5.2.3 Results and Discussion 42

5.2.4 Summary 52

5.3 Evaluation with Controlled Test Applications 52

5.3.1 Motivation 53

5.3.2 Evaluation Method 53

5.3.3 Result and Discussion 59

5.4 Further Comparison 62

5.4.1 Comparison Method 63

5.4.2 Comparison of Injection Contexts 65

5.4.3 Advanced Injection Techniques Comparison 68

5.5 Summary 73

6. CONCLUSIONS AND FUTURE WORK 75

6.1 Crawling 75

6.2 Deciding Injection Points 76

6.3 Injection Effectiveness 77

6.4 Future Work 77

APPENDIX

A. FALSE POSITIVE VALIDATIONS 79

REFERENCES 85

BIOGRAPHICAL STATEMENT 88

LIST OF ILLUSTRATIONS

Figure Page

3.1 Reflected XSS Attack Steps 13

5.1 Netsparker GUI 31

5.2 Acunetix Login Sequence Recorder 32

5.3 Acunetix Test Result 33

5.4 Skipfish Command Line Interface 34

5.5 Skipfish Report 35

5.6 Wapiti Report Overview 36

5.7 Wapiti Report In Detail 36

LIST OF TABLES

Table Page

5.1 Usability Comparison for 4 scanners 37

5.2 Statistics of Subject Applications....39

5.3 Vulnerabilities Overview 41

5.4 Type 1 Issues Scanning Results 44

5.5 Acunetix Scanning Result 46

5.6 Netsparker Scanning Result 47

5.7 Skipfish Scanning Result 48

5.8 Wapiti Scanning Result 49

5.9 OSVDB Issues Scanning Statistics....50

5.10 Non OSVDB Issues Scanning Statistics 51

5.11 Controlled Test Applications Overview 58

5.12 Controlled Test Applications Scanning Results 60

5.13 Comparison of Structural Tags....66

5.14 Comparison of Modifier Tags....67

5.15 Comparison of Element Tags 67

5.16 Comparison of Encoding Techniques....70

5.17 Comparison of Injection Techniques 72

5.18 Injection Tag Patterns Summary 74

A.1 Validations for Acunetix Results 80

A.2 Validations for Netsparker Results 82

A.3 Validations for Skipfish Results 83

CHAPTER 1

INTRODUCTION

Due to wide-spread information sharing through the internet, the software industry is paying increasing attention to various issues in web application security. Preventing Cross-site scripting (XSS) attacks is always very important. In this thesis, we give a detailed evaluation of several scanners on the XSS detection effectiveness, study their detection mechanisms, and compare their performance in different scanning phases.

Among various web application security threats, statistics have suggested that XSS vulnerabilities are among the most widely exploited. The Web Application Security Statistics Project [24] showed that XSS is the most common vulnerability type. Among all the vulnerabilities analyzed by this project, XSS accounted for 59% in their report of year 2007, and 43% in the report of year 2008. According to the 7th Website Security Statistics Report [25] by WhiteHat Security in 2009, about 65% of websites contained XSS vulnerabilities. In a report named 2011 CWE/SANS Top 25 Most Dangerous Software Errors[4], XSS was also claimed to be the most prevalent threat type. Apart from its wide spread nature, the consequences of XSS attacks can range from minor malice threats to serious security damages. Attackers can steal cookies, forge requests of valid users, redirect users to malicious websites, install Trojan horse programs, and execute malicious code on the victim's browser. Various scanners that can detect XSS

vulnerabilities have been developed. They have similar detection mechanisms at a high level, but their detection effectiveness is different. This thesis is helping us have a good understanding of strengths and limitations of different tools.

XSS attacks rely on injecting crafted attack strings in user inputs to vulnerable web applications. Since those applications do not have proper input validation mechanisms, they return responses contaminated by malicious scripts, which can perform unexpected attacks in the victim browsers. Most web application scanners detect XSS vulnerabilities by sending benign XSS attacks to the target applications and checking the responses for the "traces" of the previous injections. Although several projects have provided evaluations for web application scanners, they do not focus on the XSS detection effectiveness. Instead, they provide a general evaluation for various vulnerability types, such as SQL injection, information leak, command injection, etc. Moreover, most projects either use case studies of real-life applications, or controlled studies with crafted applications, but not both. Finally, these projects do not give an in-depth and comprehensive analysis of the detection mechanisms employed by different scanners.

In our project, we conduct a comprehensive evaluation for four popular web application scanners, including Skipfish[29], Wapiti[27], Netsparker community edition[14], and Acunetix free edition[1]. We also provide deep analysis for their detection effectiveness. Our evaluation includes case studies for real-life web applications and controlled studies with test applications created by ourselves. We choose five vulnerable web applications having XSS vulnerabilities reported by the

OSVDB website, which is online database supported by the web security community, providing information of discovered vulnerabilities in various web applications. To better represent web application vulnerabilities populated “in the wild”, we keep five applications belonging to different categories. For controlled studies, we create a total number of more than 50 simple test applications. Among them, we have base test applications, which have various XSS vulnerabilities but not any input validation mechanism. For each of base applications, we have some variation test applications, which add some input validations to the base test applications, escaping certain special characters. As discussed in chapter 2, an existing study [7] suggests that many web application scanners cannot bypass very advanced input validation mechanisms. Thus, the escaping techniques we use are simple and common, paying attention to some typical symbols such as quotation marks and angle brackets. To analyze the scanning results, we not only present the statistics, but also study how each scanner is working in detail, by tracking and analyzing network packets. We explain the detection mechanisms in different web application scanners, compare their performances in different scanning phases, such as crawling, deciding injection points, making injections, etc, and analyze the factors affecting their detection effectiveness. For example, in case studies, for a reported issue, if scanners fail to detect it, we identify whether they can only reach the vulnerable location during the crawling phase, or they can take a further action sending injections to exploit the vulnerability. We study how the injection contents look like if scanners succeed at detecting the vulnerability, capture and compare attack strings of different

scanners during the XSS injection phase, trying to discover the relationship between the variety of injection strings and their injection effectiveness.

Based on our evaluation results and analysis, we have several useful findings. Our evaluation results suggest that two scanners, Netsparker and Acunetix perform better than the other two scanners, Skipfish and Wapiti. Our analysis shows that the differences in the overall detection effectiveness are mainly due to their different performances in the phase of deciding injection points and making injections. It is worth noting that in our evaluation, the effectiveness of the crawling stage is not the only factor that affects the overall detection effectiveness. This is different from many existing projects which emphasize on the importance of crawling. By comparing injection strings, we find that even though many scanners use similar injection patterns, their injection effectiveness is different. There can be multiple reasons affecting the injection effectiveness.

The structure of this thesis is described as the following:

In chapter 2, several projects in related work are discussed, including their contributions and limitations.

In chapter 3, we give background information for the cross-site scripting vulnerability, introducing various types of XSS attacks and prevention techniques.

In chapter 4, we introduce the XSS detection mechanisms of web application scanners.

In chapter 5, we give a detailed description of our evaluation. First, we introduce web application scanners we evaluated in this project. Then we describe case studies of several real-life applications, followed by controlled studies of several test applications we created. At last, we compare the injection patterns used by different scanners.

In chapter 6, we provide concluding remarks. In particular, we discuss the key factors affecting the detection effectiveness of different scanners. Discussion for future work is also included.

CHAPTER 2
RELATED WORK

In this section, we introduce several related projects and compare our work to them.

The project that has inspired us the most is the one by Shay Chen[2]. In particular, we created our test applications based on his work. He conducted a comprehensive evaluation for over 40 popular scanners. In his evaluation, a number of test applications having XSS vulnerabilities were created. In our project, after studying some documents about XSS prevention techniques, such as the XSS prevention cheat sheet provided by OWASP (open-source application security project), we create a number of test applications based on his work. There are some differences in our projects. First, we organize the test applications and present the evaluation results in a more systematic way. We create test applications according to different scenarios. For each scenario, we create base test application(s) and optional derived test applications having certain character escaping mechanisms. Moreover, apart from the test applications from Shay's project, we add into our project several new test applications, including some variation test applications in several scenarios, and the applications in which there are some XSS vulnerabilities in javascripts.

Several projects conducted a general evaluation for various vulnerability types. For example, Jason[2] Bau's team from Stanford conducted experiments on both real-life vulnerable web applications and a controlled test bed. The vulnerability types in the

evaluation included XSS, SQL injection, Cross Channel Scripting, Information leak, etc. The scanners they used were Acunetix, HailStorm, WebInspect, Rational AppScan, McAfee Secure, N-Stalker, Qualysguard, and Nexpose. For real-life vulnerable applications, they chose Drupal, phpBB, and Wordpress, and each application had several vulnerabilities of each vulnerability type. In the evaluation with the controlled test bed, they compared the detection effectiveness for the vulnerabilities in the sub-categories they identified. They compared scanning results, execution time, and amount of data transmitted for the scanners they used. Their work and our approach adopted similar methods, since both of us include case studies for real-life web applications and controlled test applications. The difference is that they made general comparisons for various vulnerability types, but did not provide a detailed analysis on the factors causing the differences; our work focuses on the XSS detection effectiveness, and we have further analysis on scanners' performance in all scanning phases.

Some projects created effective test applications to better differentiate the scanning performance of scanners. For example, Elizabeth Fong[8] and her team from NIST created a banking application containing 3 types of vulnerabilities, XSS, SQL injection, and File inclusion. They proposed the concept of defense levels in web applications, which indicates how difficult for scanners to exploit the vulnerabilities. When constructing the test applications, they applied this idea by crafting multiple vulnerabilities in different defense levels, and making the test suite able to configure its defense levels according to the needs of different tests. Their scanning results

showed that most tools could not detect vulnerabilities having a defense level more than 2, indicating that most tools were not able to bypass some advanced input validation mechanisms. The similarity of their work to our approach is that we both apply the concept of defense levels when creating test applications. We add escaping mechanisms into our variation test applications, having different difficulties for scanners to exploit the vulnerabilities. Since very advanced escaping mechanism cannot help much in differentiating scanners' detection effectiveness, our escaping functions are quite simple and common. One difference between the two projects is that we use multiple real life vulnerable web applications for our evaluation, having more diversity and more practical challenges than their single banking application.

In the evaluation process, some projects paid attention to several practical problems which could hardly be ignored. Adam Doupe[5] and his team created a test application with several types of vulnerabilities, called Wackopicko. They evaluated 11 scanners, and they used the scanners in three modes, representing different challenges in the crawling phase. During their evaluation process, they discussed many practical problems hindering the crawling process, such as parsing malformed HTML pages, reaching pages in multi-step processes, being stuck within infinite loops, failures in automatic authentications, etc. This is similar to our evaluation experience, as we have found that scanners can fail to maintain session states in several situations, and they sometimes even fail to pass the authentications at the beginning of the crawling phase. This can make it harder for crawlers to detect web pages that can only be reached after logging into a website.

Their work had some differences comparing to our work. They had an evaluation about various vulnerability types, but we only focus on XSS issues. Their evaluation results showed that the crawling quality of a tool is vital to its overall performance, but our work suggests that many other factors also have significant impact on a scanner's detection effectiveness.

Apart from the method of using real life vulnerable applications in case studies, some people like Suto[20] conducted evaluations using test sites provided by different scanner vendors, as he believed they could be a good representation of various vulnerabilities in the wild. His evaluation had tests with two types of crawling configurations. One of them is called point and shoot tests, which used the default crawling options. Another is called trained tests, which used various advanced configurations to reach the best page coverage possible. Most scanners had only moderate improvements benefit from trained crawling. One difference comparing to our project is that our findings were based on case studies of real-life vulnerable web applications.

Jose[8] and his team proposed another way to create test applications and used them in the evaluation for three scanners, Acunetix, Rational AppScan, and QualysGuard. They injected faults into web applications to create potential security vulnerabilities, generating test cases once the vulnerabilities were confirmed. They modified two web applications by injecting typical software faults. After every injection, they used scanners to detect XSS vulnerabilities, and identified the reported issues as vulnerabilities already existed before the injection, or newly generated ones. If the newly generated issue is a true positive, they created a test case based on the injection. They

injected hundreds of faults into a single application, and found dozens of XSS and SQL injection vulnerabilities with the help of different scanners. Since the process was very time consuming and required repetitive work for every single injection, they wrote a java program to automate their evaluation process. The similarity between their method and our approach is that both projects try to use vulnerabilities in real-life web applications. The difference is that their vulnerabilities came from fault injections, which had sufficient quantity but might not be representative enough for vulnerabilities in real life; ours come from real problems reported in the community.

Many other resources have given us help when we were trying to conduct a deeper study about XSS attacks. The Open Web Application Security Project (OWASP) [19] is one of them. It is a non-profit project open to the community, providing tutorials, guidelines, methodologies, technology information, and other detailed and up-to-date resources for projects in web application security area. It is supported by a large community composed of industrial companies, educational organizations, individual students, and IT workers. It avoids affiliation with any commercial organization to ensure independence. It provides us tutorials presenting XSS attacking techniques, testing principles and guidelines, and useful resources of related work. For example, it introduces tutorial projects like WebGoat to teach web application security lessons in which users can learn to exploit vulnerabilities in several applications in the projects; it has projects like SiteGenerator allowing users to create vulnerable web sites for learning and testing; it provides prevention cheat sheets for XSS attack, illustrating attacking theories and prevention rules, which gives us

background knowledge to understand injection patterns of web application scanners, and guidance to create our test applications.

