**Lab 14-01: Binary Analysis**

**Scenario**

You are working with a crime investigation cell, and you are assigned a task to investigate some binary files. Use the Linux command-line tools to analyze and investigate the binary files.

**Solution**

You have been tasked with analyzing binary files on a Linux system using various command-line tools. The objective of this lab is to focus on 32-bit binary analysis, understanding the structure and contents of binary files, and extracting useful information from them. To accomplish this task, you will utilize several command-line tools such as file, strings, hexdump, and objdump.

By combining these tools, you aim to gain a comprehensive understanding of the 32-bit binary files under analysis and uncover any potential security vulnerabilities or interesting features within them.

|  |  |
| --- | --- |
| 1. Use the **mousepad sample.asm** command to create and open an assembly file named **sample.asm** in the **mousepad** text editor.    2. Paste the following assembly language code in the text editor and press **Ctrl + S** to save the file. It is a 32-bit assembly language program.   |  | | --- | | section .text  global \_start  \_start:  ; Write the system call number for “exit” (1) to the eax register  mov eax, 1  ; Write the exit status (0) to the ebx register  mov ebx, 0  ; Call the kernel to exit the program  int 0x80 |       3. Use the following commands to convert the assembly file into an object file and then convert the object file into a binary (ELF) file:  **nasm -f elf32 sample.asm -o sample.o**  **ld -m elf\_i386 sample.o -o sample**  4. It uses the **nasm** command to use the Netwide Assembler to convert the assembly into an object file. It then uses the **ld** command (GNU linker) to link the object file into an ELF binary executable file named **sample**.    5. Use the **file sample** command to see the file type. It shows that the file is an **ELF 32-bit LSB executable**, which is the type of 32-bit binary file.    6. Since you already know the file is an ELF binary, use the readelf -h sample command to display its header information. The Magic field contains the magic bytes that identify the file type, and the Class field indicates that the file is an ELF32 binary  7. Other headers show other useful information, like the OS (UNIX) on which the binary was created and which processor (Intel) was used by that machine.    8. Use the **hexdump sample** command to display the contents of the binary file in hexadecimal format.    9. The previous hexadecimal output may look meaningless; therefore, you can use the **hexdump -C sample | head** command to format the output and show some useful information. It shows that the file is an **ELF** file.    10. Use the **strings sample** command to extract human-readable text strings from binary files. It scans the specified file and prints any sequences of printable characters that are at least four characters long by default.    11. Use the **objdump -d sample | head** command to display the assembly language instructions from the binary file on the terminal.    12. Use the **nm sample** command to see some useful information embedded in the binary file while compiling it. It displays the symbols from the file. You can identify the variables and functions from the binary file, which is useful when you do not have access to the source code. |

**Lab 14-02: 64-Bit Analysis**

**Scenario**

As a digital forensics specialist embedded with a cybersecurity company, you are assigned to conduct a formal analysis of a collection of binary files recovered from an evidentiary device. Your objective is to determine the purpose and behavior of the binaries, identify any malicious functionality or Indicators of Compromise (IoCs), and preserve forensic integrity for potential legal proceedings. Before any analysis, confirm the chain-of-custody, obtain written authorization from the investigating authority, and ensure analysis is performed on forensic copies rather than the original media.

**Solution**

You have been assigned the task of analyzing binary files on a Linux system, with a specific focus on 64-bit binary analysis. To accomplish this, you will be utilizing a set of command-line tools, including gcc, readelf, hexdump, and objdump. The gcc compiler will be instrumental in compiling source code into 64-bit binary executables for analysis. Once compiled, readelf will aid in examining the Executable and Linkable Format (ELF) headers and other metadata of the binary files, providing insights into their structure and dependencies.

