Penetration Testing | Project: Vulner

Kenneth Wong s32
CFC190324
Trainer: Samson

Table of Contents

Introduction

Manual scanning and testing processes can be time-consuming and prone to human error which can leave vulnerabilities unnoticed. This is where such a script becomes a useful tool. The script developed here is designed to streamline the process of identifying potential vulnerabilities and perform brute-force attacks on network services.

It offers the flexibility to choose between a basic or full scan which extends functionality to include vulnerability assessments using 'Searchsploit'. By automating these tasks, the script not only saves time but also ensures a thorough and consistent approach to security testing.

Key features of this script include customizable scanning options, automated vulnerability checking, brute-forcing capabilities and efficient results management. Users can select their type of scan and cater to each specific needs.

This script leverages 'Nmap' for service detection and 'Searchsploit' for vulnerability checking, automating the process entirely. Additionally, this script includes the function of being able to perform brute-force attacks on identified services with user-provided username and password files; testing the strength of credentials and uncovering weak entry points. Subsequently, all results are saved and can be optionally zipped for easy organisation and sharing.

By reducing on the manual checks, this script allows us to focus on analysing results and addressing any vulnerabilities identified, enhancing the overall security.

Methodologies and Discussion

For this script, we decided to go with more modular functions that target specific user-related events, ensuring both clarity and easy modifications in future.

Figure 1: Function for network

```
1  #!/bin/bash
2
3  #~ Function to get the network from user input in CIDR format
4  get_network() {
5      #~ Prompt the user to enter a network address to scan and store the input in the variable 'network'
6      read -p "Enter the network to scan (e.g., 192.168.1.0/24): " network
7  }
8
```

The above function is a simple yet essential part of the script that prompts the user to input a network address in CIDR format, which is then stored in a variable called 'network'. CIDR notation is commonly used to specify IP address ranges, making it a flexible and precise method for scanning a network range. When this function is executed, it displays a prompt ('-p'), asking the user to enter a network address such as '192.168.1.0/24', which represents a specific subnet.

Figure 2: Function for filepath

```
#~ Function to determine the filepath for saving results
get_filepath() {
    #~ Prompt the user to decide whether to use the default file path for saving results (yes/no)
    read -p "Use default file path for saving results? (yes/no): " use_default
    #~ Check if the user wants to use the default path
    if [[ "$use_default" == "yes" ]]; then
        #~ Set filepath variable to the default location on the Desktop
        filepath="$HOME/Desktop/vulner_results.txt"
    else
        #~ Enter a loop to repeatedly ask for a valid directory path until one is provided
        while true; do
            #~ Prompt the user to enter a custom directory path to save the scan results
            read -p "Enter the directory path to save the scan results: " custom_path
            #~ Check if the entered path is a valid directory with the -d flag
            if [[ -d "$custom_path" ]]; then
                #~ Set the filepath variable to the specified custom path with the filename
                filepath="$custom_path/vulner_results.txt"
                #~ Exit the loop as a valid directory path has been provided
                break
            else
                #~ Inform the user that the entered path is invalid and prompt again
                echo "The path '$custom_path' does not exist. Please enter a valid directory path."
            fi
        done
    fi
}
```

The above function is responsible for determining where the scan results will be saved. It begins by prompting the user to decide if they want to use the default file path, '/home/Desktop', by responding "yes" to "Use default file path for saving results? (yes/no)." After which, the function sets the 'filepath' variable to the predefined location.

If the user opts not to use the default path, the function enters a loop and repeatedly asks the user to provide a valid path until the result is accepted. Within this loop, the function checks if the provided path is valid using the '-d' flag. If a valid directory is given, the 'filepath' variable is set to the user-specified path and saves the results into the filename 'vulner_results.txt'.

With this function, it provides flexibility for the user to save in either the default location or enter their own custom path.

Figure 3: Function for nmap scan

```
#~ Function to perform an nmap scan $1 being either basic or full
perform_nmap_scan() {
    echo "Performing $1 scan on $network..."
    #~ Run nmap command with service version detection and saving into "nmap_output" variable
    nmap_output=$(nmap -sV "$network")
    #~ Save nmap scan results to the specified file path
    echo "$nmap_output" > "$filepath"
    #~ Let user know that the nmap scan is done
    echo "$1 network scan complete. Results saved to $filepath."
}
```

The 'perform_nmap_scan()' function is designed to carry out scans on the user specified network, with the scan type being determined by the input parameter '$1', which can either be for the basic or full scan.