CHAPTER 3

CROSS-SITE SCRIPTING ATTACKS

In this chapter, we will give background knowledge of XSS attacks. We will discuss three types of XSS vulnerabilities, give several attack examples, and give an introduction to XSS prevention techniques at a high level.

Cross-Site Scripting (XSS) is one of the most common web application vulnerabilities. Many XSS attacks happen because vulnerable applications fail to sanitize malicious input at either server side or browser side, allowing them to be injected into response pages. By altering the original page structures, the injected code is able to achieve malicious intentions in victim browsers.

3.1 Reflected XSS

Traditionally, XSS vulnerabilities reside in the process when the server-side code preparing html responses to users. This is different from the DOM-based XSS after the advent of web 2.0, which will be discussed in section 3.3. Traditional XSS vulnerability can be classified as reflected XSS and stored XSS. A reflected XSS vulnerability allows users to inject malicious input which can be reflected back immediately in the response. It is considered as “non-persistent”. A typical scenario of reflected XSS attack can be:

A user types in search strings on a search website. After clicking the “search” button, the user input is supposed to be reflected back in the result page. If there are some malicious scripts within the input, and there is no proper input validation on the server side, the malicious scripts will be executed in the browser when the response is received. Based on this kind of injections, attackers can create a benign-looking URL containing XSS vectors. Users are lured to click on these URLs and the injected scripts will be run in the victim browsers.

To better illustrate the steps in a reflected XSS attack, we give a detailed example in this section. Imagine there is an online-shopping application having reflected XSS vulnerabilities. It allows registered users to post business transaction information, and communicate with other users through the message system within the website. To facilitate transactions, every registered user saves his bank account information in the web site, which can only be viewed and changed by the owner himself after authentication. Tony is one of the frequent users of this web site. He posts a message looking for sellers who can sell him some video games at a low price. Gary is an attacker. After seeing Tony’s posting, he performs the following attack steps, which are illustrated in figure 3.1.

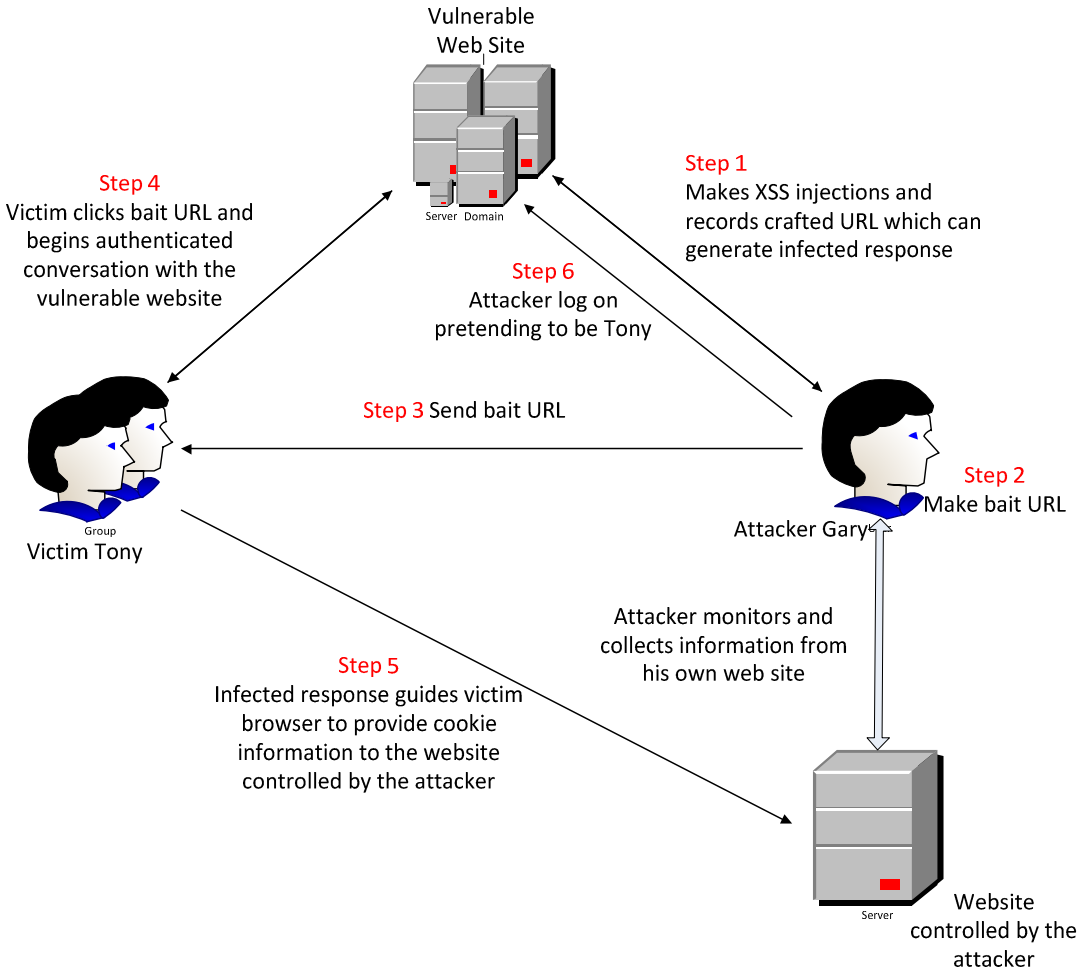


Figure 3.1 Reflected XSS Attack Steps

1. In the browse page, Gary chooses to view products of a particular seller. In an ordinary scenario, the content of the “user_name” parameter in a post message is accepted and stored in the server side, the PHP code in the server can be like:

   $name = $_POST[“user_name”];

The variable “name” will be used for composing search queries. If the user is not found in the database, the website will display a message using PHP code:

echo ”The user $name is not found”;

Since the application does not have a proper input sanitation mechanism, the content of “user_name” will be echoed back directly. After identifying the parameter “user_name” as a injection point, Gary uses a crafted script to replace the normal seller name, for example, the script used can be:

<script>var cookies=document.cookie;

document.write(“<img src=http://www.Gary.com/phisher.php?”+cookies +”>”);</script>.

When a browser tries to display the response page, the “img” tag generated will send the cookie information to www.Gary.com, which is a website created and monitored by the attacker, having certain functions in the file phisher.php to process the cookie information in requests.

2. Based on the request URL with fuzzed string in the parameter “user_name”, attacker Gary can craft an URL string which seemed benign. He may keep the part www.site.com, which is the legitimate name of the shopping website, and encode the malicious script part, making the URL looks as coming from a trust source.
3. Gary sends a message to Tony through the website message, providing him the crafted URL string, telling him to click the link to view the products that Gary can provide.

4. After logging on and seeing the new message from Gary, Tony clicks the link, and a response page containing the XSS vector is returned.
5. Tony's browser is redirected to the attacker's website. Tony's cookie information is sent to www.Gary.com and captured by the script in phisher.php
6. Using the cookie information, Gary can log on the website with the identity of Tony, and view his sensitive information stored in the account.

Note that in Figure 3.1, since the process that the attacker monitors his own website is a consistent action, the arrow representing this is different from others. The arrows for step 1 and step 4 are bidirectional. In step 1, the attacker makes a XSS injection, and records the URL reflected in the response page. In step 4, there is a conversation consists of requests and responses between the victim and the website.

In various reflected XSS attacks, step 5 and step 6 may be different, since attackers may not want to steal the victims' cookies every time. As long as they can make victims' browsers run their malicious scripts, they can do many things as they wish.

3.2 Stored XSS

Different from reflected XSS, stored XSS can accept a user's input and keep it on the web page, and it is considered as "persistent". It may have the same steps as those after step 3 in the reflected XSS attack illustrated above, but it does not require attackers to use phishing techniques

to lure victims to visit another website, and the XSS vector is injected onto the web page permanently. A typical scenario of a stored XSS attack can be:

There is a blog application allowing readers to post their messages. An attacker may post some malformed content for the value of the regular message title and body. If the website cannot validate the user inputs, malicious scripts will be injected into the attacker's posting permanently, which can be viewed by others. Whenever the posting page is visited, the malicious scripts will be run in the victim browsers.

3.3 DOM-based XSS

Apart from traditional XSS attacks, including reflected XSS attacks and stored XSS attacks, another XSS attack type is called DOM-based XSS. Unlike traditional XSS attacks which rely on having malicious payloads embedded in the reflected pages, DOM-based XSS attacks modify the DOM environment in the victim browsers, without changing the actual HTML response contents. DOM, which is the abbreviation for Document Object Model, is a convention representing objects in HTML, XHTML, or XML documents. It provides interfaces for Java scripts to manage the structure and attributes of page contents. For example, the "getElementById" method of the Document object can return a reference to the first object having the specified id in the web page; the "host" attribute of the Location object has the information of the current host name and port number.

To better illustrate DOM based XSS attacks, we take the example in the previous reflected XSS attack again. Imagine the application does not use the content stored in the variable “name” at the server side when displaying the error message. Instead, it uses JavaScript in response pages to extract the information in the DOM objects:

```
var pos=document.URL.indexOf("user=")+5;
document.write(document.URL.substring(pos,document.URL.length));
```

To print the name of the authenticated user, the JavaScript extracts the information from the “URL” attribute of the “document” object. To exploit this vulnerability, attackers may craft their request URL to:

```
http://www.site.com/products.html?user=<script>...</script>
```

the malicious script is injected after the string “user=”, polluting the DOM object environment, and the content of “document.URL” at the client side in changed. When the browser tries to display the web page, it parses the JavaScript, and the malicious script is added to the html content dynamically. Attackers can craft this URL string, making it look benign, and use it to lure victims with phishing techniques.

To prevent DOM-Based XSS attacks, input validations on the server side might not work in many cases, since the code handling the user input might not reside at the server side, and attackers may try to avoid sending malicious data to the server, leaving no track for their attacks at all. For example, attackers may use certain symbols, such as “#”, in the URL string, making the

malicious content as fragment which will not arrive at the server. For example, attackers can modify the previous URL string to:

http://www.site.com/products.html#user=<script>...</script>

The content after "#" will be interpreted as a fragment by some web browsers, and the actual URL string sent to the server will be:

http://www.site.com/products.html

Since input validations on the server side might not work in this situation, XSS preventions should be performed at the client side. In the example above, certain input validation mechanisms can be added into the JavaScript code in the response page, which is responsible for extracting the URL information from the DOM object. When a browser displays the web content, certain characters are filtered out at the client side directly.

3.4 XSS Preventions

To prevent XSS attacks, many web applications adopt certain input validation mechanisms. In this section, we discuss at a high level about common injection techniques employed by attackers, and the common mechanisms to prevent them.

Web applications usually treat user inputs as data, and place them into a context which should be treated by web browsers as data only. In this thesis, we refer this kind of context as **data context**, like <textarea>data context</textarea>. For the context which is usually treated as code having certain functionalities, we refer it as **code context**, such as <script>alert('XSS')</script>.

XSS injections usually use special characters, such as quotation marks and angle brackets, to convert data contexts to code contexts, altering the original page structures. For example, an injection into a data context between the tags <textarea> and </textarea> can change the content to:

<textarea><script>alert(“XSS”)</script></textarea>

This makes the original data context contain a code context having a java script.

The most common method of switching context is closing the current context and starting a new one, which is called **injection up**. For example, a successful injection to the “userinput” in the context userinput can change it to

<script>alert(“XSS”)</script> .

This injection has closed the data context within the original “div” tag pair by dividing it to two data contexts. A new code context containing the script is generated between the two newly generated “div” tag pairs.

Another injection type is called **injection down**, which creates a sub context within the original content, without closing it. An example can be

<INPUT TYPE=”IMAGE” SRC=”javascript:alert(“XSS”);”>

where the string “javascript:alert(“XSS”);” is provided by attackers to convert the data context into a code context containing a java script.

Based on the fact that the essence of XSS attacks is converting data contexts into code

contexts, web application developers have added functionalities of input validations to their products, including some filtering and escaping mechanisms focusing on certain symbols and characters. The filtering mechanisms are used to remove suspicious content, trying to leave no chance for misinterpretations by browsers. A disadvantage of filters is that legitimate inputs can also be filtered out unexpectedly. For this reason, the widely used method is escaping certain characters, which help the browsers to treat user inputs as a part of data only. It usually encodes special symbols in the user input and considers them as part of data content, distinguishing them from the same symbols in the original html structural code. When displaying the encoded content, they are recognized by browsers and displayed in the correct format after decoding. Table 3.1 illustrates the hex entity encoding method for several symbols. The five symbols in the table are the ones escaped by many web applications, and they are also escaped in some of our test applications in controlled studies, which will be discussed in chapter 5.

Table 3.1 HTML Escape Characters

Special Symbol	After encoded
&	&amp
<	&lt
>	&gt
“	&quot
‘	&#39

To better illustrate the concept of escaping, consider the same example used to illustrate the concept of injection up. This injection has changed the content between the tags and to:

<script>alert(“XSS”)</script>

If a proper escaping mechanism is used to encode the special symbols quotation marks and angle brackets, the content after escaping will be:

</div><script>alert("XSS")</script><div>

This can help the applications to differentiate the “ ” tag in the data context and the “ ” tag in the code context. When this content is added in the response page, the original page structure is not changed, and a new code context running the script will not be generated.

