Automated Web Scanner Report

Martin Pavlinov Zhelev

CMP320: Advanced Ethical Hacking

2022/23

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

.

Abstract

The security of web applications is very important for the modern digital landscape, because of the high reliance on web applications by businesses. If a web application is not secured properly it can lead data breaches, unauthorised access, theft of sensitive data and other malicious activities. To address this issue web applications are frequently tested by web application testers who use various tools to scan the website for vulnerabilities. This tools however take a long time to execute and are repetitive.

This report presents to development and result of an Automated Web Application scanning scripts, that was designed to automate the scanning of web applications using various tools and the subsequent report writing of the results. The script was written in Python 3.11.2, was tested on the TryHackMe platform. To develop the tools the programmer followed a systematic approach, which involved breaking down the main goal into smaller tasks that could be followed similarly to a to-do list, which ensures all features are developed effectively.

To conclude, despite of certain limitations that were faced during development such as the slow speed of Dirb and inability to test the script against real website, the script had all originally planned out features implemented. Some of the features that were implemented include ability to execute the tools, analyse their output, take screenshots of all pages that are discovered, determine if login pages exist based on the output, identify login forms in login pages, brute force the logins, and produce a neatly formatted final report in Word or PDF format.

.

Contents

.

1 INTRODUCTION

1.1 BACKGROUND

Web applications are one of the most common ways for companies to expand their business. As the reliance on web applications has grown significantly over the years, so has the importance of ensuring their security. Failing to secure a web application can allow a malicious actor to gain unauthorized access, steal sensitive information, or perform other malicious activities. Because of the high reliance on web applications, they have become a popular target for malicious attackers. “On November 23, 2022, statistics from the Statista Research Department showed that 52,212 firms throughout the globe had suffered data breaches between November 2020 and October 2021.” (EC-Council, 2022)

To ensure that all web applications that are developed are secure it is necessary they undergo web application security testing, especially if they store sensitive customer data. Web application security testing is performed by web application security experts who test the target website for common weaknesses, technical flaws or vulnerabilities that could be exploited by a malicious actor. (Natasha Urdovska, n.d.). The share of web applications containing high-severity vulnerabilities was 66 percent in 2020 and 62 percent in 2021, significantly more than in 2019 (Positive technologies, 2022).

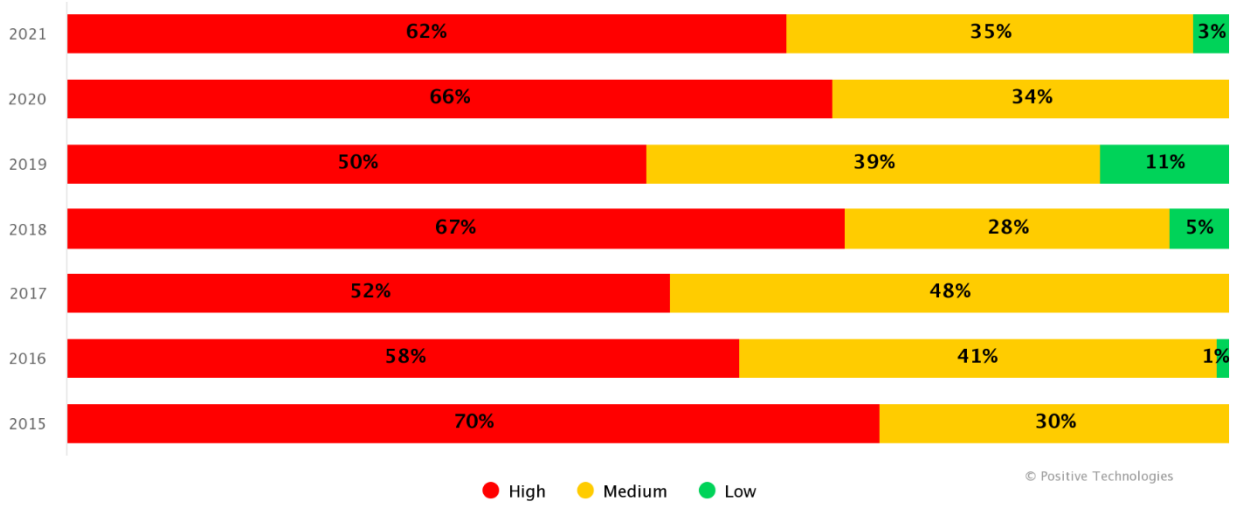

Figure 1.1 Share of vulnerable web applications by maximum vulnerability severity (Positive technologies, 2022)

The high number of vulnerabilities found in web applications show that the workload for the testers is quite significant. The web application security experts perform the testing by utilising

various tools such as Nmap, Dirb, Nikto, Hydra and others. These tools provided a way for the tester to gather information about the target web application, however they have the drawback of being very time-consuming and repetitive, especially when dealing with multiple large-scale applications. By automating the execution of the tools, the job of the tester becomes significantly easier which allows them to focus on more complex aspects of the testing such as analysing and interpreting the results. (Invicti, n.d.)

1.2 AIM

This project aims to successfully document the development of an Automated Web Application Scanning script. The main purpose of this script is to automate the scanning of various web applications by using a range of scanning tools, and to generate a report in the form of a word document. However, in order to accomplish this goal, the project was divided into several sub-aims, each focusing on specific aspects of the script's functionality and features. These sub-aims are as follows:

- Needs to be runnable in Kali Linux
- Script must be programmed in latest version of python at the time of writing – Python 3.11.2
- Has appropriate command line arguments.
- Has no GUI
- Script code is commented and understandable.
- Requires minimal user input.
- Has to successfully execute Nmap, Dirb and Nikto
- Has to take screenshot of every page discover by Dirb.
- Create a wordlist comprising of every word in the source code of website including comments.
- Discover login forms.
- Attempt brute forcing of login using word list.
- Write everything that was discovered in a neatly formatted word file.
- Has the option to create a pdf file.

2 DEVELOPMENT AND PROGRAM

2.1 DEVELOPMENT

To ease the programming the programmer split up the main goal of the script into smaller goals. Recognizing that tackling the entire task at once could lead to high complexity and potential errors, they adopted a systematic approach. This involved creating a comprehensive to-do list with all the features that needed to be implemented. The to-do list provided a clear roadmap, that ensured that all aims of the project were met. By outlining the individual tasks, the programmer could focus on solving one problem at a time, ultimately leading to a final product that functions as intended. The to-do list that was used during development was the following:

- Run Nmap, Dirb and Nikto and save their output.
- Analyse their output.
- Navigate to all pages discovered by Dirb and take a screenshot automatically.
- If Dirb or Nikto discover a login page create a wordlist comprising of the contents of webpages that were found.
- Use wordlist to brute force login page using hydra and save credentials.
- Use discovered credentials to log In and take a screenshot of what page looks like after logging in.
- Run Dirb again with the discovered credentials.
- Save output of tools, screenshots that were taken, wordlist and discovered credentials in a word file.
- Implement a function to convert docx to pdf.

Following the successful completion of the to-do list of features the programmer used the python library “argparse” to implement a help menu and the following command line switches:

- Input switch to enter target IP.
- Switch to turn on automatic login script.
- Switch to turn on screenshot taking script.
- Switch to turn on all functions.
- Switch to save a pdf file alongside the docx file.
- Switch to run only outputting to docx file part of script.

The final step was to create a requirements.txt file that contains all the dependencies that are required for the main script to be executed.

To test the script the programmer utilised the website “TryHackMe” (TryHackMe, n.d.), since it is not legal to test real websites without owner’s permission. By using the rooms on the website, the tester was able to launch different CTF rooms which have different vulnerabilities. The

script was tested against 2 different CTF rooms called “Pickle Rick” (TryHackMe & Ar33zy, 2020) and “ToolsRus” (TryHackMe & Cryillic, 2019)

The next section of the report will go over how the different features of the script were implemented by analysing snippets of the code. The full code of the program can be found in Appendix B – Full code.

2.2 PROGRAM

2.2.1 Running of tools.

The tools being ran by the script are Nmap (Nmap, n.d.), Nikto (Sullo, n.d.), Dirb (Dirb, n.d.), Hydra (Vanhauser-thc, 2022) and LibreOffice (LibreOffice, n.d.). This section will go over how and why Nmap, Nikto and Dirb were ran.

Nmap was used for scanning for vulnerabilities and open ports on the machine. Nikto was used to run a vulnerability scan on the IP. Finally, Dirb was used to find hidden directories and files on the web server.

To run the tools 3 separate functions were created that can be seen in Figure 2.1. They all take the target as an argument and create a string which contains the command and save it in a variable. Following this the command would get executed using the “subprocess” module which allows the spawning of new processes. (Python, n.d.) It can be seen that the argument “shell=True” is passed to the module. That allows the execution of the command through the shell. In the figure it can also be seen that the “runDirb” function takes an extra argument called “logged”. This argument is used to determine whether a successful login was made. If there was a successful login the command that is ran includes the found login credentials

```
# Function for running nmap
def runNmap(target):
    nmap_command = [f'sudo nmap -v -p- -A -oX {outputNmapLocation} {target}']
    subprocess.run(nmap_command, shell=True)

# Function for running nikto
def runNikto(target):
    nikto_command = [f'nikto -h {target} -output {outputNiktoLocation}']
    subprocess.run(nikto_command, shell=True)

# Function for running dirb
def runDirb(target, logged):
    # If succesfully logged in with credentials
    if logged:
        f = open(credentialsTXTLocation, "r")
        credentials = f.read()

        # Launch dirb with credentials provided
        dirbuster_command = [f'dirb http://{target} -u {credentials} -o {outputDirbLoggedLocation}']
        subprocess.run(dirbuster_command, shell = True)
    else:
        # Launch dirb without credentials
        dirbuster_command = [f'dirb http://{target} -o {outputDirbLocation}']
        subprocess.run(dirbuster_command, shell=True)
```

Figure 2.1: Functions for running of tools.

2.2.2 Getting addresses found by Dirb.

To read the addresses discovered by Dirb two functions were created that can be seen in Figure 2.2. This was done by again utilising the subprocess module. This time the module had some extra arguments passed onto it. The extra arguments “capture_output=True” and “text=True” are used to capture the result of running the command. The argument “shell=True” is used again to allow the simpler writing of the command as well as to allow the use of shell features such as pipes which are used in the command.