This function is used for both basic and full scans which makes it a versatile part of the script. It begins by informing the user that it is performing the selected scan on the network defined earlier in the script. The function then runs 'nmap' command with the service version flag ('-sV') on the network and then stores the output in the 'nmap_output' variable. Once the scan is completed, the results are then saved to the earlier specified file path.

Figure 4: Function for commencing brute force

```
#~ Function to list discovered services and perform brute force attacks
list_services_and_bruteforce() {
    #~ Determine the current directory
    current_directory=$(pwd)

    #~ Prompt the user for the username and password file names
    read -p "Enter the name of the username file (located in the current directory): " username_file
    read -p "Enter the name of the password file (located in the current directory): " password_file

    #~ Reassigning variables to the user and password files
    username_list="$current_directory/$username_file"
    password_list="$current_directory/$password_file"

    #~ Check if the files exist in the current directory
    if [[ ! -f "$username_list" ]]; then
        echo "Username file '$username_file' not found in the current directory."
        exit 1
    fi

    if [[ ! -f "$password_list" ]]; then
        echo "Password file '$password_file' not found in the current directory."
        exit 1
    fi
```

The 'list_services_and_bruteforce()' function begins by determining the current working directory with the 'pwd' command as it utilises it for the user to provide their own username and password file to initiate the brute force scan. This sets the context for locating files relative to where the script is executed.

The function constructs the full paths to these files which the user has provided by combining the current directory path with the user-provided filenames, resulting in the variables 'username_list' and 'password_list'.

After setting up these paths, the function checks if the file exists in the current directory with the '-f' flag. If either file is not found, an error message will be displayed and the script exits with a status code of '1' which ensures that the necessary files are provided before proceeding to the subsequent portions of the script.

Figure 5: Service Discovery and Extraction from Nmap Output

```
    #~ Extract hosts with relevant services from nmap output
    #~ Initialize an empty array to store hosts and services found by the nmap scan.
    services_hosts=()
    #~ Start a while loop to read each line of the nmap output.
    while read -r line; do
        #~ Check if the line contains the string "Nmap scan report for".
        if [[ "$line" == "Nmap scan report for"* ]]; then
            #~ Extract the host name or IP address from the line.
            #~ Use awk to print the last field ($NF) and tr to remove parentheses.
            host=$(echo "$line" | awk '{print $NF}' | tr -d '()')
        #~ Check if the line contains the word "open", indicating an open port.
        elif [[ "$line" == *"open"* ]]; then
            #~ Use awk with -F to split by '/' and print the first field (port number)
            port=$(echo "$line" | awk -F/ '{print $1}')
            #~ Extract the service name from the line.
            service=$(echo "$line" | awk '{print $3}')
            #~ Check if the service is one of the specified types (ftp, ssh, rdp, telnet)
            if [[ "$service" == "ftp" || "$service" == "ssh" || "$service" == "ms-wbt-server" || "$service" == "telnet" ]]; then
                #~ Append the host, port, and service to the services_hosts array.
                services_hosts+=("$host:$port ($service)")
            fi
        fi
    #~ Pipe the filtered nmap output to the while loop.
    done <<< "$(echo "$nmap_output" | grep -E 'Nmap scan report for|open')"
```

The script as seen in *Figure 5* focuses on parsing the 'nmap' output to identify hosts and services relevant for brute force attacks, such as FTP, SSH, Telnet, and RDP.

It begins by creating an empty array called 'services_hosts' to store information about the hosts and services. The function then enters a 'while' loop which reads each line of the 'nmap' output. As each

line is being processed, it checks if it contains the string “Nmap scan report for”, which is used to identify each new host detected. With this, the function extracts the host’s IP address from the line. The ‘awk ‘{print$NF}’’ prints the last field of the line and ‘tr -d ‘()’ ‘ is used to remove any parentheses around it. *(As shown in Figure 6)* With this, it then stores the resulting IP address in the ‘host’ variable.