CHAPTER 4

A GENERAL DETECTION FRAMEWORK

There are several scanners in the market trying to detect XSS vulnerabilities. In this chapter, we will discuss about their detection mechanisms. We will introduce the four scanning phases of many scanners, including crawling, deciding injection points, making injections, and reporting. In our in-depth comparison for injection mechanisms in Chapter 5, we focus on the phase making injections, which affects the injection effectiveness. To have an intuitive understanding of the injection mechanisms, we have a source code study of Skipfish, an open source web application scanner developed by Google. For the phases crawling and deciding injection points, since the evaluations focusing on them can be separate topics in related work, we do not cover detailed comparisons for scanning performance in these two phases in this thesis.

According to our scanning experience, most scanners use automated fuzz testing techniques. At first, scanners inject invalid inputs to certain input fields, which are referred as **injection points** in this thesis. Then, they search for certain predefined patterns in the response pages, trying to show that there is no proper sanitization on the inputs in vulnerable locations. A scanning process typically includes the following phases:

1. Crawling

2. Deciding injection points

3. Making Injections

3.1 Generate and send fuzzed data to injection points

3.2 Analyze response content

4. Classifying vulnerabilities and sending alerts

4.1 Crawling

Given a starting URL, the first step of scanning is crawling. By scanning html pages, scanners are able to explore subdirectories, forms, and links to other resources. A successful web application scanning process depends a lot on how much the scanner knows about the target website's structure. In order to detect a new resource, scanners make various efforts. For example, some scanners append crafted strings at the end of existing URLs, hoping to reach pages matching the generated URLs, or to be redirected to pages having similar URLs. In the crawling phase of Skipfish, crafted resource name such as "sfi9876" with various types, such as ".asp", ".pdf", "zip", etc, are appended to several existing URLs. For this reason, crawling efforts usually generate a large amount of attempting requests, and require scanners to deal with massive data transmission. For a large web application, it usually takes quite a long time to finish a complete and in-depth crawling process, or bring a high memory requirement for the machine using the scanner. To improve our user experience, certain configuration options can help us control the crawling process according to our needs. For example, Skipfish uses the "-d"

command option to limit the crawling depth to a specified number of subdirectories, and Netsparker has options to set a total page limit.

4.2 Deciding Injection Points

After the crawling phase, scanners identify injection points in the obtained resources as the fuzz targets. Certain parameters, URL directories, and fields in forms can be identified as injection points. For example, in a Get request with the URL in the format

http://www.sitename.com/index.php?&parameter1=string1&parameter2=string2/,

values of the parameters string1 and string2 can be replaced with fuzzed data, attack strings can be appended at the end of this URL. For a Post request submitting forms, field values within the forms can be identified as injection points. Since the number of pages resulting from the crawling phase is always huge, the effectiveness of deciding injection points, which reduces the scope for the later stages, can greatly improve the overall detection effectiveness and shorten the length of the scanning time.

4.3 Making Injections

Focusing on the injection points, fuzzed data is generated and inserted. Typically, fuzzing can be classified as mutation-based fuzzing and generation-based fuzzing. They are discussed below:

Mutation-based fuzzing: A fuzzing technique that mutates certain values in valid data without much knowledge of the context. This kind of fuzzers is also known as “dumb” fuzzers, since they do not know the details of the format or structure in the original data.

Generation-based fuzzing: A fuzzing technique that creates data values from scratch based on the knowledge of the data types and formats.

Most scanners adopt mutation-based fuzzing. For “Get” and “Post” requests, some scanners treat content in certain parameter fields as fuzz targets, such as the field values after the strings “name=”, “style=”, or “page=”. They are easy to be identified in the response check process. The fuzzed data always follow the same format for different applications being scanned, indicating that the data are fuzzed without knowing the actual contexts in different applications. For example, within an injection, Skipfish always uses the key word “sfi” for a tag attribute value, N-Stalker always uses the key word “nstalker” for a tag name. In this thesis, for every scanner, we refer the templates of attacking strings as **injection patterns**, which involve injection tag types and injection techniques.

After sending fuzzed data, the scanner checks whether or not the response pages contain any content matching the pattern of previous injections, which shows the existence of XSS vulnerabilities. In this process, scanners always check certain tags and parameter values used in the injection phase.

4.4 Reporting

As the final step, a scanner classifies the detected XSS vulnerabilities based on their severity levels and generates a report, which usually present the vulnerability descriptions, vulnerable locations, attack strings used in the successful injections, and request and response data focusing

on the vulnerable locations.

4.5 Case Study

To gain a better understanding of the injection mechanism, we illustrate the injection process of Skipfish, an open source tool developed by Google. In this section, we focus on its detection mechanism for reflected XSS vulnerabilities. Its code dealing with sending fuzzed data and response checking reside in the files Crawler.c and Analysis.c. Its injection patterns are consistent with the observation of the sniffed packets during scanning processes, which will be discussed in chapter 5. After the crawling phase, Skipfish modifies values within requests in the following ways:

a).Optionally append certain parameter fields with values in the format "-->\">'>'\"<sfi%06uv%06u>", where the two strings, "%06u", represent the current xss id and scan id, which are generated as different values in different injections. The scanner can identify XSS vulnerabilities by checking whether the response contains any tag matching the format of the injection tag;

b).Optionally change parameter fields with a value in the format like ".htaccess.aspx-->\">'>'\"<sfi%06uv%06u>", where two "%06u" represent the same values as described in step a.

In its response analysis phase, it mainly checks if there are some contents matching the following patterns:

a). A tag name is in the format "sfi%06uv%06u";

b). Within a tag, a parameter name is in the format "sfi%06uv%06u";

c). Within a “script” tag, for parameters not named with “src”(it checks for the “src” parameter in the scanning for URL redirection issue separately), the content contains tag in the format like "sfi%06uv%06u";

Since the xss id and scan id are unique for every injection, it is easy for Skipfish to identify the information of every injection during the response check process, helping classify the vulnerabilities in the reporting phase. For example, the scanner generates different alert information for the injections from the current scan and the injections from previous scans.

The above source code observation presents us a real-life example of the XSS injection mechanism in a web security scanner. The injection effectiveness depends a lot on the injection strings’ abilities of exploiting vulnerabilities after bypassing certain input validations. A powerful scanner should have attack strings which can cover contexts in various XSS attacks and have certain injection techniques. In chapter 5, we will compare the injection strings of different scanners in detail.

In this thesis, we will have a deep understanding about several scanners’ XSS detection effectiveness. We will compare their scanning performance in the phase of crawling, deciding injection points, and making injections. Since most scanners only provide an overall final report, with limited information of requests and responses focusing on the vulnerable locations detected, it is difficult for us to know more about their efforts in each scanning phase. For example, by

looking at the final scanning report, we have no way to know what efforts a scanner has made focusing on a vulnerable location if it cannot detect the vulnerability. Whether the scanner fails at reaching the location in the crawling phase, or it does not identify the vulnerable location as an injection point, or its injection strings are not effective enough to exploit the vulnerability, the final report cannot show anything. To make an in-depth analysis about the scanning performance in different phases, we sniff transmission packets for each scanner with Wireshark, observe and compare their efforts in details.

CHAPTER 5
EVALUATION

This chapter describes the details of our evaluation process. In the following sections, we will introduce the web application scanners we use, case studies of real life applications and controlled test applications. To study the reasons causing the different scanning performance, we also present our comparisons for injection mechanisms.

5.1 Web Application Scanners

5.1.1 Overview

This section introduces the web application scanners used in our project. In many cases, commercial scanners are easier to use. They have more advanced user interfaces helping control their scanning activities. In contrast, many open source tools have rudimentary configuration support, and some of them only rely on command line operations to configure their scanning processes. They usually take more time to set up. But on the other hand, open source scanners have no restriction for users to access their technical details, which is helpful for further studying their detection mechanisms.

Our experiments mainly focus on four scanners. Netsparker community edition and Acunetix free edition are commercial, and Skipfish and Wapiti are open source. According to the evaluation results from Shay Chen in his blog posts [3], which have been discussed in the related work section, NetSarker, Acunetix, and Skipfish have relatively low false positive rates.

For each scanner, we discuss its major features, such as user interface, reporting module, and the session maintenance mechanism, which maintains the login state, allowing the scanner to reach more web resources. They are major factors affecting our scanning experience.

5.1.2 Netsparker Community Edition

The edition we use is Netsparker community edition, version 1.7.2.13. It is a commercial scanner claimed to be free from false-positives, as described in the product website. It shares the same user interface with the professional edition. To perform automatic authentications, Netsparker allows users to use cookie strings of authenticated sessions. In our evaluation, cookie information is obtained by the network tamper tool, i.e., the tamper data plug-in of Firefox, which can help view and modify the contents in request headers and parameters. To maintain the authenticated session status, Netsparker allows users to specify the key words that should be included or excluded in the web pages being scanned, and users can use this method to detect and avoid logout pages. This feature is quite useful, since it is often that the session has a logout state when a logout page is visited, and many web pages cannot be reached afterwards. Although several advanced reporting functionalities are disabled in this free version, it still provides sufficient information, such as the severity type, background description, request and response content focusing on the reported locations, and attack strings used to exploit the vulnerabilities. The crawling results of the target websites can also be viewed in the report page.

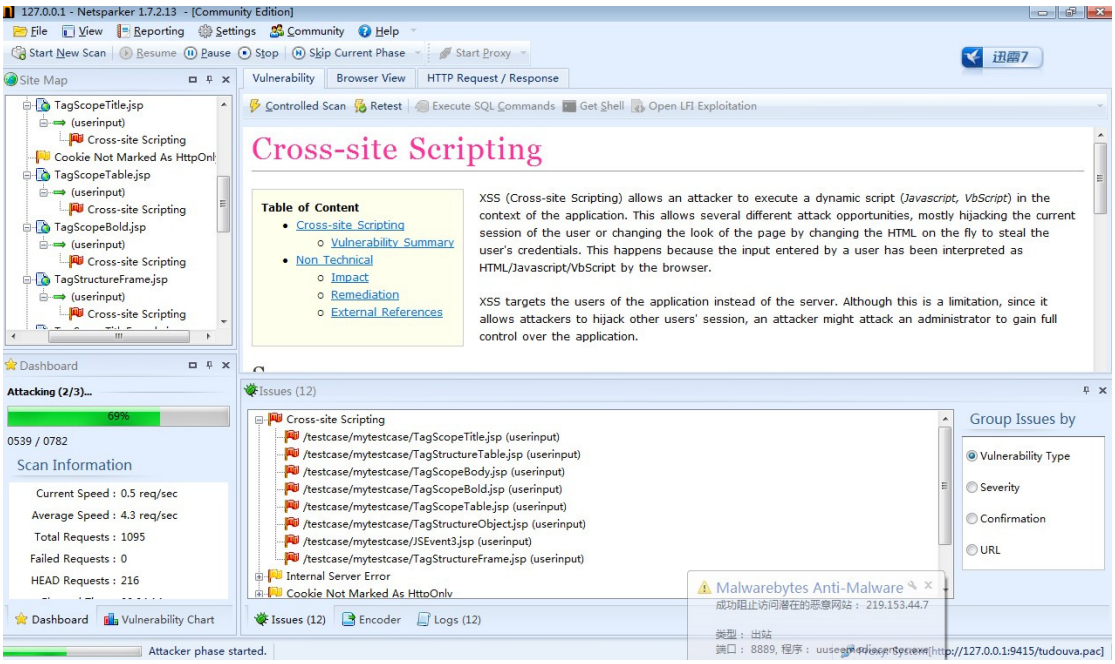


Figure 5.1 Netsparker GUI

5.1.3 Acunetix Web Vulnerability Scanner Free Edition

Acunetix free edition is another free scanner without any period limitation, and the version we use is 7.0. It has an advanced graphic user interface. To perform the login operations automatically, Acunetix has a recorder with a mini browser to record the users' logging actions, including the URLs visited, the password entered, etc. The scanner is able to retrieve the recorded information, which is referred as login sequence, to perform automatic authentications at later scanning phases. A screen shot of the login sequence recorder is presented in Figure 5.2. To maintain a valid session status for reaching more web pages, the recorder allows users to specify key words indicating whether the session is in the login state or logout state. In this way, the scanner is able to avoid pages with the specified key words and avoid unexpected status changes.

After the crawling phase, Acunetix allows users to choose which web resources should be included or excluded in later scanning phases.

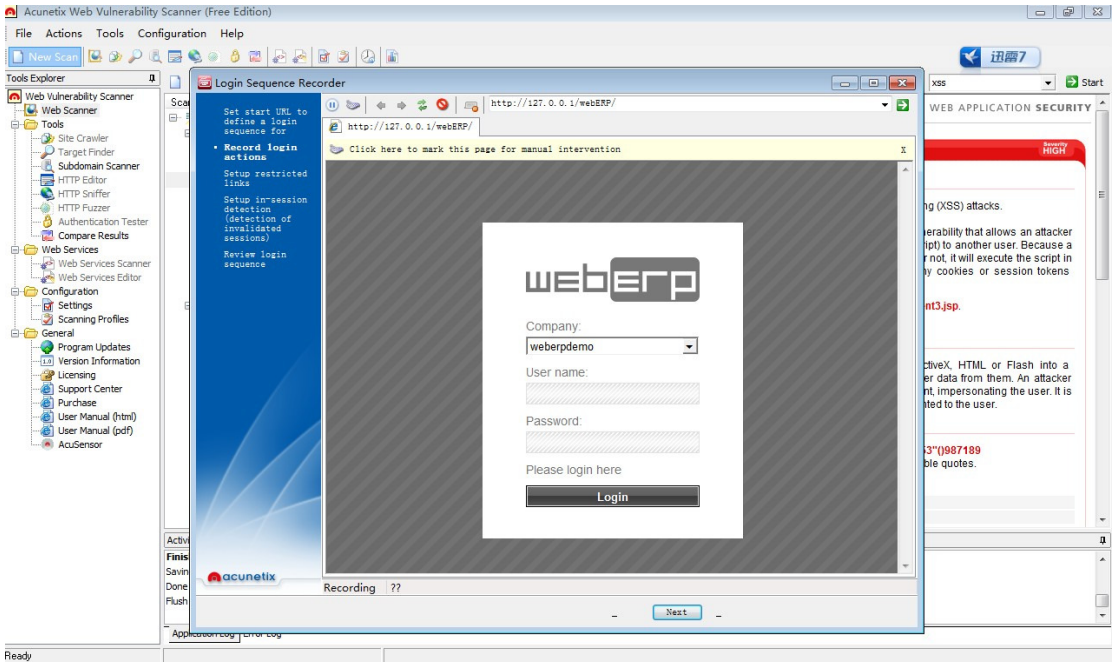


Figure 5.2 Acunetix Login Sequence Recorder

In its final report, users can see descriptions of vulnerabilities and attack strings. But it does not provide the details of the requests and responses related to the reported issues.