You aim to gain a thorough understanding of the 64-bit binary files under analysis, identify any potential vulnerabilities, and ultimately enhance your skills in binary analysis and reverse engineering.

|  |  |
| --- | --- |
| 1. Use the **mousepad sample64.c** command to create and open a C file named **sample64.c**. Write a basic C program in the file and press **Ctrl + S** to save it. The following C program is used:   |  | | --- | | #include <stdio.h>  int main() {  printf("Hello, world!\n");  return 0;  } |     2. Use the **gcc -o sample64 sample64.c** command to compile the C program and convert it into a binary file. Use **ls** to see the newly created file.    3. Use the file sample64 command to determine the file type. The output shows that it is an ELF 64-bit LSB PIE executable, which is a 64-bit binary. It is dynamically linked, meaning it depends on shared libraries at runtime.    4. Since you know the file is an ELF-type binary. Use the **readelf -h sample64** command to see the header information of the ELF file. The **Magic** header shows the magic bytes that also tell about the file type. There is a **Class** header that tells the file is **ELF64**.  5. Other headers show other useful information, like the OS (UNIX) on which the binary was created and which machine architecture it is supported by.    6. Use the **ldd sample64** to see the dynamic libraries that the executable depends on. If the output includes lines like **libc.so.6 => /lib/x86\_64-linux-gnu/libc.so.6**, it means that the executable is dynamically linked to the C library (libc).    7. Use the **hexdump -C sample64 | head** command to dump the file content (first 10 lines) in hexadecimal format and show some useful information. It shows that the file is an **ELF** file.    8. Use the objdump -d sample64 command to display the assembly instructions of the binary on the terminal. Understanding assembly language is essential for interpreting a binary’s purpose and functionality.    9. Use the **objdump -d sample64 | head** command to display only the first 10 lines of the assembly language instructions from the binary file on the terminal.    10. Use the **nm sample64** command to see some useful information embedded in the binary file while compiling it. It displays the symbols from the file. You can identify the variables and functions from the binary file, which is useful when you do not have access to the source code.    11. Use the **strings sample64** command to extract human-readable text strings from binary files. It scans the specified file and prints any sequences of printable characters that are at least four characters long by default.    12. Use the **hexdump –e’80/1 “%\_p”’ sample64** command to dump file content in the hexadecimal format. The character sequence **(%\_p)** tells hexdump to print a character in your system’s default character set. An additional option, like **80/1,** forces **hexdump** to process **80 bytes** at a time.    13. Use the **hexdump -n8 –e’8/1 “%d ””\n”’ sample64** command to print the output in decimal format. |

**Lab 14-03: Binary Analysis Methodology**

**Scenario**

As a cybersecurity analyst working for TechSecure Analytics, a firm specializing in malware research and reverse engineering, you have been assigned to demonstrate a comprehensive binary analysis methodology as part of an internal training and capability building program. The objective is to develop standardized procedures for examining binary files to understand their structure, functionality, and potential security weaknesses. The demonstration will focus on using open-source Linux command-line tools in a controlled lab environment to illustrate each stage of the binary analysis workflow.

**Solution**

You are tasked with demonstrating a binary analysis methodology using various Linux command-line tools. This methodology involves a systematic approach to analyzing binary files to gain insights into their structure, functionality, and potential vulnerabilities. To demonstrate this methodology, you will utilize a combination of command-line tools, including ndisasm, readelf, hexdump, nm, and objdump.

By demonstrating the use of these command-line tools systematically, you aim to showcase an effective binary analysis methodology that can be applied to analyze and understand various types of binary files.