The command being ran first reads the Dirb output, then pipes the output into “grep” that reads the fields that contain a specified value. Finally, the “grep” output is piped to “cut” that extracts the second section from the grepped lines which as can be seen in Figure 2.3 is the address. At the end of the function the output from the subprocess is returned in a list format by using “splitlines()” which splits a string at line breaks.

```
# Get addresses from dirb output that returned 200. Addresses are displayed as http://address
def getOKAddr():
    command = [f"cat {outputDirbLocation} | grep CODE:200 | cut -d ' ' -f 2"]
    result = subprocess.run(command, capture_output=True, text=True, shell=True)
    # Split the output into lines
    return result.stdout.splitlines()

# Get addresses from dirb output that returned 401. Addresses are displayed as http://address
def getUNAUTHAddr():
    command = [f"cat {outputDirbLocation} | grep CODE:401 | cut -d ' ' -f 2"]
    result = subprocess.run(command, capture_output=True, text=True, shell=True)
    # Split the output into lines
    return result.stdout.splitlines()
```

Figure 2.2 – Function for reading of found addresses by Dirb.

```
-----------------
DIRB v2.22
By The Dark Raver
-----------------

OUTPUT_FILE: 10.10.49.72/output/outputDirb.txt
START_TIME: Mon May 29 13:34:53 2023
URL_BASE: http://10.10.49.72/
WORDLIST_FILES: /usr/share/dirb/wordlists/common.txt

-----------------

GENERATED WORDS: 4612

---- Scanning URL: http://10.10.49.72/ ----
==> DIRECTORY: http://10.10.49.72/guidelines/
+ http://10.10.49.72/index.html (CODE:200|SIZE:168)
+ http://10.10.49.72/protected (CODE:401|SIZE:458)
+ http://10.10.49.72/server-status (CODE:403|SIZE:299)

---- Entering directory: http://10.10.49.72/guidelines/ ----
+ http://10.10.49.72/guidelines/index.html (CODE:200|SIZE:51)

-----------------
END_TIME: Mon May 29 13:39:31 2023
DOWNLOADED: 9224 - FOUND: 4
```

Figure 2.3 - Dirb output

2.2.3 Parsing Nmap and Nikto XML file.

The Nmap was saved as an ".xml" file. This was done because it allows the easier reading of specific values compared to an '.txt" file by using the "XML" elements. To read the ".xml" file the following function was created which uses the module "xml.etree.ElementTree" (Python, n.d.). The function can be seen in Figure 2.4 and Figure 2.5.

```
# Function used for extracting information from nmap xml file
def extractInfoNmap(xml_file):
    # Create xml tree
    tree = ET.parse(xml_file)

    # Get root element
    root = tree.getroot()

    # Extract scan information
    scan_args = root.get('args')
    scan_start = root.get('startstr')
    scan_version = root.get('version')

    # Extract host information
    host = root.find('host')
    host_address = host.find('address').get('addr')

    # Extract open ports information
    open_ports = []
    ports = host.find('ports')
    for port in ports.findall('port'):
        ran_scripts = []
        port_id = port.get('portid')
        port_protocol = port.get('protocol')
        service = port.find('service')
        service_name = service.get('name')
        service_product = service.get('product')
        service_version = service.get('version')
        for script in port.findall('script'):
            script_id = script.get('id')
            script_output = script.get('output')
            ran_scripts.append({
                'script_id' : script_id,
                'script_output' : script_output
            })
        open_ports.append({
            'port_id': port_id,
            'protocol': port_protocol,
            'service_name': service_name,
            'service_product': service_product,
            'service_version': service_version,
            'ran_scripts': ran_scripts
        })
```

Figure 2.4 – Function for parsing Nmap "XML". 1/2

After parsing the contents of the “XML” file the function creates and returns an output string that contains the message that will be displayed in the report. Figure 2.5

```
# Extracted info
output_string = f"Scan Information:\n" \
            f"========================================================================\n" \
            f"Scan Arguments: {scan_args}\n" \
            f"Scan Start Time: {scan_start}\n" \
            f"Scan Version: {scan_version}\n" \
            f"========================================================================\n" \
            f"Host Information:\n" \
            f"Host Address: {host_address}\n" \
            f"========================================================================\n" \
            f"Open Ports:\n" \
            f"------------------------------------------------------------------------

for port in open_ports:
    output_string += f"Port: {port['port_id']}\n" \
                     f"Protocol: {port['protocol']}\n" \
                     f"Service Name: {port['service_name']}\n" \
                     f"Service Product: {port['service_product']}\n" \
                     f"Service Version: {port['service_version']}\n" \
                     f"Scripts ran:\n"

    for script in port['ran_scripts']:
        output_string += f"-> {script['script_id']}: {script['script_output']}\n"

    output_string += f"------------------------------------------------------------------------

output_string += "========================================================================"
return output_string
```

Figure 2.5– Function for parsing Nmap “XML”. 2/2

The Nikto output was also saved as an “.xml” file as such had the very similar function created for parsing of the output file and creation of an output string that contains the message that will be displayed in the report. This can be seen in Figure 2.6 and Figure 2.7

```python
# Function used for extracting information from nikto xml file
def extractInfoNikto(xml_file, niktoVulns):
    # Create xml tree
    tree = ET.parse(xml_file)

    # Get root element
    root = tree.getroot()

    # Find niktoscan element inside the root
    niktoscan = root.find('niktoscan')

    # Find scan details element inside nikoscan
    scandetails = niktoscan.find('scandetails')

    # Extract target information
    target_ip = scandetails.get('targetip')
    target_hostname = scandetails.get('targethostname')
    target_port = scandetails.get('targetport')
    starttime = scandetails.get('starttime')

    # Extract All items
    for item in scandetails.findall('item'):
        method = item.get('method')
        description = item.find('description').text.strip()
        referencesField = item.find('references')
        if referencesField.text:
            references = referencesField.text.strip()
        else:
            references = 'Reference not available'
        uri = item.find('uri').text.strip()
        niktoVulns.append({
            'method' : method,
            'description' : description,
            'uri' : uri,
            'references' : references
        })
```

Figure 2.6- Function for parsing Nikto "XML". 1/2

```python
# Extracted info
output_string = f"Scan Information:\n" \
            f"========================================================================\n" \
            f"Target IP: {target_ip}\n" \
            f"Target hostname: {target_hostname}\n" \
            f"Target port: {target_port}\n" \
            f"Scan start time: {starttime}\n" \
            f"========================================================================\n" \
            f"Items:\n" \
            f"-------------------------------------------------------------------------------------------------

for niktoVuln in niktoVulns:
    output_string += f"Method: {niktoVuln['method']}\n" \
                     f"Description: {niktoVuln['description']}\n" \
                     f"URI: {niktoVuln['uri']}\n" \
                     f"Reference: {niktoVuln['references']}\n" \
                     f"-------------------------------------------------------------------------------------------------

output_string += "========================================================================"

return output_string
```

Figure 2.7 - Function for parsing Nikto "XML'. 2/2

2.2.4 Taking screenshots of pages.

A function called “grabScreenshot” was used To take screenshots of pages. This function can be seen in Figure 2.8. It works by utilising the “selenium” module which allows the automatization of web browsers (Selenium, n.d.). It takes the list of addresses that returned code 200 (Success), launches a new browser for each address, takes a screenshot and then quits the browser.

```
# Function for taking screenshots of all pages that returned 200
def grabScrenshot(addressesOK):
    # Create directory
    os.mkdir(screenshotsDirectory)

    # Iterate over each address
    for addressOK in addressesOK:
        # Create driver
        driver = webdriver.Firefox()

        # Open address
        driver.get(addressOK)

        sleep(1)
        # Remove ":" and "/" from address using replace()
        file_name = addressOK.replace(":", ".").replace("/", ".")

        # Take a screenshot and save it
        driver.get_screenshot_as_file(f'{screenshotsDirectory}/{file_name}.png')

        # Exit driver
        driver.quit()
```

Figure 2.8 - Function for taking screenshots.

2.2.5 Checking Nikto results for login pages.

To determine whether a login page exists the Nikto vulnerabilities are checked for the existence of a specific string which indicates that a login section was found. The argument “nikto_vulns” that is passed is filled with the vulnerabilities that were taken from the Nikto output when it was extracted by the “extractInfoNikto” function that can be seen in Figure 2.9.

```
# Function used to find if nikto found a login section
def checkIfLoginNikto(nikto_vulns):
    # Checks if login page exists
    checkMessage = 'Admin login page/section found.'

    for vuln in nikto_vulns:
        # If nikto discovered login page
        if checkMessage in vuln['description']:
            return vuln['uri']
```

Figure 2.9 - Function for checking Nikto result for login pages.

2.2.6 Creation of wordlist from page contents.

If a login page was found by the function discussed in section 2.2.5 above. The script moves onto creating getting the prerequisites for the creation of a hydra command. One of these prerequisites is a wordlist.

Web applications commonly have credentials left in the comments of the source code or in the text of the webpage, because of this a function was created that would find all take all text and comments on the webpage and extract them to a ".txt" file that can be used as a wordlist. To do this the "Request" and "Beautiful Soup" python library were used. The request library allowed the retrieval of the html of the page. (Reitz, n.d.). While beautiful soup was used to parse the web page information. (Richardson, n.d.). This function can be seen in Figure 2.10 and Figure 2.11.

```
# Functions used for retriving text and comments from the pages
def getComments(addressesOK):
    print("Getting comments")

    # Create wordlist usernames file
    wordlistUsernamesFile = open(wordlistUsernamesLocation, "w+")

    # Loop for each address that returned 200
    for addressOK in addressesOK:
        # Gate all the html code for the address
        page_html = requests.get(addressOK).text

        # Parse html using beautifulsoup
        soup = BeautifulSoup(page_html, "lxml")

        # Find all comments
        comments = soup.find_all(string=lambda text: isinstance(text, Comment))

        # Loop for each comment that was found
        for comment in comments:
            # Split aech comment onto a list of words
            words = comment.split()

            # For each word in the list of words
            for word in words:
                # Write to username wordlist file
                wordlistUsernamesFile.write(word + '\n')

        # Get all the text
        text_content = soup.get_text()

        # Split the text onto a list of words
        words = text_content.split()

        # For each word in the list of words
        for word in words:
            # Write to username wordlist file
            wordlistUsernamesFile.write(word + '\n')
```