Figure 6: Host Results for Nmap Scan

```
13  Nmap scan report for 192.168.226.128
14  Host is up (0.00024s latency).
15  All 1000 scanned ports on 192.168.226.128 are in ignored states.
16  Not shown: 1000 closed tcp ports (conn-refused)
17
18  Nmap scan report for msf1 (192.168.226.132)
19  Host is up (0.00039s latency).
```

Subsequently, the function checks if the line contains the word “open” which signifies an open port which we are able to brute force into. If an open port is found, it uses ‘awk’ to split the line by ‘/’ to extract the port number and extracts the service name with “awk ‘{print $3}’”.

The function then filters for the specific services to brute force by comparing the service name to predefined values. If it matches, the host, port and service information will be appended into the ‘services_hosts’ array which was earlier created.

This array will eventually contain a list of hosts and their services which will be used to perform brute forcing. The ‘done <<< "$(echo "$nmap_output" | grep -E 'Nmap scan report for|open')"’ portion pipes the filtered nmap output into the loop, ensuring that only relevant lines are processed.

Figure 7: Validation and Display of Discovered Services

```
96      #~ Check if the services_hosts array is empty if no services found
97      if [ ${#services_hosts[@]} -eq 0 ]; then
98          echo "No relevant services found in the scan."
99          #~ Exit the function or loop if no services are found.
100         return
101     fi
102
103     #~ Print message to indicate that the list of available hosts
104     echo "Available hosts with relevant services:"
105     #~ Iterate over the indices of the services_hosts array
106     for i in "${!services_hosts[@]}"; do
107         #~ Print the index and the corresponding service host information from the array
108         echo "[$i] ${services_hosts[i]}"
109     done
```

This section of the script as shown in Figure 7 is designed to validate the services discovered during the Nmap scan and present the output to the user in an organised manner.

It starts by checking if the array contains any entries by evaluating the length of it with ‘${#services_hosts[@]}’ being equals to zero. If the array is empty, it means that no relevant services were found during the scan and with the ‘return’ command, it exits the loop.

If it is not empty, indicating that services were found, the script moves on to display the discovered services. The script then enters a ‘for’ loop that iterates over each index in the array and each variable will take on an index number. For each infex ‘I’, the script prints the index number in square brackets followed by the corresponding host and service information from the array.

Figure 8: Validation and Display of Discovered Services

```
    #~ Select IP address and service for brute force
    read -p "Select the index of the IP to brute force: " index
    #~ Retrieve the selected entry from the services_hosts list
    selected_entry="${services_hosts[$index]}"

    #~ Split the 'selected_entry' into 'host_info' and 'service_info' by spaces, where 'host_info' conta
    IFS=' ' read -r host_info service_info <<< "$selected_entry"
    #~ Split 'host_info' into 'host' and 'port' by colons, after removing any trailing '(service)' text.
    IFS=':' read -r host port <<< "${host_info% (*}"
    #~ Remove parentheses from 'service_info' to isolate the service name.
    service="${service_info//[()]/}"

    #~ Print message indicating the start of the brute force attack for the specified host, port, and se
    echo "Starting brute force on $host:$port ($service)"
    #~ Begin case statement to handle different types of services
    case $service in
        #~ If service is FTP
        ftp)
            hydra -L "$username_list" -P "$password_list" "ftp://$host:$port" >> "$filepath"
            ;;
        #~ If service is SSH
        ssh)
            hydra -L "$username_list" -P "$password_list" "ssh://$host:$port" >> "$filepath"
            ;;
```

The user is prompted to input the index of the host they wish to brute force and the selected index is stored in the variable 'index'. Next, the entry is retrieved from 'services_hosts' array at the position specified by 'index'. The retrieved entry which contains the host, port and service is then stored into the variable 'selected_entry'.

It is then split into two parts, 'host_info' and 'service_info'. The split is done based on the spaces. 'host_info' is then split into 'host' and 'port' with the colon. The part '${host_info% (*}' removes any trailing text that starts with a space and an opening parenthesis, leaving only the host and port. Thereafter, parentheses from 'service_info' are removed.

