Network Security Assessment

Penetration Testing

Centre For Cybersecurity

Edwin Lum
S8
CFC130124

TABLE OF CONTENTS

Introduction

In today's digital landscape, it is important to ensure the security of network infrastructure. Cybersecurity threats are increasingly sophisticated, making it essential for organisations to proactively identify and mitigate potential vulnerabilities.

To address this need, this Bash script was developed to automate the penetration testing of a network through the scanning of network ports, services, and weak passwords, as well as mapping potential vulnerabilities. This script is designed to streamline the process of network security assessment, providing a thorough and efficient method for identifying security weaknesses.

The primary purpose of this script is to facilitate the early detection of vulnerabilities within a network. By automating the scanning process, the script helps cybersecurity professionals and network administrators to quickly identify open ports and services running that could be exploited, detect weak passwords that could allow unauthorised access and map known vulnerabilities associated with identified services.

Methodologies

Network Security Assessment

This script helps to automate the basic penetration testing of a network. The steps taken to scan and identify a given network are as follows:

1) Gather user input for the IP address to scan, absolute path to save the output files and basic or full scan
2) Run scans on network to find open ports and respective service versions
3) Check for seclists installation on user machine
4) Scan for weak passwords using *hydra* and *nmap* NSE scripts, while giving users the option to choose their password list
5) Scan ports for potential vulnerabilities using *nmap* NSE script and allowing user to search the results for a specific service's vulnerabilities if user chose full scan
6) Consolidate all relevant output files into a zip file

Throughout the script, *sleep* and *echo* commands are used to ensure that the output is easily readable.

Part 1

```
#!/bin/bash

#gather user input for network to scan
#while loop with regex is used as input validation for users to input blocks of digits separated by '.'
read -p 'Please key in an IP address to scan: ' ipadd

while [[ ! $ipadd =~ [0-9]+\.[0-9]+\.[0-9]+\.[0-9]+$ ]]
do
    read -p "Please key in a valid IP address: " ipadd
done

#while loop is used as input validation for users to input an existing directory in the user's machine
while :
do
    read -p 'Please key in the absolute path to save your scanned results: ' outputdir
    if [[ -d $outputdir ]]
    then
        cd $outputdir
        break
    else
        echo "Directory does not exists, please try again."
    fi
done

#gather user input for basic or full scan on network
read -p "Please choose either a basic or full scan [basic/full]: " scantype

#while loop is used as input validation for users to input only 'basic' or 'scan' strings
while [[ $scantype != 'basic' && $scantype != 'full' ]]
do
    read -p "Please key in only 'basic' or 'full': " scantype
done

echo
```

Figure 1: Part 1 gathers user input

Part 1 of the script takes in user input in preparation for the scans

Lines 5 - 10
The *read* command is used to read user input. The -p flag allows for a prompt to be displayed before reading the input. The user input for IP address is saved into the *ipadd* variable.

The *while* loop uses regular expressions (regex) to check for any user input that is not 4 blocks of digits with a '.' between each block of digits and then reading the user input for IP address, looping the user back to the *read* command on line 9 if the user input does not follow the regex.

The =~ operator is a regular expression match operator while the +$ ensures that the regex string preceding it is at the end of the line.

Lines 13 - 23
The *read* command takes the user input for the output directory and saves it into the *outputdir* variable.

The *while* loop changes directory according to the variable. If the script is unable to change directory to the output directory, a notification is printed and the user loops back to the *read* command. If the script successfully changes directory, *break* statement breaks out of the *while* loop and moves on to the next lines of the script.

Lines 26 - 32
The *read* command takes the user input for basic or full scan and saves it into the *scantype* variable.

The *while* loop checks for the *scantype* variable. If it does not match with either 'basic' or 'full' strings, the loop will repeat, asking for user input again.

Part 2

```
36  #run basic scan using nmap for tcp ports and masscan for udp ports
37  #nmap is further used for udp ports to get the service versions
38  echo "Scanning $ipadd for ports and services..."
39  echo
40  sudo nmap -sV -p- $ipadd > tcpsvscan.txt 2>&1
41  sudo masscan -pU:1-1000 $ipadd  > udpscan.txt 2>&1
42
43  #regex used to extract open udp port numbers
44  #for loop is used to run nmap on multiple open udp ports
45  udpmasscan=$(grep -Po "(?<=open\sport\s)(.*?)(?=\/udp)" udpscan.txt
46  for eachport in $udpmasscan
47  do
48      sudo nmap -sU -sV -p$eachport $ipadd >> udpsvscan.txt 2>&1
49  done
```

Figure 2: Part 2 uses *nmap* to scan for ports and services

Lines 38 - 41
A line is printed to notify the user that port and services scanning is taking place.

