Using a Web Server Test Bed to Analyze the Limitations of Web Application Vulnerability Scanners

David A. Shelly

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Joseph G. Tront, Chair
Scott F. Midkiff
Randolph C. Marchany

July 29, 2010
Blacksburg, Virginia

Keywords: Black Box Testing, Web Application Security, Web Application Scanners, Vulnerability Detection

Using a Web Server Test Bed to Analyze the Limitations of Web Application Vulnerability Scanners

David A. Shelly

(ABSTRACT)

The threat of cyber attacks due to improper security is a real and evolving danger. Corporate and personal data is breached and lost because of web application vulnerabilities thousands of times every year. The large number of cyber attacks can partially be attributed to the fact that web application vulnerability scanners are not used by web site administrators to scan for flaws. Web application vulnerability scanners are tools that can be used by network administrators and security experts to help prevent and detect vulnerabilities such as SQL injection, buffer overflows, cross-site scripting, malicious file execution, and session hijacking.

However, these tools have been found to have flaws and limitations as well. Research has shown that web application vulnerability scanners are not capable of always detecting vulnerabilities and attack vectors, and do not give effective measurements of web application security. This research presents a method to analyze the flaws and limitations of several of the most popular commercial and free/open-source web application scanners by using a secure and insecure version of a custom-built web application. Using this described method, key improvements that should be made to web application scanner techniques to reduce the number of false-positive and false-negative results are proposed.

Acknowledgements

I would like to thank Dr. Joseph Tront for his guidance during the period of my graduate research here at Virginia Tech. I would also like to thank Mr. Randy Marchany for his support during my graduate studies. Their mentorship gave me the confidence I needed to succeed in my research. Additionally, I would like to thank Dr. Scott Midkiff for serving on my advisory commitee and helping me through this involved process. I appreciate the opportunity they provided me with to pursue research in an area which I am interested in, and for allowing me to use the Virginia Tech IT Security Lab and resources for my research.

I would like to acknowledge the SANS institute, specifically Johannes Ullrich and Jason Lam, for providing Justin Stein and me with great opportunities in the area of web application security. Working with them ignited my interest in this research area and gave me an excellent starting point for my research. I would also like to thank Justin for being a research colleague of mine while completing his undergraduate research in the IT Security Lab.

I would also like to thank the members of the Virginia Tech IT Security Lab, Lee Clagett, Matthew Dunlop, John Paul Dunning, and Stephen Groat, as well as the other members of the IT Security Office, for their friendship and support during my time in the lab. The work I have done would not be the same without their teamwork, insight, and experience.

There are also several commercial vendors who provided trial licenses of their products for my research who I would like to thank. However, due to the stipulations in providing their licenses I will keep their names anonymous.

Last, but not least, I would like to thank my family and friends for their support and encouragement during this process. I would not have been able to succeed if it wasn't for the inspiration that they provided me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis investigates current limitations in the techniques used by web application vulnerability scanners. The goal of the proposed method and approach is to test web application scanners to discover where false-negative and false-positive results are being produced, and to analyze these results so that improvements can be made to web application scanner techniques. This chapter provides an overview of the research that has been conducted. Section 1.1 states the problem area which this research addresses. Background information regarding web application security and web application scanners is presented in Section 1.2. The unique research questions that this thesis addresses are reviewed in Section 1.3. An overview of the methodology used is given in Section 1.4. The expected results are discussed in Section 1.5, and finally, Section 1.6 summarizes the results and describes the outline for the rest of this thesis.

1.1 Problem Statement

Web application vulnerability scanners are preferred by many security auditors and web application administrators because they are typically very easy to use, perform tests relatively

quickly, and identify a wide variety of web application vulnerabilities. However, these tools lack the ability to detect many types of vulnerabilities that exist today because the level of artificial-intelligence currently implemented in their scanning engines is far below where it needs to be. Every web application scanner is faced with the same set of problems, these being that every web application is different, web application scanners operate based off of syntax, web application scanners do not improvise, and web application scanners are not intuitive [41]. Because of these problems, web application scanners are plagued with numerous false-positives and false-negatives. This research develops a secure and insecure version of a web application, so that a method to analyze the limitations of web application scanners can be utilized. Solutions to the problems discovered from this analysis can then be proposed in order to improve the techniques used by web application scanners to detect hard-to-find vulnerabilities.

1.2 Background

A study published by Symantec in 2010 [45] found that 75% of the enterprises surveyed admitted to having experienced a cyber attack within the past 12 months. One particular large-sized enterprised admitted that on average they experience about eight or nine attacks per week. Also, 100% of these enterprises experienced cyber losses in 2009, and reported that they spent an average of $2 million annually due to IT security breaches. In another analysis completed by the SANS Institute in 2009 [37], attacks against web applications accounted for more than 60% of the attacks seen on the Internet. Further, more than 80% of these web application vulnerability attacks were due to SQL injection or Cross-Site Scripting flaws in open-source and custom-built applications. As these numbers show, the cyber attacking business is indeed a thriving and profitable one.

The large number of cyber attacks can partially be attributed to the fact that web application vulnerability scanners are not used by web site administrators to scan for flaws.

Since the scanners are not being used, the proper adjustments are not being made to web applications, and they remain unprotected against attacks. There are many different types of web application attacks, some of which include SQL injection, buffer overflows, cross-site scripting, malicious file execution, and session hijacking. All of these attacks are very dangerous and can result in the loss of data, functionality, and money. Therefore, web application vulnerability scanners are necessary tools for administrators to use, and should be used regularly.

Research has shown however, that web application vulnerability scanners are not always capable of detecting vulnerabilities and attack vectors. As a result, they do not give an effective measurement of web application security [3, 16, 17, 44, 52]. Generally, the number of vulnerabilities that scanners cover is low, and the number of false-positives is high [3]. Improvements need to be made to the techniques used by web application vulnerability scanners so that they can provide better protection for web sites and their applications. Further motivation behind this research is given in Section 1.2.1, and definitions the reader should be familiar with are given in Section 1.2.2.

1.2.1 Motivation

Web application vulnerability scanners are tools that are used by network administrators and security professionals to help detect flaws in web sites so that they can be repaired before criminals or adversaries take advantage of them. However, since these tools currently have many flaws and limitations, they are an unsuitable method for detecting all of an application's vulnerabilities and preventing attacks. The motivation behind this research is to provide an analysis of the false-positives and false-negatives reported by web application scanners so that enhancements and improvements can be made to their techniques and better vulnerability detection can be achieved.

1.2.2 Definitions

In order to clarify the terminology that is used throughout this thesis, the definitions of some key terms and commonly misconstrued terms are given here.

Web Server - A program which manages the HTTP service to receive client requests for web-based content. The server handles client requests to ensure that they are in a valid format and that the requested resources exist, before passing the requests on to the web application for processing. Common web server software packages include Apache HTTP Server and Microsoft IIS [38].

Web Application - An application that handles the server-side logic of a web server, and is accessed over the Internet using the HTTP protocol. The three tiers usually associated with a web application are the presentation tier, logic tier, and data tier. The presentation tier is responsible for displaying or receiving data from the client, the logic tier handles the processing of input received from the presentation or data tier, and the data tier provides a mechanism for storing and retrieving data needed by the logic tier [12, 38].

Web Application Vulnerability Scanner - A type of tool that probes web applications for known flaws and vulnerabilities. Usually a set of signatures is used by the scanner to test against various elements of web applications [11]. This type of scanner utilizes what is referred to as the black-box scanning technique because it has no access to the source code of the application which it is scanning. These tools can also be referred to as web application scanners, web security scanners, web vulnerability scanners, or simply, scanners.

Web Crawling - Also referred to as web spidering, this is the technique of automatically browsing a web site in a methodical way to reveal its structure. In many cases, copies of the visited pages are made so that the web application's resources, such as embedded links and HTML code, can be accessed more quickly.

Vulnerability - In web applications, this is an area of weakness that is susceptible to attack. In order for a web application element to be vulnerable, it must be accessible by an attacker

who possesses the capability to exploit the underlying flaw.

Exploit - In web applications, this is a piece of computer code that attacks a vulnerability to cause unintended or unanticipated behavior. It may take the form of software, data strings, or command sequences, and is usually aimed at gaining control of a system, escalating user privileges, or denying service to the application.

1.3 Research Questions

The main research questions that this thesis aims to address are provided in this section. These questions are used as the basis behind the design of the test bed web server, and also to formulate the method and approach used to discover the limitations in web application scanning techniques. The test bed web server is designed to investigate the following questions:

1) What are the current limitations of web application vulnerability scanners?
2) How can the limitations of web application vulnerability scanners be improved?
3) Is the use of a controlled web server beneficial in analyzing why web vulnerability scanners produce false-positive and negative results?

1.4 Methodology Overview

This research presents a possible method to discover the limitations of web application vulnerability scanners by utilizing a test bed web server for vulnerability testing and analysis. The web application vulnerability scanners used for this research were selected based on the number and variety of vulnerabilities they probe for, the significance of the vulnerabilities they detect, and the scanning techniques that they use. The web applications that are hosted on the test bed web server are based on the Linux, Apache, MySQL, and PHP framework

(LAMP). This technology framework was selected because of its popularity and wide-spread use. Each web application vulnerability scanner is tested against the web application following a prescribed approach that includes a set of initialization, execution, classification, and analysis procedures.

By using both a secure and insecure version of a custom-built web application, false-positive and false-negative results can be linked to the techniques used by the scanners to discover vulnerabilities. This association between the techniques used and the false-positives or false-negatives reported can be used to suggest improvements for web application scanning techniques.

1.5 Expected Results

The web application was designed to have both a secure and insecure version so that both false-positive and false-negative results could be analyzed. The secure version of the web application is expected to return the majority of the false-positive results since no vulnerabilities are intentionally implemented in this version. Therefore, the only vulnerabilities that would be reported would be incorrectly identified vulnerabilities.

On the other hand, the insecure version of the web application is expected to produce false-negative results due to the fact that intentionally placed vulnerabilities may be missed by the scanners in this server implementation. Since research has shown that the current techniques used by web application scanners are not capable of detecting all vulnerabilities, it is expected that not all of the implemented vulnerabilities will be detected. Therefore, the false-negative results should show which types of vulnerabilities are not being detected by web application scanners.

Even though the insecure version of the web application server can report false-positives, it is expected that the majority of the false-positives will be identified in the secure version of the web application. A sufficient number of false-positive and false-negative results are antic-

ipated so that an analysis can be conducted for each type of vulnerability, and improvements can be proposed that will benefit web application scanner techniques.

It is also expected that if a web application scanner detects one specific type of vulnerability, then it will find all other instances of that same type of vulnerability. The techniques that web application scanners use should be consistent enough so that if they work to find one vulnerability, then a vulnerability that is caused by a similar server flaw will be detected as well.

1.6 Results and Organization

This research makes three significant contributions to the areas of web application security and web application vulnerability scanning. First, the test bed web server was developed which contains both an insecure and secure version of a web application. These two versions of the web application can be used to perform benchmark tests and analyze the coverage of web application vulnerability scanners. The second major contribution that this research presents is an effective method to analyze the limitations of web application vulnerability scanners. The test bed evaluation method, along with the web vulnerability scanner testing approach, have been shown to be successful techniques to unveil why and where vulnerability false-negatives and false-positives occur. The third, and final, contribution of this research is the proposed web application scanner improvements. The analysis of web application techniques identified the reasons why web application scanner techniques were failing, and allowed for the suggestion of new and improved techniques.

The rest of this thesis is organized as follows. Chapter 2 provides a literature review of important web vulnerability attacks, web vulnerability scanners, and web vulnerability scanner evaluations. Chapter 3 describes the design of both the initial and revised test bed web servers. Chapter 4 discusses the methodology and approach used for the tests conducted in this research. Chapter 5 presents the results and analysis that were obtained from the web

application scanner tests. Chapter 6 provides the conclusions of this research, including the lasting contributions, and the direction of future work.

Chapter 2

Review of Literature

According to Curphey and Araujo [13], there are eight categories of web application security assessment tools: source code analyzers, web application (black box) scanners, database scanners, binary analysis tools, runtime analysis tools, configuration management tools, HTTP proxies, and miscellaneous tools. The most common of these web application assessment tools are source code analyzers and web application scanners. Source code analyzers generally achieve good vulnerability detection rates, but are only useful if the web application's source code is available. On the other hand, web application vulnerability scanners are the tools which most closely mimic web application attacks, but have been known to perform rather poorly [16, 17, 44, 52].

The web application vulnerability scanner is the type of tool that is analyzed in this thesis because the techniques that these tools currently use to scan web applications need to be improved. Improving the scanning techniques of these scanners will allow them to achieve better performance and, therefore, increase their credibility. However, in order to understand and improve web application scanners, the common vulnerabilities that they aim to detect must be understood first. This review of literature is organized as follows: some of the specific vulnerabilities that web application scanners attempt to probe for will be discussed in Section 2.1, several of the most popular and researched web application scanners will be

discussed in Section 2.2, and related work regarding web application scanner evaluations and techniques will be discussed in Section 2.3.

2.1 Web Vulnerability Attacks

The number of security vulnerabilities that are being found today are much higher in applications than in operating systems [37]. This means that the attacks aimed at web applications are exploiting vulnerabilities at the application level, and not at the transport or network level like common attacks from the past. The Open Web Application Security Project (OWASP) Top 10 [50] provides one of the most highly regarded documents for web application security. The OWASP Top 10 represents a broad consensus of the current most critical web application security flaws, and the list on the release candidate for the OWASP Top 10 for 2010 includes:

- Injection (SQL, OS, LDAP)
- Cross Site Scripting (XSS)
- Broken Authentication and Session Management
- Insecure Direct Object References
- Cross Site Request Forgery (CSRF)
- Security Misconfiguration
- Failure to Restrict URL Access
- Unvalidated Redirects and Forwards
- Insecure Cryptographic Storage
- Insufficient Transport Layer Protection

This list only includes the ten most critical web application risks that exist today, but other important vulnerabilities exist, including buffer overflow exploits and malicious file execution (included on the previous version of the OWASP Top 10 for 2007 [47]). Because the total number of web application vulnerabilities that exist is extremely large, only the most relevant vulnerabilities are implemented and analyzed in this research. The web application vulnerabilities that are most relevant to this research include SQL injection, cross-site scripting (XSS), session management, buffer overflow, and malicious file execution. Information about these vulnerabilities has been taken from [12, 36, 38, 41, 50].

2.1.1 SQL Injection

A brief overview of different kinds of SQL injection attacks and their defenses will be discussed in this section. The basic definition and fundamental information regarding SQL injection techniques has been taken from Anley [2], Joshi [21], and Spett [39, 40].

Attacks

SQL injection occurs when malicious input is passed into a database interpreter without being properly validated or encoded. In this type of attack the client is attacking the web server database. The input that the attacker passes into the interpreter is crafted to be a legitimate SQL statement, but instead of returning the data that the application's developer intended, the interpreter now returns the data requested by the attacker. This type of attack is severe because not only can it expose all sensitive user and business related data, but it could even go as far as executing operating system commands or giving an attacker complete control of a web application. An example of a valid SQL query which displays information for the user "John" is:

```
SELECT info FROM users WHERE username = 'John';
```

An attacker could use the malicious user name "`' OR 1=1 -` " to cause the interpreter to display all of the user information data in the database. The corresponding SQL query would be:

```
SELECT info FROM users WHERE username = '' OR 1=1 -
```

This is one of the simplest types of SQL injection, but works because the leading single quote causes the query to break out of the single quote delimited data. Therefore the always true "`OR 1=1`" is appended to the query, and thus displays all of the user information data in the database. The double dashes "–" at the end of the query cause all of the text that would follow it to be commented out, because "–" is the comment symbol in this SQL language. Adding the comment symbol is necessary in this attack because it nullifies the rest of the syntax that the web application would normally append to the end of database query to complete the operation. Therefore, the only query that is being executed is the attacker's injected sequence, and not the web application's expected query.

Even more dangerous attacks are possible against certain SQL versions and databases as well. An example of this would be if an attacker took advantage of a web application that implements both regular and administrator users, and therefore normally logs in users with default roles, but could also log in a user with administrator roles. If the administrator user has advanced functionality and has the ability to access all of the web application's data, then the web application can be completely compromised if an attacker takes control of the administrator account. An SQL injection attack could accomplish this if a web application uses email addresses as user names and associates each user name in the database as either a regular or an administrator user. In this example the attacker will exploit a generic "Change Mailing Address" field on a web page and associate an email address of their choosing to the administrator account. The attacker would enter the following in the "Change Mailing Address" field on the web page:

```
'; UPDATE users SET username = 'attacker@email.com' WHERE
username LIKE '%admin%'; -
```

The semi-colon ";" will end the first query and allow for the attacker's query to be executed. This query will cause the email address that the attacker entered to replace the email address that matches the pattern most like "admin". All that is necessary to perform this attack is for the attacker to "guess and check" until they know that the table holding the accounts is in fact "users", and that the field holding the user names is in fact "username". After this, the user name most closely matching "admin" will be replaced with the attacker's email address, but will continue to have administrative capabilities. Now that the attacker has replaced the administrator's email address with his own, he can click the "Forgot Password" button that most user-based web applications provide, and have the administrator's password sent to him in the convenient "Password Reminder" email.

The previously mentioned attack is not always easy to execute because it is not trivial to find out the name of the table and column being used in the SQL database. This challenge is overcome by using two other types of SQL injection: blind SQL injection and error-based SQL injection. Blind SQL injection uses a series of true and false questions to take advantage of the predictability of the WHERE condition in SQL, since 1=1 will always return true. Therefore, if a record is returned when using blind SQL injection, the attacker's injected condition must have been true. Error-based SQL injection is a specific type of SQL injection that uses SQL error messages to determine the structure of the database. SQL injection statements are crafted by the attacker in a way such that the attacker can use the error responses to systematically unveil table names, column names, column data types, and even specific data entries.