A case statement is then used as it would be able to handle the different types of services instead of using a 'if-else' statement as it would be harder to follow. This allows for future modifications of the script. The 'case' statement is then closed up with '*', matching any service type that has not been explicitly mentioned in the above.

Figure 9: Function for vulnerability check

```
# Function to perform vulnerability checking with searchsploit
check_vulnerabilities() {
    echo "Checking for vulnerabilities based on nmap scan results..."

    # Define the directory for saving the searchsploit results
    results_dir="${filepath%/*}/searchsploit_results"

    # Create the directory if it does not exist
    mkdir -p "$results_dir"

    # Define the combined result file
    combined_result_file="$results_dir/combined_searchsploit_results.txt"

    # Clear the combined result file if it exists
    > "$combined_result_file"

    # Extract service names and versions from nmap output
    while read -r line; do
        if [[ "$line" == *"open"* ]]; then
            # Extract the service name from the line
            service=$(echo "$line" | awk '{print $3}')
            # Extract the version number from the line
            version=$(echo "$line" | awk '{print $4, $5}')

            # Check if the service name and version are non-empty
            if [[ -n "$service" && -n "$version" ]]; then
                echo "Searching for vulnerabilities for $service version $version..."

                echo $service $version >> $combined_result_file

                # Run searchsploit with the service name and version
                searchsploit "$service $version" >> "$combined_result_file"
```

The function ‘check_vulnerabilities’, is designed to automate the process of checking for vulnerabilities based on the ‘nmap’ scan using ‘searchsploit’. The function extracts the output and searches for vulnerabilities in the services.

It begins by setting the path of the directory where the ‘searchsploit’ results will be saved. The variable ‘filepath’ is assumed to be defined elsewhere in the script. The expression ‘${filepath%/*}’ removes the file name from the path, leaving just the directory and appends the results to it, creating a new directory path.

We then ensure that the ‘results_dir’ directory exists and then set the path for a file named ‘combined_searchsploit_results.txt’ within the ‘results_dir’ directory which will store the ‘searchsploit’ results.

‘> “$combined_results_file’ clears the contents of the file if it already exists, effectively ensuring that the file is empty before appending results in.

The following ‘while’ loop reads each line of the ‘nmap’ output and checks if it contains the word “open”. Then, the service name and version are extracted and it appends it to the ‘combined_result_file’.

Figure 10: Main function

```
#~ Main function
main() {
    #~ Call function to get the network to scan from user input
    get_network
    #~ Call function to get the file path for saving results from user input
    get_filepath

    #~ Prompt the user to choose between 'basic' or 'full' scan types
    read -p "Choose scan type (basic/full): " scan_type
    #~ Convert the user input to lowercase
    scan_type=$(echo "$scan_type" | tr '[:upper:]' '[:lower:]')

    #~ Check if the selected scan type is either 'basic' or 'full'
    if [[ "$scan_type" == "basic" || "$scan_type" == "full" ]]; then
        #~ Perform the nmap scan with the selected scan type (function)
        perform_nmap_scan "$scan_type"
        #~ List services found and perform brute force attacks on them (function)
        list_services_and_bruteforce
        #~ If the scan type is 'full', also check for vulnerabilities (function)
        if [[ "$scan_type" == "full" ]]; then
            check_vulnerabilities
        fi
```

This is the core of the script which coordinates the entire workflow. The process starts by calling the two earlier functions ‘get_network’ and ‘get_filepath’. Next, the script asks the user if they would like a “basic” or “full” scan. The user’s input is converted to lowercase with ‘tr’ for consistency. If the user selects a valid scan type, it then proceeds to perform the nmap scan and lists the services found and provides the user a list to brute force.

If the user selects a “full” scan, the script then performs the additional vulnerability checking scan. After scanning and analysis are complete, the script asks the user if they would like to zip the file. If the user agrees, the script then proceeds to zip scan results.

Conclusion

In creating this script, we faced several challenges that required careful consideration and problem-solving. One of the key issues countered was ensuring that user inputs were handled effectively especially when dealing with network addresses and the services. By using loops and conditional checks, the script could handle various scenarios such as invalid paths and scan types.

Another challenge was managing the complexity of the script, particularly when dealing with multiple tasks like the network scanning, brute-forcing and vulnerability checking. Hence, a more modular approach was made, breaking everything down into smaller focused functions.