Figure 2.10 - Function for creating wordlist from page contents. 1/2

As can be seen in Figure 2.11 after successfully saving all the scraped words in a "wordlistUsernames.txt" file a second file called "wordistPasswords.txt" that has all the text and comments that were found as well as the contents of "rockyou.txt" appended to it was created. The file "rockyou.txt" contains a list of commonly used passwords. Because the script tries to brute force both the username and password, the contents of "rockyou.txt" are appended only to the password wordlist file to reduce the time brute forcing the login page would take. This is done because brute forcing both username and password with "rockyou.txt" would be unfeasible and impractical.

```
# Sets the position in the file to the start
wordlistUsernamesFile.seek(0)

# Read the contents of the wordlist file which contains special characters
wordlistNotClean = wordlistUsernamesFile.read()

# The regex [^a-zA-Z0-9\s] will match any character that is not alphanumeric, a space, o
disallowed_chars_regex = r'[^a-zA-Z0-9\s]'

# Remove disallowed characters from the file contents
wordlistClean = re.sub(disallowed_chars_regex, '', wordlistNotClean)

# Sets the position in the file to the start
wordlistUsernamesFile.seek(0)

# Empties the file
wordlistUsernamesFile.truncate(0)

# Write the clean wordlist without special characters into the wordlist usernames file
wordlistUsernamesFile.write(wordlistClean)

# Closes the file
wordlistUsernamesFile.close()

# Opens rockyou.txt and reads it contents
rockyou = open("rockyou.txt", "r", encoding="utf-8", errors="ignore")
rockyou_contents = rockyou.read()
rockyou.close()

# Creates wordlist passwords file and opens it for writing
wordlistPasswordsFile = open(wordlistPasswordsLocation, "w")

# Prites the clean wordlist onto the opened file
wordlistPasswordsFile.write(wordlistClean)

# Prints the rockyou wordlist onto the opened file
# This is done so only one file has the contents of rockyou, because bruteforcing both u
wordlistPasswordsFile.write(rockyou_contents)
wordlistPasswordsFile.close
```

Figure 2.11 - Function for creating wordlist from page contents. 2/2

2.2.7 Getting login fields and message displayed on login fail.

Another prerequisite for building a hydra command is to find the login forms on the webpage and the message displayed when a login is failed. To do this the html code was requested and parsed using the “Requests” (Reitz, n.d.) and “BeautifulSoup” (Richardson, n.d.) libraries in the function called “getLoginFields” and “getLoginFail” that can be seen in Figure 2.12 and Figure 2.13.

```
# Function used getting all the login fields
def getLoginFields(target, uri):
    print("Getting Login Fields")

    # Gate all the html code for the address
    page_html = requests.get(f"http://{target}{uri}").text

    # Parse html using beautifulsoup
    soup = BeautifulSoup(page_html, "lxml")

    # Find login form
    login_form = soup.find("form")

    # Retrived method used for logging
    method = login_form.get("method")

    # Find input fields in login form
    fields = login_form.find_all("input")

    # List for contents of input fields
    fieldContents = []

    # Loop for each field that was found
    for field in fields:
        # Get required values
        name_field = field.get("name")
        value_field = field.get("value")
        type_field = field.get("type")

        # Append a dictionary to the list
        fieldContents.append({
            'name_field': name_field,
            'value_field': value_field,
            'type': type_field
        })

    return fieldContents, method
```

Figure 2.12 - Function for getting all login fields.

To find the display login fail message the module “difflib” was utilised to compare the html of the login page before and after running an unsuccessful login attempt. (Python, n.d.). If comparison is successful the newly created login fail message is saved.

```
# Function used for getting the message displayed on unsuccessful login
def getLoginFail(target, uri, fieldContents):
    print("Getting Login Fail Message")

    # Take the contents of the fields
    for fieldDictionary in fieldContents:
            if fieldDictionary['type'] == "text":
                username_name = fieldDictionary['name_field']
            elif fieldDictionary['type'] == "password":
                password_name = fieldDictionary['name_field']
            elif fieldDictionary['type'] == "submit":
                submit_name = fieldDictionary['name_field']
                submit_value = fieldDictionary['value_field']
            else:
                print("Automatic retrivel of login fail message unsucessfull. Create hydra command manually")

    login_url = f"http://{target}{uri}"
    usernameContents = ""
    passwordContents = ""
    submitContent = submit_value

    # Send a POST request with login credentials
    payload = {
        username_name: usernameContents,
        password_name: passwordContents,
        submit_name : submitContent
    }

    # Get html of login page before and after attempting a login
    request_html = requests.get(login_url).text
    response_html = requests.post(login_url, data=payload).text
    # Parse the HTML responses
    soup1 = BeautifulSoup(request_html, 'lxml')
    soup2 = BeautifulSoup(response_html, 'lxml')
    # Convert the parsed HTML back to strings for comparison
    html_str1 = str(soup1)
    html_str2 = str(soup2)

    # Compare the html of both
    differ = difflib.Differ()
    differences = list(differ.compare(html_str1.splitlines(), html_str2.splitlines()))
    f = open(loginFailLocation, "w+")
    for diff in differences:
        if diff.startswith('+ '):
            f.write(diff[2:])

    f.seek(0)
    login_fail_contents = f.read()
    f.close()

    # Find the index of the first ">" character
    start_index = login_fail_contents.find(">") + 1
    # Find the index of the second "<" character, starting from the position after the first ">"
    end_index = login_fail_contents.find("<", start_index)
    # Extract the desired substring
    extracted_text = login_fail_contents[start_index:end_index].strip()

    return extracted_text
```

Figure 2.13: Function for getting login fail message.

2.2.8 Running of hydra.

The hydra command can be created after getting all its prerequisites using the functions shown in section 2.2.6 and 2.2.7 above. To build and run the hydra command the function called “hydraBuilder” is used that takes all the prerequisites as arguments. Depending on the login page that was discovered the function builds a specific command. This function can be seen in Figure 2.14.

```
# Function used creating hydra command
def hydraBuilder(target, uri, is401, fieldContents, method):
    print("Creating hydra command")

    # Checks if it nees to build a command for a 401 page
    if is401:
        hydra_command = f"hydra -f -u -o {credentialsJSONLocation} -b json -L {wordlistUsernamesLocation} -P {wordlistPasswordsLocation} {target} http-get {uri}"
    else:
        # Gets messaged displayed on failed login
        login_fail = getLoginFail(target, uri, fieldContents)
        if method == "post":
            method = "http-post-form"
        else:
            print("Error getting method for hydra command. Try using hydra manually using wordlists found in wordlists directory")

        username_field = None
        password_field = None
        extra_field = None

        # Loop for each field that was found from getLoginFields
        for fieldDictonary in fieldContents:
            if fieldDictonary['type'] == "text":
                username_field = fieldDictonary['name_field']
            elif fieldDictonary['type'] == "password":
                password_field = fieldDictonary['name_field']
            elif fieldDictonary['type'] == "submit":
                extra_field = {
                    'name' : fieldDictonary['name_field'],
                    'value' : fieldDictonary['value_field']
                }
            else:
                print("Unable to setup hydra command. Use manual hyndra")

        form = f"{uri}:{username_field}=^USER^&{password_field}=^PASS^&{extra_field['name']}={extra_field['value']}:{login_fail}"
        print(f"FORM IS: {form}")
        hydra_command = f"hydra -f -u -o {credentialsJSONLocation} -b json -L {wordlistUsernamesLocation} -P {wordlistPasswordsLocation} {target} {method} '{form}'"

    subprocess.run(hydra_command, shell=True)
```

Figure 2.14 - Function for running hydra.

2.2.9 Parsing of credentials from hydra result “.json” file.

The result from running the hydra command gets saves as a “.json” file that gets parsed using the function that can be seen in Figure 2.15.

```
# Function used for extracting credentails that were found from credentials.json file created by hydra
def extractCredentials():
    #readJsonfile if it exist
    print(f"Reading credentails from {credentialsJSONLocation}")
    with open(credentialsJSONLocation, 'r') as json_file:
        data = json.load(json_file)

    login = data['results'][0]['login']
    password = data['results'][0]['password']
    # Save login and password to a file
    with open(credentialsTXTLocation, 'w') as file:
        file.write(f"{login}:{password}")
    print("Login and password saved successfully.")
    return login,password
```

Figure 2.15 - Function for parsing credentials from hydra result ".json" file.

2.2.10 Taking screenshot after logging in.

To get screenshots after logging in a function called “grabScreenLogged” was created that would take the address of the unauthorized page or login page and the credentials found by hydra as arguments. Following this it would use selenium to open the webpage and log in. This function can be seen in Figure 2.16 and Figure 2.17. If the page being opened is an unauthorised page it adds the credentials to the “URL” that is opened with selenium to authorise the access. This can be seen in Figure 2.16

```
# Function for taking screenshots of pages that require logging in
def grabScrenshotLogged(loginPageAddress, addressesUNAUTH, username, password):
    # Check if there was address that returned 401
    if addressesUNAUTH:
        # Iterate over each address
        for addressUNAUTH in addressesUNAUTH:
            # Create driver
            driver = webdriver.Firefox()

            # Get everything after http://
            split = addressUNAUTH.split("//")[1]

            # Open address usng username and password for authentication
            driver.get(f"http://{username}:{password}@{split}")

            sleep(1)

            # Remove ":" and "/" from address using replace()
            file_name = addressUNAUTH.replace(":", ".").replace("/", ".")

            # Take a screenshot and save it
            driver.get_screenshot_as_file(f'{screenshotsDirectory}/{file_name}.png')

            # Exit driver
            driver.quit()
```

Figure 2.16 - Function for taking screenshot after logging in. 1/2

While if the page being opened is a normal login page it used the “Requests” and “BeautifulSoup” python libraries again to parse the html content and find the login fields and submit button. After which it would enter the credentials in the correct fields and press the submit button to log in. This can be seen in Figure 2.17.

```
else:
    print("Does not have 401 pages")
    # Create driver
    driver = webdriver.Firefox()
    # Open address
    driver.get(f"http://{loginPageAddress}")
    # Gate all the html code for the address
    page_html = requests.get(f"http://{loginPageAddress}").text
    # Parse html using BeautifulSoup
    soup = BeautifulSoup(page_html, "lxml")
    # Find login form
    login_form = soup.find("form")
    # Find input fields in login form
    fields = login_form.find_all("input")
    # List for contents of input fields
    fieldContents = []

    # Loop for each field that was found
    for field in fields:
        # Get required values
        name_field = field.get("name")
        value_field = field.get("value")
        type_field = field.get("type")
        # Append a dictionary to the list
        fieldContents.append({
            'name_field': name_field,
            'value_field': value_field,
            'type': type_field
        })

    # Loop for each dictionary in the fieldContents lists
    for fieldDictionary in fieldContents:
        if fieldDictionary['type'] == "text":
            username_name = fieldDictionary['name_field']
        elif fieldDictionary['type'] == "password":
            password_name = fieldDictionary['name_field']
        elif fieldDictionary['type'] == "submit":
            submit_name = fieldDictionary['name_field']
        else:
            print(f"Automatic login at http://{loginPageAddress} unsuccesful. Try logging in

    # Locate username field using the name of the element for the driver to use at input
    usernameField = driver.find_element(By.NAME, username_name)
    # Locate password field using the name of the element for the driver to use at input
    passwordField = driver.find_element(By.NAME, password_name)
    # Send keys to fields
    usernameField.send_keys(username)
    passwordField.send_keys(password)
    # Locate submit button and click it
    driver.find_element(By.NAME, submit_name).click()

    sleep(1)
    # Remove ":" and "/" from address using replace()
    file_name = loginPageAddress.replace("/", ".")
    # Take a screenshot and save it
    driver.get_screenshot_as_file(f'{screenshotsDirectory}/{file_name}.png')
    # Exit driver
    driver.quit()
```