Defenses

The best way to prevent SQL injection is to have all interpreters separate untrusted data from database queries and to only accept expected input [12, 50]. In order to achieve this, all data originating from a client should have special characters escaped or sanitized into a valid format, should use an API which avoids the use of an interpreter entirely, or should use prepared statements and parameterized interfaces. The "mysqli->prepare" mechanism will create prepared statements for MySQL in PHP, and the following code sequence will escape all special characters:

```
if ( !get_magic_quotes_gpc() ) {
    $safe_string = mysql_real_escape_string( $original_string );
}
```

Also, to protect against blind SQL injection and error message based SQL injection, SQL error reporting should be disabled in conjunction with the other safety measures mentioned above. In order to disable error reporting for MySQL the "@" character should precede commands to suppress on-screen error reporting. These defensive measures should be implemented to mitigate SQL injection attacks.

2.1.2 Cross Site Scripting

Three of the main types of cross-site scripting (XSS) attacks, as well as some defensive techniques to protect against them, will be reviewed in this section. For more detail regarding XSS attacks, refer to Endler [14].

Attacks

XSS occurs when a web application includes malicious code in a web page that is sent to a client's browser without proper content validation. In this type of attack the web page server is attacking the client machine. When the web page is viewed by the client it will execute the malicious script that the attacker embedded into the web page. XSS is the most prevalent web application security flaw [50] due in a large part to its simplicity and resulting severity. Some of the attacks that this type of vulnerability can result in are the hijacking of a user's session, the defacement of websites, the insertion of hostile content, and the redirection of users' requests.

Reflective XSS is a type of XSS attack that can occur when a victim follows a URL which contains malicious scripting that is executed when the web page is rendered. This is commonly done by sending victims legitimate looking e-mail messages that contain malicious script in the message's URL. Once the HTTP request from the URL is processed, the HTML content is received and displayed in the victim's browser, thus executing the malicious script. An example of a URL containing a reflective XSS attack that would execute some type of malicious script included the function "malicious()" would be:

```
http://www.targetsite.com/display.php?user=<script>malicious()</script>
```

Stored XSS is another type of XSS attack. This type of attack occurs when the malicious script is uploaded into the database back end of a server without input validation, and is later retrieved by the web application to be embedded into a web page. This causes every user who visits the infected web page to execute the malicious script in his or her browser. An example of where this vulnerability can be found is a web application that uses a comment section to allow users to view and leave feedback about a product. If an attacker were to leave a comment which included malicious script, the script would be stored as a comment for that product in the database, and then executed every time a user clicks to

view the page holding the comments for that product. An example of this type of attack is a script crafted to steal a user's cookie and save it in a remote site for exploitation at a later time so that they can perform actions as if they were the victim (such as bank transactions, e-mail correspondence, etc...). The following script would execute such an attack if stored in a web application's database and then executed in a client's web browser:

```
<script>document.write('<img src="http://www.attackersite.com
'+document.cookie+'")</script>
```

A third type of XSS attack is Document Object Model, or DOM-based, XSS. This is a different kind of XSS attack because it occurs on the client side when the user is processing the content, instead of on the server side when the web application is retrieving information to put in a web page. The Document Object Model is the standard model that represents HTML and XML content of a web page. The DOM can be modified in this type of attack to execute a malicious script in the victim's browser. An example of this type of attack would be to exploit a web page that uses some embedded JavaScript in to set the default language for the client using a variable in the URL. An example of this would be:

```
http://www.mysite.com/index.html#default=English
```

The malicious script that would exploit this would simply need to replace the variable "English" in the URL. A URL that shows this type of DOM-based XSS attack would be:

```
http://www.mysite.com/index.html#default=<script>malicious()</script>
```

Because everything after the # in a URL is not sent to the server by the browser, the malicious script would not be detected by the server even if the server was performing input validation. Therefore the script would be echoed into the page (DOM) when the browser

renders it, and would result in the attacker's script being executed [49].

Defenses

The same rules described above that apply to protecting against SQL injection, apply to protecting against XSS as well. All user supplied input should be validated and properly escaped before being included in the output web page [50]. This requires the same escaping technique as before, which in PHP is:

```
if ( !get_magic_quotes_gpc() ) {
    $safe_string = mysql_real_escape_string( $original_string );
}
```

Also, proper output encoding will ensure that the browser treats the possibly dangerous content as text, and not as active content that could be executed. The "htmlentities()" and "htmlspecialchars()" functions in PHP will check output to make sure that it is HTML encoded.

The best way to avoid DOM-based XSS vulnerabilities is to have all client-side input passed to the server for proper validation. However, if using variables in the DOM cannot be avoided, then input validation should occur in the script itself. A check to confirm that the string being written to the HTML page consists of only alphanumeric characters should be completed so that no scripting characters are allowed. A downside of this defensive technique is that the security check is viewable in the HTML code to the attackers, and therefore is easily understandable and attackable [22]. An example of a script which checks that only alphanumeric characters exist in a string is:

```
if (original_string.match(/^[a-zA-Z0-9]+$/)) {
```

```
    document.write(original_string);
}
```

These defensive techniques, although not infallible due to the mentioned limitations, will add a level of protection against attacks that aim to exploit XSS vulnerabilities.

2.1.3 Session Management

Session management vulnerabilities can exist when a web application does not provide a secure mechanism for maintaining a user's state, both while the user is interacting with the web application, and after the user finishes his session. Attacks to exploit vulnerable session management schemes, as well as defensive strategies to counter these exploits, will be discussed in this section.

Attacks

Session management attacks use flaws in the authentication or session management functionality of a web application to impersonate users by stealing a victim's session IDs. Session IDs (sometimes referred to as session tokens) are pseudo-unique strings that are generated by web applications to maintain session state [28]. Flaws involving session management can result in an attacker controlling some (or even all) of the accounts registered with an application. This could give an attacker full control of a web application if the session ID of a privileged account, such as an administrator account, becomes compromised. Two of the most common methods used by web applications to handle session management is to either include session IDs in the URL, or to use cookies. Exposing session IDs in URLs or in cookies gives attackers the ability to impersonate other users. An example of a URL that exposes a user's session ID would be:

```
http://www.bank.com/myaccount;sessionid=5DJ7FD65SG7HJJ77NBDSF4S33S
```

If the attacker can convince a user to send this URL to them, or if the attacker can view this URL while it is being transmitted, then the attacker will be able impersonate this user. Using cookies to store session IDs is a better approach then using a URL because the ID value is obfuscated and more difficult to abuse since it is not embedded in the URL [28]. However, cookies are still vulnerable to attacks because they can be logged (e.g., using an XSS attack to remotely store a user's cookie) or socially engineered.

The way in which session IDs are created may result in vulnerabilities as well. If a session ID is simply created using a trivial process, such as a sequential counter or a time stamp, then it may be possible for an attacker to retrieve the session ID of a particular user. Many web sites limit the number of incorrect user name/password attempts to log into a web page, but do not limit the number of incorrect session ID attempts that may be sent to a web server. If this is the case, then an attacker may be able to launch a brute-force attack against a web server to discover a legitimate session ID, and thus hijack a user's session.

An example of this type of attack would be if an attacker used a tool which constantly visited a web site to receive a session ID and provided an output of all the session IDs received. If the attacker were to run this tool during a time when a potential victim was logging into a web application, then the attacker would know which session ID the victim received, and would be able to mimic that user. An example of output from a type of attack like this can be seen in Figure 2.1. The output shows that the session IDs are indeed sequential, and since there is a gap between the session ID ending in 47 and the session ID ending in 49, the attacker can infer that a legitimate user was given the session ID ending in 48.

Defenses

There are several ways to protect against session management attacks. Making sure that session IDs are not predictable is the first step in providing security for a user's session. A trusted source that is capable of providing randomness, such as a pseudo-random number

Figure 2.1: The output of an example attack to steal a user's session ID.

generator, should be used to generate session IDs [12]. In PHP, the PHPSESSID variable can be used as a random session ID to maintain a user's state. Figure 2.2 shows an HTTP header of a web application that is using the PHPSESSID variable as the session ID in the cookie.

Session IDs should be destroyed by the web server when a user logs out of the web application. This will prevent an attacker from being able to use a session ID to mimic another user if the user's session ID was for some reason confiscated. To protect even further, session IDs should have a timeout value that causes them to expire after a certain period of time. This way, if a user does not explicitly log out of a web application, the session will expire anyway, and the session ID will not be reusable if an attacker is able to confiscate it.

Also, to protect session IDs from being stolen while in transit, HTTPS or TLS (SSL) connections should be implemented to encrypt and protect session IDs. This is especially true if cookies are used that contain sensitive information such as user names, passwords, or user

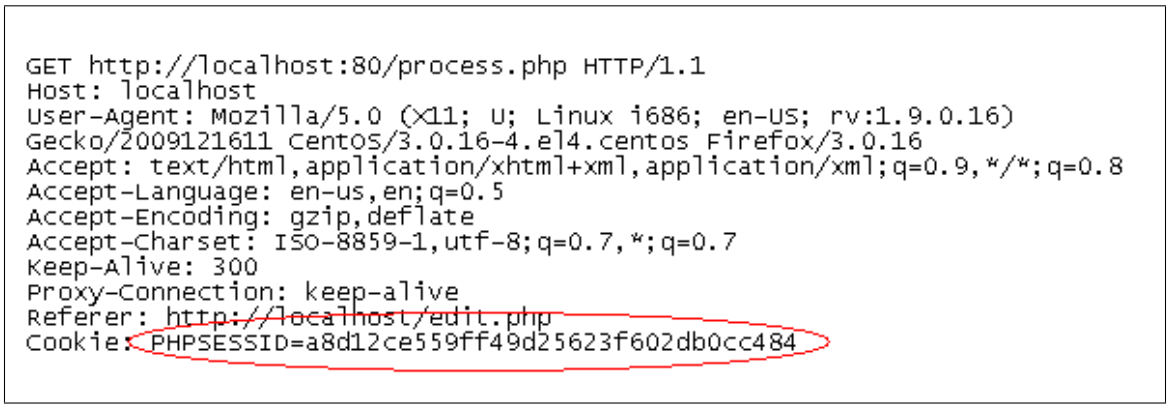


Figure 2.2: An example of a cookie using the PHPSESSID variable as a user's session ID.

privileges. If this is the case, a message authentication code (MAC) should be calculated for the cookie to preserve its integrity, and then the cookie should be encrypted to prevent eavesdropping. All of these defensive measures should be implemented in one way or another to ensure the maximum level of security to protect against session management vulnerabilities.

2.1.4 Buffer Overflow

Buffer overflow vulnerabilities have always been a significant problem in computer security because they provide hackers with an attack vector to gain control of a remote computer system. An explanation of how buffer overflow attacks work, and how to protect against them will be discussed in this section.

Attacks

The buffer overflow attack is a popular attack that affects not only web applications, but all computer programs written without proper bounds checking. The vulnerability occurs when an application copies user supplied data into a memory buffer that is not large enough to hold it. This results in the buffer overflowing and causes the subsequent memory blocks to be written with the attacker's data. This type of attack can be used to corrupt data, crash programs, or even execute malicious code on a web server. However, these vulnerabilities

are often hard to find, and are even harder to exploit [41, 48].

A common way that attackers find these vulnerabilities is by sending large amounts of data as input to a program, and then analyzing the application's response to see if anything unusual occurs. If an attacker suspects that a program is susceptible to a buffer overflow he will craft his input so that the information on the call stack of the vulnerable application is overwritten. This causes the function's return pointer to be set to a value so that it executes the malicious code that the attacker just inserted into the victim system as data. An example of where a buffer overflow vulnerability could exist is:

```
char buf[64];
gets(buf);
```

This would result in a buffer overflow vulnerability because the "gets()" function reads an arbitrary amount of data into a stack buffer. Since the size of the data being read into "buf[]" is not being checked beforehand, more than 64 characters could be read-in and overflow the stack.

According to [48], almost all web servers, application servers, and web application environments are susceptible to buffer overflow vulnerabilities. Languages such as Java and Python are immune to these attacks, and are only vulnerable if the overflow takes place inside the interpreter itself because they automatically provide bounds checking on their functions. Several buffer overflow vulnerabilities have been found in PHP functions over the years. Some examples of these include the "htmlentities()," "htmlspecialchars()," "wordwrap()," and "popen()" functions. The security expert e.wiZz! discovered that the "popen()" function is vulnerable to a buffer overflow in PHP versions 4.2-5.2 [15]. The proof of concept code that was provided is:

```
$_buff = str_repeat("A",9999);
$handle = popen('/whatever/', $_buff);
```

```
echo $handle;
```

If the user supplied input exceeds the memory buffer, the following warning will be displayed if a vulnerable version of PHP is being used:

Warning: popen(/bin/ls,): Invalid argument in **/var/www/html/buf.php** on line **3**

If a version of PHP is being used that is not vulnerable to the "popen()" buffer overflow vulnerability, the size of the user supplied input will be checked and the output would be:

Warning: popen(/bin/ls,): Result too large in **/var/www/html/buf.php** on line **3**

Defenses

To protect against buffer overflow vulnerabilities, code should be written so that the web application only accepts input that is within a specified boundary or size. For web applications, this means validating all input from HTTP request methods so that if a large amount of data is received it can be handled properly.

Many buffer overflow vulnerabilities are due to insecure functions in development languages or server products. The best way to protect against a buffer overflow in these cases is to not use the vulnerable functions or products at all. If this is not possible, then the user should either upgrade to a newer version of the language or product, or apply a patch (if one exists) that corrects the problem. These defensive measures will protect against buffer overflow vulnerabilities if implemented correctly.

2.1.5 Malicious File Execution

Malicious file execution attacks are possible against vulnerable web applications when user supplied input is processed by a web server's file or stream functions without properly being inspected for hostile content. Possible malicious file execution attacks, as well as defenses against these attacks, will be described in this section.

Attacks

Malicious file execution attacks allow hackers to achieve internal system compromise, perform remote code execution, and install remote root kits [47]. These types of attacks can affect both the client and the server and often result in Denial of Service (DoS), data theft, session hijacking, and even complete system compromise. One particular type of malicious file execution attack is the Remote File Include (RFI) attack. This type of attack occurs when a web application is tricked into including non-approved remote files with malicious code by accepting filenames or files from an attacker. An example of PHP code that is vulnerable to this attack is:

```
$file = $_POST["filename"];
include($file.".php");
```

If the web application does not sanitize the input, a remotely hosted file containing an exploit could be uploaded instead of an expected local or remote file. This exploit could be uploaded either by a local file name, or a URL, because PHP is configured by default to allow both these types of inclusion. For example, when a server executes a transaction or operation that requests a file, an attacker could upload a malicious script that is hosted on a remote machine by using the following as the filename :

```
http://attackersite.com/malicious_file
```

This will result in the remote file being included, and therefore executed, on the server. Other PHP commands that are also vulnerable to this same type of attack are "include_once," "fopen," "file_get_contents," "require," and "require_once" [4].

Defenses

To defend against these attacks, the same counter-measures should be taken which apply to SQL injection and XSS. Escaping special characters will limit the attacks because URLs will be treated as text, and not as active links. The "get_magic_quotes_gpc()" function in PHP checks if a string should be escaped. An even better approach would be to only accept known good input because this would prevent harmful characters from ever being included in the filename to begin with.

In PHP, the "register_globals", "allow_url_fopen" and "allow_url_include" properties can be disabled in the PHP configuration file to prevent problems related to malicious file execution as well. These properties allow file system functions to use a URL to retrieve data from remote locations, but their functionality is not necessary in most web applications [47]. Therefore, these mitigation techniques should be followed to prevent attacks caused by malicious file execution vulnerabilities.

2.2 Web Application Vulnerability Scanners

There are several web application vulnerability scanners that test for popular vulnerabilities in web servers and web applications. These tools can either be academic research projects, free/open-source applications, or commercial software products. Tools are developed in academia by members of universities who are interested in improving and studying web application vulnerability scanners, but are generally not available for purchase or commercial use. The open-source/free tools are available to the public, but are generally not as up-to-

date and accurate as the commercial tools. These tools do however, give users the ability to customize their tool and gain a greater understanding of the security of their web applications. Commercial tools usually give more comprehensive results than open-source/free tools, but can cost anywhere from just under $100.00 to over $6000.00 [1, 9]. Specific web vulnerability scanners from these three categories that automatically scan for and detect the most common web application vulnerabilities will be reviewed in this section.

2.2.1 Web Application Scanners in Academia

One of the categories of web application vulnerability scanners includes those that are developed in academia. These scanners are different from free/open-source and commercial scanners because the researchers who work on them are continuously evaluating them and also discuss not only where their design succeeds, but where their design is limited and requires future work. These scanners are not available for public use, so they cannot be used in this analysis of web vulnerability scanner limitations, but reviewing the techniques and methods used by these scanners will help in understanding how other web application scanners work.

Huang et al. developed a web application scanner called WAVES that attempts to reduce the number of potential side effects of black-box testing [18, 19]. The auditing process of web application scanners can cause permanent modifications, or even damage, to the state of the application it is targeting. This is a drawback that both commercial and open-source/free web application scanners share, and is why the authors introduced a testing methodology that would allow for harmless auditing. Their experimental results found that WAVES was unable to detect any new vulnerabilities that were not already detected by a static source code analyzer they had developed. Also, WAVES was unable to discover all of the vulnerabilities that the static source code analyzer had found (detected only 80% of the vulnerabilities found by the static analyzer). The authors believe their tool failed in part because it did not have complex procedures able to detect all data entry points, and because

it was unable to observe HTML output.

Another academic black-box approach was developed by Antunes and Viera as described in [3]. Their web vulnerability scanner was used to identify SQL injection vulnerabilities in 262 publicly available web services. The first step in their approach was to prepare for the tests by obtaining information regarding the web service in order to generate the workload (valid web service calls). The second step was to execute the tests. This was accomplished by using a workload emulator that acted as a web service consumer, and by using an attack load generator that automatically generated attacks by injecting them into the workload test calls. The final step in their approach was to analyze the responses by using a set of well-defined rules which would identify vulnerabilities and exclude potential false-positives. Their results showed that they achieved a detection coverage rate of 81% in the scenario where they had access to the known number of vulnerabilities, and maintained a false-positive rate of 18% in their optimistic interpretation. These results are better than those of the commercial tools that the authors analyzed, and suggests that it is possible to improve the effectiveness of vulnerability scanners.