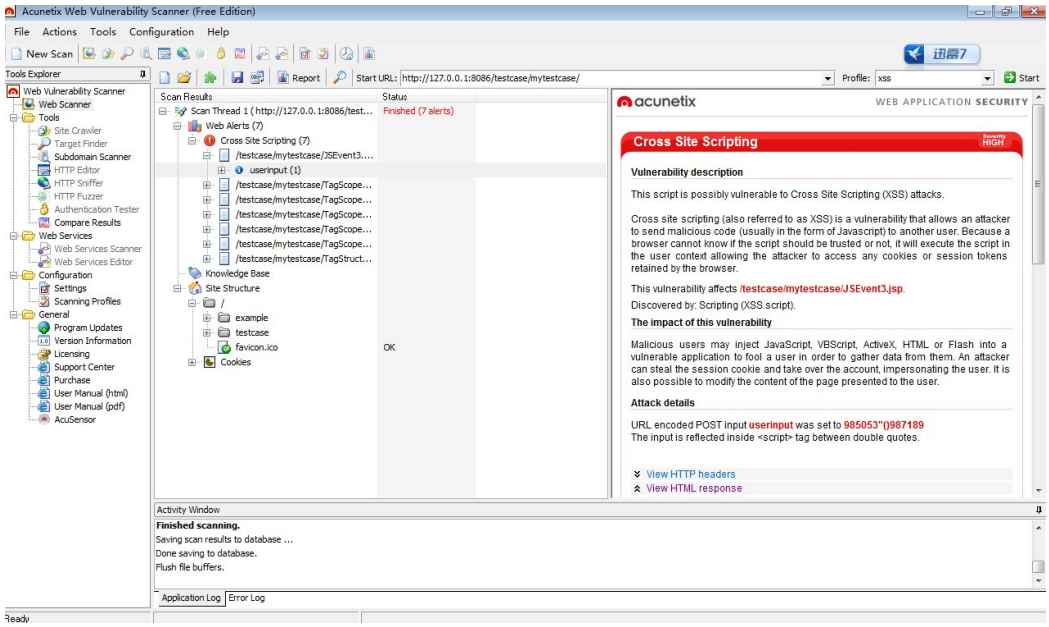


Figure 5.3 Acunetix Test Result

5.1.4 Skipfish

Skipfish is an open source scanner written in C, and it need to be compiled and run in the Linux environment. The version we use is 1.84b. According to the descriptions on the project website, it is highly efficient at maintaining a network transmission rate of more than 200 requests per second. This allows extensive brute force functionalities in its crawling process. For example, in order to reach more resources in a particular folder, users can specify numerous guessed file names in a wordlist, which can be configured before scanning. The high request transmission rate can help the scanner try every entry in the wordlist in a brute force way. For this reason, it is

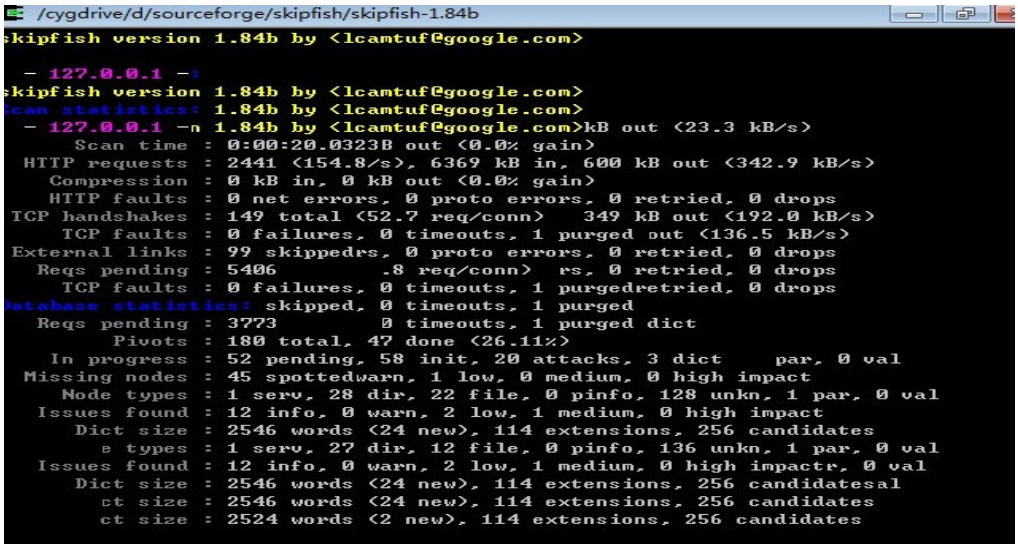


Figure 5.4 Skipfish Command Line Interface

important for users to learn various crawling configuration options and choose their own scanning strategies, such as picking a suitable wordlist. It may take a very long time to finish a complete crawling process if all extensive crawling configurations are enabled.Skipfish only has command line mode to configure scanning. To perform automatic authentications, it uses a “–C” option to append cookie strings to the request content. Our evaluation experience shows that this method can help bypass certain authentication pages, but the performance is not as good as that of some commercial scanners. The report is generated in html format, including the information of the vulnerable locations, vulnerability descriptions, and the request and response data used to indentify vulnerabilities.

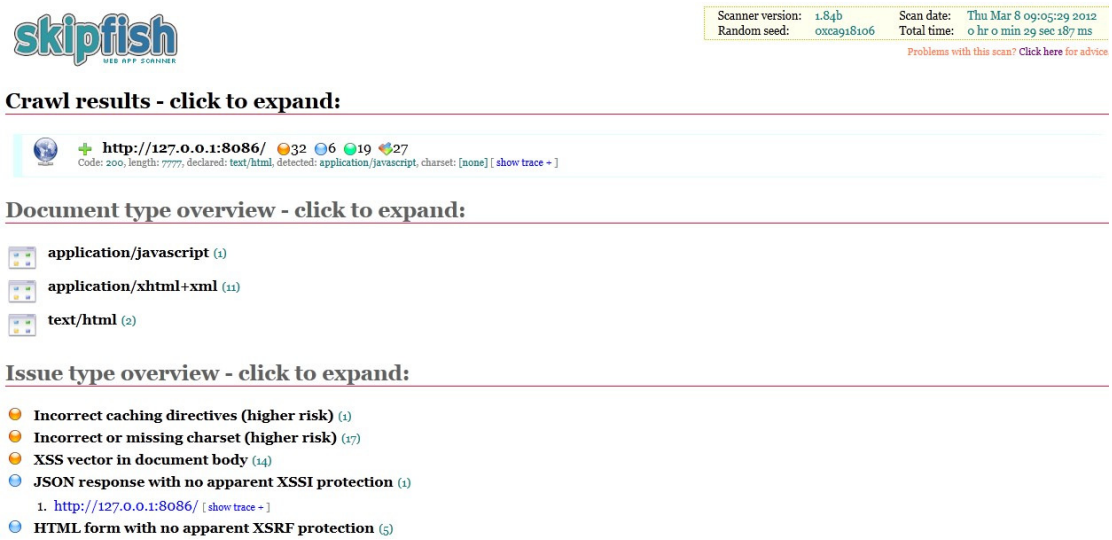
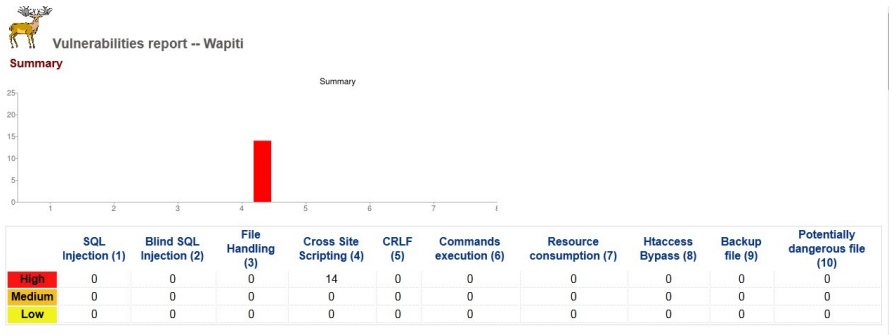


Figure 5.5 Skipfish Report

5.1.5 Wapiti

Wapiti is an open source scanner written in Python, and the version we use is 2.2.1. Similar to Skipfish, it only uses a command line interface to configure its scanning processes. To perform automatic authentications, it has programs to generate cookie files according to the login page URLs and credential data provided by the users. Different from cookie strings which are used directly by many scanners, the cookie files embed the cookie information in their text content, and the scanner can obtain the information from the files to perform authentication activities. Our evaluation experience shows that it cannot bypass authentication web pages in several scanning runs. The scanner also has commands to exclude certain URLs during the crawling phase, avoiding logout pages.



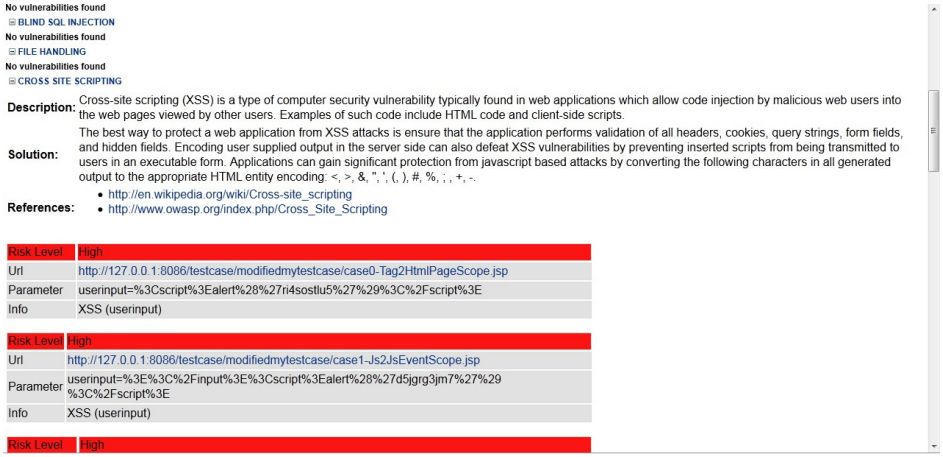
Vulnerabilities report -- Wapiti

Summary

	SQL Injection (1)	Blind SQL Injection (2)	File Handling (3)	Cross Site Scripting (4)	CRLF (5)	Commands execution (6)	Resource consumption (7)	Htaccess Bypass (8)	Backup file (9)	Potentially dangerous file (10)
High	0	0	0	14	0	0	0	0	0	0
Medium	0	0	0	0	0	0	0	0	0	0
Low	0	0	0	0	0	0	0	0	0	0

Figure 5.6 Wapiti Report Overview

.



No vulnerabilities found

BLIND SQL INJECTION

No vulnerabilities found

FILE HANDLING

No vulnerabilities found

CROSS SITE SCRIPTING

Description: Cross-site scripting (XSS) is a type of computer security vulnerability typically found in web applications which allow code injection by malicious web users into the web pages viewed by other users. Examples of such code include HTML code and client-side scripts.

Solution: The best way to protect a web application from XSS attacks is ensure that the application performs validation of all headers, cookies, query strings, form fields, and hidden fields. Encoding user supplied output in the server side can also defeat XSS vulnerabilities by preventing inserted scripts from being transmitted to users in an executable form. Applications can gain significant protection from javascript based attacks by converting the following characters in all generated output to the appropriate HTML entity encoding: <, >, &, ", ', (,), #, %, ; , +, -.

References:
- http://en.wikipedia.org/wiki/Cross-site_scripting
- http://www.owasp.org/index.php/Cross_Site_Scripting

Risk Level	High
Url	http://127.0.0.1:8086/testcase/modifiedmytestcase/case0-Tag2HtmlPageScope.jsp
Parameter	userinput=%3Cscript%3Ealert%28%27ri4sostlu5%27%29%3C%2Fscript%3E
Info	XSS (userinput)

Risk Level	High
Url	http://127.0.0.1:8086/testcase/modifiedmytestcase/case1-Js2JsEventScope.jsp
Parameter	userinput=%3E%3C%2Finput%3E%3Cscript%3Ealert%28%27d5jgrg3jm7%27%29 %3C%2Fscript%3E
Info	XSS (userinput)

Risk Level	High

Figure 5.7 Wapiti Report In Detail

The reports are generated as html pages, which present brief vulnerability descriptions, vulnerable locations and parameters, and attack strings. They do not have detailed information about the crawling results, such as detailed content in requests and responses focusing on the vulnerable locations, vulnerability classifications in deeper levels, and more detailed descriptions, etc.