Figure 2.17- Function for taking screenshot after logging in. 2/2

2.2.11 Automatically login into page.

All of the functions from section 2.2.5, 2.2.6, 2.2.7, 2.2.8, 2.2.9 and 2.2.10 related to the login functionality of the script are ran from a function called “autoLogin” that can be seen in Figure 2.18 and Figure 2.19.

```
# Function used for auto login into pages
def autoLogin(addressesUNAUTH, addressesOK, nikto_vulns, is401, logged):
    # Create directories
    os.mkdir(wordlistDirectory)
    os.mkdir(credentailsDirectory)
    os.mkdir(extrasDirectory)

    # Check if nikto discover a login page and set uri to uri of login page
    uri = checkIfLoginNikto(nikto_vulns)

    # If login page exist uri is set to a value and this executes
    if uri:
        print("Nikto login found. Creating wordlist")
        print("Login page uri:", uri)

        # Gets all the text from the pages that returned 200
        getComments(addressesOK)

        # Gets the information about the login field and the method used for login
        fieldContents, method = getLoginFields(target, uri)

        # Used to build hydra command
        hydraBuilder(target, uri, is401, fieldContents, method)

    # If there is pages that returned 401
    elif addressesUNAUTH:
        is401 = True
        print("401 page exists. Creating wordlist from page contents")
        print("401 page Address:", addressesUNAUTH)

        # Gets all the text from the pages that returned 200
        getComments(addressesOK)

        # Gets the url for page that returned 401 in the format http://10.10.10.10/uri
        url = addressesUNAUTH[0]

        # Splits url at every '/'
        parts = url.split("/")

        # Gives everything to the right of the 3rd '/' which is the uri
        result = "/".join(parts[3:])

        # Sets the uri to be /uri
        uri = "/" + result

        # Method/login field are not needed for http-get login so they are set to none
        fieldContents = None
        method = None
        hydraBuilder(target, uri, is401, fieldContents, method)
```

Figure 2.18 - Function for auto login. 1/2

```
with open(credentialsJSONLocation, 'r') as json_file:
    data = json.load(json_file)

# Extract login and password and from json file if it contains a sucessfull result
if len(data['results']) > 0:

    # Extract username and password from json file
    username,password = extractCredentials()

    # Set logged to true to indicate login credentials were found
    logged = True

    # Take screenshots of pages after logging in
    loginPageAddress = target + uri

    if screenshotScript:
        grabScrenshotLogged(loginPageAddress, addressesUNAUTH, username, password)

    # Run dirb again with logged set to true which makes it use credentials
    runDirb(target, logged)
```

Figure 2.19 - Function for auto login. 2/2

2.2.12 Creation of ".docx" report.

To create a neatly formatted ".docx" report the script utilise the "python-docx" library which is used for the creating and updating Microsoft Word ".docx" files (Canny, 2021). Four functions were created that were called "saveOutputWord", "addImageWord", "addWordlistWord" and "addCredentialsWord". They were all used to add a specific result from running the script to a Microsoft Word document. These functions can be seen in Figure 2.20, Figure 2.21, Figure 2.22 and Figure 2.23

```
# Function for saving output to word file
def saveOutputWord(filename, tool_name, output):
    # Opening docx file
    document = Document(filename)

    # Create a table and add it to document
    table = document.add_table(rows=2, cols=1)

    # Get cell on first row
    toolNameCell = table.cell(0, 0)
    toolNameCell.text = tool_name

    # Styling
    toolNameCell_paragraph = toolNameCell.paragraphs[0]
    toolNameCell_paragraph.style = document.styles['Heading 1']
    toolNameCell_paragraph.alignment = WD_PARAGRAPH_ALIGNMENT.CENTER

    # Get cell on second row
    contentCell = table.cell(1, 0)
    contentCell.text = output

    document.save(filename)
    print(f"{tool_name} saved to {filename}")
```

Figure 2.20- Function for adding tool output to ".docx" report.

```python
# Function adding created images to word file
def addImageWord(filename):
    # Opening docx file
    document = Document(filename)

    # Create heading
    document.add_heading("Captured Screenshots", level=0)

    # iterate over screenshots in screenshots directory
    for screenshot in os.listdir(screenshotsDirectory):

        # Create path to screenshots by joining hte screenshots directory location and the screens
        screenshotFile = os.path.join(screenshotsDirectory, screenshot)

        # checking if file exists at location
        if os.path.isfile(screenshotFile):
            # Create a table and add it to document
            table = document.add_table(rows=2, cols=1)

            # Get cell on first row
            nameCell = table.cell(0, 0)
            nameCell.text = f"Screenshot: {screenshotFile}"

            # Style
            nameCell_paragraph = nameCell.paragraphs[0]
            nameCell_paragraph.style = document.styles['Heading 1']
            nameCell_paragraph.alignment = WD_PARAGRAPH_ALIGNMENT.CENTER

            # Get cell on second row
            imageCell = table.cell(1, 0)
            imageCell_paragraph = imageCell.paragraphs[0]

            # Adding picture to paragraph in cell
            imageCell_paragraph.add_run().add_picture(screenshotFile, width=Cm(15), height=None)

    document.save(filename)
    print(f"Added picture to {filename}")
```

Figure 2.21 - Function for adding created screenshots to ".docx" report.

```python
# Function for adding created wordlist to word file
def addWordlistWord(filename):
    # Opening docx file
    document = Document(filename)

    # Creating a heading
    document.add_heading("Created wordlist", level=0)

    # Opening wordlist
    wordlistUsernamesFile = open(wordlistUsernamesLocation)

    # Reading wordlist
    content = wordlistUsernamesFile.read()

    # Adding contents of wordlist file to document
    document.add_paragraph(content)

    document.save(filename)
    print(f"Added comments to {filename}")
```

Figure 2.22 - Function for adding created wordlist to ".docx" report.

```python
# Function for adding created credentials to word file
def addCredentialsWord(filename):
    # Opening docx file
    document = Document(filename)

    # Creating a heading
    document.add_heading("Discovered credentials", level=0)

    # If credentials worked for logging in
    if os.path.exists(credentialsTXTLocation):
        # Open credentials file
        credentialsTXTFile = open(credentialsTXTLocation)

        # Read credentials files
        content = credentialsTXTFile.read()

        # Adding contents of credentials file to document
        document.add_paragraph(content)
    else:
        document.add_paragraph("No credentials were found")

    print(f"Added credentials to {filename}")

    document.save(filename)
```

Figure 2.23 - Function for adding found credentials to ".docx" report.

The functions for adding content to the report would all get ran by a function called "createReport" that would create the initial document which the content would be added to. This function can be seen in Figure 2.24 and Figure 2.25.

```python
# Function used for demonstrating report creation from provided output folder
def createReport(logged):
    if os.path.isdir(reportsDirectory):
        shutil.rmtree(reportsDirectory)
    os.mkdir(reportsDirectory)
    #Create document
    document = Document()
    # Add a title
    document.add_heading('Scans Output', level=0)
    document.save(docxReportLocation)

    # Saves nmap output in docx
    nmap_output = extractInfoNmap(outputNmapLocation)
    saveOutputWord(docxReportLocation, 'Nmap Scan Output', nmap_output)

    # Variables used for storing all vulnerabilities discovered by nikto
    nikto_vulns = []
    # Saves nikto output in docx
    nikto_output = extractInfoNikto(outputNiktoLocation, nikto_vulns)
    saveOutputWord(docxReportLocation, 'Nikto Scan Output', nikto_output)

    # Saves dirb output in docx
    dirb_output = open(outputDirbLocation, "r")
    saveOutputWord(docxReportLocation, 'Dirbuster Scan Output', dirb_output.read())
    dirb_output.close()
```

Figure 2.24 - Function for creating "docx" report. 1/2

```
    # Saves dirb logged output in docx
    if logged:
        dirbLogged_output = open(outputDirbLoggedLocation, "r")
        saveOutputWord(docxReportLocation, 'Dirbuster Logged Scan Output', dirbLogged_output.read())
        dirbLogged_output.close()

    # Runs if user used the -S switch to turn the screenshot script on or the -0 switch to turn using
    if screenshotScript or outputOnly:
        # Adds images to docx
        addImageWord(docxReportLocation)

    # Runs if user used the -L switch to turn the login script on or the -0 switch to turn using outp
    if loginScript or outputOnly:
        # Adds wordlist to docx
        addWordlistWord(docxReportLocation)
        # Adds discovered credentials to docx
        addCredentialsWord(docxReportLocation)
```

Figure 2.25- - Function for creating "docx" report. 2/2

2.2.13 Converting ".docx" report to ".pdf".

LibreOffice was used to convert the created Microsoft Word Document report in the function called "converDocxToPdf" that can be seen in Figure 2.26. It worked by running a command using the "subprocess" python module that launched LibreOffice in headless mode to control the application without having a user.

```
# Function for conveting docx file to pdf
def convertDocxToPdf():
    # Read the DOCX file
    command = [f"libreoffice --headless --convert-to pdf {docxReportLocation} --outdir {pdfReportLocation}"]
    subprocess.run(command, shell=True)
```

Figure 2.26 - Function for converting ".docx" report to ".pdf".