Root privileges is used for the *nmap* command through the use of *sudo*. The -sV flag is used for service scan and -p- flag runs the command for every port. IP address is called from the user input for *ipadd* variable.
The output is saved into a tcpsvscan.txt file, with 2>&1 used to combine the standard error and standard output into the standard output stream and they are saved into the file, making the script run cleaner.

Root privileges is also used for the *masscan* command. The command was only run for the first 1000 ports using the -pU:1-1000 flag as the script will run faster.

Regex was used to extract the port numbers using regex lookahead and lookbehind to extract the substring between 'open port ' and '/udp', with the port numbers saved into the variable, *udpmasscan*.

To get the service versions of multiple open ports, a *for* loop was run on *udpmasscan*, using *nmap* with the -sU flag for a more comprehensive scan on the udp ports. The output will then be appended into the udpsvscan.txt file.

Part 3

```
51  #check for seclists installation
52  #seclists installed if it is not found on user machine
53  echo "Checking if seclists is installed on your machine..."
54  sleep 2
55  sudo updatedb
56  if [[ $(locate seclists) == *"/usr/share/seclists"* ]]
57  then
58      echo "Seclists is installed on your machine."
59  else
60      echo "Installing Seclists on your machine..."
61      sudo apt-get update > /dev/null
62      sudo apt-get -y install seclists > /dev/null
63      echo "Seclist installation complete."
64  fi
65
66  echo
```

Figure 3: Part 3 checks for seclists installation

Lines 53 - 66
A line is printed to notify the user on the check for seclists installation.

sudo updatedb updates the database of file names for *locate* command. The *if* statement is used to locate the directory for seclists. If the seclists directory is not found, it will be installed on the user machine. To reduce any unnecessary output, /dev/null was used to discard any standard output and standard error.

Part 4A

```
68  #let user choose password list to use for brute force attack
69  #while loop and if statements used as input validation for user to input either 'y' or 'n'
70  while :
71  do
72      read -p "Please choose if you want to use your own password list for brute force attack [y/n]: " pwlist
73      if [[ $pwlist == 'y' ]]
74      then
75          read -p "Please give the absolute path for your preferred password list: " pwpath
76          pwfile=$(find "$pwpath")
77          if [[ "$pwfile" ]]
78          then
79              echo "$pwpath selected."
80              echo
81              sleep 2
82              break
83          else
84              echo "$pwpath is invalid."
85          fi
86      elif [[ $pwlist == 'n' ]]
87      then
88          pwpath=/usr/share/seclists/Passwords/darkweb2017-top10.txt
89          echo "Default $pwpath used."
90          echo
91          sleep 2
92          break
93      else
94          echo "Incorrect input."
95      fi
96  done
```

Figure 4: Part 4A lets user choose password list for brute force attack

Lines 70 - 96

The *while* loop is used to ensure that the user inputs only ‘y’ or ‘n’ for choosing the password list to be used for brute force attack.

The *read* command is used with -p flag to get the user to input whether they want to use their own password list. The input is saved into the *pwlist* variable.

The *if* statement is used to verify that the *pwlist* variable is either ‘y’ or ‘n’, otherwise it would loop back to the original *read* command.

If the user chose ‘y’, the user would need to input an absolute path for the password list, which is saved into the *pwpath* variable. The variable is checked using the *find* command and saved into another variable, *pwfile*. If the user chose ‘n’, the default seclists password list would be used, and the *break* statement breaks out of the *while* loop.

The nested *if* statement checks for *pwfile* variable to be true. If true, *break* statement breaks out of the *while* loop and if otherwise, the user is directed back to the first *read* command in the *while* loop.