Different methods were also explored for certain tasks. Alternative tools for brute-forcing were considered but ultimately chose Hydra due to its robustness. The script was initially written in Python but ended up changing over to Bash to maintain simplicity and focus on the fundamentals.

Writing this script thoroughly taught us the importance of error handling and creating a user-friendly design regardless of being at either end of the script. By anticipating potential issues and providing clear feedback to the user, we were able to create a script that is both functional and intuitive.

This script streamlines and automates crucial tasks in network security assessments, thereby enhancing efficiency and effectiveness. By combining all the above-mentioned functions into a single workflow, this script saves time and effort that would otherwise be a tedious manual process. Automation ensures that tasks are performed consistently, reducing the likelihood of human error and ensuring that no critical steps are overlooked. Such a consistency is essential as it is important to maintain accurate and reliable assessments.

Additionally, this script simplifies the handling of results by organising and optionally zipping them, making it easier to share the results. For cybersecurity professionals and penetration testers, this script provides a valuable yet versatile tool that improves productivity and be able to more systematically identify potential risks.

Recommendation

The script currently supports brute-forcing for a limited set of services. Future versions could include additional services based on user requirements thus broadening the scope of the script's utility.

The integration with 'searchsploit' is functional but could be refined. For example, incorporating more advanced filtering or searching techniques could improve the accuracy of the results. It currently represents the best solution available based on current proficiency but future improvements could lead to more accurate vulnerability assessments.

For a start, the script could be integrated with other security tools and databases to expand the script's functionality.

As cybersecurity threats evolve, regularly updating the script to incorporate the latest tools, techniques and threat intelligence is crucial. This ensures that this script will constantly remain relevant in our rapidly changing landscape.

In summary, while this script provides a solid foundation for network security assessments, there is room for improvement and expansion in future.

References

"Bash Single vs Double Brackets," Baeldung, retrieved August 17, 2024, from https://www.baeldung.com/linux/bash-single-vs-double-brackets.

"Where is the Path to the Current User's Desktop Directory Stored?" Unix & Linux Stack Exchange, retrieved August 17, 2024, from https://unix.stackexchange.com/questions/391915/where-is-the-path-to-the-current-users-desktop-directory-stored.

"Save Current Directory in Variable Using Bash," Stack Overflow, retrieved August 17, 2024, from https://stackoverflow.com/questions/13275013/save-current-directory-in-variable-using-bash.

"Finding Bash Shell Array Length Elements," Cyberciti.biz, retrieved August 18, 2024, from https://www.cyberciti.biz/faq/finding-bash-shell-array-length-elements/.

"Checking if Length of Array is Equal to a Variable in Bash," Stack Overflow, retrieved August 18, 2024, from https://stackoverflow.com/questions/13101621/checking-if-length-of-array-is-equal-to-a-variable-in-bash.

"Check if Array is Empty in Bash," Server Fault, retrieved August 19, 2024, from https://serverfault.com/questions/477503/check-if-array-is-empty-in-bash.

"Bash For Loop Array," Cyberciti.biz, retrieved August 19, 2024, from https://www.cyberciti.biz/faq/bash-for-loop-array/.

"Accessing Array Index Variable from Bash Shell Script Loop," Unix & Linux Stack Exchange, retrieved August 19, 2024, from https://unix.stackexchange.com/questions/278502/accessing-array-index-variable-from-bash-shell-script-loop.

"Bash Case Statement," Linuxize, retrieved August 20, 2024, from https://linuxize.com/post/bash-case-statement/.

"Bash Script to Create a Directory," Ask Ubuntu, retrieved August 20, 2024, from https://askubuntu.com/questions/952582/bash-script-to-create-a-directory.

"Hack The Box Lab: Exploring Remote Desktop Exploitation," Medium, retrieved August 20, 2024, from https://medium.com/@jaber_5689/hack-the-box-lab-exploring-remote-desktop-exploitation-7c957274a501#:~=Task%205%3A%20Service%20on%20Port%203389&text=The%20Nmap%20scan%20yields%20the,is%20active%20on%20this%20port.

OpenAI. (2024). *ChatGPT* (GPT-4). Retrieved from https://chatgpt.com/