2.2.14 Implementation of command line switches.

To implement command line switches the "argparse" module was used. It allows the creation of user-friendly command-line interfaces and generates help and usage messages automatically (Python, n.d.). You can see how this was implemented in Figure 2.27 and Figure 2.28.

```
parser = argparse.ArgumentParser()
parser.add_argument("-T", metavar="<target>", action="store", type=str, help="Target IP address", required=True)
parser.add_argument("-L", action="store_true", help="Use to attempt auto logging in. WARNING! CAN TAKE A LONG TIME")
parser.add_argument("-S", action="store_true", help="Use to take screenshots of pages automatically")
parser.add_argument("-A", action="store_true", help="Turn all scripts on. Use if testing on a live machine")
parser.add_argument("-P", action="store_true", help="Create a pdf document of report aswell. REQUIRES LIBRE OFFICE")
parser.add_argument("-O", action="store_true", help="Use provided example scan folder to create report in both docx

args = parser.parse_args()
```

Figure 2.27 - Command line switches and help menu implementation 1/2.

```
parser = argparse.ArgumentParser()
parser.add_argument("-T", metavar="<target>", action="store", type=str, help="Target IP address", required=True)
parser.add_argument("-L", action="store_true", help="Use to attempt auto logging in. WARNING! CAN TAKE A LONG TIME")
parser.add_argument("-S", action="store_true", help="Use to take screenshots of pages automatically")
parser.add_argument("-A", action="store_true", help="Turn all scripts on. Use if testing on a live machine")
parser.add_argument("-P", action="store_true", help="Create a pdf document of report aswell. REQUIRES LIBRE OFFICE")
parser.add_argument("-O", action="store_true", help="Use provided example scan folder to create report in both docx

args = parser.parse_args()

target = None
loginScript = False
screenshotScript = False
outputOnly = False
pdf = False

if args.T:
    target = args.T

if args.L:
    # Code for switch1
    loginScript = True
    print("Auto login is enabled")

if args.S:
    # Code for switch2
    screenshotScript = True
    print(f"Screen shot taking is on")

if args.A:
    loginScript = True
    screenshotScript = True
    print(f"All scripts are on")

if args.P:
    pdf = True
    print(f"Will create pdf report alongside docx report")

if args.O:
    outputOnly = True
    pdf = True
    print(f"Demonstrating creation of report with a provided scan folder")
```

Figure 2.28 - Command line switches and help menu implementation 2/2.

In Figure 2.29 you can see what the help menu for the program looks like when the user displays it using the “-h” switch.

```
└─$ python3 script.py -h
usage: script.py [-h] -T <target> [-L] [-S] [-A] [-P] [-O]

options:
  -h, --help   show this help message and exit
  -T <target>  Target IP address
  -L           Use to attempt auto logging in. WARNING! CAN TAKE A LONG TIME
  -S           Use to take screenshots of pages automatically
  -A           Turn all scripts on. Use if testing on a live machine
  -P           Create a pdf document of report aswell. REQUIRES LIBRE OFFICE
  -O           Use provided example scan folder to create report in both docx and pdf format. Used only
               to demonstrate creation of report with a provided output folder
```

Figure 2.29 - Help menu

2.2.15 Creation of “requirements.txt”.

To provide the user of the script of way of easily installing all the python module required for the execution of the script the programmer created a “requirements.txt file using “pipreqs”. (Kravcenko, n.d.). This can be seen in Figure 2.30 and Figure 2.31

```
┌──(kali㉿kali)-[~/Desktop/script]
└─$ pip3 install pipreqs
Defaulting to user installation because normal site-packages is not writeable
Collecting pipreqs
  Downloading pipreqs-0.4.13-py2.py3-none-any.whl (33 kB)
Requirement already satisfied: docopt in /usr/lib/python3/dist-packages (from pipreqs) (0.6.2)
Collecting yarg
  Downloading yarg-0.1.9-py2.py3-none-any.whl (19 kB)
Requirement already satisfied: requests in /usr/lib/python3/dist-packages (from yarg→pipreqs) (2.28.1)
Installing collected packages: yarg, pipreqs
  WARNING: The script pipreqs is installed in '/home/kali/.local/bin' which is not on PATH.
  Consider adding this directory to PATH or, if you prefer to suppress this warning, use --no-warn-script-location.
Successfully installed pipreqs-0.4.13 yarg-0.1.9
```

Figure 2.30 - Installing pipreqs.

```
┌──(kali㉿kali)-[~/Desktop/script]
└─$ python3 -m pipreqs.pipreqs .
WARNING: Import named "Requests" not found locally. Trying to resolve it at the PyPI server.
WARNING: Import named "Requests" was resolved to "requests:2.31.0" package (https://pypi.org/project/requests/).
Please, verify manually the final list of requirements.txt to avoid possible dependency confusions.
WARNING: Import named "selenium" not found locally. Trying to resolve it at the PyPI server.
WARNING: Import named "selenium" was resolved to "selenium:4.9.1" package (https://pypi.org/project/selenium/).
Please, verify manually the final list of requirements.txt to avoid possible dependency confusions.
INFO: Successfully saved requirements file in ./requirements.txt
```

Figure 2.31 - Creation of “requirements.txt”.

The reason this approach for creating “requirements.txt” was chosen instead of running “pip freeze > requirements.txt” is because “pip freeze” saves all packages installed on the system. By using “pipreqs” the programmer was able to save only the packages that are required for the execution of the script. The contents of “requirements.txt can be seen in Appendix A – Requirements.txt

Using the provided with the script “requirements.txt” file the user can run “pip install -r requirements.txt” to install all the required packages. This can be seen in Figure 2.32.

```
┌──(kali㉿kali)-[~/Desktop/script]
└─$ pip install -r requirements.txt
Defaulting to user installation because normal site-packages is not writeable
Requirement already satisfied: beautifulsoup4==4.11.2 in /usr/lib/python3/dist-packages (from -r requireme
nts.txt (line 1)) (4.11.2)
Requirement already satisfied: python_docx==0.8.11 in /usr/lib/python3/dist-packages (from -r requirements
.txt (line 2)) (0.8.11)
Collecting Requests==2.31.0
  Downloading requests-2.31.0-py3-none-any.whl (62 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 62.6/62.6 kB 319.6 kB/s eta 0:00:00
Collecting selenium==4.9.1
  Downloading selenium-4.9.1-py3-none-any.whl (6.6 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 6.6/6.6 MB 11.7 MB/s eta 0:00:00
Requirement already satisfied: charset-normalizer<4,≥2 in /usr/lib/python3/dist-packages (from Requests==
2.31.0→-r requirements.txt (line 3)) (3.0.1)
Requirement already satisfied: idna<4,≥2.5 in /usr/lib/python3/dist-packages (from Requests==2.31.0→-r r
equirements.txt (line 3)) (3.3)
```

Figure 2.32 - Installing required packages.

3 DISCUSSION

3.1 DISCUSSION

An issue that was faced by the tester was the inability to test against real websites which meant tools such as “Whois” were not implemented for the purpose of this project, because the “TryHackMe” CTF rooms do not have any domain information that could be retrieved.

Alongside this another issue was the slow speed of Dirb, which following some research the programmer was able to determine was caused by the fact that it can’t be multithreaded to improves its performance.

The final issue was the lack of testing against rooms other than the ones used during development which could have allowed the easy implementation of more potential scenarios that the script could face.

Despite that in the end the programmer successfully developed The Automated Web Application Scanning script and was able to fulfil its main purpose of automating the scanning of various web applications and generating a neatly formatted report containing all the finding in the form of a Word or PDF document. The finished script had the following features:

- To ensure the script works in Kali Linux it was tested extensively on the operating system and was found to be functional without compatibility issues.
- Script was programmed in Python 3.11.2 which ensured that it benefits from the latest features of the programming language.
- Command line arguments are implemented, to allow the user to customise the running of the script based on the preferences. The following command line arguments were added:
  - "-T <target>" - Used to input target IP Address.
  - “-L” – Enables auto logging in script.
  - “-S” – Enables automatic page screenshot taking script.
  - “-A” – Turns all scripts on
  - “-P” – Creates a “.pdf” report alongside the “.docx” report.
  - “-O” – Allows the demonstration of writing to word file by using a provided output folder.
- The script requires no user input to run after a target is supplied.
- Can create comprehensive wordlist comprising of every word found in the HTML code of the website.

- It is able to identify Login forms successfully. Alongside this it can generate a hydra command to attempt to brute force the login forms with the generated wordlist, which allows the script to identify weak passwords.
- Finally, it creates a neatly formatted Word and PDF document contains all the information that was discovered during its execution.

In conclusion the programmer was able to successfully meet the aims of the project. The script achieved its purpose of scanning web applications and generating a detailed report in both a Word and PDF format.

3.2 FUTURE WORK

As future work the programmer would try to add more web applications scanning tools to the script, which were not added to this initial version due to time constrains. Possible new tools that could be added and the reason they would be added are:

- Whois – Would allow the gathering of information about domains names.
- Dmitry – Check subdomains, whois, netcraft etc.
- O-saft – Check SSL certificates
- Tlssled – Check for TLS vulnerabilities
- Whatweb – Could be used to identify the technologies used by the target website.
- Wapiti – This vulnerability scanner would allow more information about vulnerabilities to be found.
- Skipfish – Further vulnerability scanning.
- SQLMap – Test target website for SQL injections
- Wfuzz – replace Dirb. Wfuzz can be multithreaded which makes it faster than Dirb that does not have that option.

Finally in terms the use of Hydra the script could be improved to accommodate the brute forcing of other types of login pages. It could also be modified to allow the user to give the script a username to use.

4 REFERENCES

Canny, S., 2021. *Python-docx.* [Online]
Available at: https://pypi.org/project/python-docx/
[Accessed 22 May 2023].

Dirb, n.d. *Dirb.* [Online]
Available at: https://dirb.sourceforge.net/
[Accessed 21 May 2023].

EC-Council, 2022. *What is web application security, and why is it important?.* [Online]
Available at: https://www.eccu.edu/blog/technology/what-is-web-application-security-and-why-is-it-important/
[Accessed 29 May 2023].

Invicti, n.d. *Why Web Vulnerability Testing Needs to be Automated.* [Online]
Available at: https://www.invicti.com/blog/web-security/automatic-web-application-vulnerability-testing-detection/
[Accessed 29 May 2023].

Kravcenko, V., n.d. *Pipreqs 0.4.13.* [Online]
Available at: https://pypi.org/project/pipreqs/
[Accessed 27 May 2023].

LibreOffice, n.d. *LibreOffice.* [Online]
Available at: https://www.libreoffice.org/
[Accessed 21 May 2023].