Part 4B

```
 98  echo "Scanning $ipadd for weak passwords..."
 99  echo
100
101  #function checks if each port is open and then brute forces each port using hydra
102  #arg passed to choose the port and the login credentials are extracted using regex
103  bruteport () {
104  if [[ $(cat tcpsvscan.txt) == *"open  $1"* ]]
105  then
106      echo "Commencing brute force attack on $1..."
107      hydra -L /usr/share/seclists/Usernames/top-usernames-shortlist.txt -P $pwpath $ipadd $1 > hydra$1.txt 2>&1
108      if [[ $(cat hydra$1.txt) == *"login: "* ]]
109      then
110          echo "Brute force attack successful."
111          sleep 2
112          echo "The $1 login name(s):"
113          echo "$(grep -Po '(?<=login\:\s)(.*?)(?=\s\s\spassword)' hydra$1.txt)"
114          sleep 2
115          echo "The corresponding $1 password(s):"
116          echo "$(grep -Po '(?<=password\:\s).*' hydra$1.txt)"
117          sleep 2
118      else
119          echo "Brute force attack failed."
120          sleep 2
121      fi
122  else
123  echo "The $1 port is not open."
124  sleep 2
125  fi
126  echo
127  }
128
129  bruteport ftp
130  bruteport ssh
131  bruteport rdp
```

Figure 5: Part 4B checks for weak passwords using *hydra*

Lines 98 - 131

The function, *bruteport*, is defined to reduce repetitive code while brute forcing the various ports, ftp, ssh and rdp. The *$1* variable allows the service to be passed as an argument while calling for the function.

In the function, an *if* statement is used to check for the open port stated in the argument using a string, 'open *$1*', from the tcpsvscan.txt file. The user will be informed if the port is not open.

If the port is open, users are informed of the commencement of the brute force attack on the respective service. The *hydra* command is used, with username list pre-defined in the command and password list used based on the user selection in Part 4A. IP address is also called from the variable, *ipadd*, given by the user in Part 1. The output is saved into a hydra*$1*.txt file.

The nested *if* statement checks for successful brute force attack using the text file. A successful brute force attack would print the login names and corresponding passwords using regex lookahead and lookbehind to check for the relevant information. The -Po flag allows the *grep* command to read Perl regex and only match nonempty parts of the line.
The user will also be informed if the brute force attack failed.

The last few lines show the *bruteport* function being called with the service to be brute forced being passed as the argument.

Part 4C

```
133  #For brute force of Telnet port, NSE script is used instead
134  if [[ $(cat tcpsvscan.txt) == *"open  telnet"* ]]
135  then
136      echo "Commencing brute force attack on telnet..."
137      sudo nmap --script-updatedb > /dev/null
138      nmap -p23 --script telnet-brute --script-args \
139      userdb=/usr/share/seclists/Usernames/top-usernames-shortlist.txt,passdb=$pwpath,telnet-brute.timeout=8s $ipadd > nsetelnet.txt 2>&1
140      if [[ $(cat nsetelnet.txt) == *"Valid credentials"* ]]
141      then
142          echo "Brute force attack successful."
143          sleep 2
144          echo "The login credentials are:"
145          echo "$(grep -Po '(?<=\|\s\s\s\s\s)(.*?)(?=\s\-\sValid)' nsetelnet.txt)"
146      else
147          echo  "Brute force attack failed."
148      fi
149  else
150      echo "The telnet port is not open."
151      sleep 2
152  fi
```

Figure 6: Part 4C checks for weak password uses NSE script for telnet

Lines 134 – 152

The *if* statement is used to check for open telnet ports from the tcpsvscan.txt file. The user will be informed if the port is not open.

If the telnet port is open, the user is informed of the brute force attack on telnet. The database of NSE scripts is first updated. The *nmap* command is then used to run the NSE script, telnet-brute to check for weak passwords on telnet for the network chosen by the user. The '\' in the command splits the command into 2 lines to make it more readable. The arguments are passed to show the paths for the lists of usernames and passwords, as well as the script timeout value. The output is saved into the nsetelnet.txt file.
hydra is not used to brute force telnet as it is unreliable compared to telnet-brute.

The *if* statement then checks for successful brute force attacks by checking for a string in the file showing 'Valid credentials'. Users will be informed of a successful brute force attack and the login credentials will be printed using regex lookahead and lookbehind.
The user will be informed if the brute force attack fails.