5.1.6 Summary

Table 5.1 gives a comparison for various usability-related features of the scanners mentioned above.

Table 5.1 Usability Comparison for 4 scanners

		Netsparker CE	Acunetix free edition	Skipfish	Wapiti
Overall Usage	GUI	Yes	Yes	No	No
Crawling	Stop after Crawling	No (disabled)	Yes	No	No
	Exclude URL	Yes	Yes	Yes	Yes
Session Maintenance	Login method	Cookie	Login Sequence	Cookie	Cookie File
	Exclude Logout	Yes	Yes	Yes	Yes
Reporting	Show crawl result	Yes	Yes	Yes	No
	Severity classification	Yes	Yes	Yes	Yes
	Request& Response detail	Yes	No	Yes	No
	Attack pattern	Yes	Yes	Yes	Yes

5.1.7 Other Tools

In the evaluation for injection patterns, we used some other tools, including IBM Rational AppScan version 7.8 trial edition, NStalker free edition 2012, and Nikto version 2.1.4. IBM AppScan is quite powerful at detecting various types of vulnerabilities, but its trial edition has a 30 days period limitation and can only scan applications deployed on the test websites specified by the vendor. NStalker does not support session maintenance in its free edition. Nikto has many

features for detecting problems on the server side, but XSS detection is not its specialty. For these reasons, we do not evaluate these scanners in our case studies of real-life web applications.

5.2 Case Studies

5.2.1 Subject Applications

In our case studies of real-life web applications, we select several vulnerable web applications reported to have XSS vulnerabilities in the Open Source Vulnerability Database (OSVDB), which is online database providing information about security vulnerabilities of various web applications in the market. It is created and maintained by the web security community. Companies and individuals can better collaborate with each other through their postings. We believe that its reported vulnerabilities can represent various real-life security issues. The reasons are the following. First, the database is aimed at providing unbiased and most current security postings, and the information is independent from product vendors. Second, the database covers over 80,000 vulnerabilities, spanning over 40,000 products from over 4,000 researchers, according to the description in the website. To compose a group of real-life test applications, according to the information in several XSS issue postings in OSVDB, we randomly pick 5 vulnerable open source products belonging to different categories, with the versions reported to have vulnerabilities. Some statistics about them are shown in table 5.2.

eFront is an online learning application which allows administrators to create and edit their studying materials and share with others. Users can use it to create and upload online tests, manage uploaded content such as text files, videos, and flashes. It can also help users

communicate with others through messages and forums within the site.

webERP is an open source accounting and business management web application written in PHP. It can manage company invoices, orders, and tickets, analyze market sales, and provide strategic decision supports. It has hundreds of PHP files, and XSS vulnerabilities in a number of files have been reported.

Table 5.2 Statistics of Subject Applications

	Version	Size on Disk	Number of Files	Number of Lines of Code	Reported Issue IDs
eFront	3.6.9	23MB	818	62328	76745,76750
webERP	4.0.2	65MB	534	193492	77194,72048
Kaibb	2.0.1	500KB	66	6652	76603,76604
OsCommerce	2.2	2MB	529	31968	79330,78619
Orangehrm	2.5.0.6	10MB	570	160794	77416,77417,71286

Kaibb is a small but comprehensive bulletin board application written in PHP. Its functionalities include login authentication and message board discussion.

OsCommerce is an e-commerce solution used by thousands of shop owners, application developers, and service providers. It has typical shopping cart functionalities such as searching and browsing product information, making and tracking orders, and managing personal accounts. It allows site administrators to employ a MySQL database storing information of products, manufacturers, customer accounts, and comment messages.

OrangeHRM is a human resource management application written in PHP. According to the information in its official website, OrangeHRM has been claimed to be the world's most popular open source human resource management software. It can help users to manage employee information, and evaluate employee performance.

A reported issue usually provides the information of product version, publish and disclosure date, vulnerability descriptions, and available solutions if there are some. A summary of OSVDB reported issues about our 5 real-life vulnerable applications are listed in table 5.3, where each issue is identified with a unique OSVDB ID.

Table 5.3 Vulnerabilities Overview

OSVDB 76745	Located at index.php of efront Does not validate parameter ‘course’ and ‘message_type’ Allows user to craft an URL by appending an attack string to the end.
OSVDB 76750	Located at administrator.php of efront Does not validate parameter ‘ctg’,’user’, ‘view_calendar’
OSVDB 77194	Allows user to craft an URL by appending an attack string to the end. This type of vulnerability can be found in various locations.
OSVDB 72048	Located at AccountGroups.php Does not validate parameter ‘CompanynameField’
OSVDB 76603	Located at index.php of Kaibb does not validate the 'Referer' HTTP header
OSVDB 76604	Located at acp/index.php of Kaibb does not validate the 'Referer' HTTP header
OSVDB 79330	Located at index.php Does not validate parameter “Cart”
OSVDB 78619	Allows user to craft URL by appending to the end in various locations.
OSVDB 77416	Located at index.php Does not validate parameter 'uniqcode' and 'isAdmin'
OSVDB 77417	Located at lib/controllers/CentralController.php Allows user to craft URL by appending to the end.
OSVDB 71286	Located at templates/recruitment/jobVacancy.php Does not validate parameter 'recruitcode'

5.2.2 Evaluation Method

For every case study application, our evaluation uses scanners Acunetix, Netsparker, Wapiti, and Skipfish. For each scanner, we use its default configuration for XSS detection. In the crawling phase, all applications require session maintenance mechanisms, and some applications are very complex to crawl. To have better scanning results, for a particular scanner and application

combination, we run the scanning for several times, and the results of multiple runs are not exact the same, as there might be differences at maintaining the session status for different runs. During every scanning run, we use Wireshark[28], a well-known network traffic analyzer, to capture transmission packets, and the packet data are used in further analysis about scanning performance. Since many scanning processes involve thousands or millions of request and response packets, we create several small programs to analyze the packet text files exported from Wireshark.

5.2.3 Results and Discussion

We divide the OSVDB reported issues described in table 5.3 into the following three types:

1) The reported issue does not provide specific vulnerable URLs, files, or parameters, only gives a general description.

2) The reported issue gives information of specific vulnerable URLs and files, which lack proper input validations for particular parameters.

3) The reported issue gives information of vulnerable URLs, where the attacking strings can be appended to generate attacks.

In the following paragraphs, we will discuss the scanning performance of the four scanners. First, we will compare their scanning results for the OSVDB type 1 issues. Since this type of issue only provides a general description for multiple vulnerable locations, for all problems detected by

scanners matching the issue description, we will validate false positives. Then, we will give each scanner a table to present its detection results for the OSVDB type 2 and type 3 issues, making an in-depth comparison about the performance in different scanning phases. Next, we will compare detection performance for the issues not reported in OSVDB website, but detected by our scanners. The validation of false positives for them is included. At last, we will have a discussion for their overall performance and draw our conclusions.

In our evaluation, the OSVDB type 1 issues include the issue with OSVDB ID 77194 from the application webERP and the issue with OSVDB ID 78619 from the application OsCommerce. We use table 5.4 to show the detection performance of the four scanners. It is possible that for a reported type 1 issue, during a scanning run, there are too many detected vulnerable locations for us to validate the false positives. We only select a few of them for validation. For example, in the situation that dozens of vulnerable locations are reported to have the vulnerability matching the description of a type 1 issue, we only select the top 10 locations, according to the time being detected in the scanning report. If there are over hundreds of vulnerable locations reported, we select 15 of them. To validate the vulnerabilities in these locations, we try to perform actual XSS attacks using the attacking strings provided by the scanners. For some large applications, the scanning results for different scanning runs of the same scanner might not always be the same, as the scanner might have different performance in session maintenance every time. For this situation, Table 5.4 shows an approximately average number of the detected issues.

Table 5.4 Type 1 Issues Scanning Results

	OSVDB 77194	OSVDB 78619
Acunetix	XSS issues are reported at around 50 locations in average. There is 0 false positives out of 10 selected issues	XSS issues are reported in 31 positions of 17 files. There is 0 false positive out of 10 selected issues
Netsparker	It can detect hundreds of related XSS issues. There is 0 false positive out of 15 selected issues	XSS are reported at around 50 locations in average. There is 0 false positive out of 10 selected issues
Skipfish	It cannot detect any XSS issue	Various XSS issues are at 15 locations.
Wapiti	Failed at session authentication	No XSS issues reported

We can see that Netsparker and Acunetix detected more vulnerabilities than Skipfish. Wapiti failed at bypassing the login phase for the application webERP, and could not find any XSS issue for the application OsCommerce.

For the type 2 and type 3 issues, which provide specific vulnerable locations, we summarize the scanning results of each scanner in a separate table. Since all issues are reported in OSVDB website, we do not validate false positives for them. For each vulnerable location or parameter, we use the term **attack level** to describe how many scanning phases the scanner has accomplished focusing on it. The descriptions of attack level values are listed below:

1. For the situation that the reported URL or parameter does not appear in any request header, we conclude that the scanner has failed in the crawling phase, and the attack level is marked as 0;

2. For the situation that there are requests making the reported vulnerable location or parameter as a request target, but no attack strings have been used for injections, we conclude that the scanner has failed at the phase of deciding injection points, and the attack level is marked as 1;
3. For the situation that the scanner has sent fuzzed data to the vulnerable location or parameter, but no vulnerability is reported, we conclude that the scanner has failed at the injection phase, and the attack level is marked as 2;
4. For the situation that the vulnerability has been correctly reported, we mark the attack level as 3, and list the attack string used in the successful injection.

In each table, for the situations with attack level value less than 3, since there is no successful injection, we do not list the attack strings used by scanners.

Table 5.5 Acunetix Scanning Result

Vulnerability Location	Attack Level	Attack pattern
OSVDB 76745 at “message_type”	3	“onmouseover=prompt(983420) bad=”
OSVDB 76745 at “course”	3	“onmouseover=prompt(983420) bad=”
OSVDB 76745 URL appending	2	
OSVDB 76750 at “ctg”	1	
OSVDB 76750 at “user”	1	
OSVDB 76750 at “view-calendar”	1	
OSVDB 72048	0	
OSVDB 76603	1	
OSVDB 76604	0	
OSVDB 79330	0	
OSVDB 77416 at 'isAdmin'	3	“onmouseover=prompt(927004) bad="
OSVDB 77416 at 'uniqcode'	3	“onmouseover=prompt(9540484) bad="
OSVDB 77417	1	
OSVDB 71286	0	

Table 5.6 Netsparker Scanning Result

	Attack Level	Attack pattern
OSVDB 76745 at “message_type”	1	
OSVDB 76745 at “course”	1	
OSVDB 76745 URL appending	3	/index.php/’style=’x:expre/**/ssion(alert(9))
OSVDB 76750 at “ctg”	3	/administrator.php?ctg=’stYle=x;expre/**/ssion(alert(9))’
OSVDB 76750 at “user”	3	/administrator.php?user=’stYle=x;expre/**/ssion(alert(9))
OSVDB 76750 at “view-calendar”	2	
OSVDB 72048	1	
OSVDB 76603	1	
OSVDB 76604	0	
OSVDB 79330	2	
OSVDB 77416 at 'isAdmin'	1	
OSVDB 77416 at 'uniqcode'	3	'"--></style></script><script>alert(0x005124)</script>,
OSVDB 77417	2	
OSVDB 71286	0	

Table 5.7 Skipfish Scanning Result

	Attack Level	Attack pattern
OSVDB 76745 at “message_type”	1	
OSVDB 76745 at “course”	2	
OSVDB 76745 URL appending	3	/index.php/.htaccess.aspx-->">'>'"<sfi000023v821211>”;
OSVDB 76750 at “ctg”	1	
OSVDB 76750 at “user”	1	
OSVDB 76750 at “view-calendar”	1	
OSVDB 72048	1	
OSVDB 76603	1	
OSVDB 76604	0	
OSVDB 79330	0	
OSVDB 77416 at 'isAdmin'	1	
OSVDB 77416 at 'uniqcode'	2	
OSVDB 77417	1	
OSVDB 71286	0	

Table 5.8 Wapiti Scanning Result

	Attack Level	Attack pattern
OSVDB 76745 at “message_type”	0	
OSVDB 76745 at “course”	2	
OSVDB 76745 URL appending	1	
OSVDB 76750 at “ctg”	0	
OSVDB 76750 at “user”	0	
OSVDB 76750 at “view-calendar”	0	
OSVDB 72048	0	Failed at session authentication
OSVDB 76603	0	
OSVDB 76604	0	
OSVDB 79330	0	
OSVDB 77416 at 'isAdmin'	0	Failed at session authentication
OSVDB 77416 at 'uniqcode'	0	Failed at session authentication
OSVDB 77417	0	Failed at session authentication
OSVDB 71286	0	Failed at session authentication

Among the above 4 scanners, Wapiti can only detect very few XSS issues, and its run time is relatively short. By observing the packets transmitted, we find that Wapiti fails to perform auto authentications in several scanning runs, since it can only reach the login page and some related resources in the crawling phase for several applications. For the other three scanners, i.e., Acunetix, Netsparker, and Skipfish, each of them can detect several XSS issues. Table 5.9 shows the statistics of the scanners’ performance for the 14 vulnerable locations in total. We count the number of locations where scanners have achieved attack levels no less than 1, 2, and 3

respectively.