|  |
| --- |
| 1. Use the **file wildfire-test-elf-file** command to see the file type. It shows that the file is an **ELF 64-bit LSB executable**, which is the type of 64-bit binary file. It is a dynamically linked file, which means it uses dynamically linked libraries.  **Note:** A sample file, **wildfire-test-elf-file,** is used as an example in this lab.    2. Since you know the file is an ELF-type binary. Use the **readelf -h** **wildfire-test-elf-file** command to see the header information of the ELF file. The **Magic** header shows the magic bytes that also tell about the file type. There is a **Class** header that tells the file is **ELF64**.  3. Other headers show other useful information, like the OS (UNIX) on which the binary was created and which machine architecture it is supported by.    4. Use the **strings wildfire-test-elf-file** command to extract human-readable text strings from binary files. It scans the specified file and prints any sequences of printable characters that are at least four characters long by default.      5. Use the **ndisasm -a -p intel wildfire-test-elf-file** command to disassemble the binary file into its equivalent assembly language code.      6. Use the **hexdump wildfire-test-elf-file** command to dump the file content in hexadecimal format, but it looks meaningless for now.    7. Use the **hexdump -C wildfire-test-elf-file** command to dump the file content in hexadecimal format and show some useful information. It shows that the file is an **ELF** file.    8. Use the **hexdump -C wildfire-test-elf-file | head** command to dump the file content (first 10 lines only) in hexadecimal format.    9. Use the **objdump -d wildfire-test-elf-file** command to display the assembly language instructions from the binary file on the terminal. It is essential to understand the assembly language to interpret what the binary file is intended for or what its functionality is.        10. Use the **nm wildfire-test-elf-file** command to see some useful information embedded in the binary file while compiling it. It displays the symbols from the file. You can identify the variables and functions from the binary file, which is useful when you do not have access to the source code.      11. Use the **hexdump --canonical** **wildfire-test-elf-file** command to convert the raw data into ASCII format to make it somewhat meaningful.      12. Use the **hexdump –e’80/1 “%\_p”’** **wildfire-test-elf-file** command to dump file content in the hexadecimal format. The character sequence (%\_p) tells hexdump to print a character in your system’s default character set. An additional option, like 80/1, forces **hexdump** to process 80 bytes at a time.    13. Use the **hexdump –e’80/1 “%\_p””\n”’ wildfire-test-elf-file** command to add new lines with the help of **“\n”** option.    14. Use the **hexdump -n8 –e’8/1 “%d ””\n”’ wildfire-test-elf-file** command to print the output in decimal format. |

**Lab 14-04: Bypassing No Execute on the Stack**

**Scenario**

An attacker aims to exploit a vulnerability in a program by injecting malicious code into the stack and executing it. However, the system has enabled No Execute (NX) protection, preventing the execution of code on the stack. This security measure makes traditional stack-based buffer overflow attacks ineffective.

**Solution**

To bypass No Execute (NX) protection on the stack, attackers often resort to techniques like Return-Oriented Programming (ROP) or Jump-Oriented Programming (JOP). These methods involve chaining together existing code fragments, known as "gadgets," within the program's memory space to execute malicious behavior without directly injecting executable code onto the stack. By carefully selecting and arranging these gadgets, attackers can circumvent NX protection and achieve their objectives. Defenders can counter these attacks by implementing security measures such as stack canaries, Address Space Layout Randomization (ASLR), Control Flow Integrity (CFI), and additional stack memory protections, which make it more challenging for attackers to construct viable exploit chains.

**Note:** The information provided here about bypassing No Execute (NX) protection is for educational purposes only. Any attempt to bypass security measures, including NX protection, may violate laws and regulations without proper authorization. It is essential to adhere to ethical guidelines and legal requirements when conducting security research or testing. Unauthorized exploitation of vulnerabilities can lead to legal consequences, including criminal charges. Always obtain proper authorization and permission before engaging in security testing, and ensure compliance with applicable laws and regulations.

|  |  |
| --- | --- |
| 1. You need to paste the code into the text editor in **Kali Linux** and save it with the extension **.c.**   |  | | --- | | **#include <stdio.h>**  **#include <string.h>**  **int main() {**  **char realPassword[20];**  **char givenPassword[20];**  **strncpy(realPassword, "fffffffffffffff", 20);**  **printf("Enter the password: ");**  **fgets(givenPassword, sizeof(givenPassword), stdin);**  **givenPassword[strcspn(givenPassword, "\n")] = 0;**  **if (0 == strncmp(givenPassword, realPassword, 20)) {**  **printf("SUCCESS!\n");**  **} else {**  **printf("FAILURE!\n");**  **printf("givenPassword: %s\n", givenPassword);**  **printf("realPassword: %s\n", realPassword);**  **return 0;**  **}** |   **2.** Then open your terminal and compile the code using **gcc –o filename filename.c**command***.***    3. Now, debug your file by using the **gdb ./filename** command. The gdb is a prompt indicating that you are currently using the GNU Debugger (GDB) command-line interface. GDB is a powerful tool used for debugging programs written in C, C++, and other languages. When you see gdb in your terminal, you can enter GDB commands to inspect and debug your program.    4. You can use all of these commands to debug.   * **run:** Starts the execution of the program being debugged. * **break <line\_number> or break <function\_name>:** Sets a breakpoint at a specific line number or function. The program will pause execution when it reaches the specified breakpoint. * **next or n:** Executes the following line of code in the program. * **step or s:** Executes the following line of code, stepping into function calls if encountered. * **print <variable> or p <variable>:** Prints the value of a variable. * **info locals:** Displays information about local variables in the current scope. * **info breakpoints:** Shows information about currently set breakpoints. * **quit or q:** Exits GDB.     5. Now, set the saved register info. Repeat this process 3-4 times and use different register locations.    6. Now, run your bypass file. It is a bypass of the stack that you can compare with code. |