2.2.2 Free/Open-Source Web Application Scanners

Many open-source and free web application scanners are available for black-box testing and analysis. Some of these applications provide extensive functionality with the ability to be customized and expanded to meet the needs of users. Others however do not provide a great deal of usability and have a limited amount of functionality, and therefore can only test for a few web application vulnerabilities. Three of the more thorough and robust free/open-source scanners, Grendel-Scan [6], Wapiti [43], and W3AF [35], will be reviewed.

Grendel-Scan [6] is an open-source web application security testing tool which has an automated testing module for detecting common web application vulnerabilities. It has the ability to find simple web application vulnerabilities, but its designers state that no automated tool can identify complicated vulnerabilities, such as logic and design flaws. Grendel-Scan tests

for SQL injection, XSS attacks, and session management vulnerabilities, as well as other vulnerabilities.

Wapiti [43] is a free web application vulnerability scanner and security auditor. It performs black-box analysis by scanning the web pages of a web application in search of scripts and forms where data can be injected. After the list of scripts and forms is gathered, Wapiti injects payloads to test if the scripts are vulnerable. Wapiti scans for remote file inclusion errors, SQL and database injections, XSS injections, and other vulnerabilities.

W3AF [35] is exactly what it stands for, a Web Application Attack and Audit Framework. The goal of the project is to create a framework which can find and exploit web application vulnerabilities easily. The project's long term objectives are for it to become the best open source web application scanner, and the best open source web application exploitation framework. Also, the designers want the project to create the biggest community of web application hackers, combine static code analysis and black box testing into one framework, and become the nmap [23] of the web. W3AF incorporates a great deal of plug-ins into its framework, and is capable of testing for SQL injection, XSS attacks, buffer overflow, malicious file execution, and session management vulnerabilities. Table 2.1 provides a comparison of the vulnerabilities that the free and open source web application scanners search for.

Table 2.1: A comparison of the relevant vulnerabilities detected by free/open-source web application scanners.

	Grendel-Scan	Wapiti	W3AF
SQL Injection	X	X	X
Cross Site Scripting (XSS)	X	X	X
Session Management	X	-	-
Malicious File Execution	-	X	X
Buffer Overflow	-	-	X

2.2.3 Commercial Web Application Scanners

Commercial web application scanners are generally licensed to companies or organizations that wish to test their web applications for vulnerabilities so that they can fix security holes before they are maliciously exploited. Since a data breach can result in the loss of personal information of thousands of customers, and the loss of millions of dollars [45], companies are willing to pay large sums of money for these applications. These commercial applications compete against each other for market share, and therefore do not want to disclose their scanner's limitations or restrictions. However, an approach to analyze these limitations and restrictions is proposed in this thesis. Some of the features of popular commercial web application scanners will be discussed below.

Cenzic [8] sells a web application scanner tool called Hailstorm which utilizes stateful testing. Stateful testing tools are designed to behave like human testers by taking what seems to be an application's insignificant or disparate weaknesses, and combining them together into serious exploits [11]. The key benefits that Hailstorm claims are the ability to identify major security flaws in target applications, to help with internal compliance policies, to avoid vulnerabilities that lead to downtime, and to assess applications for commonly known vulnerabilities. Cenzic provides a 7-day free trial of Hailstorm Core which can detect vulnerabilities including SQL injection, XSS , and session management.

Acunetix Web Vulnerability Scanner [1] is another black-box tool which claims in-depth checking for SQL injection, XSS, and other vulnerabilities with its innovative AcuSensor Technology. This technology is supposed to quickly find vulnerabilities with a low number of false-positives, pinpoint where each vulnerability exists in the code, and report the debug information as well. Acunetix also includes advanced tools to allow penetration testers to fine tune web application security tests, and has many more features to scan websites with different scan options and identities. The only vulnerability that the free edition of the software detects is XSS, but a 30-day trial version of the product is available that also can detect SQL injection, file execution, session management, and manual buffer overflow

attacks.

N-Stalker [29] provides a suite of web security assessment checks to enhance the overall security of web applications. It is founded on the technology of Component-oriented Web Application Security Scanning, and allows users to create their own assessment policies and requirements, enabling them to check for more than 39,000 signatures and infrastructure security checks. Vulnerabilities checked for include SQL injection, XSS attacks, buffer overflows, and session management attacks, but the evaluation edition only lasts for a 7-day period.

Netsparker [24] is a web application vulnerability scanner developed by Mavituna Security Ltd. Netsparker is focused on eliminating false-positives, and uses confirmation and exploitation engines to ensure that false-positives are not reported. The engines also allow the users to see the actual impact of the attacks instead of text explanations of what the attack could do. Because of the techniques Netsparker uses, Mavituna Security claims that it developed the first false-positive free web application scanner. Netsparker scans for all types of XSS injection, SQL injection, malicious file execution, and session management vulnerabilities.

Burp Scanner [33] is a web application vulnerability scanner that is part of Burp Suite Professional. Burp Suite Professional is the commercial version of Burp Suite, which is an integrated platform for attacking and testing web applications. Burp Suite provides a number of tools, including an interception web proxy, web spider, application intruder, session key analyzer, and data comparer. The professional version includes Burp Scanner which can operate in either passive or active mode, and either manual scan or live scan mode. The vulnerabilities it searches for include SQL injection, XSS injection, and session management vulnerabilities.

Rational AppScan [20] is licensed by IBM for advanced web application security scanning. The AppScan tool automates vulnerability assessments and tests for SQL injection, XSS attacks, buffer overflows, and other common web application vulnerabilities. AppScan can generate advanced remediation capabilities in order to ease vulnerability remediation, sim-

plify results with the Results Expert wizard, and test for emerging web technologies. Rational AppScan provides an unlimited evaluation period for its standard edition; however, with the evaluation license the software is only capable of testing a test web site provided by AppScan.

BuyServers Ltd. [5] sells a web vulnerability scanner called Falcove which is a 2-in-1 scanning and penetration tool, meaning that it not only tries to detect vulnerabilties, but is capable of exploiting them as well. Falcove utilizes a crawler feature that checks for web vulnerabilities, audits dynamic content (password fields, shopping carts), and generates penetration reports that explain the security level of the tested web site. However, BuyServers Ltd. no longer supports the trial version of the product that detects SQL injection, XSS, and file execution attacks.

HP's WebInspect [7] software provides web application security testing and assessment for complex web applications. WebInspect claims fast scanning capabilities, broad security assessment coverage, and accurate web application security scanning results. HP also believes WebInspect identifies security vulnerabilities that are undetectable by traditional scanners by using innovative assessment technologies such as simultaneous crawl and audit, and concurrent application scanning. HP WebInspect scans for data detection and manipulation attacks, session and authentication vulnerabilities, and server and general HTTP vulnerabilities, but does not currently provide a working evaluation version of the product.

NT OBJECTives' NTOSpider [32] is a web application security scanner that claims to provide automated vulnerability assessment with unprecedented accuracy and comprehensiveness. NTOSpider identifies application vulnerabilities and ranks threat priorities, as well as produces graphical HTML reports. NT OBJECTives' proprietary S^3 Methodology and Data Sleuth intelligence engine are employed for automation and accuracy, and checks vulnerabilities on a case-by-case basis, which provides context-sensitive vulnerability checking. NTOSpider checks for SQL injection, XSS attacks, and session management vulnerabilities, but does not provide a trial version for evaluation.

Table 2.2 provides a summary of the relevant vulnerabilities that each of the evaluation

Table 2.2: A comparison of the relevant vulnerabilities detected by evaluation versions of commercial web application scanners.

	Hailstorm	N-Stalker	Netsparker	Acunetix	Burp Scanner
SQL Injection	X	X	X	X	X
Cross Site Scripting (XSS)	X	X	X	X	X
Session Management	X	X	X	X	X
Malicious File Execution	-	-	X	X	-
Buffer Overflow	-	X	-	X	-

versions of the commercial web application scanners detect.

2.3 Related Evaluation Applications and Techniques

Several studies evaluate the usefulness of web application vulnerability scanners. However, no significant previous work has been identified that specifically aims at finding ways for tool improvement by using a vulnerability benchmark system. In Section 2.3.1 some of the web applications available for conducting web vulnerability scanner evaluations will be introduced. Section 2.3.2 describes some of the approaches and results found in the literature regarding web vulnerability scanner evaluations.

2.3.1 Web Vulnerability Scanner Evaluation Applications

There are a number of tools such as WebMaven's BuggyBank [34], and Foundstone's Hacme Bank [25], Hacme Shipping [26], and Hacme Travel [27] which can be used both to evaluate web vulnerability scanners and to train users about web application security. These applications emulate several security flaws such as SQL injection, buffer overflows, cross-site scripting, and authentication/authorization flaws. These applications allow users to safely and legally attempt real exploits against web applications so that they can learn about security flaws and fix these issues in their own applications.

Web applications such as these can also be used as testing environments for web application

vulnerability scanners. Scanners can be evaluated based on the known number of built in vulnerabilities in the web applications, and the corresponding number of vulnerabilities that the scanners detect. However, these applications are not built to mimic the functionality of real web applications, and therefore do not always represent the type of behavior actually seen on the web. Also, some web application scanner vendors are aware of the fact that users test their scanners against these vulnerable applications, and therefore pre-program their tools to report the vulnerabilities that have already been discovered [44].

2.3.2 Web Vulnerability Scanner Evaluation Techniques

Most evaluation techniques consist of using existing web applications that are publicly available on the web for judging web vulnerability scanner performance [17, 44, 52]. This technique is common because it allows researchers to test scanners against multiple websites that they believe include vulnerabilities. Another type of evaluation technique [16] is based on developing a benchmark web application where the features installed and tested are controllable. The test bed web server approach described in this paper follows this latter technique for conducting its scanner limitation analysis.

Application security consultant, Larry Suto, conducted an experiment in order to find the accuracy and time needed to run several commercially available web application security scanners [44]. The author conducted his tests using a 'Point and Shoot' method, as well as a 'Trained' scan method, against vendors' own test sites. In the 'Trained' scan, the tools were made aware of all the pages that they were supposed to test in order to mitigate the limitations of the crawler in the results. The 'Point and Shoot' method utilized only the default configuration without specifically being made aware of the pages to test. The results found that the best scanners achieved 94%, 62%, and 55% accuracy rates in 'Trained' mode, but that the other scanners only averaged a 39% accuracy rate overall (61% false-negatives). The results also showed that only moderate improvements were gained from normal training, and that the scanners that missed the most vulnerabilities tended to also report the most

false-positives.

An experiment comparing web vulnerability scanning tools for SQL injection and cross-site scripting attacks was conducted by Fonseca, Vieira, and Madeira [17]. In order to make clear the coverage and false-positive rates of vulnerability scanning tools, they proposed a method to benchmark and evaluate automatic web vulnerability scanners by using software fault injection techniques. The software faults were injected into the code of two web applications. They were then checked with three commercial web vulnerability scanners to test if the potential vulnerabilities created by the injected faults were detected. Their results showed that the percentage of false-positives is considerably high, ranging from 20% to 77%. The authors' analysis found that most of the false-positives were caused by errors issued by the web application in normal execution due to the fault injected. The authors' results also found that 9% of the total vulnerabilities injected were not detected by the automatic vulnerability scanners. The authors believe that some of these false-negatives occurred because the vulnerability scanners were using a black-box approach that has trouble detecting second order vulnerabilities.

Another experiment conducted by Vieira, Antunes, and Madeira used web security scanners to detect vulnerabilities in web services to further show the limitations of scanners due to the high number of false-positives and low coverage of vulnerabilities [52]. The authors evaluated 300 publicly available web services for SQL injection, XPath injection, code execution, buffer overflow, user-name/password disclosure, and server path disclosure vulnerabilities. The number of false-positives reported was high for three out of the four scanners, and the coverage of vulnerabilities was less than 20% for two of the scanners. Further analysis found that all the XPath injection and code execution vulnerabilities were detected (no false-positives), but that all of the user-name/password disclosure vulnerabilities were in fact false-positives. Their results also showed that SQL injection is the most predominant vulnerability (accounting for almost 85% of all the vulnerabilities detected).

Fong et al. [16] developed an extensive test suite that could be configured to turn vulnera-

bility types on and off at different defense levels. The vulnerabilities that were implemented ranged from those easily exploitable to the unbreakable. Their evaluation of the test suite used both open-source and proprietary web application scanners, but the tests were limited to only cross-site scripting, SQL injection / blind SQL injection, and file inclusion. Their results showed that there is a noticeable decrease in the tools' detection ability as the level of defense increases. This suggests that the test suite is an effective method to distinguish web scanners based on their vulnerability detection rate. The authors also believe that an implementation technique, such as the one utilized, would be useful in identifying areas for tool improvement. This is the objective of the test bed approach presented in this thesis.

These evaluations confirm that current web application vulnerability scanner techniques are lacking in their approach. The high number of false-positives and false-negatives that are reported need to be reduced, and a technique to consistently achieve relatively low false-positive and false-negative rates should be developed. The design of a test bed web server that can be used to analyze these limitations is discussed in Chapter 3.

Chapter 3

Test Bed Design

Two prototypes were used to conduct the analysis on the limitations of web application vulnerability scanners. The initial test bed prototype only included 11 total instances of 10 different vulnerabilities, but the revised test bed prototype included 50 total instances of five different vulnerabilities. The initial prototype was beneficial for learning different kinds of web application technologies and vulnerabilities, but was not practical for completing a full analysis on the limitations of web application scanners. The revised prototype was therefore developed so that more instances of testable vulnerabilities could be included. A description of the initial test bed prototype is given in Section 3.1, and the design of the revised test bed prototype is discussed in Section 3.2.

3.1 Initial Test Bed Prototype

The initial test bed design is based on the WebMaven BuggyBank web application which was originally developed by David Rhoades of Maven Security and released under the GNU General Public License (GPL) [34]. WebMaven is described as an interactive learning environment for learning web application security safely and legally. The web site is designed

to mimic an online-banking application, and emulates several security flaws in its operation. BuggyBank is also used as a benchmark application for testing web vulnerability scanners and audit tools, as mentioned previously in Section 2.3.1. Several of the key client and server-side features of the initial test bed prototype are described in Section 3.1.1, the vulnerabilities and security flaws that exist in the initial prototype are reviewed in Section 3.1.2, and issues with the initial prototype design are discussed in Section 3.1.3.

3.1.1 Overview and Features

The initial prototype was installed on an Apache server and is primarily composed of a combination of different Perl CGI scripts. The web application also includes HTML pages for instructions and documentation, and back-end databases for storing session information, passwords, and account information. In order to build the CGI script that provides the main functionality of the web application, users must select which vulnerabilities they want embedded in the web application. Figure 3.1 shows a screen-shot of the PHP page that is used to build the CGI script. Depending on which vulnerabilities the users choose to

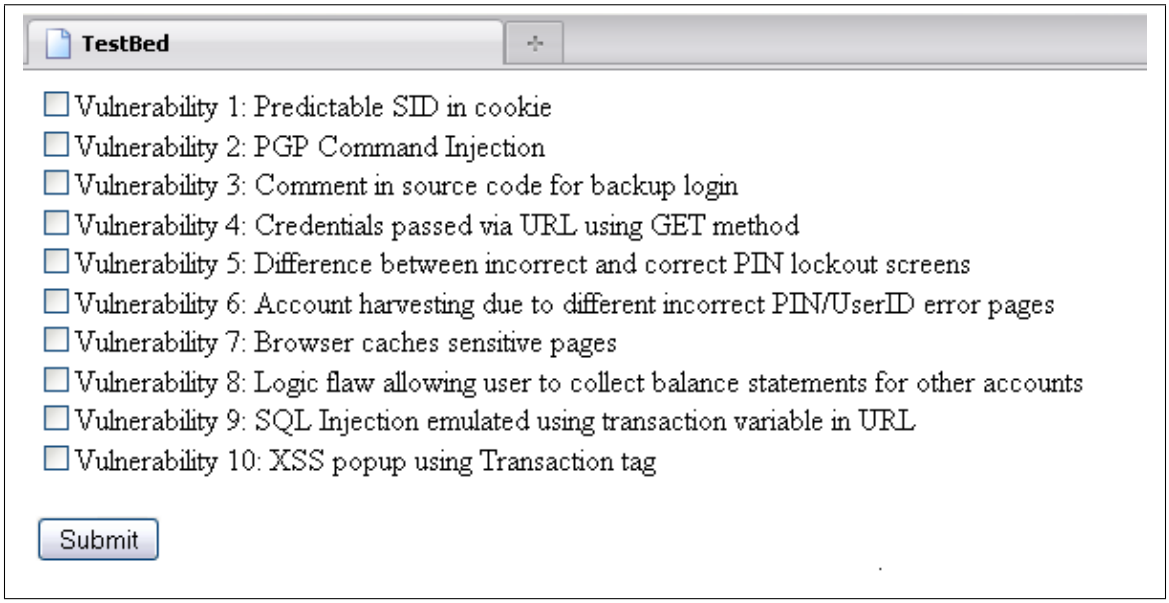
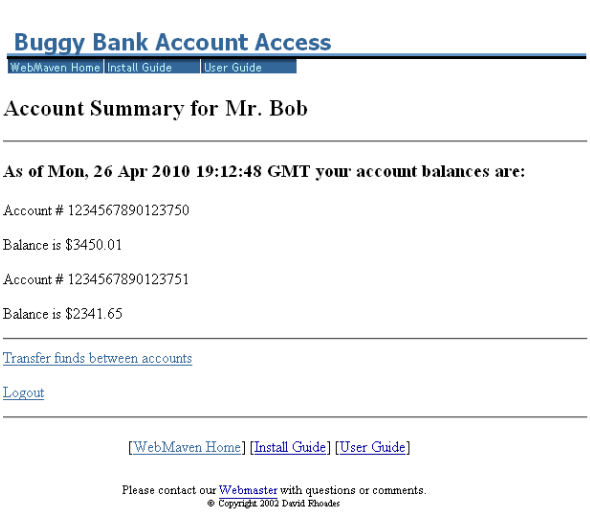


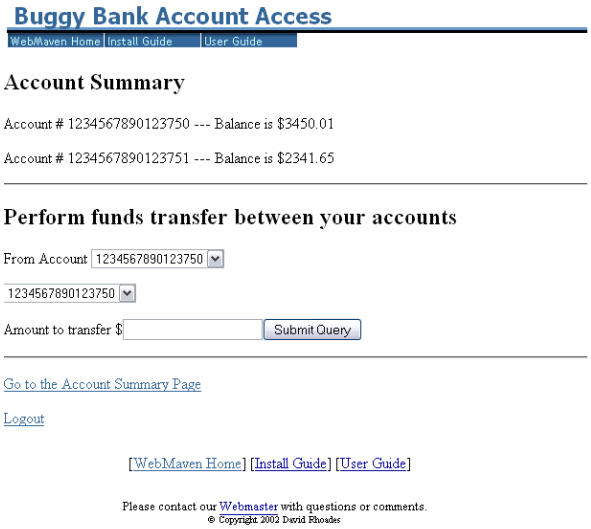
Figure 3.1: The Vulnerability Selection Page for the initial test bed prototype.

implement, different CGI scripts are selected and combined to form the complete CGI script for the application.