Table 5.9 OSVDB Issues Scanning Statistics

	Attack Level>=1	Attack Level>=2	Attack Level>=3
Acunetix	10	5	4
Netsparker	12	7	4
Skipfish	11	3	1

For the OSVDB type 2 and type 3 issues, we have following observations for the performance of these three scanners

a. Three scanners have similar performance during the crawling phase, since they all can reach more than 10 out of a total of 14 vulnerable locations.
b. The commercial scanners Acunetix and Netsparker have better performance in the phase of deciding injection points. Among the vulnerable locations they reached in the crawling phase, they both choose at least half of them as the targets to make injections. Skipfish can only decide 3 out of 10 injection points.
c. Acunetix and Netsparker detect more vulnerabilities, and they have better injection effectiveness. Acunetix identifies 4 vulnerabilities based on 5 injection points, and Netsparker identifies 4 vulnerabilities based on 7 injection points. Skipfish can only identify 1 vulnerability based on 3 injection points.

Apart from the XSS issues reported in OSVDB website, scanners have detected several

other XSS issues which are not reported before. We use table 5.10 to present the comparison focusing on the detection performance for them. For each application, we give the total number of issues detected by different scanners respectively, and the false positive rates. To identify false positives, we try to make benign XSS attacks according to the attack string information in report descriptions. If our attacks are successful, the reported issues are identified as true positives; otherwise, if we find input sanitization mechanisms focusing on the reported locations in our source code study, and the injection within response cannot cause actual XSS attacks as expected, we identify the reported issues as false positives. For example, an injection can cause the response page contain a database error message injected with part of attack string content. The scanner recognizes this content and generates a warning. But actually, the original attack string has been sanitized, having certain special symbols escaped, and the remaining content in the response page cannot cause actual XSS attacks. For these situations, we identify the reported issue as false positive.

Table 5.10 Non OSVDB Issues Scanning Statistics

	Acunetix		Netsparker		Skipfish		Wapiti	
	Count	FP Rate	Count	FP Rate	Count	FP Rate	Count	FP Rate
eFront	0		0		0		0	
webERP	0		3	0%	0		0	
Kaibb	1	100%	1	100%	0		0	
Oscommerce	0		0		0		0	
OrangHRM	6	0%	2	100%	0		0	

5.2.4 Summary

For the five web applications we studied, the commercial scanners Netsparker and Acunetix have better performance in all phases except crawling, and there is no significant difference between them two. The factors affecting a scanner's detection effectiveness can be found in various scanning phases. In the crawling phase, except for the difference of the crawling algorithm, whether it is able to successfully manage the session status is important. In the injection phase, the ability of choosing appropriate injection points and fuzzed data is important. In our evaluation with the test applications created by ourselves, which will be described in the next section, we will have a more detailed comparison for the performance in phases after crawling.

5.3 Evaluation with Controlled Test Applications

As we have seen in the case studies of real-life web applications, except for the crawling phase, the scanners have different performance in the phase of deciding injection points and making injections, which significantly impact their overall XSS detection effectiveness. In this section, we evaluate our scanners with test applications created by ourselves. Most of our applications are created based on the work of Shay Chen, as discussed in the related work section, and we have made some improvements. Since each application has only one PHP file, all scanners do not have significant difference in the crawling phase. For this reason, we are able to focus on the performance excluding the crawling phase. In our test applications, we are trying to embody various contexts where XSS vulnerabilities can reside, and several simple escaping mechanisms, which are often used in many web applications for sanitizing user input. In this

section, at first, we identify the significance of the evaluation with controlled test applications. Next, we introduce our test applications. At last, we present our scanning results and discuss about them in detail.

5.3.1 Motivation

The evaluation using real-life vulnerable web applications has the advantage of providing practical scanning experience, but it has certain limitations as well. First, the vulnerabilities of selected applications might not be complete enough, since it is difficult for them to cover various XSS vulnerabilities. Second, when we are trying to analyze the injection effectiveness only, it is very difficult to isolate it from other factors in the crawling process, such as session maintenance, since correlations among various files in large real-life applications are complicated and achieving consistent session maintenance after login operations might be difficult for some scanners. To have a more complete, convincing evaluation about the injection effectiveness, we need to have a series of test applications which are simple enough to eliminate the influence of crawling, and complete enough to cover various XSS vulnerability scenarios.

5.3.2 Evaluation Method

Our test applications are written in JSP. Since we are trying to use them to represent various XSS vulnerabilities in different contexts, each of them only achieves a simple and common use case of real life applications, such as page login, message posting, etc. To simulate the web applications in real life and embody the concept of defense levels, which has been discussed in

the related work section, we add to some applications some escaping mechanisms for special symbols, like what we usually see in many real-life applications. In this section, we will first discuss about the XSS vulnerability contexts that our test applications are designed to cover, and then introduce the escaping techniques used by some of the scanning applications.

Most XSS attacks rely on injections in web pages containing various html tags. Malicious input can be injected into different HTML contexts, such as within the contents of tag attributes, between the two tags of a pair, etc. According to the relationship between the injection content and existed HTML tags, we classify the XSS issues as tag scope issues and tag structure issues, and they are discussed in the following paragraphs.

In a tag scope issue, the input is reflected back in the content between a pair of html tags. For example, in the following html code, the malicious input, which is marked in bold font, is injected between the tags <textarea> and </textarea>:

<textarea**> <script>alert("XSS")</script>**</textarea>

In our test applications, in order to embody the tag scope issues of various tags, we choose to create applications having XSS vulnerabilities in the tag scopes of <body>, <b>, <table>, <textarea>, and <title>. One reason for choosing these 5 tags is that they are good representations of various html tags. For example, the tags <body> and <title> can represent tags helping lay out page structures, since the <body> tag defines the scope of the "body" part in an

html page, and the <title> tag defines the title content; the <b> tag is an example of the tag modifying attributes for page content, since it can change the font to be bold for the content within its scope; the tags <table> and <textarea> represent tags used to define page elements, which are optional in web content. Another reason for choosing them is that examples exploiting the vulnerabilities related to these tags can be easily found in related XSS attack introduction documents, such as XSS cheat sheet from ha.ckers.org.

Different from tag scope issues, in a tag structural issue, injections are in the value of a tag attribute, which usually reside in the tag content within an angle bracket pair. For example in the following code, the user input, which is displayed in bold font, is injected into the value of the “src” attribute in the ‘img” tag:

<img src=”**userinput**”>

Our test applications use tag <Frame>, <Object>, <Table>, and <img> to cover several tag structure issues, in which the injected content is in the attribute content. One reason for choosing them is that several examples of XSS injections in these 4 tags can be found in the XSS introduction document, XSS Cheat Sheet [20].

There are some tag structure issues in which the malicious data is reflected back in dynamic content, such as javascript or event-handler code. Some examples are listed in the following code, where the reflected user input is marked in bold font:

Within JS event: <script>alert(‘**userinput**’)</script>

Within JS expression: <script>string x=’**userinput**’</script>

Event-handler code: <body onload=”x=’**userinput**’”>

VB script: <script type=”text/vbscript”>alert(‘**userinput**’)</script>

To cover this kind of issue, we create test applications in which inputs can be reflected back within Java scripts, VB scripts, and other event handler code.

Except for the situation that malicious input is reflected back in the functional part of a page which actually organize page content or implement certain functionalities, user inputs can also be reflected back in comments and exception messages. For example, user input can be found within the following comment:

HTML Comment: “<!—“ +**userinput** + “-->”

If the user input is not properly validated, an attacker may make injections to jump out of the comment context. For example, he might change his input to “--><script>alert('xss')</script><!—“, and the script will be separated from the comment area and be executed.

Apart from implementing various XSS vulnerability contexts, some of our test applications adopt simple input validation mechanisms. An effective scanner should be able to exploit the vulnerabilities after bypassing these validation mechanisms. According to the work by Fong[7],

which has been discussed in the related work section, most scanners cannot detect vulnerabilities in applications with advanced defense mechanisms. For this reason, our test applications only have simple escaping mechanisms. We do not consider escaping all the special characters that have been handled in several web security libraries, such as ESAPI of OWASP project and Microsoft Anti Cross-Site Scripting Library. We add simple escaping functions for symbols <, >, ‘, “, and &. The reason of choosing them is that they are described as “5 characters significant in XML”, in the XSS Prevention Cheat Sheet provided by OWASP project [19]. As we will see in the result and discussion section, adopting the escaping mechanisms about these 5 symbols is helpful for distinguishing scanners’ performance in bypassing validation mechanisms.

To help understand the descriptions for our test applications in this thesis, we define a few terms below:

Scenario: Describe situations having a particular type of context where XSS vulnerabilities reside. A scenario is involved with a type of context, such as comments, tag scope, or tag structure.

Test Case(s): A group of test application(s) focusing on a particular scenario.

Base Test Application(s): Test application(s) about the simplest situation of a scenario where no escaping mechanism is implemented.

Table 5.11 Controlled Test Applications Overview

Scenarios	Description	Variations
Scenario 0	Input is reflected back in page content not involved with any tag content.	No variation
Scenario 1	HTML tag JS event for button input type, input is integrated into event code.	1,2,3,4
Scenario 2	Input is reflected back in VB event code within an html tag.	1,2,3,4
Scenario 3	Input is assigned to a variable which is reflected in Javascript context within tag attribute.	1,2,3,4
Scenario 4	Input is assigned to a variable which is reflected in Javascript content.	1,2,3,4
Scenario 5	Input reflected in Javascript content directly.	1,2,3,4
Scenario 6	Input is reflected in the <script> a tag attribute value.	1,2,3,4
Scenario 7	Input reflected in VB script context within a tag attribute.	1,3,4
Scenario 8	Input reflected in VB script content.	1,3,4
Scenario 9	Input is reflected in javascript comment. Have cases for single line comment and double line comment.	No variation
Scenario 10	Input is reflected in VB script comment.	No variation
Scenario 11	Input is reflected in html comment.	No variation
Scenario 12	Input is reflected in exception message.	No variation
Scenario 13	Have test applications for tag scope issues, including applications about tags <body>, <bold>, <table>, <textarea>, and <title>.	No variation
Scenario 14	Have test applications for tag structure issues, focusing on tags <frame>, <table>, <img>, and <object>	1,2,3,4 only for <img> and <object>

Variation Test Application(s): A group of test applications derived from a base test application by adding different escaping mechanisms, which usually has 4 situations:

1. Escaping “<”, “>”, and single quotation marks,
2. Escaping “<”, “>”, and double quotation marks,
3. Escaping “<”, “>”, and both quotations marks,
4. Escaping “<”, “>”, “&”, and both quotation marks.

All scenarios are described in table 5.11: For each scenario, we give a brief description. The numbers in the “Variations” column lists the types of the variation test applications that are used, corresponded to the 4 variation test application types described above.

5.3.3 Result and Discussion

We use Acunetix, Netsparker, Skipfish, and wapiti to scan the test applications. Since the structure of these test applications are quite simple, and we do not have to perform auto authentications and maintain the session statuses, all scanners are able to crawl them successfully. Since all scanners report at most one issue for each small test application, which is the only XSS vulnerability the application has, there is no false positive for all scanners. Table 5.12 shows the scanning results for the four tools.

Table 5.12 Controlled Test Applications Scanning Results

	Acunetix		Netsparker		Skipfish		Wapiti	
	Base	Variations	Base	Variations	Base	Variations	Base	Variations
Application 0	Y		Y		Y		Y	
Application 1	Y		Y		Y	1/4	Y	
Application 2	Y		Y		N		Y	
Application 3	Y	3/4	Y		Y		Y	
Application 4	Y	3/4	Y	3/4	Y	1/4	N	
Application 5	Y		Y		N		N	
Application 6	Y	2/4	Y	2/4	N		Y	
Application 7	Y		Y		N		Y	
Application 8	Y	2/3	Y	2/3	N		N	
Application 9A	Y		N		N		N	
Application 9B	Y		N		N		N	
Application 10	Y		N		Y		N	
Application 11	Y		Y		Y		N	
Application 12	Y		Y		Y		N	
Application 13A	Y		Y		N		Y	
Application 13B	Y		Y		N		Y	
Application 13C	Y		Y		N		Y	
Application 13D	Y		Y		N		N	
Application 13E	Y		Y		Y		Y	
Application 14A	N		Y		Y		Y	

Table 5.12 - Continued

Application 14B	Y		Y		N		Y	
Application 14C	Y	2/4	Y	3/4	N	1/4	Y	
Application 14D	Y	2/4	Y	2/4	N	1/4	Y	
Base Tests	22/23		20/23		9/23		14/23	
Variation Tests	14/38		12/38		4/38		None	

For applications without corresponding variation test applications, we use Y or N to indicate whether their vulnerabilities are detected by the scanner or not; for each test application having variation test applications, we add a column in the n/m format to show that n out of its m variation test applications are reported to have XSS vulnerabilities. In the last two rows, we count the total number of base test applications and variation test applications detected by different scanners respectively. We use the format n/m to identify that n out of m applications have been reported to have vulnerabilities. From the table, we have following observations:

a. Acunetix and Netsparker have better performance than skipfish and wapiti. They both can detect the vulnerabilities in most base test applications, and some variation test applications.

b. Among Acunetix and Netsparker, Acunetix has slightly better performance in scanning for both base applications and variation applications.

c. Among Skipfish and Wapiti, Skipfish can detect more vulnerabilities in variation applications, but Wapiti can detect more vulnerabilities in base applications.