Part 5

```
154  #if user chose full scan, run NSE vulners script to check for vulnerabilities
155  #results of vulnerabilities are summarised into another text file to show only the open ports and all enumerated CVEs related to each port
156  if [[ $scantype == 'full' ]]
157  then
158      echo
159      echo "Scanning $ipadd ports for vulnerabilities..."
160      nmap -sV --script vulners $ipadd > vuln.txt
161      grep -Ei ' open  |cve' vuln.txt > vulnsummary.txt
162      read -p "Please select the service with vulnerabilities that you would like to search for: " readvuln0
163      readvuln=${readvuln0,,}
164      grep $readvuln vuln.txt | cut -f 1 -d ' '| grep '[0-9]' | awk -F/ '{print$1}' >> vulnport.txt
165      sleep 2
166
167      for number in $(cat vulnport.txt)
168      do
169          nmap -sV --script vulners 192.168.163.132 -p$number | grep -Ei " $readvuln |cve" >> vuln$readvuln.txt
170      done
171
172      if [[ $(cat vuln$readvuln.txt) == *" $readvuln "* ]]
173      then
174          echo "Vulnerabilities in $readvuln have been found."
175          sleep 2
176          echo
177          cat vuln$readvuln.txt
178          sleep 2
179          echo
180          echo "Your search results have been saved in vuln$readvuln.txt."
181          sleep 2
182      else
183          echo "No vulnerabilities have been found in $readvuln."
184          sleep 2
185      fi
186  fi
187
188  echo
```

Figure 7: Part 5 scans the network for vulnerabilties

Lines 156 - 188

The *if* statement checks for user input choice of full scan. If full scan was chosen, a line is then printed to inform the user that the ports will be scanned for vulnerabilities.

The *nmap* command is used to run the NSE script, vulners, to check for vulnerabilities in the IP address given by the user and the full output is saved in vuln.txt. The *grep* command is used to check for open ports and CVE enumerated, and then summarise the original output in vuln.txt into vulnsummary.txt. The -Ei flag allows the command to read extended regex and ignore the case sensitivity in the string.

User input is then taken for the service with vulnerabilities that the user wants to search for. It is saved into *readvuln0* variable. The *readvuln0* variable is then converted to lowercase and saved as the *readvuln* variable.

The vuln.txt is then parsed to obtain only the port numbers and appended to the vulnport.txt file. A *for* loop is used to run the *nmap* with the vulners script on multiple ports running the specified service in the *readvuln0* variable. The output is appended into the vuln*$readvuln*.txt file.

The *if* statement checks for the service in the vuln*$readvuln*.txt file. If the service has vulnerabilities, it will be found in the output file. The user will be informed and the details of the vulnerabilities will be printed out for the user.
If there are no vulnerabilities found for the service, a line will be printed to inform the user.

Part 6

```
#all results are saved into the networkvuln.zip file and all other working files in the output directory are removed
zip -m networkvuln.zip tcpsvscan.txt udpscan.txt udpsvscan.txt hydraftp.txt hydrassh.txt \
hydrardp.txt nsetelnet.txt vuln.txt vulnsummary.txt vuln$readvuln.txt > /dev/null

echo "All results are saved in 'networkvuln.zip' file."
sleep 2
echo "You may retrieve the service versions of the respective tcp and udp ports from tcpsvscan.txt and udpsvscan.txt in the zip file."
sleep 4
echo "You may retrieve the weak passwords of the respective ports (if any) from hydraftp.txt, hydrassh.txt, hydrardp.txt and nsetelnet.txt in the zip file."

if [[ $scantype == 'full' ]]
then
    rm vulnport.txt
    sleep 4
    echo "You may retrieve the respective complete and summarised vulnerabilities (if any) from vuln.txt and vulnsummary.txt in the zip file."
fi
```

Figure 8: Part 6 consolidates the files in a zip file and informs the user

Lines 191 - 205
The text files containing the output are zipped into the networkvuln.zip file. The -m flag moves the files from the output directory into the zip file.

The next few printed lines inform the users the contents of each file.

The last *if* statement checks for the *$scantype* variable to be 'full' and removes the additional working file, vulnport.txt. The last line printed informs the user of where to find the output after the vulnerability scan.

Discussion

Network Security Assessment

The script is designed to enhance network security by automating the scanning of ports, services, weak passwords, and mapping potential vulnerabilities. It also allows users to input target network details and performs comprehensive scans, compiling the results into a zip file for easy access and further analysis.

Advantages of the Script

The script uses regex and *while* loops to ensure the user inputs are valid. This reduces the likelihood of user errors and ensures the script operates on correct inputs.