The main features that the initial test bed web server implemented include the ability to view a summary of a user's accounts, and the ability to transfer funds between a user's accounts. Two known accounts and their corresponding PIDs are supplied with the application to allow users to log-in to the application and interact with its functionality. The account summary page shows the current balances of all of the accounts belonging to the user who is logged in. The account transfer page allows users to select which account to transfer money from, the amount of money to transfer, and the account to transfer money to. Figure 3.2a shows a screen-shot of the account summary page, and Figure 3.2b shows a screen-shot of the account transfer page.



(a) A screenshot of the Account Summary Page.



(b) A screenshot of the Account Transfer Page.

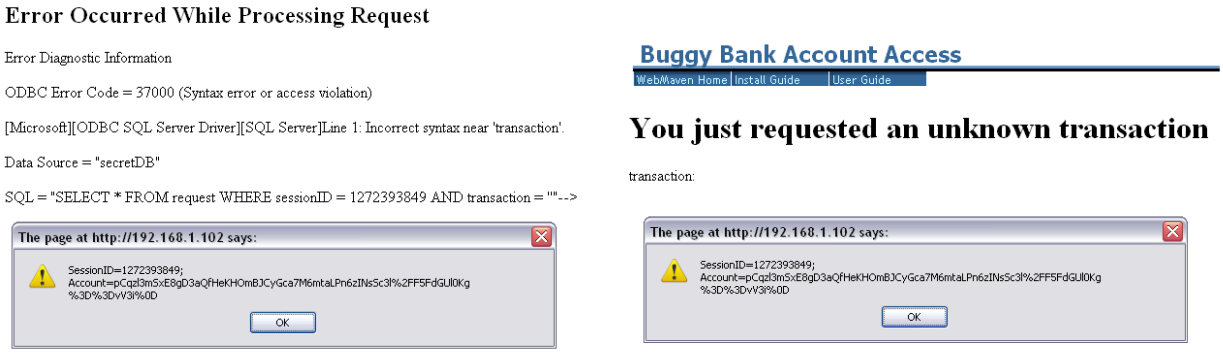
Figure 3.2: The main features of the initial test bed web server prototype.

3.1.2 Vulnerabilities Implemented

The initial test bed prototype is a modified version of the BuggyBank web application that allows users to choose which vulnerabilities and security flaws to include in the application. The list, which can be seen in Figure 3.1, includes vulnerabilities and flaws that can lead to information disclosure, account hijacking, and system compromise. There are several security flaws and vulnerabilities embedded in the web application, but only SQL injection, XSS, and session management attacks are of interest for the analysis of web application scanner techniques. The flaws and vulnerabilities involving source code comments, insecure passing of credentials, and browser caching can be detected by web application vulnerability scanners, but are trivial to find and not of interest. The security risks involving logic flaws, and PIN or User ID harvesting are more obscure vulnerabilities, and require a greater level of artificial intelligence than current web application vulnerability scanners possess in order to be detected. All of these vulnerabilities and flaws represent security risks however, and should be considered when reviewing web applications since not all vulnerabilities are detectable by web application vulnerability scanners.

One of the embedded vulnerabilities in BuggyBank that is detectable by web application scanners is XSS. In this web application, XSS attacks can occur due to improper input validation on the transaction tag. The transaction tag is not scrubbed for malicious characters, and therefore allows malicious script to be executed. Figures 3.3a and 3.3b show the two cases where XSS vulnerabilities exist in the web application due to this insecure practice.

Another vulnerability that exists in the BuggyBank web application is a result of the predictable session ID which is placed in the cookie. The session ID is generated by taking the current epoch time and converting it to a substring of only the first 10 characters. Because the session ID is being created in this way, session hijacking is very easy since the session ID corresponds to the time in seconds when the user was logged in. For example, the session ID with the epoch value 1272400834 corresponds to the date 27 April 2010 16:40:34, and the session ID with epoch value 1272400835 corresponds to the date 27 April 2010 16:40:35.



(a) The reflected XSS vulnerability in the SQL Error Page

(b) The reflected XSS vulnerability due to an unknown transaction.

Figure 3.3: The XSS vulnerabilities in the initial prototype.

The last vulnerability of interest that is present in the BuggyBank web application is SQL injection. When the transaction tag is given an SQL statement instead of a legitimate transaction, a database error screen is reported. Although this SQL injection attack does not report any user or account information, it does disclose information about the database and server. Disclosing this type of information is dangerous because it can be used to devise future attacks specifically aimed at those types of systems. Figure 3.4 shows the error page caused by SQL injection in the BuggyBank server.

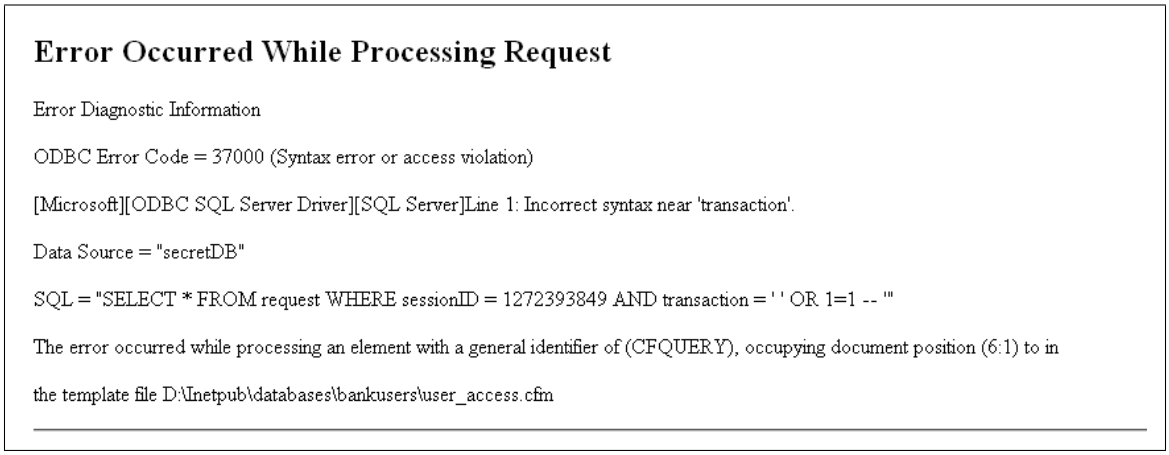


Figure 3.4: The SQL injection vulnerability in the initial prototype.

3.1.3 Issues

This initial prototype system works well as a benchmark application for testing and detecting web application vulnerabilities. However, it is not suitable for a full analysis of web application vulnerability scanner techniques because of its lack of features and functionality. This lack of functionality results in only being able to analyze four instances of vulnerabilities, which is not enough to achieve meaningful results. A revised test bed prototype that included many more instances of vulnerabilities was needed so that this issue could be resolved.

Also, the initial test bed web server implemented some vulnerabilities that are not detectable by web application scanners, and implemented some vulnerabilities that are not the most crucial to web application security. The vulnerabilities that have been identified as the most critical for detection by web application vulnerability scanners are SQL injection, XSS, session management, buffer overflow, and malicious file execution attacks. The initial prototype only included three of these vulnerabilities, but included seven other vulnerabilities that are either trivial and easy to find, or complicated and impossible to be detected by web application scanners.

Additionally, the web application technologies that were used to create the BuggyBank web application were not the most popular or current technologies used on the web. Web application vulnerability scanners look for technologies such as JavaScript, PHP, and MySQL. This issue is addressed in the revised test bed prototype because the more popular technologies, such as these, are represented.

3.2 Revised Test Bed Prototype

The revised test bed prototype is based on a web application called Hokie Exchange. Hokie Exchange is a notional community site where users submit, assess, categorize, and purchase

different hi-tech products. Hokie Exchange implements several of the most popular vulnerabilities found in web applications. To provide a sufficient number of vulnerabilities for web application scanners to search for, multiple instances of these vulnerabilities were placed throughout the application. An overview of the Hokie Exchange web application is given in Section 3.2.1. Key client features of the revised test bed prototype are described in Section 3.2.2, and key server features of the revised test bed prototype are described in Section 3.2.3. The type and number of vulnerabilities that exist in the web application are presented in Section 3.2.4.

3.2.1 Test Bed Overview

The test bed web server consists of both a secure version and an insecure version of the Hokie Exchange web application. The purpose of the secure version of the web application is primarily to test for and analyze false-positive results. The secure version of the revised test bed prototype does not include any intentional implementations of the five vulnerability types analyzed in this thesis. Since the secure version of the web application does not have any of these vulnerabilities implemented, the results that the web application scanners produce should only contain false-positives. However, other types of vulnerabilities exist in the web application because the test bed web server uses out-of-date versions of web server technologies, which include security holes. Design flaws and coding mis-practices likely result in unintended vulnerabilities as well. These unintentional vulnerabilities are not necessarily a drawback, because they provide more data for the analysis of web application scanner techniques.

The purpose of the insecure version of the web application is to produce false-negative results for analysis. Five different types of vulnerabilities are intentionally implemented in this version of Hokie Exchange. False negatives occur when the web application vulnerability scanners do not detect all of these intentionally implemented vulnerabilities. Even though the majority of the results obtained during the tests of the insecure version of the web application

should be false-negatives, false-positives can be observed as well. If a web application scanner marks something as a vulnerability when it actually is not, the result will be categorized as a false-positive.

The many client-side and server-side features of the Hokie Exchange web application are discussed in the following two sections. Even though there is both an insecure and secure version of the web application, the overall functionality of the client and server features remains the same. The versions are only modified to add or remove vulnerabilities.

3.2.2 Client-Side Features

The main client-side features of the Hokie Exchange web application include the ability to submit, assess, and categorize hi-tech items. The first page that users are shown when visiting the Hokie Exchange web application is Index.php. This page links to Reminder.php, Register.php, and Process.php. The main purpose of the Index.php page is for users to submit log-in credentials to enter the web application.

Reminder.php allows users to enter their e-mail address to receive their account's password in a reminder e-mail. Register.php processes the usernames and passwords that are entered to register a new user for the web application. Both of these pages are accessible without logging into the web application.

Process.php is the main page of Hokie Exchange once user's are logged in. This page processes the submitted user credentials to log-in users as either administrators or normal users. Process.php links to Templates.php, Categorize.php, Assessments.php, Edit.php, and Logout.php, as well as Admin.php and Validate.php if the user is an administrator.

Templates.php works with Upload_file.php to handle the processing of uploaded file templates. File templates are XML documents compressed in either .zip or .tar formats. Categorize.php gives users the ability to categorize new items into existing categories of product types, or a new user specified category. Assessments.php allows users to view and add feed-

back on all of the categorized items in the web application, and purchase these items as well. Logout.php is also accessible through Process.php and handles the termination of a user's session.

Edit.php, along with Edit2.php, allows users to configure their account information. The users can modify their username, change their password, view their account statistics, and update the mailing address and credit card number associated with their accounts. A mailing address and credit card number must be registered with a user in order for them to purchase an item.

Administrator users can also visit Admin.php and Validate.php. Admin.php works with Admin2.php to allow administrators to upgrade regular users to administrator role, and to add other usernames and passwords to the list of users who can access uploaded item templates. The Validate.php page gives administrators the ability to approve or delete uploaded item templates, approve or deny item categorizations, and view validated item templates. A layout of the structure of the Hokie Exchange web application can be seen in Figure 3.5.

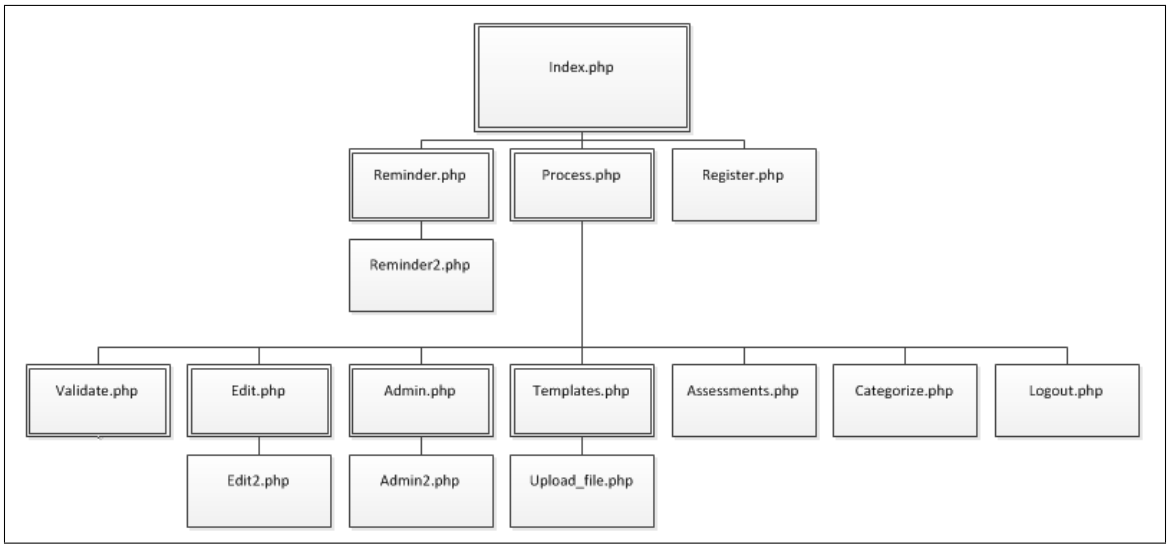


Figure 3.5: The layout of the Hokie Exchange web application.

3.2.3 Server-Side Features

The web server follows the Linux, Apache, MySQL, and PHP framework. CentOS 4.4 is the version of Linux that is used for the operating system, while Apache 2.0 is the server version, MySQL 4.1 is the database version, and PHP 4.3.9 is the scripting language version. These out-of-date versions of web server technologies were purposely selected because they include security holes that allow for the implementation of certain vulnerabilities. For instance, buffer overflow and malicious file execution attacks can be implemented since the out-of-date versions of the web technologies used on the server do not include patches for functions and operations that have been found vulnerable.

The back-end database on the web server is comprised of four primary tables: Users, Categories, Categorize, and Items. The Users table holds all of the account information for each user of the web application. The main information included in this table is the users' e-mail addresses, MD5 hashed passwords, roles, dates registered, shipping addresses, and credit card numbers.

The Categories table includes all of the categories that are available for selection to categorize newly added items. These categories include Computer, Laptop, Wireless Phone, Net Book, Video Game, Computer Accessory, Phone Accessory, Networking Device, Television, Speakers, Gaming Console, Home Entertainment System, Media Storage Device, and Other. The Other category is used for users to enter category names that are not found in this list.

Once a newly uploaded item is categorized, it moves to the Categorize table. This table includes the name of each item, the user who uploaded it, any of the submitter's comments, the proposed category for each item, and whether or not the item is awaiting approval. If an administrator does not approve an item's categorization it will be removed from this table.

On the other hand, if an item's proposed categorization is approved by an administrator, the item will move to the Items table. The Items table includes all of the items that are available for review and purchase. This table is similar to the Categorize table, and includes

each item's name, comments, category, and number sold.

Each table in the database is filled with a sufficient amount of dummy data so that the web application follows normal behavior when scanned by web application vulnerability scanners. The database contains 11 registered users (10 regular, one administrator), 10 items for purchase/assessment, 15 uncategorized items, and five categorized items waiting approval. The web server also contains five validated item template files, and five uploaded template files that are awaiting administrator approval.

3.2.4 Vulnerabilities Implemented

The revised test bed prototype has vulnerabilities intentionally implemented in the insecure version of the Hokie Exchange web application. The vulnerabilities that are implemented include SQL injection, XSS injection, session management flaws, malicious file execution, and buffer overflow vulnerabilities. All of these vulnerabilities exist in the insecure version of the web application because proper input validation is ignored in one way or another. However, the proper input validation techniques to prevent these vulnerabilities have been followed in the secure version of the Hokie Exchange web application.

The most prominent vulnerability that exists in the web application is SQL injection. There are 20 different areas in the web application where SQL injection can be carried out by injecting statements in web site forms, inputs, or cookie variables. Both error-based SQL injection and blind SQL injection can be performed against the web application since SQL responses are either reflected back to the user immediately, or after viewing other pages in the web application. An example of SQL injection which displays error information that can be used to learn the structure of a database table in the Hokie Exchange web application can be seen in Figure 3.6.