Natasha Urdovska, n.d. *The Importance of Web Application Security Testing.* [Online]
Available at: https://www.it-labs.com/the-importance-of-web-application-security-testing
[Accessed 29 May 2023].

Nmap, n.d. *Nmap.* [Online]
Available at: https://nmap.org/
[Accessed 21 May 2023].

Positive technologies, 2022. *Threats and vulnerabilities in web applications 2020–2021.* [Online]
Available at: https://www.ptsecurity.com/ww-en/analytics/web-vulnerabilities-2020-2021/
[Accessed 29 May 2023].

Python, n.d. *argparse — Parser for command-line options, arguments and sub-commands.* [Online]
Available at: https://docs.python.org/3/library/argparse.html
[Accessed 23 May 2023].

Python, n.d. *difflib — Helpers for computing deltas.* [Online]
Available at: https://docs.python.org/3/library/difflib.html
[Accessed 20 May 2023].

Python, n.d. *subprocess — Subprocess management.* [Online]
Available at: https://docs.python.org/3/library/subprocess.html
[Accessed 20 May 2023].

Python, n.d. *xml.etree.ElementTree — The ElementTree XML API.* [Online]
Available at: https://docs.python.org/3/library/xml.etree.elementtree.html
[Accessed 20 May 2023].

Reitz, K., n.d. *Requests 2.31.0.* [Online]
Available at: https://pypi.org/project/requests/
[Accessed 20 May 2023].

Richardson, L., n.d. *Beautifulsoup4.* [Online]
Available at: https://www.crummy.com/software/BeautifulSoup/
[Accessed 20 May 2023].

Selenium, n.d. *Selenium 4.9.1.* [Online]
Available at: https://pypi.org/project/selenium/
[Accessed 20 May 2023].

Sullo, n.d. *Nikto web server scanner.* [Online]
Available at: https://github.com/sullo/nikto
[Accessed 21 May 2023].

tryhackme & Ar33zy, 2020. *TryHackMe | Pickle Rick.* [Online]
Available at: https://tryhackme.com/room/picklerick
[Accessed 20 May 2023].

tryhackme & Cryillic, 2019. *TryHackMe | ToolsRus.* [Online]
Available at: https://tryhackme.com/room/toolsrus
[Accessed 20 May 2023].

TryHackMe, n.d. *TryHackMe | Cyber Security Training.* [Online]
Available at: https://tryhackme.com/
[Accessed 20 May 2023].

Vanhauser-thc, 2022. *Hydra.* [Online]
Available at: https://github.com/vanhauser-thc/thc-hydra
[Accessed 20 May 2023].

APPENDICES

APPENDIX A – REQUIREMENTS.TXT

```
beautifulsoup4==4.11.2
python_docx==0.8.11
Requests==2.31.0
selenium==4.9.1
```