The script is generally automated except for several user inputs. A fallback to a default password list from seclists helps users run the script with minimal configuration. The automated installation if seclists is not installed on the user machine further ensures the script has all necessary resources available without manual intervention.

Saving the scan results into a zip file allows for easy review and sharing of findings. This also helps in maintaining a clean output directory by removing intermediate files.

As each feature is developed as a separate module, future code improvements and debugging would be more straightforward.

The option for both basic and full scan provides flexibility to the user on how thorough the scan is required.

Areas for Improvement for the Script

The script is unable to run using IPv6 addresses, and hence may not be compatible to be used on every network.

The script requires root privileges for certain operations, which can be a security concern if the script is run on sensitive systems. The script can also only be run on accounts that have root privileges.

The *masscan* used for UDP scan only scans for the first 1000 ports to decrease the time needed for the script to complete running.

There is also no valid check for the user input password list, beyond ensuring that the password list is on the user machine. If the absolute path for the password list is an invalid file, it could lead to an error when the user password list is passed into *hydra* command.

The default password list is very short to decrease the time required to run the script, which would lead to a higher chance of brute force attack not returning weak passwords.

The use of the *bruteport* function for *hydra* also limits the brute force attacks on the respective services to only their default ports. If the ports for the services are swapped, the brute force attack would automatically fail.

The script has minimal error handling for failed commands or unexpected issues like network issues or permission errors. This can lead to incomplete scans or missing results without clear indications of what went wrong.

Output of Script

```
┌──(kali㉿kali)-[~]
└─$ bash projvulner.sh
Please key in an IP address to scan: 192.168.163.132
Please key in an output directory to save your scanned results: /home/kali/testmsf
Please choose either a basic or full scan [basic/full]: full

Scanning 192.168.163.132 for ports and services ...

Checking if seclists is installed on your machine ...
Seclists is installed on your machine.

Please choose if you want to use your own password list for brute force attack [y/n]: y
Please give the absolute path for your preferred password list: /usr/share/seclists/Passwords/citrix.txt
/usr/share/seclists/Passwords/citrix.txt selected.

Scanning 192.168.163.132 for weak passwords ...

Commencing brute force attack on ftp ...
Brute force attack successful.
The ftp login name(s):
ftp
ftp
ftp
The corresponding ftp password(s):
nsroot
v9Yx*6uj
Unidesk1

Commencing brute force attack on ssh ...
Brute force attack failed.

The rdp port is not open.

Commencing brute force attack on telnet ...
Brute force attack successful.
The login credentials are:
user:user
```

Figure 9: Output of script from Part 1 to Part 4C

The entire script runs automatically, only requiring several inputs like the IP address to be scanned, output directory to save the results and deciding whether a basic or full scan is required in Part 1.

Part 4A also requires user input to choose the password list to be used, and the absolute path of the password list if a user password list is chosen.

The output in Part 4B and 4C shows the login credentials if successful brute force attack occurs for any of FTP, SSH, RDP or Telnet.