The second most prominent vulnerability in the web application is XSS injection, with 17 areas available for exploitation. These XSS vulnerabilities overlap with some of the SQL

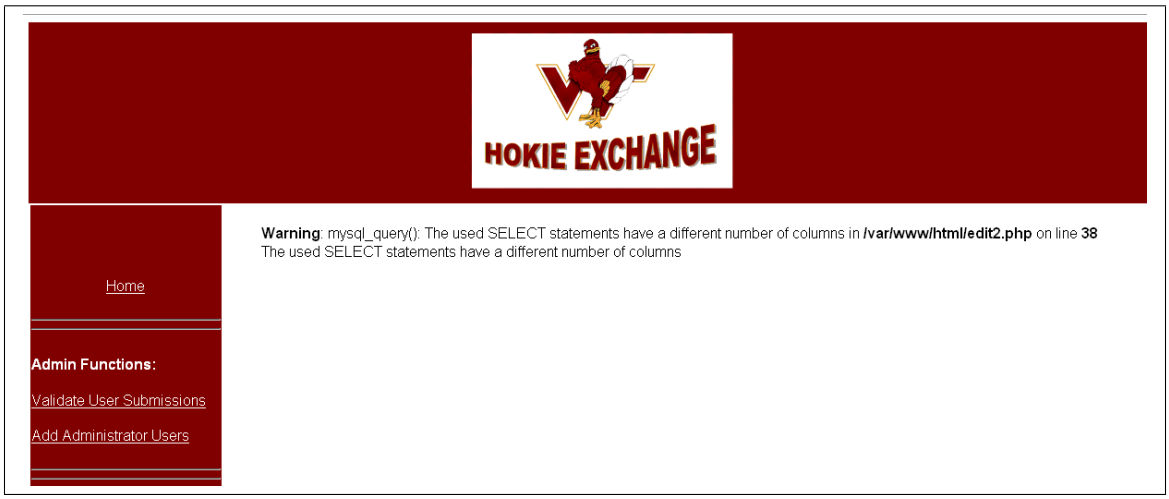


Figure 3.6: An SQL injection attack which causes an error message to be displayed that can be used to learn the structure of a table in the Hokie Exchange database.

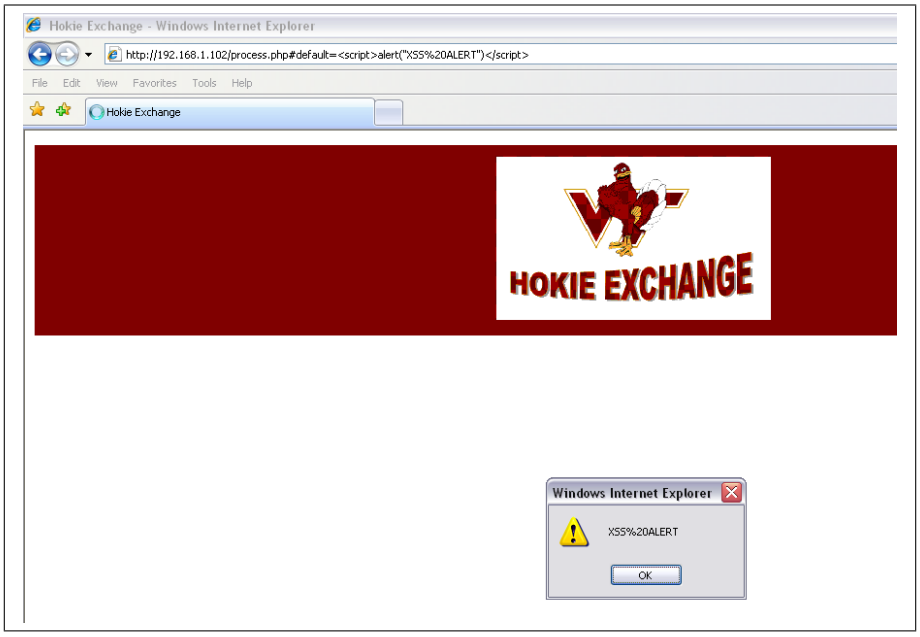
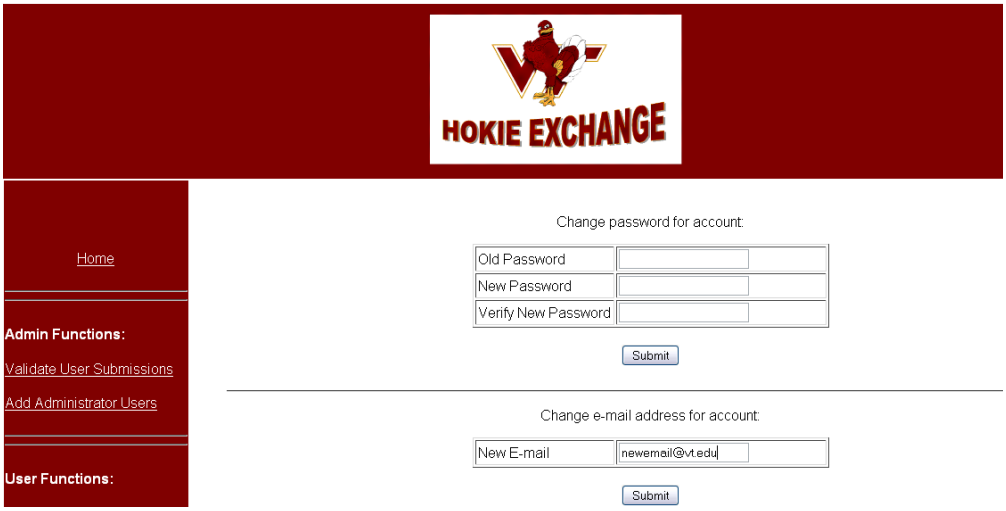
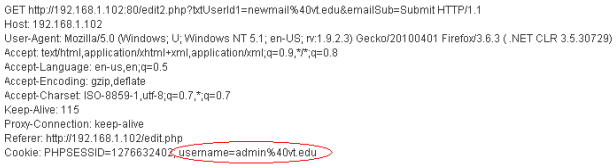


Figure 3.7: An alert box which appears due to XSS injection in the DOM of Internet Explorer.

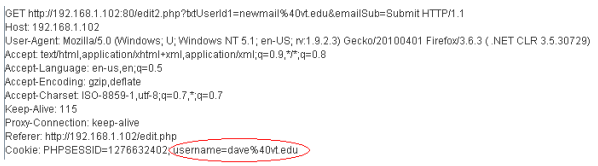
areas of injection since both of these injection techniques rely on improper validation of user supplied data. Three different types of XSS injection can be carried out against the vulnerable web application: reflected XSS, stored XSS, and DOM-based XSS. Figure 3.7 shows an example of a reflected DOM-based attack in Internet Explorer.



(a) The new email address that should change the email address of the current user.



(b) The unmodified username 'admin@vt.edu'.



(c) The modified username 'dave@vt.edu' which will result in the changing the username for 'dave@vt.edu' to 'newemail@vt.edu'.

Figure 3.8: A session management vulnerability that allows for the exploitation of unprotected cookie variables.

There are 10 cases of poor session management techniques within the insecure web application as well. The first session management vulnerability implemented is a predictable session ID. The session ID is set using a ten character representation of the current epoch time and is therefore vulnerable to a brute force session hijacking attack. Besides this vulnerability, there are nine instances in the web application where a session can be manipulated by exploiting unprotected sensitive cookie variables. The web application sets a username variable in the cookie that is used throughout the web application to handle user transactions. An attack which exploits this session management vulnerability to modify another user's account e-mail address can be seen in Figure 3.8.

There are two areas within the insecure web application that are vulnerable to malicious file execution. The type of malicious file execution vulnerability that is implemented is remote

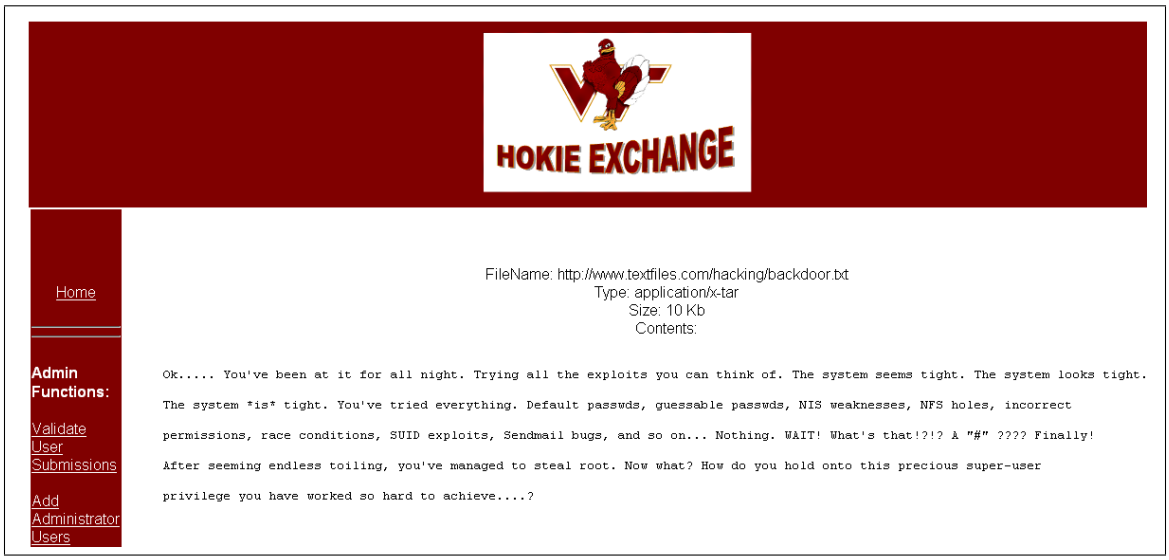


Figure 3.9: The result of an RFI attack that retrieves a remote file by exploiting the filename of an uploaded file.

file inclusion. This vulnerability exists when unexpected scripts or files from remote hosts are retrieved and executed on a web server. An example of an RFI attack that uses the filename of an archived file to retreive a remote file can be seen in Figure 3.9.

The last vulnerability that is implemented in the insecure version of the web application is a buffer overflow. Buffer overflow vulnerabilities are exploited by giving a function more data than it is expecting. The 'popen' function is vulnerable to a buffer overflow in the version of PHP used by the test bed web server. By exploiting the inputs of this function a buffer overflow attack can be executed. Figure 3.10 shows what is displayed when an unexpected input is given to the function that is vulnerable to a buffer overflow.

Several steps were taken to ensure that these vulnerabilities were correctly implemented. The PHP configuration file was modified so that all PHP and MySQL errors could be displayed, and so that the web server could request remote URLs. Furthermore, the HTTP methods were set to GETs instead of POSTs for the majority of the transactions in the web application. This modification causes the HTTP variables to be included in the URL of requests, instead of in the HTTP body. There are 50 total vulnerabilities used in the insecure version of the

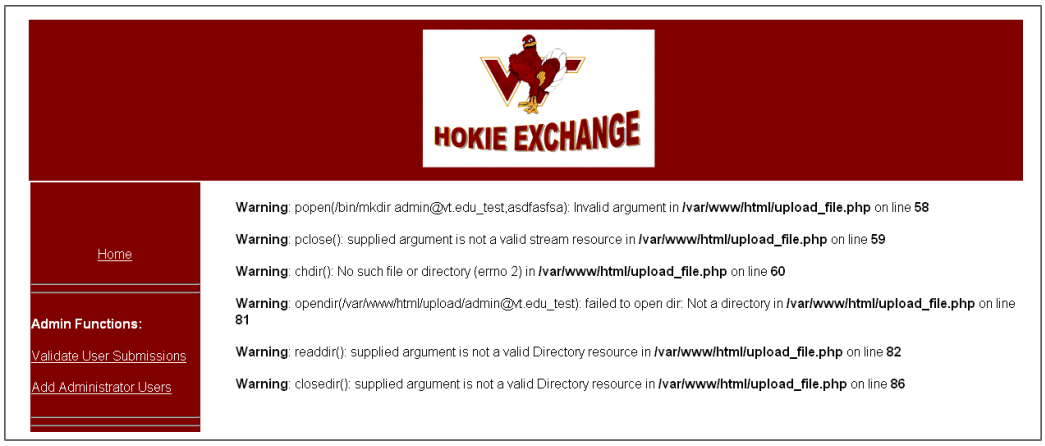


Figure 3.10: The error messages that are displayed after modifying a vulnerable input variable in a function that is susceptible to a buffer overflow.

Table 3.1: A summary of vulnerabilities implemented in the Hokie Exchange web application.

	Number Implemented
SQL Injection - Form Input	12
SQL Injection - Cookie Variable	8
XSS Injection - Reflected	10
XSS Injection - Stored	6
XSS Injection - DOM-based	1
Session - Predictable SID	1
Session - Sensitive Cookie Variable	9
Remote File Inclusion	2
Buffer Overflow	1
Total	**50**

web application to analyze web application scanner techniques. Table 3.1 summarizes the type and number of vulnerabilities implemented in the Hokie Exchange web application. The source code for the secure version of the Hokie Exchange web application can be downloaded from http://www.security.vt.edu/SL27/shelly_files/HokieExchange_Secure.zip. The source code for the insecure version of the Hokie Exchange web application can be downloaded from http://www.security.vt.edu/SL27/shelly_files/HokieExchange_Unsecure.zip. For all other questions regarding the Hokie Exchange web application, please contact the Virginia Tech IT Security Office by emailing Randy Marchany at marchany@vt.edu. Chapter 4 describes the method and approach that is used to analyze why web application scanners produce false-negative and false-positive results when scanning for these vulnerabilities.

Chapter 4

Methodology and Approach

A reliable method to unveil the shortcomings of current web application vulnerability scanner techniques needs to be followed to allow for in-depth analysis and tool improvement. To effectively analyze why web application scanner techniques are failing, it is necessary to discover how web application vulnerabilities are exploited, and understand how web vulnerability scanners work. A sufficient amount of background research has been conducted in both of these areas. Section 2.1 reviewed web vulnerability attacks and revealed that most web application vulnerabilities exist because of poor coding practices that lead to insecure areas of web applications. A method to test for vulnerability scanner limitations can be devised by knowing how to effectively attack and defend these insecure areas of web applications. Section 2.2 reviewed web application vulnerability scanners and techniques. This background research found that scanners which utilize black-box testing techniques generally attempt to locate vulnerabilities by mapping out a web application's structure, discovering the technologies which are in use, and pinpointing the areas to test for weaknesses. This background information is used to help develop a test bed web server method to analyze the limitations of web application scanners. Section 4.1 explains why this method is suitable for analyzing the limitations of web application vulnerability scanners. A thorough testing approach which aims to expose the flaws in current web vulnerability scanning techniques is

then described in Section 4.2.

4.1 Test Bed Evaluation Method

This section describes the reason why a test bed web server is used to evaluate web application scanners in order to find the limitations of their techniques. Section 4.1.1 describes the technologies that were specifically selected to build the test bed web server, Section 4.1.2 explains why the test bed method provides a high level of consistency, and finally Section 4.1.3 describes how this method benefits from the use of both white-box and black-box analysis techniques.

4.1.1 Test Bed Technologies

The vulnerabilities analyzed in this research can exist in all web application technologies, and are not language or platform specific. As such, the LAMP (Linux, Apache, MySQL, PHP) software stack has been chosen to be used as the underlying architecture of the test bed server because of the widespread use and popularity of its technologies. LAMP is a fast-growing open source enterprise software stack, which relies heavily on MySQL. MySQL has had over 100 million copies of its software distributed and is the world's most popular open source database software [42].

Apache has consistently been the most popular HTTP server since 1996. The latest web server survey conducted in January 2010 by Netcraft [31] found that Apache owns about 55% of the market share for top servers across all domains. Apache's highest percentage of market share was in late 2005 when it owned over 70% of the market share. Apache can be run on almost all operating systems, but Linux based systems allow for easy installation and management.

Another important aspect of LAMP is the programming language component. The primary

options for the language are either PHP, Perl, or Python. PHP has been selected because of its popularity, and because of its ease of integration with MySQL and Apache. According to a PHP usage statistics study conducted in 2007, there were close to 21 million Internet domains hosting web services with PHP installed. Additionally, PHP was found to be the most popular Apache HTTP Server module [51].

PHP has also been chosen because it is one of the more vulnerable web development languages, as can be seen by viewing the Common Vulnerabilities and Exposures (CVE) system [46]. The CVE is a dictionary of publicly-known information security vulnerabilities and exposures, and is maintained by MITRE Corporation through funding from the National Cyber Security Division of the United States Department of Homeland Security. The entries in this system are given unique identifiers so that they can be searched using the National Vulnerability Database (NVD) [30]. PHP related vulnerabilities in the NVD accounted for almost 30% of all vulnerabilities recorded in 2009. Also, about 82% of the labeled PHP vulnerabilities were caused by lack of input sanitation, such as SQL injection and XSS [10].

Because the most popular web technologies are being used as part of the test bed evaluation method, the analysis of web vulnerability scanner limitations can be applied to the majority of web applications and servers. Therefore, the results will be representative of current web application practices and techniques.

4.1.2 Controlled Environment

In order for an experiment to achieve meaningful results, the independent, dependent, and controlled variables need to be selected appropriately. In this test method, the web server technologies are kept constant (the controlled variables), and the number of deliberate vulnerabilities (the independent variables) is modified. By keeping the web server constant, and by changing the number of purposely implemented vulnerabilities, the vulnerabilities (the dependent variables) that are detected and omitted by the web application scanners can be observed and analyzed. This method provides a set of false-positive, and false-negative

results, that can be analyzed in order to discover what is causing these false results.

Using a single test bed web server is beneficial because it provides a controlled web application environment that will remain unchanged and consistent during the testing process. If an inconsistent method was used where web vulnerability scanners tested several web applications that had significantly different numbers of pages, server side technologies, and other levels of complexity, then analysis would be much harder to complete. Limiting the factors that change in the tests focuses only on the deficiencies of web vulnerability scanning techniques, and not on other issues that may arise due to differing variables.