**Lab 14-05: 64-Bit Exploitation and ROP Fundamentals**

**Scenario**

Assume a leading technology firm specializing in cybersecurity solutions is conducting a training workshop on 64-bit exploitation and Return-Oriented Programming (ROP) fundamentals for its cybersecurity team. The workshop aims to enhance their skills in identifying and mitigating advanced exploitation techniques cybercriminals use.

**Solution**

A leading cybersecurity firm conducts specialized training on 64-bit exploitation and Return-Oriented Programming (ROP) fundamentals for its cybersecurity team. The training comprises expert-led lectures, hands-on labs, and real-world scenario analysis, equipping participants with skills to identify and mitigate advanced exploitation techniques cybercriminals use. Emphasis is placed on secure coding practices, vulnerability mitigation strategies, and developing ROP chains. Upon completion, participants receive a certification, enhancing their credibility as cybersecurity professionals. Continuous learning is encouraged through access to resources and community forums, ensuring ongoing readiness to defend against evolving cyber threats. This proactive approach strengthens the company's cybersecurity posture and commitment to safeguarding sensitive data and critical infrastructure.

**Note:** The information provided on 64-bit exploitation and Return-Oriented Programming (ROP) fundamentals is intended for educational purposes only. While efforts have been made to ensure accuracy, the content should not be considered professional cybersecurity advice. Individuals should exercise caution and conduct further research or seek guidance from qualified cybersecurity professionals before applying any techniques discussed in the context of real-world scenarios. The exploitation of vulnerabilities without proper authorization is illegal and unethical. The responsibility for any consequences resulting from the misuse or misinterpretation of the information lies solely with the individual.

|  |  |  |
| --- | --- | --- |
| 1. You need to paste the following code into the text editor in **Kali Linux** and save it with the extension **.c.**   |  | | --- | | **// vulnerable\_program.c**  **#include <stdio.h>**  **#include <string.h>**  **void vulnerable\_function(char \*input) {**  **char buffer[64];**  **strcpy(buffer, input);**  **}**  **int main() {**  **char input[100];**  **printf("Enter input: ");**  **fgets(input, sizeof(input), stdin);**  **vulnerable\_function(input);**  **return 0;**  **}** |   2. Paste the following code into the text editor in **Kali Linux** and save it with the extension **.py.**   |  | | --- | | **from pwn import \***  **# Connect to the vulnerable program**  **p = process('./vulnerable\_program')**  **padding = b'A' \* 64**  **rop\_chain = b'YOUR\_ROP\_CHAIN\_HERE'**  **payload = padding + rop\_chain**  **p.sendline(payload)**  **# Interact with the program (optional)**  **p.interactive()** |   3. Open your terminal and compile the C language Code. Using the following command: **gcc –o filename –fno-stack-protector filename.c.**    4. Now, run the debugger for this file. Use the command **gdb ./filename*.***    5. Now, for the debugger, use the **info function.** You will see the two functions.    6. Use the following command, **disassemble main,** to see the assembly code behind the main function. You can see there are three main functions.    7. Now, use the command **disassemble vulnerable\_function** to see the assembly code behind this function.    8. Execute the exploit file. To run the exploit file, copy the Python code and save it under the name **exploit.py**.    9. Now go to the terminal and run the file by using the **python3 filename**command**.** It will run the exploit and wait until it is done. |

**Lab 14-06: Analyze ELF Executable File using Detect It Easy (DIE)**

**Scenario**

TechFam Security, a cybersecurity firm managing the Linux-based infrastructure of multiple clients, has encountered unusual system behavior on several client machines. Initial investigations indicate the presence of suspicious ELF executable files that attackers may craft to evade detection while performing unauthorized system activities such as data exfiltration and privilege escalation. The security team needs to identify the origins, structure, and potentially malicious behavior of these ELF files without executing them to prevent further compromise.