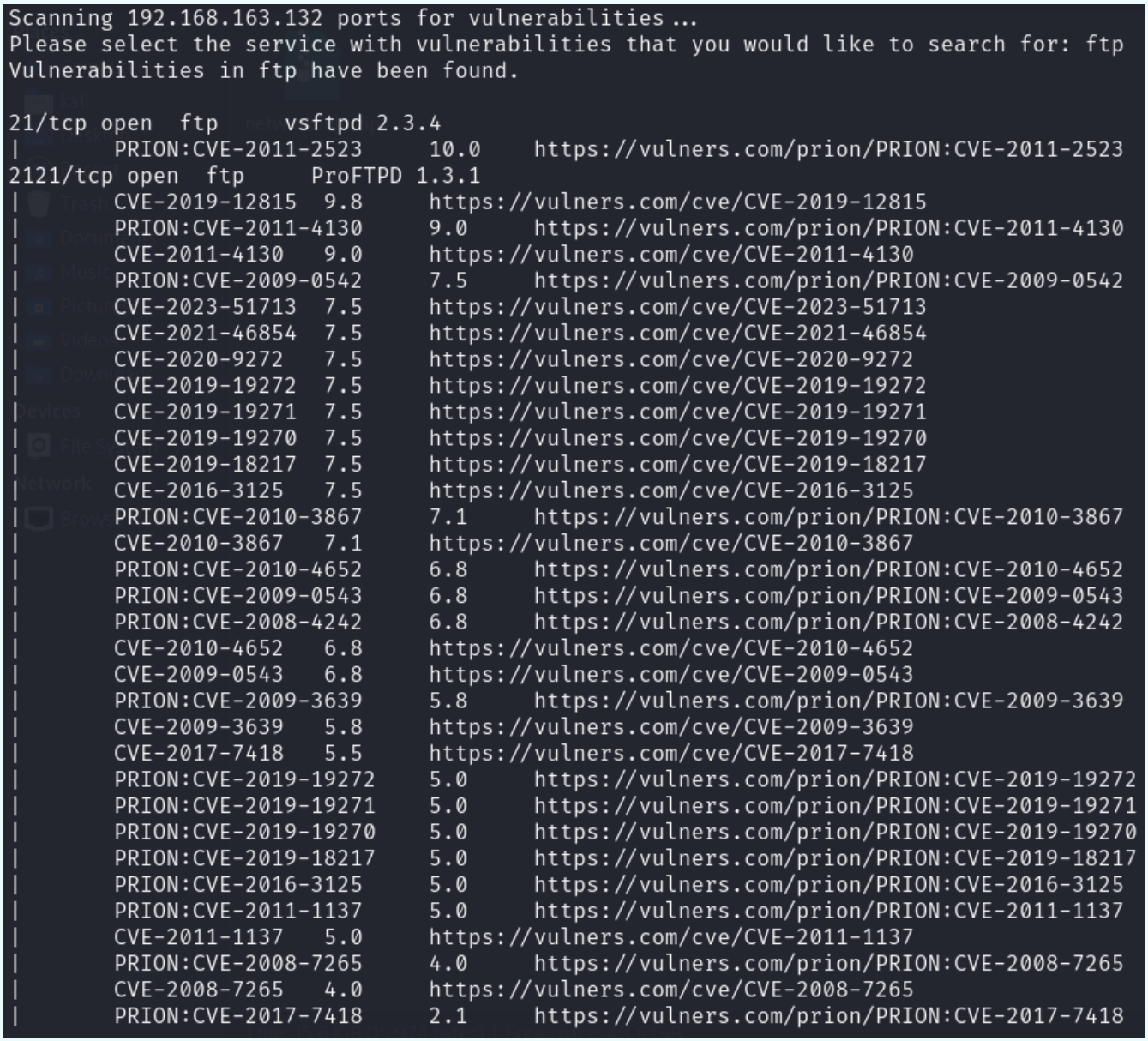

Figure 10: Output of script from Part 5

The output for Part 5 shows all the vulnerabilities enumerated from the *nmap* NSE script, vulners.

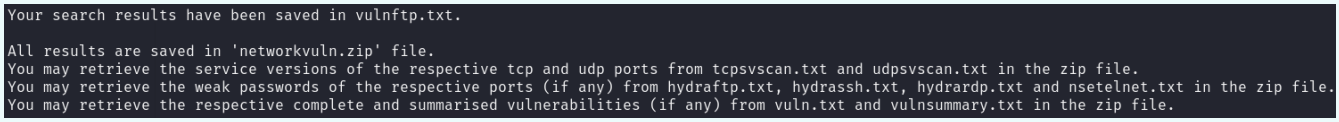

Figure 11: Output of script from Part 6

The output for Part 6 informs the user of where the files are saved and the name of the zip file.

Potential Uses of Script

The script can help cybersecurity professionals to identify weak passwords and misconfigured services. Thus, they are able to harden the network by addressing the weaknesses in network configuration to enhance overall security.

In the event of a security incident, this script can be used to quickly assess the network for potential vulnerabilities that may have been exploited. This rapid assessment can help incident responders to identify the root cause and scope of the incident more efficiently.

This script could potentially be useful for Small and Medium Enterprises (SMEs) as they often lack the resources for extensive security teams or expensive commercial scanning tools. This script provides an accessible and cost-effective solution for SMEs to perform basic security scans and protect their networks.

Conclusion

This Bash script is a versatile tool for a wide range of cybersecurity activities, from network hardening to incident response. By automating the scanning process and providing comprehensive results, it helps organisations of all sizes to improve their security posture and defend against cyber threats.