The technologies (Linux, Apache, MySQL, and PHP) are kept constant, but the number of deliberate vulnerabilities will be modified. Because the source code being used was developed in-house, an expected set of results is known, and there will not be an unknown quantity of vulnerabilities. This method is an effective way in analyzing web vulnerability scanners because the vulnerabilities that are being improperly identified or omitted by the scanners can be located. From this the exact number of false-positives and false-negatives that exist in each of the scanner's results can be determined.

4.1.3 Black Box and White Box Analysis

The test method benefits from the use of both black-box and white-box analysis. Black-box analysis is a type of penetration testing where the operation of an application is analyzed in search of vulnerabilities. White-box analysis is another vulnerability detection technique that is only possible when the source code of an application is available to be inspected for security flaws [52].

Black-box analysis is the testing technique used by web application scanners to find web application vulnerabilities when the internals of the applications are not known. This type of analysis takes the perspective of a client side user, and relies on learning the behavior of a web application and manipulating different kinds of user supplied input. Black-box analysis is

used as part of the test bed method to better understand the limitations in web vulnerability scanner techniques. The ability to view the web application scanner requests and responses over HTTP, allows the vulnerabilities that are incorrectly identified or unidentified to be analyzed.

White-box analysis is generally expected to achieve better vulnerability detection rates than black-box testing because the actual source code of the web application is available for review. Since other analysis techniques [17, 44, 52] tested web application scanners against web applications that were not under the direct control of the experimenters, access to the source code was not possible, and therefore white-box analysis could not be performed. These analysis techniques were therefore unable to verify the actual number and location of web application vulnerabilities. The test bed web server does not have this limitation because it is intentionally built to have an exact number of vulnerabilities in specific locations.

Having access to the source code is extremely beneficial for other reasons as well. Not only can vulnerabilities be confirmed as correctly identified or misidentified, but the code can be analyzed to see *why* a certain vulnerability may have been misidentified or omitted. Vulnerabilities may be inappropriately discovered or ignored by web application scanners because of many factors surrounding the context, complexity, and placement of insecure code in a web application. Using a test method that uses both white-box and black-box analysis techniques will allow these limitations to be understood and will explain why vulnerabilities are being incorrectly identified.

4.2 Vulnerability Scanner Testing Approach

This section describes the approach that was taken to test the web application vulnerability scanners against the revised test bed web server. There were eight total web application vulnerability scanners evaluated for use against the web server test bed, three free/open-source and five commercial. Only six of these were selected for analysis in order to maintain

anonymity and refrain from disclosing which scanners were specifically used. The decision to use a specific web vulnerability scanner was based on the number and variety of vulnerabilities it probes for, the significance of the vulnerabilities it detects, and the scanning techniques that it uses.

Section 4.2.1 will describe the four main phases of the testing approach to analyze web vulnerability scanners, and Section 4.2.2 will identify some known limitations of the approach.

4.2.1 Testing Procedures

The experimental study consisted of four major phases: The first phase was to initialize the vulnerability scanner and web server in preparation for the tests. The second phase was to execute the tests and capture the results. The third phase was to classify the results as false-positives or false-negatives. The final phase of the study was to analyze the results and identify limitations in the web application scanner techniques.

Initialization

The setup for the experiments consisted of one computer acting as a client with a web browser, and another computer acting as a web server with both the functions of an application server and a database server. The vulnerability scanners were all installed on the single client machine and were connected over a Local Area Network (LAN) with the test bed web server. This setup allowed for the use of Wireshark [53] to capture all of the network traffic sent from the web vulnerability scanner machine to the web server, and vice versa. To reduce the amount of unnecessary traffic that was captured, the LAN did not have any other machines connected to it nor did it have Internet access. Using a packet analyzer such as Wireshark allowed for the capturing of the web vulnerability scanner requests for later analysis. A diagram of the test set up can be seen in Figure 4.1.

The procedure for initializing the test bed web server for each web vulnerability scan con-

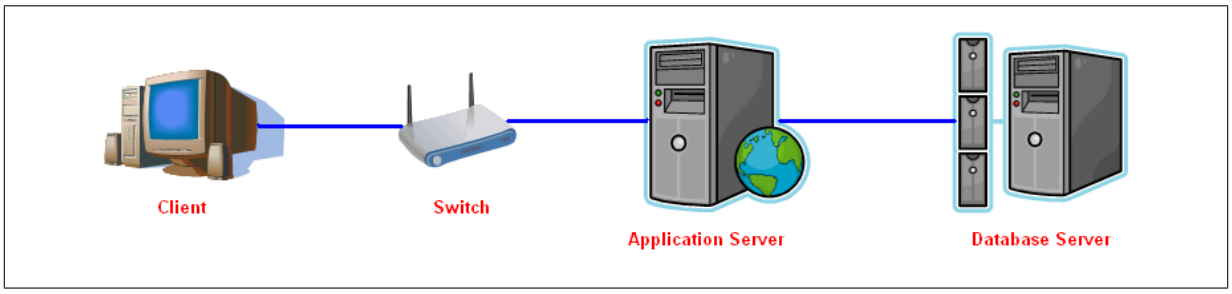


Figure 4.1: The setup for the experiments connected in a Local Area Network (LAN).

sisted of several steps. First, specific PHP files, initialization files, and configuration files were used depending on whether or not the secure version or the insecure version of the test bed web server was being tested. Once these files were in place, the initialization procedure could begin. This procedure was necessary in order to return the web server to a clean state, and ensure that no previous test runs would leave any modifications to the server. The following procedure was executed before every test:

1) Restore the MySQL database to its original state
2) Delete all client side and server side cookies
3) Restore the entire web server directory with a clean backup
4) Restart the web server

Execution

After the initialization process was complete, the testing procedure could begin. All findings during the execution of the web application scans were carefully documented and reported so that a thorough analysis could be conducted. The testing procedure consisted of the following steps:

1) Configure the web application vulnerability scanner (primarily default settings)

2) Start Wireshark packet capturing
3) Execute the web application vulnerability scan
4) Stop Wireshark packet capturing and save the trace
5) Save the contents of the database, cookie files, and scanner results

This testing procedure was repeated twice for every web application scanner; once for the secure version of the test bed web server and once for the insecure version. A total of twelve scans were completed. Each scan had its database records, cookie files, and scanner results documented and saved for analysis.

Classification

After the tests were complete, the scanner results were reviewed in order to determine if all legitimate vulnerabilities were detected or if incorrect vulnerabilities were found. If a vulnerability was reported by the scanner, but was indeed not an actual vulnerability, then it was classified as a false-positive. If a vulnerability was not detected by the web application scanner, then it was classified as a false-negative. All other vulnerabilities detected by the scanners were ignored since this analysis was only concerned with the five specific types of vulnerabilities implemented.

The objective of the secure version of the test bed web server was to primarily find false-positives. Since this version of the web server did not have any intentional vulnerabilities built into it, only incorrectly identified vulnerabilities would be present in the web vulnerability scanner results. Results were classified as false-positives if they met any of the following conditions:

1) The scan result is due to an application robustness problem (error page, format exception, etc.) and not a vulnerability
2) Normal operation of the web application results in the same error/problem

3) Source code analysis determines scan result is incorrect

4) The results duplicate a vulnerability that has already been accounted for

Note: Repeated input forms that are displayed recursively on a web page are not considered duplicate vulnerabilities if they are detected multiple times by a web application scanner (e.g. user comment boxes)

The insecure version of the test bed web server was used to primarily find false-negatives. This version of the web server had a specific number of vulnerabilities implemented so that false-negative identification could be easily computed. After reviewing each scanner's results, the number of false-negatives was calculated by subtracting the number of legitimate vulnerabilities detected by the scanner from the total number of vulnerabilities implemented in the server.

Analysis

The final phase of the experimental process was to analyze the scanner results. Analysis of the false-positives and false-negatives were conducted using the packet traces of all the HTTP requests and responses. In order to find the limitations in web vulnerability scanner techniques, the actual HTTP requests were viewed so that the scanner's approach could be discovered. Once the approach was discovered, mistakes in its functionality could be identified and explanations could be given for any limitations in its techniques.

Analysis was also conducted using the client and server cookies, database entries, and scanner test results. This analysis, as well as taking into account the context of the web application source code, assisted in determining why false-positives and false-negatives occurred. This final phase of the approach was what revealed possible improvements to web application vulnerability scanner techniques.

4.2.2 Limitations of Approach

An approach was followed that attempted to produce the most meaningful results and allow for a thorough analysis. However, there were some limitations to this approach. This research focused on web applications written in PHP and did not consider other web development languages such as Python, Perl, Ruby, .NET, and Java since they are not as popular. Therefore, the results found may only be specific to the configuration used (LAMP). For more generalized results and analysis, additional web based technologies will need to be considered. Regardless, the results obtained are still significant since the LAMP configuration is the most popular web server configuration, as explained earlier.

Also, this approach did not take into consideration vulnerabilities that occur in popular dynamic based web technologies. Technologies such as AJAX, Flash, and image maps are not present in this implementation of the web server and therefore cannot be analyzed. JavaScript was the only dynamic technology implemented, because there are only a select few web application scanners that check for vulnerabilities in these technologies. However, since these dynamic technologies are becoming more and more popular, all web application scanners should search for their security flaws. Future studies that analyze the limitations of web application scanner techniques will need to consider the vulnerabilities caused by these technologies.

Another limitation of this approach is that vulnerabilities could be accidentally included in the web application. Scanner results could include vulnerabilities that were not intentionally put into the web server, and could cause the false-positive and false-negative results to be inaccurate. Because of this limitation, a detailed analysis of scanner results needed to be completed. If a vulnerability existed that had not been included in the original known number of vulnerabilities, then the known number of vulnerabilities would need to be adjusted to reflect the addition of this new vulnerability.

The last limitation, and probably the most significant limitation, of the approach was that the knowledge of the inner workings of the commercial web application vulnerability scanners

was not known. If the scanners were developed in house, or if access to the source code to see how their algorithms worked was available, then the analysis would have been much easier because exactly what each scanner was doing would be known. Since this was not possible, the analysis had to rely on using the packet traces, cookie files, database entries, and scanner test results instead.

Although limitations have been identified in this approach, the methodology is sound and can produce a significant study. Because these limitations have been noted now, they can be addressed in future work and in other experiments aimed at discovering deficiencies in web vulnerability scanning techniques. The results and analysis achieved by following this method and approach are presented in Chapter 5.

Chapter 5

Results

Six of the eight reviewed web application scanners were selected for the analysis of the limitations in web application scanning techniques. Both commercial and free/open-source web application scanners were evaluated. Choosing only six of these scanners for the tests and analysis was sufficient because the scanners selected covered all of vulnerabilities implemented in the insecure version of the web application. Using six out of the eight web application scanners reviewed ensured anonymity among the selected scanners as well. An overview of the test techniques used for the tests is given in Section 5.1. The results obtained after conducting the tests of the six web application vulnerability scanners is presented in Section 5.2. An analysis of the limitations observed from these tests, and corresponding improvements, are presented in the final section of this chapter, Section 5.3.

5.1 Test Techniques

Three tests were conducted for each web application vulnerability scanner. One test targeted the insecure version of the initial test bed prototype, and the other two tests targeted the secure and insecure versions of the revised test bed prototypes. The experimental approach

for each test followed the initialization procedure described in Section 4.2.1 and execution procedure described in Section 4.2.1. These procedures ensured that not only would a clean version of the web application be used for each test, but also that all of the data relevant to revealing the flaws in the web application scanner's techniques would be captured as well.

The web application scanners were configured using mostly the default settings so that they would test for the implemented vulnerabilities, as well as any other vulnerabilities for which they were programmed to scan. This allowed the scanners to perform scans without any major user-defined configuration or option management. However, if the initial test was found to be unsuccessful due to the scanner failing to spider the entire web application for all relevant links, the settings of the scanner were adjusted until it was capable (to the best of its ability) of scanning the majority of the web application. This ensured that each scanner achieved meaningful results for every type of vulnerability for which it was scanning for.

After each successful test, the scanner's results were reviewed to determine which vulnerabilities, and specifically which kind of each vulnerability, were detected. This process not only revealed what vulnerabilities were being most commonly detected, but also where scanners were looking for vulnerabilities in web applications. However, further analysis needed to be conducted to unveil why these vulnerabilities were being detected, and why the other vulnerabilities were not being detected. The vulnerabilities that were detected are presented in Section 5.2, and the analysis of these results is presented in Section 5.3.

5.2 Observations

Tests were conducted against the revised test bed prototype, as well as against the initial test bed prototype. The tests against the initial test bed prototype ensured that the correct web application scanner configuration was being used, and also provided more results to be analyzed. The results for the initial test bed prototype will be reviewed first, but since there were only three scannable vulnerabilities implemented in the initial prototype, the results

based on these tests are minimal. The results for the revised test bed prototype provided the most insight as to which vulnerabilities were successfully or unsuccessfully detected. The coverage/false-negative rates, as well as false-positive rates, are summarized for each type and sub-type of vulnerability.

In these results, the letter used to correspond to each scanner (e.g., Scanner A, Scanner B, etc...) does not represent the same scanner in each table of results. For instance, Scanner A in the tables of SQL injection false-positive and false-negative results, is not necessarily the same scanner as Scanner A in the tables of XSS injection false-positive and false-negative results. This differentiation is necessary in order to provide anonymity between the scanners and their results.

5.2.1 Initial Test Bed Prototype

The results obtained from the tests that targeted the initial test bed prototype can be seen in Table 5.1. There were 4 total instances of vulnerabilities implemented in the initial prototype. These vulnerabilities included a form-based SQL injection vulnerability, a predictable SID for session management, and two reflected XSS vulnerabilities. As it can be seen, the number of false-positives ranged from 0 to 201, and the number of false-negatives ranged from 1 to 4.

Every scanner except for Scanner E was able to detect the SQL injection vulnerability that resulted in an error page. Also, every scanner except for Scanner B was built to scan for session management vulnerabilities, but the only web application vulnerability scanner that was able to detect the predictable session ID vulnerability was Scanner D. Likewise, Scanner F was the only scanner able to detect the reflected XSS vulnerability caused by the transaction variable (See Figure 3.8c). So, even though none of the scanners were capable of detecting all of the vulnerabilities individually, all of the vulnerabilities were able to be detected by at least one scanner.

Table 5.1: Initial Test Bed Prototype Results

	Reported	False Positives	Detected	False Negatives
Scanner A	6	4 (66.7%)	2 (50%)	2 (50%)
Scanner B	2	0 (0%)	2 (66.7%)	1 (33.3%)
Scanner C	204	201 (98.5%)	3 (75%)	1 (25%)
Scanner D	10	7 (70%)	3 (75%)	1 (25%)
Scanner E	0	0 (0%)	0 (0%)	4 (100%)
Scanner F	4	2 (50%)	2 (50%)	2 (50%)

5.2.2 Revised Test Bed Prototype

The revised test bed prototype had a significant increase in the number of implemented vulnerabilities in the insecure version of its web application. Compared with the four scannable vulnerabilities implemented in the initial test bed prototype, the revised test bed prototype had a 1150% increase with its implementation of 50 scannable vulnerabilities. The false-negative and false-positive results observed when testing for these vulnerabilities in the insecure version of the web application, as well as the results observed when testing the secure version of the web application, are presented in this section.

False Negatives

Results were classified as false-negatives or false-positives by following the classification procedure described in Section 4.2.1. False-negatives were tested by using the insecure version of the web application, and were observed in every category of vulnerability, and by every single web application scanner. The DOM-based XSS and buffer overflow vulnerabilities had the highest percentage (100%) of false-negatives. The type of vulnerability which had the lowest percentage (40.3%) of false-negatives was the form input type of SQL injection, which is reviewed first.

All six of the scanners selected were capable of testing for SQL injection. The SQL injection vulnerabilities were either caused by insecure form inputs or cookie variables. Table 5.2 presents the false-negative results for the form input type of SQL injection. Every instance

Table 5.2: SQL Injection Results - Form Inputs

	Detected	Implemented	False Negatives
Scanner A	9	12	3
Scanner B	11	12	1
Scanner C	8	12	4
Scanner D	2	12	10
Scanner E	6	12	6
Scanner F	7	12	5
Total	43 (59.7%)	72	29 (40.3%)

Table 5.3: SQL Injection Results - Cookie Variables

	Detected	Implemented	False Negatives
Scanner A	3	8	5
Scanner B	0	8	8
Scanner C	0	8	8
Scanner D	0	8	8
Scanner E	0	8	8
Scanner F	0	8	8
Total	3 (6.3%)	48	45 (93.7%)

of this type of SQL injection vulnerability that was implemented in the insecure version of the web application was detected by at least one scanner. Scanner B detected the most (11 out of 12), and Scanner D detected the least (2 out of 12). The overall detection rate of the scanners was 59.7%, and the corresponding false-negative rate was thus 40.3% for this method of SQL injection.

Manipulating cookie variables to perform SQL injection was the other method of SQL injection that could be executed. There were eight instances of this type of SQL injection in the insecure version of the web application. Table 5.3 presents the false-negative results for this type of injection. Only one scanner, Scanner A, was able to detect SQL injection vulnerabilities based on cookie variable manipulation. The overall detection rate of the scanners, and corresponding false-negative rate, was therefore 6.3%, and 93.7%, respectively.