5.4 Further Comparison

From the case studies of real-life web applications, we found that the performance in the phase of making injections is affecting the overall evaluation result. To better understand the detection performance of various scanners, we used test applications created by ourselves to get rid of the influence of crawling and the incompleteness of real life applications. In this section, we conduct an in-depth study about the injection mechanisms of the various scanners, with the help of Wireshark, which can capture and analyze data packets during network transmission. Wireshark has filters for capturing packets and displaying their information, which can help us track the data transmissions for web pages of interests and remove irrelevant information in thousands or millions of packets. Comparing to several message logging tools often used on the server side, Wireshark can provide more detailed information for requests and responses in different network layers, and its exporting module can generate report documents for further analysis. By analyzing the packet data exported from Wireshark, we can summarize the attack strings used by each scanner. We compare their injection efforts by comparing their attack patterns within the attack strings. Our comparison criteria include various tag types and injection techniques used in attack strings. The objective of our work is having more detailed knowledge about how each scanner is working in its injection phase, hoping to find some correlations

between its detection performance and its injection effort.

In this section, we will first give an overview of several XSS attack techniques. Then we will compare html tags, encoding techniques, and advanced injection techniques used in injection strings of different scanners. We will give some discussion and a summary at last. One reason for choosing those comparison criteria is that there are examples about them in related documents discussing XSS injections, such as the XSS Cheat Sheet [20], HTML Code Injection and Cross-site Scripting [18], and Advanced XSS Knowledge [17]. From our observation, we can see that those tags and injection techniques are also commonly used in various web application scanners.

5.4.1 Comparison Method

To better understand and evaluate injection techniques, we describe common XSS attack techniques, from the perspective of a XSS attacker. Below we give some examples where a page has been polluted by XSS injection strings:

<IMG SRC="javascript:alert('XSS');">

<BODY ONLOAD=alert(document.cookie)>

<A HREF="javascript:document.location='http://www.attackersite.com/'">XSS</A>

Attackers inject scripts having unexpected functionalities to the locations where data are are supposed to be received. To prevent from malicious inputs, web applications apply certain input validation techniques，like the escaping mechanisms we tried in creating controlled test applications. Thus, attackers can always create injections to bypass these preventions. Generally

speaking, most injection techniques make effort in two aspects:

a. Where to make injections, what are the contexts of the injection;

b. How the injections will be made, which advanced injection techniques are adopted to bypass the defense mechanisms.

Since the scanners send benign injections which have formats similar to actual XSS attacks, we evaluate their injection techniques in the above two aspects. During the process of scanning real-life web applications and test applications we created, we use Wireshark to capture packets for each scanner. Based on the XSS issues reported, with the help of the display filter of Wireshark, we randomly select several confirmed issues and use their destination host IP numbers, URLs and parameter information as search criteria to filter out irrelevant packets. We export the content of packets in a format which can be analyzed by the small Java programs we wrote. We study the injection effort each scanner has made to successfully exploit the focused vulnerabilities. After combining the analysis results of different vulnerabilities, for each scanner we come up with a list of **injection patterns**, which are the formats of common attack strings that the scanner uses. For example, if a request has the following URL:

http://www.somesite.com/index.php?input="><ScRiPt>alert('tg7x4l60vu')</sCrIpT>

The attack string will be “"><ScRiPt>alert('tg7x4l60vu')</sCrIpT>”, and the injection pattern obtained will be ><ScRiPt>….</sCrIpT>, which uses “a” tag, “script” tag, and an injection technique obfuscating tag names.

Comparing to other scanners which have tried various tags and injection techniques, Skipfish only has what is claimed to be “a complex string that is guaranteed to break out of many different parsing modes” by its developer Michał Zalewski, in his blog article, Understanding and Using Skipfish[29]. Since there is not enough variety of Skipfish’s attack strings, we only include the other 3 scanners in the controlled case studies, Netsparker, Acunetix, and Wapiti.

5.4.2 Comparison of Injection Contexts

Typically, malicious input can be injected into the context of various html tags. To better analyze the injections of different scanners, we classify several tags that are often used for injections. As described in the previous section, these tags come from related XSS tutorial documents and our study for injection patterns in different scanners.

First, there are some tags organizing html page structures, such as <body> and <title>. In this thesis, we call them **structural tags**. If the structural tags in a web page are not properly validated, the injections can exploit the vulnerabilities, and the malicious inputs can be reflected back. The comparison results for the structural tags used by the 6 scanners in their attack strings are reported in table 5.13, where “yes” indicates the scanner uses this tag in its injection, and “no” indicates that it does not.

Table 5.13 Comparison of Structural Tags

Structural Tags	IBM App Scan	Nikto	Netsparker	NStalker	Acunetix	Wapiti
<body>	no	no	yes	no	yes	no
<iframe>	yes	no	yes	no	yes	no
	no	no	no	no	yes	no
<title>	yes	yes	no	yes	no	no
Total Count	2	1	2	1	3	0

Secondly, there are several html tags used to define the attribute values of HTML elements or the format of a web page. For example, we use the <style> tag to define the HTML page format, like <style type="text/css">……<style>; we use the <base> tag to define the default link address, like <base href=www.attacker.com />, etc. In this thesis, we call them **modifier tags**. According to our observation, this type of injection tag is rarely adopted by the scanners we evaluated. A detailed comparison is reported in table 5.14.

Table 5.14 Comparison of Modifier Tags

Modifier tags	IBM App Scan	Nikto	Netsparker	NStalker	Acunetix	Wapiti
<meta>	no	no	no	no	no	no
<base>	no	no	no	no	no	no
<style>	yes	yes	yes	no	no	no
Total Count	1	1	1	0	0	0

Table 5.15 Comparison of Element Tags

Element tags	IBM App Scan	Nikto	Netsparker	NStalker	Acunetix	Wapiti
<img>	yes	yes	yes	no	yes	yes
<script>	yes	yes	yes	yes	yes	yes
<input>	no	yes	no	no	no	no
<table>	no	no	no	no	no	no
<object>	no	no	yes	no	no	yes
<form>	no	yes	no	no	no	yes
<textarea>	yes	no	no	yes	yes	no
<a>	yes	yes	yes	yes	yes	yes
Total Count	4	5	4	3	4	5

Finally, among the various tag patterns in our observation, many tags can be classified as

tags representing a page element. For example, the <script> tag indicates that the input might be embedded into a script; the <textarea> tag means that the input data might be reflected back within a text field. In this thesis, we call them **element tags**. Web application scanners usually try several element tags in their injections. A comparison about element tags can be found in table 5.15.

5.4.3 Advanced Injection Techniques Comparison

The comparison of injection tags can help us gain some understanding about the variety of the injection methods used by different scanners. However, scanners also distinguish themselves from their injection techniques, which are used to bypass input sanitization mechanisms. For a scanner, the more advanced injection skills it adopts, the more likely it will detect vulnerabilities while others cannot.

As we know, the plain text of attacker's input, such as <script>alert('XSS attack')</script>, is relatively easy to be sanitized by web applications and recognized by users. There are always some escaping mechanisms in web applications, and some servers even have methods to escape certain symbols automatically. For example, if the PHP setting "magic_quotes_gpc=ON" is set, every single quote and double quote are escaped with backslash. Attackers usually try to evade the sanitization mechanisms by encoding their input. This technique can work in many situations since filters may fail to recognize the attack strings, but browsers can always interpret the content correctly after decoding the content, and run malicious code. Some typical encoding

techniques can be concluded as:

1). URL encoding

This encoding technique is widely used by most scanners. The following is an example of String before and after encoding:

String before encoding: <IMG SRC=javascript:alert('XSS')>

String after encoding: %3CIMG%20SRC%3Djavascript%3Aalert('XSS')%3E

2). UTF-8 representation in XML

This is another very popular encoding method. An example is:

String before encoding: <IMG SRC=javascript:alert('XSS')>

String after encoding:

<IMGSRC=javascript:a&#108

;ert('XSS')>

3). Hex representation in XML

An example is

String before encoded: <IMG SRC=javascript:alert('XSS')>

String after encoded:

<IMG

SRC=&#x6A&#x61&#x76&#x61&#x73&#x63&#x72&#x69&#x70&#x74&#x3A&#x61&#x6C&#x65

&#x72&#x74&#x28&#x27&#x58&#x53&#x53&#x27&#x29>

4). Base 64 encoding

An example is:

String before encoding: <IMG SRC=javascript:alert('XSS')>

String after encoding: PElNRyBTUkM9amF2YXNjcmlwdDphbGVydCgnWFNTJyk+

Table 5.16 Comparison of Encoding Techniques

Encoding Techniques	IBM App Scan	Nikto	Netsparker	NStalker	Acunetix	Wapiti
Function to encode like fromCharCode()	no	no	yes	no	no	yes
UTF-8 Unicode encoding	no	yes	no	yes	no	yes
Hex encoding	no	no	no	yes	no	no
Base 64 encoding	no	no	yes	no	no	no
HTML entities(&quot, etc)	no	yes	no	no	yes	no
URL encoding	yes	yes	yes	yes	yes	yes
Totoal Count	1	3	2	3	2	3

5). Html entities

Some characters are reserved in html, such as the symbol “<“can be represented as “&lt”, and the symbol “&” can be represented as “&amp” etc. The following is an example:

Original String: <IMG SRC=javascript:alert(“XSS”)>

String after using html entities: <IMG SRC=javascript:alert("XSS")>

6). Using certain functions to generate input strings

An example is the static method fromCharCode() function. For example, we can use the function call “document.write(String.fromCharCode(72,69,76,76,79))” to generate string “HELLO”.

Table 5.16 illustrates the comparison for encoding techniques.

Except for the above encoding techniques, there are several other advanced injection techniques making effort to bypass input validation mechanisms. They generate variations to their injection strings. Some of them are described below:

To avoid the situation that the input within an injected string is sanitized, as it matches certain fixed key words defined by the filters, attackers mutate XSS expressions in their injections to break the match, by adding other characters. An example can be found in Netsparker, where, some injection strings are in the following format

“</a style=x:expre/**/ssion(Netsparker(0xXXXXXX))>”

The normal word “expression” is divided by the comment symbol “/**/”.

Since some browsers are capable of appending the closing bracket for incomplete tags

automatically, HTML tags in some injections are left unclosed, such as

<IMG SRC="javascript:alert('XSS')";

Sometimes web applications put certain tag names on their “bad word list”. As some browsers might not strictly check for tag structures, some injections use malformed tags, which are called obfuscated tags in this thesis. For example, tag “<script>” can be mutated as <sCrIpT>.

Table 5.17 Comparison of Injection Techniques

Advanced Injection Techniques	IBM App Scan	Nikto	Netsparker	NStalker	Acunetix	Wapiti
Embedded characters in XSS expression	yes	no	yes	no	no	yes
VB script	yes	no	no	no	no	no
Extraneous open brackets, No closing script tags, etc	no	yes	yes	yes	yes	yes
Anonymous tags	yes	no	no	yes	no	no
Obfuscated tags	no	yes	yes	no	yes	yes
Total Count	3	2	3	2	2	3

Apart from Javascript, VB script also are used, such as

<IMG SRC='vbscript:msgbox("XSS")'>.

Anonymous tags are found in some injections. For example, NStalker uses the <nstalker> tag, and Skipfish uses the <sfi> tag.

Table 5.17 gives a detailed comparison among several tools.

5.5 Summary

For Netsparker, Acunetix, and Wapiti, we can conclude that:

In the comparison for the injection tags, Netsparker and Acunetix have slightly better variety than Wapiti. Except for element tags, Wapiti does not use any modifier or structural tags.

For injection techniques, the three scanners have quite similar variety.

Overall, the performance of the injection patterns of these three scanners is very close. Since the evaluations of real-life vulnerable web applications and controlled test applications suggest that Wapiti has worse performance in XSS detection effectiveness, the observation of injection patterns cannot clearly indicate this point.

By comparing injection patterns of 6 tools together, we can see that all scanners have similar performance in tag patterns. The total number of tag patterns out of totally 15 tags each scanner uses can be found in table 5.18:

Table 5.18 Injection Tag Patterns Summary

IBM AppScan	Nikto	Netsparker	NStalker	Acunetix	Wapiti
7	7	7	8	7	5

No scanner has outstanding performance in the total number of injection patterns.

All scanners have similar performance in encoding techniques. Most of them adopt 2 or 3 encoding methods.

All scanners have similar performance in the advanced injection techniques. Most of them have 2 or 3 types of injection techniques.

By simply observing injection patterns, all the scanners make some effort in injections, and it is difficult to tell which one is better so far. According to the evaluation results from case studies of real-life applications and test applications we created, there are some differences in the injection effectiveness of the four scanners, but the minor differences in their injection patterns cannot clearly indicate this. Apart from the injection patterns, there are also many other factors influencing the injection effectiveness, and we will discuss about them in chapter 6.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

Based on our evaluation, we can see that among Netsparker, Acunetix, Wapiti, and Skipfish, commercial scanners Netsparker and Acunetix have overall better performance in both case studies of real life web applications and controlled studies of the test applications we created. Comparing to Wapiti, Skipfish has better performance in the real-life application case studies, and it can bypass some validation mechanisms in controlled test applications, whereas, Wapiti cannot. We have analyzed the differences of each scanner's efforts behind the scanning reports. In the following paragraphs, for the phases of crawling, deciding injection points, and making injections, we discuss the scanners' performance, summarize our observations, and propose our improvements.