Users are given the option to do a quick basic scan or a more comprehensive full scan. In both scans, the *nmap* command scans for open ports and service versions of each port while *hydra* checks for weak passwords for several of the services. The network vulnerabilities are also scanned when a full scan is selected. The *nmap* NSE script, vulners, scans for the vulnerabilities and the user will be able to search for a specific service to check for vulnerabilities. All the results are then saved into a zip file.

This script is designed to simplify and streamline the process of network vulnerability assessment, making it accessible and efficient for cybersecurity professionals and organisations of all sizes. The automation increases efficiency of cybersecurity professionals, creating additional bandwidth for them to handle analytical work and strategic planning. The ability to search the results of vulnerability scanning also allows for more targeted remediation efforts and efficient prioritisation of security tasks.

In conclusion, this Bash script is a valuable addition to the cybersecurity toolkit, offering an efficient and effective means of conducting network vulnerability assessments. Its scanning capabilities, combined with user-friendly features and organised result management, make it a powerful tool for maintaining network security. As cybersecurity threats continue to evolve, such automated solutions are essential for staying ahead of potential vulnerabilities and ensuring the safety of digital assets.

Recommendations

Network Security Assessment

The input validation for IP address could give users an initial option of picking IPv6 and split the input validation for IPv4 and IPv6 into separate regex expressions, giving the option of using the script on networks that exclusively use IPv6.

The script can be customised to improve the *masscan* UDP scan to include all ports for more extensive scanning of the ports and services. In addition, longer default password and user lists can be used to improve the chances of finding weak passwords. However, more time would be required to complete the scan.

The basic scan could also be further improved by allowing user customisation through specifying custom port ranges for scanning, to lower time taken for running the script when a more focused scan is required.

Running vulnerability scans and brute force attacks should be done in a controlled, isolated environment. The networks and machines scanned can be replicated and tested in virtual machines to avoid unintended consequences.

To improve the *bruteport* function defined in the script, the port numbers from the *nmap* scan could be extracted from the tcpsvscan.txt and passed into the function. This would ensure greater reliability on the *hydra* brute force attack if the services on the network are not on their default port numbers.

Increase scalability of script by optimising script and allowing users to scan networks using CIDR. This would take much longer but all the devices in the network can be scanned at once and the scanned results can be easily consolidated.

Security of the script can be improved by implementing input sanitisation to prevent code injection attacks. Implementing logging of script actions would also leave an audit trail to track usage.

Error handling in the script can be enhanced by adding *set* -e command at the start of the script to ensure that the script terminates if any command exits with a non-zero status.

References

1. “Logo.” Bash Logo Media Assets, bashlogo.com/. Accessed 15 June 2024.

2. “Check for IP Validity.” Stack Overflow, stackoverflow.com/questions/13777387/check-for-ip-validity. Accessed 15 June 2024.

3. “How Do I Check If a Directory Exists or Not in a Bash Shell Script?” Stack Overflow, stackoverflow.com/questions/59838/how-do-i-check-if-a-directory-exists-or-not-in-a-bash-shell-script. Accessed 15 June 2024.

4. “Regex Match All Characters between Two Strings.” Stack Overflow, stackoverflow.com/questions/6109882/regex-match-all-characters-between-two-strings. Accessed 15 June 2024.

5. “Discussion: How to Run in Silent Mode/Concurrently · Issue #2021 · Facebookresearch/Hydra.” GitHub, github.com/facebookresearch/hydra/issues/2021. Accessed 15 June 2024.

6. “Bash - What Does " 2>&1 " Mean?” Stack Overflow, stackoverflow.com/questions/818255/what-does-21-mean. Accessed 15 June 2024.

7. “How to Check If a String Contains a Substring in Bash.” Stack Overflow, stackoverflow.com/questions/229551/how-to-check-if-a-string-contains-a-substring-in-bash. Accessed 15 June 2024.

8. “How to Grep for Multiple Strings, Patterns or Words.” Knowledge Base by PhoenixNAP, 5 May 2020, phoenixnap.com/kb/grep-multiple-strings. Accessed 15 June 2024.

9. “Bash Scripting Guide: How to Ensure Exit on Error.” Ioflood.com, ioflood.com/blog/bash-exit-on-error/#:~:text=The%20simplest%20way%20to%20make. Accessed 15 June 2024.