All six of the scanners selected scanned for XSS injection. The different types of XSS injection implemented in the insecure version of the web application were reflected, stored, and DOM-

Table 5.4: XSS Injection Results - Reflected

	Detected	Implemented	False Negatives
Scanner A	4	10	6
Scanner B	5	10	5
Scanner C	6	10	4
Scanner D	8	10	2
Scanner E	0	10	10
Scanner F	3	10	7
Total	26 (43.3%)	60	34 (56.7%)

Table 5.5: XSS Injection Results - Stored

	Detected	Implemented	False Negatives
Scanner A	0	6	6
Scanner B	2	6	4
Scanner C	0	6	6
Scanner D	2	6	4
Scanner E	0	6	6
Scanner F	0	6	6
Total	4 (11.1%)	36	32 (88.9%)

based. The results for the reflected XSS injection vulnerability are presented in Table 5.4. There was only one scanner, Scanner E, that was unable to detect any instances of this type of vulnerability. All other scanners detected at least three out of the ten implemented instances, and achieved an overall detection rate of 43.3% and a corresponding false-negative rate of 56.7%.

Stored XSS injection was the second type of XSS injection implemented in the insecure version of the web application. The results for this stored type of XSS injection vulnerability can be seen in Table 5.5. Although all of the scanners selected tested for XSS injection, only two were able to detect any instances of this type of XSS injection. This resulted in an overall detection rate of only 11.1% and a false-negative rate of 88.9%.

DOM-based XSS injection is different than the previous two types of XSS injection because the injected script is only handled by the client and therefore never processed by the server. This caused some problems for the web application scanners because the DOM-based type

Table 5.6: XSS Injection Results - DOM-based

	Detected	Implemented	False Negatives
Scanner A	0	1	1
Scanner B	0	1	1
Scanner C	0	1	1
Scanner D	0	1	1
Scanner E	0	1	1
Scanner F	0	1	1
Total	0 (0%)	6	6 (100%)

Table 5.7: Session Management Results - Predictable SID

	Detected	Implemented	False Negatives
Scanner A	0	1	1
Scanner B	1	1	0
Scanner C	0	1	1
Scanner D	0	1	1
Scanner E	0	1	1
Total	1 (20%)	5	4 (80%)

of XSS injection was undetectable by any of the scanners, as can be seen in Table 5.6. All six scanners searched for this type of vulnerability, but achieved a detection rate of 0%, and therefore a false-negative rate of 100%.

Five of the scanners selected checked for session management vulnerabilities. The two types of session management vulnerabilities that were implemented in the insecure version of the web application were predictable session IDs and sensitive cookie variables. The results from the predictable SID tests can be seen in Table 5.7. Only one scanner, Scanner B, reported that the cookie may contain a weakly created SID that could easily be stolen using a brute force hijacking technique. All of the other four scanners did not report anything about the predictability of the SID, and caused the detection rate to fall to 20% and the false-negative rate to reach 80%.

Table 5.8 presents the results achieved by the scanners that searched for the nine instances of insecure cookie variable vulnerabilities. Scanner A and Scanner B were both able to detect one of these vulnerabilities. They both reported the same instance of the vulnerability, which

Table 5.8: Session Management Results - Sensitive Cookie Variable

	Detected	Implemented	False Negatives
Scanner A	1	9	8
Scanner B	1	9	8
Scanner C	0	9	9
Scanner D	0	9	9
Scanner E	0	9	9
Total	2 (4.4%)	45	43 (95.6%)

Table 5.9: Malicious File Execution Results- Remote File Inclusion

	Detected	Implemented	False Negatives
Scanner A	1	2	1
Scanner B	0	2	2
Scanner C	0	2	2
Total	1 (16.7%)	6	5 (83.3%)

was the very first instance in the logging-in phase of the web application. No instances after this phase were detected resulting in a detection rate of only 4.4% and a false-negative rate of 95.6%.

The results achieved by the three scanners that checked for the remote file inclusion type of malicious file execution vulnerability are presented in Table 5.9. Two instances of remote file inclusion were implemented in the insecure web application. The only instance that was detected however was the instance in the validation page where the administrator could select the validated templates that he wanted to view. Scanner A was the only scanner that was able to detect this vulnerability, and the resulting detection rate and false-negative rate were therefore 16.7% and 83.3%, respectively.

The buffer overflow results can be seen in Table 5.10. This was the other vulnerability type that was undetectable by any of the scanners. The three scanners reported some known security issues with the versions of Apache and PHP that were being used, but did not mention the vulnerable function that was being used in the insecure version of the web application. This resulted in a detection rate of 0% and a false-negative rate of 100%.

Table 5.10: Buffer Overflow Results

	Detected	Implemented	False Negatives
Scanner A	0	1	1
Scanner B	0	1	1
Scanner C	0	1	1
Total	0 (0%)	3	3 (100%)

False Positives

The secure version of the web application was primarily used to test for false-positives. However, only one false-positive was reported for the secure version throughout all six of the web application scanner results. On the other hand, the insecure version of the web application did produce false-positive results. Table 5.11 presents the insecure version's false-positive results for SQL injection. The two scanners that reported the most SQL injection vulnerabilities, Scanner B and Scanner C, were also the only two scanners to produce false-positive results. Scanner A reported the third most SQL injection vulnerabilities, but had the highest total number of correctly identified SQL injection vulnerabilities. The overall percentage of correctly reported vulnerabilities was 79.3% and the corresponding false-positive percentage was 20.7%.

False-positives were also reported by two of the scanners that scanned for XSS injection vulnerabilities. The false-positive results for the insecure version are presented in Table 5.12. Scanner D detected the most XSS injection vulnerabilities, while Scanner A and Scanner B

Table 5.11: False Positives - SQL Injection

	Reported	Correct	False Positives
Scanner A	12	12	0
Scanner B	15	11	4
Scanner C	16	8	8
Scanner D	2	2	0
Scanner E	6	6	0
Scanner F	7	7	0
Total	58	46 (79.3%)	12 (20.7%)

Table 5.12: False Positives - XSS Injection

	Reported	Correct	False Positives
Scanner A	8	4	4
Scanner B	8	7	1
Scanner C	7	7	0
Scanner D	10	10	0
Scanner E	0	0	0
Scanner F	3	3	0
Total	36	31 (86.1%)	5 (13.9%)

Table 5.13: False Positives - Remote File Inclusion

	Reported	Correct	False Positives
Scanner A	2	1	1
Scanner B	0	0	0
Scanner C	0	0	0
Total	2	1(50%)	1 (50%)

reported the most false-positive results. These two scanners did, however, detect the second most vulnerabilities. The overall percentage of correctly reported vulnerabilities was 86.1% and the corresponding false-positive rate was 13.9%.

Table 5.13 presents the false-positive results that were reported by the three scanners that searched for remote file inclusion vulnerabilities. Scanner A was the only scanner that reported a false-positive; however it was also the only scanner that correctly detected a RFI vulnerability. The overall percentage of correctly reported vulnerabilities was thus 50%, with a corresponding false-positive percentage of also 50%.

Although five of the selected scanners searched for session management vulnerabilities, none of them produced any false-positive results. Also, no false-positives were reported from the three scanners that searched for buffer overflow vulnerabilities. An analysis of why these false-positives and false-negatives occurred is presented in the next section.

5.3 Analysis

The results presented in the previous section verify that there are indeed limitations in the current techniques used by web application vulnerability scanners. By analyzing the results and data obtained through the web application scanner tests, the main areas where web application scanners require improvements have been revealed. This analysis was conducted for each of the different types of vulnerabilities to see why the techniques used by the web application scanners produced so many false-positive and false-negative results. From this analysis, key improvements to scanner techniques have been unveiled as well. The analysis for each vulnerability type, as well as proposed scanning improvements for the web application security area they address, are explained in sections 5.3.1 through 5.3.5, and other findings are discussed in Section 5.3.6.

5.3.1 SQL Injection

The web application scanners produced both false-negative and false-positive results when scanning for SQL injection vulnerabilities. Analysis could be completed on both of these areas, even though there were many more false-negative results than false-positive results. The analysis and proposed improvements for reducing false-negatives and false-positives are discussed in this section.

False Negatives

The SQL injection vulnerabilities that were found by the web application scanners were detected primarily because SQL error pages were reflected back to the client. The scanners were then able to infer that SQL injection was possible because the error page responses contained part of the SQL string that they had tested with. However, there were some cases where the web application scanners were not able to recognize the SQL error responses in the returned pages. This may be because the techniques used by the scanners to parse the

information on the returned pages are unable to recognize all of the different types of error or warning statements. This problem can be avoided by having scanners mark any input form that causes an error or warning to be displayed as a suspected vulnerability. If this is done, at the very least, the web application auditor will be alerted as to where information disclosure can occur due to error or warning statements.

The majority of SQL injection false-negatives were caused because the web application scanners did not enter data into all of the required fields to complete a transaction. For instance, when a scanner attempted to perform a register or log-in transaction, only one of the two or three user-name/password fields was being filled out. All of these input forms need to be filled with data in order for the transaction to go through, and have its inputs processed by the server. If input forms are not being supplied with the proper data, SQL injection vulnerabilities cannot be detected because the output needed to reveal these vulnerabilities is not being returned.

The other false-negatives were caused because the scanners did not attempt to test if cookie variables were susceptible to SQL injection. This problem can be easily fixed by adding cookie variables to the list of inputs that should be checked for SQL injection. There were some scanners that did attempt to test at least some of the pages for SQL injection using cookie variables. The scanner that was able to detect some of the locations where cookie variables were susceptible to SQL injection was only able to do so in cases where other input forms on the page did not need to be completed as well . The other vulnerabilities were missed because, as mentioned before, the scanner did not complete all of the forms that were required to make the vulnerability visible.

In order for these shortcomings to be avoided, every possible combination of form inputs should be tested on every page of the web application. The web application vulnerability scanners should first crawl the web application to determine all of the inputs that are associated with each page of the application. Once this process is complete, the scanners should go through SQL injection tests for every input on a page. In order to test if every input is

vulnerable, all other inputs should be supplied with dummy data, until all combinations of requests have been performed for each transaction. The dummy data does not necessarily have to be the same for every input form, but should be used in the case where password forms are being used so that no problems arise due to passwords not matching. This technique will ensure that all combinations of requests are sent to the server, and that every input form that could be vulnerable to SQL injection is tested.

False Positives

The primary reason why false-positive results were generated during the scans for SQL injection vulnerabilities in the insecure version of the web application was because duplicate entries were being reported. Duplicates occurred because the tests were being conducted using both GET and POST methods, and also because the same instance of a vulnerability was being tested for using multiple exploit scenarios. The problem with these techniques is that even after an input area is found to be vulnerable, the scanners continue to test the same area, and therefore report that more vulnerabilities exist for an input than actually do.

Scanners should test using both the GET and POST methods, but only report a vulnerability for the one method that is used by the web application for that transaction. The exception for this case would be if both methods are supported by the web application for the transaction because they complete slightly different operations. For example, a "`GET[settings]`" transaction may just pass a username variable to view an account's settings, but a "`POST[settings]`" transaction may pass a users' custom-defined configuration settings to the server as well, therefore performing a slightly different operation. Also, scanners should not report that multiple vulnerabilities exist if several exploit scenarios succeed on the same input area. Using multiple exploits to verify that an input area is indeed vulnerable is recommended, but only one vulnerability should be reported in the end.

The one false-positive observed in the tests against the secure version of the web application was due to a limitation in the blind SQL injection technique used by the scanner. Since the

injected SQL string did not return an HTML page that was different from the HTML page returned without the injected string, the scanner marked the input as being susceptible to SQL injection. The reasoning behind this is that a different HTML page should be returned when an invalid parameter is supplied to an input form that is not vulnerable. However, in this case, the injected SQL string was being escaped before being entered into the database, and was then not being reflected back on the page.

Another technique that could be used to test if an input is susceptible to blind SQL injection is to use a time delay statement as part of the SQL query. In this type of test, the database will delay returning the response page if the input is indeed vulnerable to SQL injection. If the time needed to receive the response is not the same as the configured test time, then either the input is not vulnerable or the database does not perform time delays, and other similar techniques should be used. The improved techniques presented in this section will still report SQL injection vulnerabilities, but will also reduce the number of false-positives caused by duplicates or blind SQL injection issues.

5.3.2 XSS Injection

False-negatives and false-positives were also both produced during the web application scanners' tests for XSS injection vulnerabilities. The analysis on XSS injection showed that the false-negatives and false-positives were being caused for very similar reasons as the SQL injection false-negatives and false-positives. This is because both types of vulnerabilities are caused by using injection, and therefore rely on similar methods to both test for, and detect, vulnerable inputs. The analysis and techniques for reducing false-negatives and false-positives are discussed in this section.

False Negatives

When web application scanners detected XSS injection vulnerabilities it was because the injection strings they tested with were being reflected in response pages that the scanners could easily parse. Even when stored XSS vulnerabilities were detected, it was because the injected script included SQL characters that caused an SQL error page to be reported back. The injected XSS string that was used in these tests would then also be displayed on the error page and cause the web application scanner to report it as a reflected XSS vulnerability. In these cases, the scanners detected the input form where the stored XSS vulnerability was possible. However, it was being detected through reflected techniques. A drawback of this technique is that in order for it to work, the input must not only be vulnerable to XSS, but also to SQL injection.

Although this method will work when a XSS injection string is being directly stored in a SQL database, it will not work for every single instance where stored XSS is possible. The web application scanners should use a unique injection string for each test so that they can parse every page in the web application to determine if that unique test string is being stored and then reflected on another page. If it finds the string and detects that it was properly escaped, then it will not be reported as a vulnerability. However, if it finds the string and it is in the exact format that it was tested with, then it is indeed a vulnerability. The web application scanners should also crawl the web application multiple times in search of the stored XSS injection strings. In many cases stored XSS injection vulnerabilities do not become apparent until the application is accessed many times.

Just like the main cause for SQL injection false-negatives, the main reason XSS injection false-negatives occurred was because the web application scanners were not completing all of the required fields to reveal a vulnerability. Only one of the scanners was capable of detecting the reflected XSS vulnerability in the account registration transaction, where all three of the fields needed to be supplied with data in order to expose the vulnerability. Similar false-negatives were also caused by the web application scanners because they did

not test all of the possible user supplied input forms or variables on a web page. Specifically, the "`#default=`" variable for the DOM-based XSS injection was not tested by any of the web application scanners for vulnerabilities. Since this parameter is not included in the requests to the server, the scanners must not record it as a testable parameter. Therefore they do not check it for vulnerabilities. Cookie variables and uploadable files were overlooked by most of the scanners when testing for XSS injection as well. None of the scanners found the reflected XSS injection vulnerability that could be detected by inserting malicious script into the contents of a file, and only one scanner found the vulnerability caused by inserting a XSS injection string as part of the name of an uploaded file.

The same techniques used to reduce SQL false-negatives can also be used here. The method of attempting every possible combination of form inputs should be used to test every page of a web application. All parameters should be tested for injection, including cookie variables, filenames, and file contents. To test for DOM-based XSS injection, the scanners should search every page for embedded scripts and then attempt to exploit the parameters that are used in those scripts.

False Positives

The primary reason why false-positives occurred when web application scanners tested for XSS injection vulnerabilities was because of duplicates. Scanners reported the same input form as being vulnerable multiple times because they continued to test the input form even after it had already been reported as being susceptible to a XSS vulnerability. In one case, a scanner recognized that an input was vulnerable with one set of parameters, but then tested it again by changing some of the other parameters on the page. The same exploit string was being used, but since the request looked different, due to the other changed parameters, the scanner reported the same input form as having two vulnerabilities.

The same techniques used to reduce SQL injection false-positives due to duplicates can be used here as well. All types and combinations of input parameters should be checked for XSS

injection. Once an input is found to be vulnerable, it should not be reported more than once. Completing multiple tests on the same input is a good practice for verification, but reporting that the same input is responsible for multiple XSS injection vulnerabilities is incorrect, and skews the vulnerability results. Following this technique will reduce false-positives, but will also maintain the same coverage of vulnerability detection.

5.3.3 Session Management

Numerous false-negative results were observed in the session management vulnerability area because only two web application scanners were capable of detecting session management vulnerabilities. These two scanners were also only capable of discovering three of the implemented vulnerabilities. Even though the poor performance of the web application scanners resulted in many false-negatives to analyze, there were no false-positives to analyze. The analysis on the session management false-negatives, as well as proposed false-negative reduction techniques, are presented in this section. Since no false-positives occurred, no analysis could be conducted on how to reduce session management false-positives.

False Negatives

Nearly all of the scanners that checked for session management vulnerabilities determined that the cookies were not marked as HttpOnly, meaning that the cookie variables could be read by client-side scripts. Even though the web application scanners were able to determine that the cookie variables were unprotected, most of the scanners did not attempt to manipulate them. The scanners that did attempt to manipulate session variables only did so by using injection strings that tested if file or information disclosure was possible. The scanners did not, however, check to see if the cookie variables could be manipulated in a way that would change the outcome of a transaction in favor of another user.

The web application scanner that detected the unprotected cookie variable vulnerability was

only able to do so because it inspected the cookie variables and information on its initial connection to the web application. From this initial connection, the scanner was also able to detect that it may be possible to use a brute force technique to hijack the session ID. Although the scanner was capable of detecting the unprotected cookie variable in the initial connection, it was not able to detect where the other unprotected cookie variables could be manipulated throughout the rest of the application. This limitation resulted because the scanner only tested for session management vulnerabilities using the information obtained during the initial connection to the application, and not throughout the rest of the application.

In order to avoid these shortcomings, web application scanners should attempt to tamper with all unprotected session variables throughout the application to see if a session can be manipulated. Similar to the techniques proposed to improve SQL injection and XSS vulnerability detection, scanners should add cookie variables to the list of inputs to be checked for application vulnerabilities. In these types of tests, cookie variables should be altered when completing different transactions in the web application to see if the altered variables cause the server to respond with different outputs. If a different response is returned than what is normally expected (besides a log-in or log-out page), the variable can be reported as being vulnerable. Also, scanners should analyze the session IDs used for connecting to the web application. The IDs should be analyzed to make sure that they use a strong enough algorithm to protect against brute-force attacks. These proposed techniques will better detect session management vulnerabilities, and will in turn reduce the number of false-negative results.