APPENDIX B – FULL CODE

```
import shutil
import subprocess
import os
from docx import Document
from docx.shared import Cm
from docx.enum.text import WD_PARAGRAPH_ALIGNMENT
from selenium import webdriver
from selenium.webdriver.common.by import By
from time import sleep
from bs4 import BeautifulSoup, Comment
import requests
import xml.etree.ElementTree as ET
import difflib
import re
import json
import argparse

parser = argparse.ArgumentParser()
parser.add_argument("-T", metavar="<target>", action="store", type=str,
help="Target IP address", required=True)
parser.add_argument("-L", action="store_true", help="Use to attempt auto logging
in. WARNING! CAN TAKE A LONG TIME")
parser.add_argument("-S", action="store_true", help="Use to take screenshots of
pages automatically")
parser.add_argument("-A", action="store_true", help="Turn all scripts on. Use if
testing on a live machine")
parser.add_argument("-P", action="store_true", help="Create a pdf document of
report aswell. REQUIRES LIBRE OFFICE")
parser.add_argument("-O", action="store_true", help="Use provided example scan
folder to create report in both docx and pdf format. Used only to demonstrate
creation of report with a provided output folder")
```

```
args = parser.parse_args()

target = None
loginScript = False
screenshotScript = False
outputOnly = False
pdf = False

if args.T:
    target = args.T

if args.L:
    # Code for switch1
    loginScript = True
    print("Auto login is enabled")

if args.S:
    # Code for switch2
    screenshotScript = True
    print(f"Screen shot taking is on")

if args.A:
    loginScript = True
    screenshotScript = True
    print(f"All scripts are on")

if args.P:
    pdf = True
    print(f"Will create pdf report alongside docx report")

if args.O:
    outputOnly = True
    pdf = True
    print(f"Demonstrating creation of report with a provided scan folder")

targetDirectorry = f'./{target}'
outputDirectory = f'{target}/output'
screenshotsDirectory = f'{target}/screenshots'
wordlistDirectory = f'{target}/wordlists'
credentailsDirectory = f'{target}/credentials'
reportsDirectory = f'{target}/reports'
extrasDirectory = f'{target}/extras'

outputNmapLocation = f'{outputDirectory}/outputNmap.xml'
outputNiktoLocation = f'{outputDirectory}/outputNikto.xml'
```

```
outputDirbLocation = f'{outputDirectory}/outputDirb.txt'
outputDirbLoggedLocation = f'{outputDirectory}/outputDirbLogged.txt'
docxReportLocation = f'{reportsDirectory}/{target}-scanReport.docx'
pdfReportLocation = f'{reportsDirectory}/'

wordlistUsernamesLocation = f'{wordlistDirectory}/wordlistUsernames.txt'
wordlistPasswordsLocation = f'{wordlistDirectory}/wordlistPasswords.txt'
credentialsTXTLocation = f'{credentailsDirectory}/credentials.txt'
credentialsJSONLocation = f'{credentailsDirectory}/credentials.json'

loginFailLocation = f'{extrasDirectory}/loginfail.txt'

def main():
    # Logged variable set to TRUE when succesfull hyndra
    logged = False
    # Is401 variable set to TRUE when there is a page that returns code 401 -
Unauthorized Access
    is401 = False

    # # Create directories
    # if os.path.isdir(targetDirectorry):
    #     shutil.rmtree(targetDirectorry)
    # os.mkdir(targetDirectorry)
    # os.mkdir(outputDirectory)

    # # Run the scans and get the output
    # runNmap(target)
    # runNikto(target)
    # runDirb(target, logged)

    #Get addresses
    addressesOK = getOKAddr()
    addressesUNAUTH = getUNAUTHAddr()

    # Runs if user used the -L switch to turn the login script on
    if screenshotScript:
        grabScrenshot(addressesOK)

    nikto_vulns = []
    # Adds content to nikto vulns list
    extractInfoNikto(outputNiktoLocation, nikto_vulns)

    # Runs if user used the -L switch to turn the login script on
    if loginScript:
        autoLogin(addressesUNAUTH, addressesOK, nikto_vulns, is401, logged)
```

```
    createReport(logged)

    # Runs if user used the -P switch to turn output to pdf on
    if pdf:
        convertDocxToPdf()

# Function for running nmap
def runNmap(target):
    nmap_command = [f'sudo nmap -v -p- -A -oX {outputNmapLocation} {target}']
    subprocess.run(nmap_command, shell=True)

# Function for running nikto
def runNikto(target):
    nikto_command = [f'nikto -h {target} -output {outputNiktoLocation}']
    subprocess.run(nikto_command, shell=True)

# Function for running dirb
def runDirb(target, logged):
    # If succesfully logged in with credentials
    if logged:
        f = open(credentialsTXTLocation, "r")
        credentials = f.read()

        # Launch dirb with credentials provided
        dirbuster_command = [f'dirb http://{target} -u {credentials} -o
{outputDirbLoggedLocation}']
        subprocess.run(dirbuster_command, shell = True)
    else:
        # Launch dirb without credentials
        dirbuster_command = [f'dirb http://{target} -o {outputDirbLocation}']
        subprocess.run(dirbuster_command, shell=True)

# Get addresses from dirb output that returned 200. Addresses are displayed as
http://address
def getOKAddr():
    command = [f"cat {outputDirbLocation} | grep CODE:200 | cut -d ' ' -f 2"]
    result = subprocess.run(command, capture_output=True, text=True, shell=True)
    # Split the output into lines
    return result.stdout.splitlines()

# Get addresses from dirb output that returned 401. Addresses are displayed as
http://address
def getUNAUTHAddr():
    command = [f"cat {outputDirbLocation} | grep CODE:401 | cut -d ' ' -f 2"]
```

```
    result = subprocess.run(command, capture_output=True, text=True, shell=True)
    # Split the output into lines
    return result.stdout.splitlines()

# Function used for extracting information from nmap xml file
def extractInfoNmap(xml_file):
    # Create xml tree
    tree = ET.parse(xml_file)

    # Get root element
    root = tree.getroot()

    # Extract scan information
    scan_args = root.get('args')
    scan_start = root.get('startstr')
    scan_version = root.get('version')

    # Extract host information
    host = root.find('host')
    host_address = host.find('address').get('addr')

    # Extract open ports information
    open_ports = []
    ports = host.find('ports')
    for port in ports.findall('port'):
        ran_scripts = []
        port_id = port.get('portid')
        port_protocol = port.get('protocol')
        service = port.find('service')
        service_name = service.get('name')
        service_product = service.get('product')
        service_version = service.get('version')
        for script in port.findall('script'):
            script_id = script.get('id')
            script_output = script.get('output')
            ran_scripts.append({
                'script_id' : script_id,
                'script_output' : script_output
            })
        open_ports.append({
            'port_id': port_id,
            'protocol': port_protocol,
            'service_name': service_name,
            'service_product': service_product,
            'service_version': service_version,
```

```
            'ran_scripts': ran_scripts
        })

    # Extracted info
    output_string = f"Scan Information:\n" \
                f"======================================================================
======\n" \
                f"Scan Arguments: {scan_args}\n" \
                f"Scan Start Time: {scan_start}\n" \
                f"Scan Version: {scan_version}\n" \
                f"======================================================================
======\n" \
                f"Host Information:\n" \
                f"Host Address: {host_address}\n" \
                f"======================================================================
======\n" \
                f"Open Ports:\n" \
                f"----------------------------------------------------------------------
----------------------------------------------------\n"

    for port in open_ports:
        output_string += f"Port: {port['port_id']}\n" \
                         f"Protocol: {port['protocol']}\n" \
                         f"Service Name: {port['service_name']}\n" \
                         f"Service Product: {port['service_product']}\n" \
                         f"Service Version: {port['service_version']}\n" \
                         f"Scripts ran:\n"

        for script in port['ran_scripts']:
            output_string += f"-> {script['script_id']}:
{script['script_output']}\n"

        output_string += f"----------------------------------------------------------
----------------------------------------------------------------\n"

    output_string +=
"======================================================================="
    return output_string

# Function used for extracting information from nikto xml file
def extractInfoNikto(xml_file, niktoVulns):
    # Create xml tree
    tree = ET.parse(xml_file)

    # Get root element
```

```
    root = tree.getroot()

    # Find niktoscan element inside the root
    niktoscan = root.find('niktoscan')

    # Find scan details element inside nikoscan
    scandetails = niktoscan.find('scandetails')

    # Extract target information
    target_ip = scandetails.get('targetip')
    target_hostname = scandetails.get('targethostname')
    target_port = scandetails.get('targetport')
    starttime = scandetails.get('starttime')

    # Extract All items
    for item in scandetails.findall('item'):
        method = item.get('method')
        description = item.find('description').text.strip()
        referencesField = item.find('references')
        if referencesField.text:
            references = referencesField.text.strip()
        else:
            references = 'Reference not available'
        uri = item.find('uri').text.strip()
        niktoVulns.append({
            'method' : method,
            'description' : description,
            'uri' : uri,
            'references' : references
        })

    # Extracted info
    output_string = f"Scan Information:\n" \
                f"========================================================================\n" \
                f"Target IP: {target_ip}\n" \
                f"Target hostname: {target_hostname}\n" \
                f"Target port: {target_port}\n" \
                f"Scan start time: {starttime}\n" \
                f"========================================================================\n" \
                f"Items:\n" \
                f"------------------------------------------------------------------------\n"
```

```
    for niktoVuln in niktoVulns:
        output_string += f"Method: {niktoVuln['method']}\n" \
                         f"Description: {niktoVuln['description']}\n" \
                         f"URI: {niktoVuln['uri']}\n" \
                         f"Reference: {niktoVuln['references']}\n" \
                         f"-------------------------------------------------------------
-------------------------------------------------------------\n"

    output_string +=
"================================================================================"

    return output_string

# Function for taking screenshots of all pages that returned 200
def grabScrenshot(addressesOK):
    # Create directory
    os.mkdir(screenshotsDirectory)

    # Iterate over each address
    for addressOK in addressesOK:
        # Create driver
        driver = webdriver.Firefox()

        # Open address
        driver.get(addressOK)

        sleep(1)
        # Remove ":" and "/" from address using replace()
        file_name = addressOK.replace(":", ".").replace("/", ".")

        # Take a screenshot and save it
        driver.get_screenshot_as_file(f'{screenshotsDirectory}/{file_name}.png')

        # Exit driver
        driver.quit()

# Function used to find if nikto found a login section
def checkIfLoginNikto(nikto_vulns):
    # Checks if login page exists
    checkMessage = 'Admin login page/section found.'

    for vuln in nikto_vulns:
        # If nikto discovered login page
        if checkMessage in vuln['description']:
            return vuln['uri']
```

```
# Functions used for retriving text and comments from the pages
def getComments(addressesOK):
    print("Getting comments")

    # Create wordlist usernames file
    wordlistUsernamesFile = open(wordlistUsernamesLocation, "w+")

    # Loop for each address that returned 200
    for addressOK in addressesOK:
        # Gate all the html code for the address
        page_html = requests.get(addressOK).text

        # Parse html using beautifulsoup
        soup = BeautifulSoup(page_html, "lxml")

        # Find all comments
        comments = soup.find_all(string=lambda text: isinstance(text, Comment))

        # Loop for each comment that was found
        for comment in comments:
            # Split aech comment onto a list of words
            words = comment.split()

            # For each word in the list of words
            for word in words:
                # Write to username wordlist file
                wordlistUsernamesFile.write(word + '\n')

        # Get all the text
        text_content = soup.get_text()

        # Split the text onto a list of words
        words = text_content.split()

        # For each word in the list of words
        for word in words:
            # Write to username wordlist file
            wordlistUsernamesFile.write(word + '\n')

    # Sets the position in the file to the start
    wordlistUsernamesFile.seek(0)

    # Read the contents of the wordlist file which contains special characters
    wordlistNotClean = wordlistUsernamesFile.read()
```

```
    # The regex [^a-zA-Z0-9\s] will match any character that is not alphanumeric,
a space, or a newline.
    disallowed_chars_regex = r'[^a-zA-Z0-9\s]'

    # Remove disallowed characters from the file contents
    wordlistClean = re.sub(disallowed_chars_regex, '', wordlistNotClean)

    # Sets the position in the file to the start
    wordlistUsernamesFile.seek(0)

    # Empties the file
    wordlistUsernamesFile.truncate(0)

    # Write the clean wordlist without special characters into the wordlist
usernames file
    wordlistUsernamesFile.write(wordlistClean)

    # Closes the file
    wordlistUsernamesFile.close()

    # Opens rockyou.txt and reads it contents
    rockyou = open("rockyou.txt", "r", encoding="utf-8", errors="ignore")
    rockyou_contents = rockyou.read()
    rockyou.close()

    # Creates wordlist passwords file and opens it for writing
    wordlistPasswordsFile = open(wordlistPasswordsLocation, "w")

    # Prites the clean wordlist onto the opened file
    wordlistPasswordsFile.write(wordlistClean)

    # Prints the rockyou wordlist onto the opened file
    # This is done so only one file has the contents of rockyou, because
bruteforcing both username and password will be too long
    wordlistPasswordsFile.write(rockyou_contents)
    wordlistPasswordsFile.close

# Function used getting all the login fields
def getLoginFields(target, uri):
    print("Getting Login Fields")

    # Gate all the html code for the address
    page_html = requests.get(f"http://{target}{uri}").text
```

```
    # Parse html using beautifulsoup
    soup = BeautifulSoup(page_html, "lxml")

    # Find login form
    login_form = soup.find("form")

    # Retrived method used for logging
    method = login_form.get("method")

    # Find input fields in login form
    fields = login_form.find_all("input")

    # List for contents of input fields
    fieldContents = []

    # Loop for each field that was found
    for field in fields:
        # Get required values
        name_field = field.get("name")
        value_field = field.get("value")
        type_field = field.get("type")

        # Append a dictionary to the list
        fieldContents.append({
            'name_field': name_field,
            'value_field': value_field,
            'type': type_field
        })

    return fieldContents, method

# Function used for getting the message displayed on unsuccessful login
def getLoginFail(target, uri, fieldContents):
    print("Getting Login Fail Message")

    # Take the contents of the fields
    for fieldDictionary in fieldContents:
            if fieldDictionary['type'] == "text":
                username_name = fieldDictionary['name_field']
            elif fieldDictionary['type'] == "password":
                password_name = fieldDictionary['name_field']
            elif fieldDictionary['type'] == "submit":
                submit_name = fieldDictionary['name_field']
                submit_value = fieldDictionary['value_field']
            else:
```

```
                print("Automatic retrivel of login fail message unsucessfull.
Create hydra command manually")

    login_url = f"http://{target}{uri}"
    usernameContents = ""
    passwordContents = ""
    submitContent = submit_value

    # Send a POST request with login credentials
    payload = {
        username_name: usernameContents,
        password_name: passwordContents,
        submit_name : submitContent
    }

    # Get html of login page before and after attempting a login
    request_html = requests.get(login_url).text
    response_html = requests.post(login_url, data=payload).text
    # Parse the HTML responses
    soup1 = BeautifulSoup(request_html, 'lxml')
    soup2 = BeautifulSoup(response_html, 'lxml')
    # Convert the parsed HTML back to strings for comparison
    html_str1 = str(soup1)
    html_str2 = str(soup2)

    # Compare the html of both
    differ = difflib.Differ()
    differences = list(differ.compare(html_str1.splitlines(),
html_str2.splitlines()))
    f = open(loginFailLocation, "w+")
    for diff in differences:
        if diff.startswith('+ '):
            f.write(diff[2:])

    f.seek(0)
    login_fail_contents = f.read()
    f.close()

    # Find the index of the first ">" character
    start_index = login_fail_contents.find(">") + 1
    # Find the index of the second "<" character, starting from the position
after the first ">"
    end_index = login_fail_contents.find("<", start_index)
    # Extract the desired substring
    extracted_text = login_fail_contents[start_index:end_index].strip()
```

```
    return extracted_text

# Function used creating hydra command
def hydraBuilder(target, uri, is401, fieldContents, method):
    print("Creating hydra command")

    # Checks if it nees to build a command for a 401 page
    if is401:
        hydra_command = f"hydra -f -u -o {credentialsJSONLocation} -b json -L
{wordlistUsernamesLocation} -P {wordlistPasswordsLocation} {target} http-get
{uri}"
    else:
        # Gets messaged displayed on failed login
        login_fail = getLoginFail(target, uri, fieldContents)
        if method == "post":
            method = "http-post-form"
        else:
            print("Error getting method for hydra command. Try using hydra
manually using wordlists found in wordlists directory")

        username_field = None
        password_field = None
        extra_field = None

        # Loop for each field that was found from getLoginFields
        for fieldDictonary in fieldContents:
            if fieldDictonary['type'] == "text":
                username_field = fieldDictonary['name_field']
            elif fieldDictonary['type'] == "password":
                password_field = fieldDictonary['name_field']
            elif fieldDictonary['type'] == "submit":
                extra_field = {
                    'name' : fieldDictonary['name_field'],
                    'value' : fieldDictonary['value_field']
                }
            else:
                print("Unable to setup hydra command. Use manual hyndra")

        form =
f"{uri}:{username_field}=^USER^&{password_field}=^PASS^&{extra_field['name']}={ex
tra_field['value']}:{login_fail}"
        print(f"FORM IS: {form}")
        hydra_command = f"hydra -f -u -o {credentialsJSONLocation} -b json -L
{wordlistUsernamesLocation} -P {wordlistPasswordsLocation} {target} {method}
```

```
'{form}'"

    subprocess.run(hydra_command, shell=True)

# Function used for auto login into pages
def autoLogin(addressesUNAUTH, addressesOK, nikto_vulns, is401, logged):
    # Create directories
    os.mkdir(wordlistDirectory)
    os.mkdir(credentailsDirectory)
    os.mkdir(extrasDirectory)

    # Check if nikto discover a login page and set uri to uri of login page
    uri = checkIfLoginNikto(nikto_vulns)

    # If login page exist uri is set to a value and this executes
    if uri:
        print("Nikto login found. Creating wordlist")
        print("Login page uri:", uri)

        # Gets all the text from the pages that returned 200
        getComments(addressesOK)

        # Gets the information about the login field and the method used for
login
        fieldContents, method = getLoginFields(target, uri)

        # Used to build hydra command
        hydraBuilder(target, uri, is401, fieldContents, method)

    # If there is pages that returned 401
    elif addressesUNAUTH:
        is401 = True
        print("401 page exists. Creating wordlist from page contents")
        print("401 page Address:", addressesUNAUTH)

        # Gets all the text from the pages that returned 200
        getComments(addressesOK)

        # Gets the url for page that returned 401 in the format
http://10.10.10.10/uri
        url = addressesUNAUTH[0]

        # Splits url at every '/'
        parts = url.split("/")
```

```
        # Gives everything to the right of the 3rd '/' which is the uri
        result = "/".join(parts[3:])

        # Sets the uri to be /uri
        uri = "/" + result

        # Method/login field are not needed for http-get login so they are set to
none
        fieldContents = None
        method = None
        hydraBuilder(target, uri, is401, fieldContents, method)

    with open(credentialsJSONLocation, 'r') as json_file:
        data = json.load(json_file)

    # Extract login and password and from json file if it contains a sucessfull
result
    if len(data['results']) > 0:

        # Extract username and password from json file
        username,password = extractCredentials()

        # Set logged to true to indicate login credentials were found
        logged = True

        # Take screenshots of pages after logging in
        loginPageAddress = target + uri

        if screenshotScript:
            grabScrenshotLogged(loginPageAddress, addressesUNAUTH, username,
password)

        # Run dirb again with logged set to true which makes it use credentials
        runDirb(target, logged)

# Function used for extracting credentails that were found from credentials.json
file created by hydra
def extractCredentials():
    #readJsonfile if it exist
    print(f"Reading credentails from {credentialsJSONLocation}")
    with open(credentialsJSONLocation, 'r') as json_file:
        data = json.load(json_file)

    login = data['results'][0]['login']
    password = data['results'][0]['password']
```

```
    # Save login and password to a file
    with open(credentialsTXTLocation, 'w') as file:
        file.write(f"{login}:{password}")
    print("Login and password saved successfully.")
    return login,password

# Function for taking screenshots of pages that require logging in
def grabScrenshotLogged(loginPageAddress, addressesUNAUTH, username, password):
    # Check if there was address that returned 401
    if addressesUNAUTH:
        # Iterate over each address
        for addressUNAUTH in addressesUNAUTH:
            # Create driver
            driver = webdriver.Firefox()

            # Get everything after http://
            split = addressUNAUTH.split("//")[1]

            # Open address usng username and password for authentication
            driver.get(f"http://{username}:{password}@{split}")

            sleep(1)

            # Remove ":" and "/" from address using replace()
            file_name = addressUNAUTH.replace(":", ".").replace("/", ".")

            # Take a screenshot and save it
            driver.get_screenshot_as_file(f'{screenshotsDirectory}/{file_name}.pn
g')

            # Exit driver
            driver.quit()
    # Runs if there was no 401 pages, but there was a login page
    else:
        print("Does not have 401 pages")

        # Create driver
        driver = webdriver.Firefox()

        # Open address
        driver.get(f"http://{loginPageAddress}")

        # Gate all the html code for the address
        page_html = requests.get(f"http://{loginPageAddress}").text
```

```
        # Parse html using BeautifulSoup
        soup = BeautifulSoup(page_html, "lxml")

        # Find login form
        login_form = soup.find("form")

        # Find input fields in login form
        fields = login_form.find_all("input")

        # List for contents of input fields
        fieldContents = []

        # Loop for each field that was found
        for field in fields:
            # Get required values
            name_field = field.get("name")
            value_field = field.get("value")
            type_field = field.get("type")

            # Append a dictionary to the list
            fieldContents.append({
                'name_field': name_field,
                'value_field': value_field,
                'type': type_field
            })

        # Loop for each dictionary in the fieldContents lists
        for fieldDictionary in fieldContents:
            if fieldDictionary['type'] == "text":
                username_name = fieldDictionary['name_field']
            elif fieldDictionary['type'] == "password":
                password_name = fieldDictionary['name_field']
            elif fieldDictionary['type'] == "submit":
                submit_name = fieldDictionary['name_field']
            else:
                print(f"Automatic login at http://{loginPageAddress} unsuccesful.
Try logging in manually with credentials found in credentials/credentials.txt")

        # Locate username field using the name of the element for the driver to
use at input
        usernameField = driver.find_element(By.NAME, username_name)

        # Locate password field using the name of the element for the driver to
use at input
        passwordField = driver.find_element(By.NAME, password_name)
```

```
        # Send keys to fields
        usernameField.send_keys(username)
        passwordField.send_keys(password)

        # Locate submit button and click it
        driver.find_element(By.NAME, submit_name).click()

        sleep(1)

        # Remove ":" and "/" from address using replace()
        file_name = loginPageAddress.replace("/", ".")

        # Take a screenshot and save it
        driver.get_screenshot_as_file(f'{screenshotsDirectory}/{file_name}.png')

        # Exit driver
        driver.quit()

# Function for saving output to word file
def saveOutputWord(filename, tool_name, output):
    # Opening docx file
    document = Document(filename)

    # Create a table and add it to document
    table = document.add_table(rows=2, cols=1)

    # Get cell on first row
    toolNameCell = table.cell(0, 0)
    toolNameCell.text = tool_name

    # Styling
    toolNameCell_paragraph = toolNameCell.paragraphs[0]
    toolNameCell_paragraph.style = document.styles['Heading 1']
    toolNameCell_paragraph.alignment = WD_PARAGRAPH_ALIGNMENT.CENTER

    # Get cell on second row
    contentCell = table.cell(1, 0)
    contentCell.text = output

    document.save(filename)
    print(f"{tool_name} saved to {filename}")

# Function adding created images to word file
def addImageWord(filename):
```

```
    # Opening docx file
    document = Document(filename)

    # Create heading
    document.add_heading("Captured Screenshots", level=0)

    # iterate over screenshots in screenshots directory
    for screenshot in os.listdir(screenshotsDirectory):

        # Create path to screenshots by joining hte screenshots directory
location and the screenshot name
        screenshotFile = os.path.join(screenshotsDirectory, screenshot)

        # checking if file exists at location
        if os.path.isfile(screenshotFile):
            # Create a table and add it to document
            table = document.add_table(rows=2, cols=1)

            # Get cell on first row
            nameCell = table.cell(0, 0)
            nameCell.text = f"Screenshot: {screenshotFile}"

            # Style
            nameCell_paragraph = nameCell.paragraphs[0]
            nameCell_paragraph.style = document.styles['Heading 1']
            nameCell_paragraph.alignment = WD_PARAGRAPH_ALIGNMENT.CENTER

            # Get cell on second row
            imageCell = table.cell(1, 0)
            imageCell_paragraph = imageCell.paragraphs[0]

            # Adding picture to paragraph in cell
            imageCell_paragraph.add_run().add_picture(screenshotFile,
width=Cm(15), height=None)

    document.save(filename)
    print(f"Added picture to {filename}")

# Function for adding created wordlist to word file
def addWordlistWord(filename):
    # Opening docx file
    document = Document(filename)

    # Creating a heading
    document.add_heading("Created wordlist", level=0)
```

```
    # Opening wordlist
    wordlistUsernamesFile = open(wordlistUsernamesLocation)

    # Reading wordlist
    content = wordlistUsernamesFile.read()

    # Adding contents of wordlist file to document
    document.add_paragraph(content)

    document.save(filename)
    print(f"Added comments to {filename}")

# Function for adding created credentials to word file
def addCredentialsWord(filename):
    # Opening docx file
    document = Document(filename)

    # Creating a heading
    document.add_heading("Discovered credentials", level=0)

    # If credentials worked for logging in
    if os.path.exists(credentialsTXTLocation):
        # Open credentials file
        credentialsTXTFile = open(credentialsTXTLocation)

        # Read credentials files
        content = credentialsTXTFile.read()

        # Adding contents of credentials file to document
        document.add_paragraph(content)
    else:
        document.add_paragraph("No credentials were found")

    print(f"Added credentials to {filename}")

    document.save(filename)

# Function for conveting docx file to pdf
def convertDocxToPdf():
    # Read the DOCX file
    command = [f"libreoffice --headless --convert-to pdf {docxReportLocation} --
outdir {pdfReportLocation}"]
    subprocess.run(command, shell=True)
```

```
# Function used for demonstrating report creation from provided output folder
def createReport(logged):
    if os.path.isdir(reportsDirectory):
        shutil.rmtree(reportsDirectory)
    os.mkdir(reportsDirectory)
    #Create document
    document = Document()
    # Add a title
    document.add_heading('Scans Output', level=0)
    document.save(docxReportLocation)

    # Saves nmap output in docx
    nmap_output = extractInfoNmap(outputNmapLocation)
    saveOutputWord(docxReportLocation, 'Nmap Scan Output', nmap_output)

    # Variables used for storing all vulnerabilities discovered by nikto
    nikto_vulns = []
    # Saves nikto output in docx
    nikto_output = extractInfoNikto(outputNiktoLocation, nikto_vulns)
    saveOutputWord(docxReportLocation, 'Nikto Scan Output', nikto_output)

    # Saves dirb output in docx
    dirb_output = open(outputDirbLocation, "r")
    saveOutputWord(docxReportLocation, 'Dirbuster Scan Output',
dirb_output.read())
    dirb_output.close()

    # Saves dirb logged output in docx
    if logged:
        dirbLogged_output = open(outputDirbLoggedLocation, "r")
        saveOutputWord(docxReportLocation, 'Dirbuster Logged Scan Output',
dirbLogged_output.read())
        dirbLogged_output.close()

    # Runs if user used the -S switch to turn the screenshot script on or the -0
switch to turn using output folder only on
    if screenshotScript or outputOnly:
        # Adds images to docx
        addImageWord(docxReportLocation)

    # Runs if user used the -L switch to turn the login script on or the -0
switch to turn using output folder only on
    if loginScript or outputOnly:
        # Adds wordlist to docx
        addWordlistWord(docxReportLocation)
```

```
        # Adds discovered credentials to docx
        addCredentialsWord(docxReportLocation)

# Runs if user used the -O switch to turn using output folder on
if outputOnly:
    createReport(True)
    convertDocxToPdf()
else:
    main()
```