**Solution**

TechFam Security hires you as a Certified Penetration Tester because they want you to hack and find their vulnerabilities ethically. The Executable and Linkable Format (ELF) is a generic executable file format in Linux environments. The ELF header, sections, and segments contain three main components. Each component plays an independent role in the loading and execution of ELF executables. The static analysis of an ELF file involves investigating an ELF executable file without running or installing it. It also involves accessing the binary code and extracting valuable artifacts from the program. Numerous tools can be used to perform static analysis on ELF files. We will use the Detect It Easy (DIE) tool to analyze the ELF file in this lab.

Detect It Easy (DIE) is an application used to determine the types of files. Apart from Windows, DIE is also available for Linux and MacOS. It has an entirely open architecture of signatures. It can easily add its algorithms to detect or modify the existing signatures. A signature-based detection method detects the file’s compiler, linker, and packer.

|  |
| --- |
| 1. Open the browser on the Windows 10 machine. Copy and paste the following link: **https://drive.google.com/drive/folders/17Tl19hPliPwCNlkGilMj8tf65ebpVvhl** to download the **Detect It Easy** setup file. After downloading the file, unzip it and double-click the **die.exe**. **The Open File – Security Warning window** pop-up appears. Click **Run**.    2. **Detect It Easy** window appears. Click the ellipses icon (. . .) next to the **File Name** text field.    3. The **Open file** window appears; navigate to the **Downloads** folder, select the **ELF Test File**, and click **Open**. Download the **ELF Test File** from the link provided in **step # 1**.    4. **Detect It Easy** automatically scans the file, and the result shows the Operating system, compiler, and language details in the middle pane. The results may be different when you perform this lab.    5. Click on the **Advanced** checkbox. Then, click the **File info** button.    6. **Info** window appears. You can observe file name, size, MD5, SHA1, Entropy, and entry points. After viewing the information, click **Close** to close it.    7. Click the **Hash** button.    8. The **Hash** window appears. Here, information related to the hash can be viewed. Click **Close** to close the window.    9. Click the **Entropy** button.    10. The **Entropy** window appears. Observe the status, size, and entropy graph. Click **Close** to close the window. You can further explore MIME, Hex, Signatures, and Demangle functions. It concludes the demonstration of ELF file analysis using Detect It Easy (DIE). |

**Lab 14-07: Perform Malware Disassembly using IDA and OllyDbg**

**Scenario**

Cybersecurity experts at SecureNet Solutions, a global software development firm, have identified unusual network traffic and suspicious executable files that may indicate a targeted malware attack. Initial investigations suggest that an unknown malware sample is designed to exploit system vulnerabilities, disrupt software operations, and potentially exfiltrate sensitive client data.

Given the critical need to understand the malware’s capabilities, the security team must comprehensively analyze the malicious code without executing it. Static malware analysis is essential to extract valuable insights, including the malware’s operational logic, entry points, potential exploits, and payload delivery mechanisms.

**Solution**

As a Certified Penetration Tester tasked with securing SecureNet Solutions’ infrastructure, your objective is to perform malware disassembly using Interactive Disassembler (IDA) and OllyDbg. This process will allow you to deconstruct the executable file and examine its assembly code to identify the malware’s behavior and intent.

Static malware analysis involves dismantling executable files into binary format to study their functionalities and features without executing the code. This process identifies programming languages, APIs, and critical program logic to assess threat potential. Tools such as IDA Pro and OllyDbg facilitate this process by disassembling and debugging the code. IDA Pro maps the execution of binary programs by creating symbolic assembly language representations, aiding in understanding malicious code even when obfuscated. Its built-in debugger allows analysts to step through code execution, bypassing obfuscation and enhancing analysis depth. OllyDbg specializes in binary code debugging by tracing registers, identifying API calls, constants, strings, and routines. This makes it essential to analyze malware without source code. These tools combined provide powerful insights into malware behavior and potential threats.

There is a new debugging option, “Set permanent breakpoints on system calls.” When active, it requests OllyDbg to set breakpoints on KERNEL32.UnhandledExceptionFilter(), NTDLL.KiUserExceptionDispatcher(), NTDLL.ZwContinue(), and NTDLL.NtQueryInformationProcess().