5.3.4 Malicious File Execution

Even though only three web application scanners searched for malicious file execution vulnerabilities, both false-positives and false-negatives were reported. An analysis of the false-negatives and false-positives is discussed in this section. Improved techniques to reduce false-negatives and false-positives are proposed in this section as well.

False Negatives

Only Scanner A was capable of detecting a remote file inclusion vulnerability. This type of malicious file execution vulnerability was detected because the scanner set the parameter value of the vulnerable input to a remote file. The remote file was then requested and returned as the output of the server response. This scanner was also capable of testing the other parameter where remote file inclusion was possible as well. However, it could not detect this second vulnerability because in order for the remote file inclusion to be found in this case, an archived file needed to be uploaded to the server. Once this file is unarchived by the server, the filename of the contained file would carry out the attack. The scanner failed to detect this remote file inclusion vulnerability because it only attempted to test the name of the uploaded file, and not the name of the archived file.

The other scanners that failed to detect the remote file inclusion vulnerabilities did so because they did not attempt to provide the susceptible parameters with inputs that would disclose the vulnerability. Similar to how session management false-negatives were observed, the scanners provided the form inputs with exploit strings that could result in local file or information disclosure, but not malicious file execution. Besides this, the web application scanners did not attempt to upload files. Though some of the scanners noted that the web application supported file uploading capabilities, and that this could result in security issues, the scanners did not attempt to test the inputs to find out if vulnerabilities actually existed. Also, there were cases where scanners failed to detect remote file inclusion because they tested with encoded or obscured URLs to retrieve remote files, instead of plaintext URLs. For instance, there were tests where the scanner attempted to use a URL embedded in JavaScript, but instead of the server responding with the expected remote file, the server responded with a “Could not open file” error.

To protect against these limitations, the scanners should attempt not only to test every possible input for remote file inclusion vulnerabilities, but should also use both obscured and plaintext URLs for the tests. Web application scanners should also attempt to upload

many different types of files to the server, if the server supports file uploading. The files that are uploaded should contain exploits in both their file contents and in their filenames so that all types of vulnerabilities are tested for, and not only malicious file execution vulnerabilities. These improved techniques will test if malicious files can be executed, and if remote files can be requested through vulnerable inputs in a web application.

False Positives

There was one false-positive observed by the web application scanners when searching for remote file inclusion vulnerabilities. The false-positive was caused because the scanner reported a duplicate remote file inclusion vulnerability. The same mistake made by the scanner that produced XSS injection false-positives was repeated, and produced a malicious file execution false-positive as well. Once the vulnerable input was detected, the scanner continued to test that same input for the same type of vulnerability. The change that the web application scanner made to the other parameters on the page caused the other requests to appear different, and therefore report a different vulnerability. This faulty technique caused the web application scanner to detect the same vulnerability twice.

The same techniques used to diminish false-positives caused by duplicates in XSS and SQL injection can be used for malicious file execution vulnerabilities as well. All possible combinations of inputs should be attempted to test for vulnerabilities; however, only one remote file inclusion vulnerability should be reported per input form. If this technique is followed, along with the techniques proposed to reduce false-negatives, false-positives will be reduced because once a vulnerable input is found, it will only be reported once.

5.3.5 Buffer Overflow

Three scanners also scanned for buffer overflow vulnerabilities in the web applications. However, none of these scanners were able to detect the implemented buffer overflow vulnerability

in the insecure version of the web application. An analysis of why these false-negatives occurred, and techniques to reduce these false-negatives are given in this section. Since no false-positives were reported for buffer overflow vulnerabilities, no analysis could be conducted on how to reduce them.

False Negatives

The scanners that checked for buffer overflow vulnerabilities did so by analyzing the HTTP header that was received from the initial connection to the web application. The scanners checked a database to see if the versions of Apache and PHP that were in use had any known buffer overflow vulnerabilities associated with them. The techniques used by the web application scanners to search for buffer overflow vulnerabilities were the same techniques used to find session management vulnerabilities. That is, the web application scanners determined the vulnerabilities by only analyzing the information obtained during the initial connection to the application, and not by scanning for vulnerabilities throughout the rest of the application. Some scanners did attempt to manipulate variables to see if file/information disclosure was possible, or if parameters could be exploited to execute operating system commands. However, these scanners did not attempt to overload variables that access the memory of the server in search of inputs that are susceptible to buffer overflow attacks.

A technique to improve how web application scanners detect buffer overflow vulnerabilities would be to attempt to perform actual overflows on the functions used in a web application. Scanners should compare the expected output of a web page that is returned during normal operation with the web page that is returned after a parameter is given an invalid input. This is similar to the technique that should be used to test if session variables are unprotected, except every type of user-supplied input should be tested to see if invalid or unexpected data causes the web application to crash or return an error page. These improved techniques will test for zero-day vulnerabilities since the scanners will be actively scanning the web application for vulnerable inputs. This will reduce the number of false-negatives that are

caused by scanners missing vulnerabilities that are unique to custom web applications, and therefore not listed in vulnerability databases.

5.3.6 Other Findings

The exact reason why false-negatives occurred was difficult to determine in some cases because the techniques that the web application scanners used to detect vulnerabilities were not completely known. There were some tests where an injected condition or technique should have caused the server to respond with an output that would have revealed the vulnerability, but for some reason did not. There were also cases where the web application scanners were executing the correct tests and were causing the vulnerable output to be reflected back in the response page, but were unable to recognize the string that confirmed the vulnerability. For these instances, it could not be completely explained why the false-negatives occurred.

There are possible explanations, however, for these observed false-negatives. For instance, these results may be due to the overload of requests that the server has to handle. The overload that the server is faced with may result in it not operating the way it is supposed to, and therefore, not revealing all of the vulnerabilities that it should. On the other hand, the false-negatives could be due to the fact that the scanners are not handling all of the responses they are receiving. Web application scanners may need to adjust their techniques so that they can monitor the current connection speed that they have with the server. By doing so, they can throttle the number of connections that they make to the server. These techniques will improve the scanner's vulnerability detection because all of the responses will be handled correctly, and thus matched correctly with their corresponding requests.

Other reasons why scanners may have missed vulnerabilities could be due to failures in their spidering techniques. These types of failures would have caused the scanners not to be able to grab all of the links and parameters when crawling through a web application. For instance, there were instances when web application scanners did not even check the initial log-in page for vulnerabilities. The scanners only looked for vulnerabilities that could be detected once

logged in to the web application, and did not consider vulnerabilities that could be located on the log-in page itself. Scanners should always check the log-in page for vulnerabilities. They should also improve their spidering techniques so that all pages and variables within the scope of the web application are tested. A summary of the contributions of this research, as well as areas of future work, are reviewed in the final chapter, Chapter 6.

Chapter 6

Conclusion

This research has shown that a test bed web server hosting both secure and insecure versions of web applications is a suitable method for determining why web application vulnerability scanners produce false-negative and false-positive results. Web application scanners have been found to perform relatively well when detecting the most simple kinds of SQL and XSS injections. However, much work still needs to be done to improve scanner techniques to detect non-traditional instances of these vulnerabilities, as well as session management, malicious file execution, and buffer overflow vulnerabilities. From the analysis of false-negatives and false-positives, suggestions could be noted that would improve web application scanner techniques.

There was some deviation from the expected results, however. It was believed that the secure version of the web application would produce the majority of the false-positive results, but this turned out not to be the case. Most of the false-positives that were observed were due to scanners duplicating vulnerabilities while testing the insecure version of the web application. The lack of false-positives observed in the tests of the secure version of the web application can be attributed to the fact that the techniques used by the web application scanners were unable to find a majority of the vulnerabilities implemented in the insecure version of the web application. Therefore, the techniques used by the web application scanners can be

assumed to not test enough of the input parameters to mistakenly categorize something as a vulnerability. The lack of the number of tests of user supplied input therefore drives the number of false-positives down.

However, when it comes to web application security, false-positives should be preferred over false-negatives. If the techniques used by the web application scanners locate a majority of the actual implemented vulnerabilities, the security cost associated with reporting extra false-positives will be low, and the overall security of the web application will be high. If this is the case, the web application administrator or auditor will be presented with a list of vulnerabilities that cover a majority of the actual vulnerabilities, and will only have to review the results to determine which are false-positives and not actual vulnerabilities.

The results obtained from this research also suggest that it is better to use multiple web application scanners when testing applications for security vulnerabilities. The web application scanners achieved much better vulnerability detection rates combined than did any one individually. The contributions of this thesis are further discussed in Section 6.1, and directions for future work of this research are discussed in Section 6.2.

6.1 Contributions

This research made three significant contributions. The first being that this research developed both a secure and insecure version of a web application that could be used for web application scanner benchmark testing. The secure version of the web application can be used for detecting false-positive results, since no vulnerabilities are actually implemented in this version. The insecure version of the web application can be used for detecting false-negative results because the number and type of vulnerabilities that are implemented is known.

The second major contribution that this thesis presents is a method to analyze the limitations of web application vulnerability scanners. The limitations, being false-negatives and false-

positives, were successfully documented and the reasons behind why they occurred were discovered. The method presented has been verified to produce results and can be applied to other types of web application vulnerabilities and web application scanners.

Improvements for web application scanner techniques were also identified, and are the third major contribution of this research. Techniques to reduce the number of false-negatives and false-positives were suggested for every type of vulnerability that produced these results. These improved techniques can be applied to web application scanners so that they can achieve better detection rates, and therefore help prevent web application attacks.

6.2 Future Work

In the future, this research can expand to include an analysis of other web application technologies and server configurations. One of the limitations that has been noted to the approach used in this thesis is that it only considers one specific technology framework. Future work can include the evaluation of other configurations so that results can be obtained from all of the technologies represented in web applications. Other Web 2.0 technologies, such as Ajax, Flash, Ruby, and Python, can be included once enough web application vulnerability scanners are reviewed that include them in the list of technologies that they search for. The popularity of these Web 2.0 technologies should increase in the near future, and will therefore play a major part in where vulnerabilities exist in web applications. By increasing the functionality and complexity of the test bed web server, it will more accurately mimic the design of web applications of today and of the future.

Another area of future work is to use the analysis produced on the limitations of web application scanners to develop a black-box web application scanner. The improved techniques that are suggested in this thesis can be followed to create a scanner that achieves better performance than the current tools in the market. By limiting the number of false-positives and false-negatives produced, the scanner created will out-perform its competition.

Short term goals for future work include submitting papers developed from this research to conferences related to web application security. The first targeted conference is OWASP AppSec DC 2010. This conference contains tracks for web application security testing, vulnerabilities/exploits in the web application world, and application security case studies. The paper prepared for this conference will also be shared with the organizations that provided trial licenses of their software for this research. The organizations that provided trial licenses were interested in how their products performed, but were also interested in any suggested improvements that would come out of this research. Lastly, it is expected that if accepted to the OWASP AppSec DC 2010 conference, the secure and insecure versions of the web application will then be included as a supported OWASP project. All of the source code and related web application information has been documented so that future work can continue from where this research left off.

Bibliography

[1] Acunetix. Acunetix Web Application Security. Available at: `http://www.acunetix.com/`, 2010.

[2] C. Anley. Advanced SQL Injection In SQL Server Applications, 2002.

[3] N. Antunes and M. Vieira. Detecting SQL Injection Vulnerabilities in Web Services. In *Dependable Computing, 2009. LADC '09. Fourth Latin-American Symposium on*, pages 17–24, Sept. 2009.

[4] R. Auger. The Web Application Security Consortium - Remote File Inclusion. Available at: `http://projects.webappsec.org/Remote-File-Inclusion`, 2009.

[5] BuyServers Ltd. Falcove Web Vulnerability Scanner. Available at: `http://www.buyservers.net`, 2008.

[6] D. Byrne and E. Duprey. Grendel-Scan. Available at: `http://www.grendel-scan.com/`.

[7] Carahsoft Technology Corp. HP WebInspect software. Available at: `http://www.carahsoft.com/hp/products/webinspect`, 2009.

[8] Cenzic, Inc. Hailstorm Core and Hailstorm Starter. Available at: `http://www.cenzic.com`, 2010.

[9] CodeScan Labs. CodeScan Developer - Security at the Source. Available at: `http://www.codescan.com/`, 2009.

[10] F. Coelho. PHP-related vulnerabilities on the National Vulnerability Database . Available at: `http://www.coelho.net/php_cve.html`, 2009.

[11] A. Conry-Murray. Web application security for all. *Network Magazine*, 20(2):31 – 51, 2005.

[12] M. Curphey. A Guide to Building Secure Web Applications and Web Services, September 2002.

[13] M. Curphey and R. Arawo. Web application security assessment tools. *Security & Privacy, IEEE*, 4(4):32–41, July-Aug. 2006.

[14] D. Endler. The Evolution of Cross-Site Scripting Attacks, May 2002.

[15] e.wiZz! PHP Buffer Overflow(popen). Available at: `http://seclists.org/bugtraq/2009/Jan/73`, January 2009.

[16] E. Fong, R. Gaucher, V. Okun, P. E. Black, and E. Dalci. Building a test suite for web application scanners. *Hawaii International Conference on System Sciences*, 0:479, 2008.

[17] J. Fonseca, M. Vieira, and H. Madeira. Testing and Comparing Web Vulnerability Scanning Tools for SQL Injection and XSS Attacks. In *Dependable Computing, 2007. PRDC 2007. 13th Pacific Rim International Symposium on*, pages 365–372, Dec. 2007.

[18] Y. W. Huang and D. Lee. Web Application Security-Past, Present, and Future. pages 183–227. 2005.

[19] Y. W. Huang, C. H. Tsai, D. Lee, and S. Y. Kuo. Non-detrimental web application security scanning. In *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, pages 219–230, Nov. 2004.

[20] IBM. Rational AppScan Standard Edition. Available at: `http://www-01.ibm.com`, 2010.

[21] S. Joshi. SQL Injection Attack and Defense. Available at: `http://www.securitydocs.com/library/3655`, September 2005.

[22] A. Klein. DOM Based Cross Site Scripting or XSS of the Third Kind. Available at: `http://www.webappsec.org/projects/articles/071105.shtml`, July 2005.

[23] G. F. Lyon. NMAP.ORG. Available at: `http://nmap.org/`.

[24] Mavituna Security Ltd. Netsparker Web Application Security Scanner. Available at: `http://www.mavitunasecurity.com`, 2010.

[25] McAfee, Inc. Hacme Bank v2.0. Available at: `http://www.foundstone.com/us/resources/proddesc/hacmebank.htm`, May 2006.

[26] McAfee, Inc. Hacme Shipping v1.0. Available at: `http://www.foundstone.com/us/resources/proddesc/hacmeshipping.htm`, June 2006.

[27] McAfee, Inc. Hacme Travel v1.0. Available at: `http://www.foundstone.com/us/resources/proddesc/hacmetravel.htm`, June 2006.

[28] J. Melbourne and D. Jorm. Penetration Testing for Web Applications (Part Three). Available at: `http://www.securityfocus.com/infocus/1722`, August 2003.

[29] N-Stalker. N-Stalker The Web Security Specialists. Available at: `http://nstalker.com`, 2010.

[30] National Institute of Standards and Technology. National Vulnerability Database. Available at: `http://web.nvd.nist.gov`, 2010.

[31] Netcraft. Market Share for Top Servers Across All Domains August 1995 - January 2010. Available at: `http://news.netcraft.com/archives/web_server_survey.html`, January 2010.

[32] NT OBJECTives. NTOSpider. Available at: `http://www.ntobjectives.com`, 2010.

[33] PortSwigger. Burp Scanner. Available at: `http://portswigger.net/`.

[34] D. Rhoades. WebMaven. Available at: `http://www.mavensecurity.com/WebMaven.php`, 2002.

[35] A. Riancho. W3AF-Web Application Attack and Audit Framework. Available at: `http://w3af.sourceforge.net/`.

[36] J. Riden, R. McGeehan, B. Engert, and M. Mueter. Know Your Enemy: Web Application Threats. Available at: `http://honeynet.org/papers/webapp`, 2007.

[37] SANS. The Top Cyber Security Risks. Available at: `http://www.sans.org/top-cyber-security-risks/`, September 2009.

[38] J. Scambray and M. Shema. Hacking Web Applications Exposed: Web Application Security Secrets and Solutions, 2002.

[39] K. Spett. Blind SQL Injection: Are Your Web Applications Vulnerable?, 2002.

[40] K. Spett. SQL Injection: Are Your Web Applications Vulnerable?, 2002.

[41] D. Stuttard and M. Pinto. The Web Application Hacker's Handbook: Discovering and Exploiting Security Flaws. Available at: `http://proquest.safaribooksonline.com/9780470170779`, October 2007.

[42] Sun Microsystems. About MySQL. Available at: `http://www.mysql.com/about/`, 2008.

[43] N. Surribas. Wapiti. Available at: `http://www.ict-romulus.eu/web/wapiti/`.

[44] L. Suto. Analyzing the accuracy and time costs of web application security scanners. February 2010.

[45] Symantec. State of Enterprise Security 2010, September 2010.

[46] The MITRE Corporation. Common Vulnerabilities and Exposures. Available at: `http://cve.mitre.org/`, 2010.

[47] The OWASP Foundation. OWASP Top 10 - 2007, 2007.

[48] The OWASP Foundation. Buffer Overflow. Available at: `http://www.owasp.org/index.php/Buffer_Overflow`, February 2009.

[49] The OWASP Foundation. DOM Based XSS. Available at: `http://www.owasp.org/index.php/DOM_Based_XSS`, October 2009.

[50] The OWASP Foundation. OWASP Top 10 - 2010, 2009.

[51] The PHP Group. Usage Stats for April 2007. Available at: `http://us.php.net/usage.php`, January 2009.

[52] M. Vieira, N. Antunes, and H. Madeira. Using web security scanners to detect vulnerabilities in web services. In *Dependable Systems & Networks, 2009. DSN '09. IEEE/IFIP International Conference on*, pages 566–571, 29 2009-July 2 2009.

[53] Wireshark Foundation. Wireshark. Available at: `http://www.wireshark.org/`, 2010.