6.1 Crawling

Although we have not evaluated scanners' performance in the crawling phase in detail, there are factors in this phase influencing the scanning experience, such as the effectiveness of session management. Only with well-controlled session management can the functionalities of a scanner be utilized to its limit. In case studies for real-life web applications, due to several failures in keeping stable session states, Wapiti could not reach enough web resources in the crawling phase during several scanning runs, and did not reach its full

potential in XSS detection. In order to improve this, scanners should have good usability in the functionalities like auto login and session maintenance. Many scanners have options to use the cookie string of an authenticated session in their configurations, which is easy to use and control, but this is the only method for many scanners. If the most frequent login method is not working, a good scanner should have several other methods as back up, since this can greatly increase a scanner's usability.

6.2 Deciding Injection Points

In the case studies of real-life web applications, it is clear that the ability of deciding injection points is affecting the overall scanning performance. Scanners Netsparker, Acunetix, and Skipfish, can reach a similar number of web pages in the crawling phase. However, Netsparker and Acunetix can identify more injection points. Due to their better performance in this step, they both are able to send injections to more locations which are actually vulnerable to XSS attack, thus have better scanning results. We find in Skipfish's source code, an injection request usually stores the information about which parameters should be fuzzed. Scanners should have certain algorithms to choose which web resources should be chosen as injection targets, and the differences in these algorithms make scanners have different performance in the phase deciding injection points.

6.3 Injection Effectiveness

The phase making injections involves the process of sending fuzzed data and analyzing response content. Based on our evaluation results, scanners have close performance in the injection patterns we focused on. So far, we have not found direct and obvious connections between the variety of injection patterns and the overall detection effectiveness.

Perhaps there are many other factors influencing the injection effectiveness. At first, the injection strategies of different scanners might be different. As we see in Skipfish source code, for different parameters the scanner has different injection actions, either replacing normal values with attack strings or appending different injection strings. Second, the logic in content analysis might also be a factor. To identify locations affected by previous injections, many scanners search for certain tag names and content injected with attacking strings. This pattern searching process might not be complete, and it is also possible to generate false positives.

6.4 Future Work

Our evaluation only includes the XSS detection performance of the four scanners so far. We can evaluate more tools or the newest version for the four tools in the future.

Netsparker and Acunetix have very close performance according to our evaluation results. To distinguish the performance of different scanners, we can have more case studies of real-life vulnerable web applications, written in different languages. We can also improve our test applications to cover more scenarios, e.g., the scenario that malicious input is injected into CSS

properties.

We did not make a detailed comparison about the crawling performance. Since the crawling process is always separated from the injection process, and it is very important, we plan to evaluate crawling abilities of different scanners in the future.

.

APPENDIX A

FALSE POSITIVE VALIDATIONS

Table A.1 Validations for Acunetix Scanning Results

	Vulnerability Location	T/F	Description
eFront	Index.php at “course”	TP	OSVDB 76745, type 2
	Index.php at “message_type”	TP	OSVDB 76745, type 2
webERP	URL appended at index.php	TP	OSVDB 77194, type 1 Attack string: %22onmouseover=prompt('XSS')%3E
	URL appended at accountgroups.php	TP	
	URL appended at agedsuppliers.php	TP	
	URL appended at bankreconciliation.php	TP	
	URL appended at contracts.php	TP	
	URL appended at customertransinquiry.php	TP	
	URL appended at custwherealloc.php	TP	
	URL appended at dailysalesinquiry.php	TP	
	URL appended at factors.php	TP	
	URL appended at glaccounts.php	TP	
Kaibb	rss.php at forum	FP	Part of input is reflected within SQL error message. There is a proper input escaping mechanism preventing actual XSS attacks
OrangHRM	index.php at “repcode”	TP	%22onmouseover=prompt('XSS')%3E
	centralcontroller.php at captureState	TP	%22%3E%3Cscript%3Ealert('xss') %3C%2Fscript%3E%3C
	centralcontroller.php at Loc_name	TP	
	centralcontroller.php at PageNo	TP	
	centralcontroller.php at TxtFromdate	TP	%22%20onmouseover%3dprompt %28998236%29%20bad%3d%22

Table A.1 - Continued

	centralcontroller.php at TxtTodate	TP	
OsCommerce	URL appended at advanced_search.php	TP	OSVDB 78619, type 1 Attack string: %22onmouseover=prompt(998974)%3E
	URL appended at conditions.php	TP	
	URL appended at contact_us.php	TP	
	URL appended at create_account.php	TP	
	URL appended at shipping.php	TP	
	URL appended at default.php	TP	
	URL appended at password_forgotten.php	TP	
	URL appended at privacy.php	TP	
	URL appended at product_info.php	TP	
	URL appended at reviews.php	TP	

Table A.2 Validations for Skipfish Scanning Results

	Vulnerability Location	T/F	Description(attack string)
eFront	Index.php (URL appending)	TP	OSVDB 76745, type 2. Attack string: %22onmouseover=prompt('XSS')%3E
OsCommerce	advanced_search.php confirmed attack	TP	OSVDB 78619, type 1 Attack string: %22onmouseover=prompt('XSS')%3E
	create_account.php	TP	
	conditions.php	TP	
	contact_us.php	TP	
	create_account_success.php	TP	
	login.php	TP	
	logoff.php	TP	
	password_forgotten.php	TP	
	privacy.php	TP	
	product_reviews_write.php	TP	
	reviews.php	TP	
	shipping.php	TP	
	shopping_cart.php	TP	
	specials.php	TP	
	tell_a_friend.php	TP	

Table A.3 Validations for Netsparker Scanning Results

	Vulnerability Location	T/F	Description(attack string)
eFront	Index.php at “course”	TP	OSVDB 76745, type 2. Attack string: %27%22%20ns=netsparker(0x0003E2)%20
	Index.php (URL appending)	TP	OSVDB 76745, type 2. Attack string: %22onmouseover=prompt('XSS')%3E
webERP	index.php	TP	OSVDB 77194, type 1, URL appending Attack string: %22onmouseover=prompt('XSS')%3E
	SelectCreditItems.php	TP	
	SelectCustomer.php	TP	
	SelectProduct.php	TP	
	doc/Manual/ManualContents.php	TP	
	PrintCustStatements.php	TP	
	SalesAnalRepts.php	TP	
	SalesGraph.php	TP	
	SelectOrderItems.php	TP	
	CustomerReceipt.php	TP	
	DailySalesInquiry.php	TP	
	SelectContract.php	TP	
	Contracts.php	TP	
	SuppPaymentRun.php	TP	
	PDFRemittanceAdvice.php	TP	
	UserSettings.php(parameters DisplayRecordsMax and email)	TP	Issue not reported. Attack string: %27%22%20ns=netsparker(0x0003E2)%20
	SelectSupplier.php(parameters Keywords, SupplierCode)	TP	Issue not reported. Attack string: %27%22%20ns=netsparker(0x0003E2)%20
	CustWhereAlloc.php(parameter TransNo)	TP	Issue not reported. Attack string: %27%22%20ns=netsparker(0x0003E2)%20
Kaibb	Index.php	FP	Issue not reported. URL appending. No actual XSS attack can be generated

Table A.3 - Continued

OrangHRM	lib/controllers/CentralController.php (parameter sortOrder0)	FP	Issue not reported. There are escaping mechanisms, and no actual XSS attack can be generated
	/orangehrm/templates/hrfunct/emppop.php (parameter sortOrder0, sortOrder1)	FP	
OsCommerce	advanced_search.php:	TP	OSVDB 78619, type 1 Attack string: %22onmouseover=prompt('XSS')%3E
	create_account_success.php	TP	
	contact_us.php	TP	
	login.php	TP	
	password_forgotten.php	TP	
	privacy.php	TP	
	product_info	TP	
	reviews.php	TP	
	shipping.php	TP	
	shopping_cart.php	TP	

REFERENCES

[1] Acunetix Web Application Security. http://www.acunetix.com/. Accessed June 20, 2012.

[2] Bau, J., Bursztein, E., Gupta, D., and Mitchell, J. State of the Art: Automated Black-Box Web Application Vulnerability Testing. 2010 IEEE Symposium on Security and Privacy. 2010. pp. 332-345.

[3] Chen, S. Web Application Scanners Accuracy Assessment. Security Tools Benchmarking. December 26, 2010.
http://sectooladdict.blogspot.com/2010/12/web-application-scanner-benchmark.html. Accessed February 14th, 2012.

[4] CWE. 2011 CWE/SANS Top 25 Most Dangerous Software Errors. September 13, 2011. http://cwe.mitre.org/top25/. Accessed May 17th, 2012.

[5] Doup´e, A., Cova, M., and Vigna, G. Why Johnny Can't Pentest: An Analysis of Black-box Web Vulnerability Scanners. Proceedings of the 7th international conference on Detection of intrusions and malware, and vulnerability assessment. 2010, pp. 111-131.

[6] Fong, E. and Okun, V. Web application scanners: definitions and functions. Proceedings of the 40th Annual Hawaii International Conference on System Sciences. 2007, pp. 280b.

[7] Fong, E., Gaucher, R., Okun, V., and Black, P.E. Building a Test Suite for Web Application Scanners. Proceedings of the 41st Hawaii International Conference on System Sciences. 2008, pp. 478.

[8] Fonseca, J., Vieira, M., and Maderia, H. Testing and comparing web vulnerability scanning tools for SQL injection and XSS attacks. 13th IEEE International Symposium on Pacific Rim Dependable Computing. 2007, pp. 365-372.

[9] Gordeychik, S. Web Application Security Statistics. Web Application Security Consortium.http://projects.webappsec.org/w/page/13246989/Web%20Application%20Security%20Statistics. Accessed March 20th, 2012.

[10] Hammersland, R. and Snekkenes, E. Fuzz testing of web applications.

http://www.idi.ntnu.no/emner/tdt60/Hammersland-FTW.pdf. Accessed April 10th, 2012.

[11] Kiezun, A., Guo,P.J., Jayaraman, K., and Ernst, M.D. Automatic creation of SQL Injection and cross-site scripting attacks. Proceedings of the 31st International Conference on Software Engineering. 2009. pp. 199-209.

[12] Klein, A. DOM Based Cross Site Scripting or XSS of the Third Kind. July 4, 2005. http://www.webappsec.org/projects/articles/071105.shtml. Accessed May 16th, 2012.

[13] Loh, P.K.K. and Subramanian, D. Fuzzy classification metrics for scanner assessment and vulnerability reporting. The IEEE Transactions on Information Forensics and Security. 2010. Volume 5, Issue 4, pp. 613 – 624.

[14] Mavituna Security. http://www.mavitunasecurity.com/netsparker/. Accessed June 12th, 2012.

[15] McAllister, L., Kirda,E., and Kruegel, C. Leveraging User Interactions for In-Depth Testing of Web Applications. Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection. 2008. pp. 191-210.

[16] Menczer, F., Pant, G., Srinivasan, P., Ruiz, M. E. Evaluating topic-driven web crawlers. Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval. New York, NY, USA. 2001. pp. 241-249.

[17] Novacaine. Advanced XSS Knowledge. March 23, 2010.
http://www.exploit-db.com/papers/13646/. Accessed July 8th, 2012

[18] Ollmann, G. HTML Code Injection and Cross-site Scripting.
http://www.technicalinfo.net/papers/CSS.html. Accessed May 6th, 2012.

[19] Open Web Application Security Project (OWASP): OWASP Top Ten Project. http://www.owasp.org/index.php/Top_10 (2010). Accessed April 6th, 2012.

[20] RSnake: XSS (Cross Site Scripting) Cheat Sheet. http://ha.ckers.org/xss.html. Accessed March 7th, 2012.

[21] Suto,L. Analyzing the Effectiveness and Coverage of Web Application Security Scanners. October, 2007. http://www.stratdat.com/webscan.pdf. Accessed March 20th, 2012.

[22] Suto,L. Analyzing the Accuracy and Time Costs of Web Application Security Scanners. February, 2010.
http://ha.ckers.org/files/Accuracy_and_Time_Costs_of_Web_App_Scanners.pdf. Accessed March 20th, 2012.

[23] Wassermann, G., Su, Z. Sound and precise analysis of web applications for injection vulnerabilities. Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation. 2007. Volume 5, Issue 4, pp. 32-41.

[24] Web Application Security Consortium. Web Application Security Statistics Project 2007. https://files.pbworks.com/download/Ay5UB8nfZw/webappsec/13247068/wasc_wass_2007.pdf ?ld=1. Accessed March 10th, 2012.

[25] WhiteHat Security. 7th WebSite Security Statistics Report. May 19, 2009. http://www.slideshare.net/jeremiahgrossman/whitehat-security-website-security-statistics-report-q109. Accessed April 15th, 2012.

[26] Wagner, S. The use of application scanners in software product quality assessment. Proceedings of the 8th international workshop on Software quality. New York, NY, USA. 2011. pp. 42-49.

[27] Wapiti – Web Application Vulnerabiliy Scanner / Security Auditor. http://wapiti.sourceforge.net/. Accessed October 11th, 2011.

[28] Wireshark. http://www.wireshark.org/. Accessed August 20th, 2011.

[29] Zalewski, M. Understanding and Using Skipfish.
http://lcamtuf.blogspot.com/2010/11/understanding-and-using-skipfish.html. Accessed March 15th, 2012.

[30] Zalewski, M., Heinen, N., Roschke, S. Skipfish – Web Application Security Scanner. http://code.google.com/p/skipfish/wiki/SkipfishDoc. Accessed June 20th, 2011.

BIOGRAPHICAL STATEMENT

Dengfeng Xia was born on 26th January 1986. His current research interests include Software Testing and Web Security. He obtained his Master of Science degree from the University of Texas at Arlington in May 2012.