**Note:** Use the following link: **https://drive.google.com/drive/folders/1-EwJfULSpaNO2e2YLrAz\_0FwYCk8C-g-** to download all tools used in this lab to perform static malware analysis.

|  |
| --- |
| 1. Turn on the **Windows 10** virtual machine. Navigate to the **Downloads** folder after downloading all the tools from the link provided in the note. Then, go inside the **IDA** folder.    2. Double-click on the **idafree84\_windows.exe** file.    3. The IDA Freeware setup window appears. Click on the **Next >** button.    4. Select the **I accept the agreement** radio button. Then, click on the **Next >** button.    5. Leave the installation directory as default and click on the **Next >** button.    ,6. Click on the **Next >** button to install IDA Freeware.    7. Click on the **Finish** button.    8. In the Windows search bar, type **IDA Freeware 8.4**. Then, click on the **IDA Freeware 8.4** to open it.    9. If the **IDA License** window appears, click the **I Agree** button.    10. The **User interface telemetry** window appears, uncheck **Yes, I want to help improve IDA** checkbox, and click on the **OK** button.    11. The **IDA: Quick start** pop-up appears; click the **New** button to select a malicious file for disassembly.    12. The **IDA** main window, along with the **Select file to disassemble** the window, appears.    13. In the **Select file to disassemble** window, navigate to **Downloads\Klez Virus Live!** folder, select **face.exe,** and click on the **Open** button.    14. The **Load a new file** window shows up; by default, the **Portable executable for 80386 (PE) [pe64.dll]** option is chosen; click the **OK** button. If a **Warning** pop-up appears, click on **OK**. If a **Please confirm** dialog-box appears, read the instructions carefully, and then click on **Yes**.    15. IDA finishes evaluating the imported malicious file and shows the results on the **IDA View-A** tab.    16. In the **IDA View-A** section, right-click anywhere and choose **Text view** from the context menu to view the text information of the malicious file uploaded to IDA for analysis.    17. This reveals the malicious file's text view, allowing its information to be analyzed.    18. Maximize the IDA window. To view the flow of the uploaded malicious file, navigate to **View → Graphs** and click **Flow chart**.    19. A **Graph** window appears with the flow. You may zoom in and adjust the screen to view this more clearly.    20. Close the **Graph** window, go to **View → Graphs**, and click **Function** **calls** from the menu bar.    21. A window demonstrating the **Call flow** displays; zoom in for an enhanced view. Once the analysis is complete, close the WinGraph32 Call flow window.    22. Click the **HexView-1** tab to view the hex value of the malicious file.    23. Click the **Imports** tab to view a list of all functions that the executable calls.    24. Close all open windows. In the **Save database** pop-up, click on the **OK** button.    25. Navigate to the **Downloads/OllyDbg** folder. Double-click on the **Ollydbg.exe**. If an **Open File - Security Warning** pop-up appears, click on **Run**.    26. If an **Old DLL** dialog box appears, click on **Yes**. If an OllyDbg warning notice regarding administrative permissions comes up, click **OK**.    27. The **OllyDbg** main window appears. When you launch OllyDbg for the first time, several sub-windows might appear in the main window; close all of them.    28. Select **File** from the menu bar and click **Open**.    29. The **Select 32-bit executable** window appears; navigate to the **Downloads** folder, select **tini.exe**, and click the **Open** button.    30. The output appears in a window named **CPU - main thread, module tini**, maximize the window.    31. Select **View** in the menu bar, and then choose **Log**.    32. A **Log data** window appears in OllyDbg, displaying the log details. The **Log data** shows the program's start point and calls to recognized functions. After completing your analysis, close the **Log data** window.    33. Select **View** in the menu bar, then choose **Executable modules**.    34. A window named **Executable modules** appears in OllyDbg, displaying all executable modules.    35. Double-click any module to view the complete information of the selected module. In this lab, we are selecting the **757D0000** module. The results might differ when you perform this lab. This will take you to the **CPU - main thread** window.    36. Select **View** in the menu bar, and then select **Memory map**.    37. A **Memory map** window appears in OllyDbg, displaying all memory mappings. Close the **Memory map** window.    38. Select **View** in the menu bar, and then choose **Threads**.    39. A window named **Threads** appears in OllyDbg, displaying all threads. This way, you can scan files and analyze the output using OllyDbg. |