RIT | Rochester Institute of Technology of Dubai

Department of Electrical Engineering and Computing

Computing Security

CSEC 202 Reverse Engineering Fundamentals

Spring 2024

Sections: 600, 601, and 602

Homework Assignment #4:

Advanced Dynamic Analysis

"GDB Debugging Mastery and Reverse Engineering"

Release Date: April 21, 2024

Due Date: **April 28**, 2024 – 11:59:59 p.m. (GST= GMT+4)

Instructor: Emad AbuKhousa (eakcad@rit.edu)

Homework Assignment #4:

Advanced Dynamic Analysis

Please pay close attention to the following essential guidelines and expectations:

- **Submission Deadline: Extensions will not be granted under any circumstances. This is a one-week assignment, and sufficient time is required to grade all submissions before finals. Please manage your time effectively to meet this deadline**
- **Instructor Support Window: The** instructor will not provide support 2 days before the assignment's deadline. It's crucial to start early to maximize the opportunity for receiving guidance and support. Procrastination may limit your ability to seek help.
- **Support Availability**: Support for this assignment will be available only during office hours. Make sure to bring up any questions or concerns during these designated times to receive assistance.
- **Communication Channels:** WhatsApp is not considered an official channel for office hours or assignment support. Please use the designated communication methods provided by your instructor for all correspondence related to this assignment.

Academic Integrity:

- Zero Tolerance for Plagiarism: Plagiarism and paraphrasing of any kind will be strictly penalized. All submissions will be thoroughly checked for originality. **We have robust measures in place to detect malpractice, so do not underestimate the system's ability to identify cheating.**

- Consequences of Academic Misconduct: Being caught in acts of academic dishonesty, such as plagiarism, will result in a score of zero for this assignment. Further disciplinary actions may follow according to the academic conduct policies. It is imperative that you adhere to the highest standards of academic integrity in your work.

These guidelines are put in place to ensure a fair and conducive learning environment for all students. Adhering to these rules is not only about avoiding penalties but also about cultivating professionalism, responsibility, and integrity in your academic pursuits.

Remember:

Start working on your assignment early to navigate through potential challenges with ample time for adjustments.

Objectives:

The overarching goal of this assignment is to deepen your understanding and proficiency in software security and debugging techniques. Through hands-on exercises, you will learn to navigate and utilize the GNU Debugger (GDB) to inspect and manipulate program execution, explore the fundamentals of buffer overflow vulnerabilities, and develop effective strategies to identify and exploit these vulnerabilities in a controlled environment. Additionally, you will apply reverse engineering skills to uncover hidden functionalities within the code and craft a practical exploit using Python, enhancing your ability to analyze and secure software systems. This comprehensive approach will equip you with the critical skills necessary for advanced debugging, reverse engineering, and cybersecurity analysis, preparing you for more complex challenges in the field of computer security.

Rubric:

Homework Assignment #4 Rubric	
Criteria	**Pts**
A detailed Report (not a copy-paste from this write up!): a. Craft a well-organized, step-by-step report detailing your entire analysis process. b. Include clear explanations for each step, accompanied by relevant screenshots. c. Discuss the result and purpose of each tool or command you employed, demonstrating your understanding. d. Format your report professionally in a standard Word document using consistent styles and headings. e. Include a cover page listing the assignment details, team members, their contributions, and this rubric for reference. f. Avoid using the color "**RED**" within the report. **The group report should be submitted by a single team member, and the names of all team members must be listed on the MyCourses submission form.**	10%
Part 1: Introduction to Debugging with GDB [30%] • Compilation for 32-bit Architecture [5%] • Setting and Testing Breakpoints [5%] • Inspecting Variables and Memory [10%] • Stack Frame and Register Analysis [10%]	30%
Part 2: Code Cracking with GDB [20%] • Program Operation Interception [5%] • Password Discovery [5%] • Execution Hijacking [10%] • Reverse Engineering to C [10%]	30%

Part 3: Attacker Bad Input Quiz [30%] • Correct Quiz Answers [10%] • GDB Program Behavior Analysis [20%]	30%
Bonus: Part 4: Smash the Stack for Fun and Profit [20%] • GDB Exploration [10%] • Shell Python Exploit Development [10%]	20%
"Us vs. the CAT" round 4 **Show the Cat who is in control!**	

Part 1 [30%] Introduction to Debugging with GDB:

Objective:

The objective of thispart is to introduce you to the basics of debugging using the GNU Debugger (GDB). You will learn how to compile programs for debugging, set breakpoints, inspect variables and memory, manipulate program execution, and analyze the call stack and registers. This hands-on exercise will enhance your understanding of program behavior and is essential for effective problem-solving in software development.

```c
#include<stdio.h>

int main()
{
    int x;
    int a = x;
    int b = x;
    int c = a + b;
    printf("%d\n", c);
    return 0;
}
```

Directions:

To complete this assignment, you will need a C programming environment and GDB installed on your system. Make sure to follow each step carefully, as this will prepare you for more complex debugging tasks in your future projects.

1. Compile for 32-bit Architecture:

Compile your C program (**ex1.c**) using the following command to ensure it includes debugging information and is set up for a 32-bit architecture:

```
$ gcc -m32 -g ex1.c -o ex1
```

This command tells GCC to generate a 32-bit executable and include debugging information that GDB can use.

2. Start GDB:

```
$ gdb ./ex1
```

This initializes GDB with your compiled program.

3. Setting Breakpoints:

Set a breakpoint at the start of the main function:

```
(gdb) break main
```

After running the program with run, disassemble the main function to find the memory address just before the printf call, and set a breakpoint there:

```
(gdb) disas main

(gdb) break *address
```

4. Variables and Memory Inspection:

Inspect and modify variables:

```
print x
print a
set var x=25
```

Locate the assembly command following the printf call to find the return address.

5. Stack Frame and Registers:

Get information about the current stack frame:

```
info frame ( or i f)
```

View register values just before the printf function executes:

```
info registers ( or i r)
```

6. Function Parameters:

Analyze how arguments are passed to printf:

```
x/2w $esp
x/2d $esp
x/2x $esp
x/s *$edx
```

7. Print Stack Frame:

Print 20 DWORDs from the current stack pointer:

```
x/20x $esp
```

8. Stepping into Functions:

Step into the printf function to observe its execution:

step (or s or si)

9. Printing the Stack inside Functions:

While inside printf, before returning to main, print the stack frame and stack contents:

```
info frame
```

```
x/20s $esp
```

10. Return Address and Base Pointer:

Examine the return address and base pointer to understand the stack layout:

```
x/gx $ebp+4
info register ebp
```

Complete these steps to gain a deeper insight into how your program executes, which will be critical for debugging more complex applications. Document your findings and any insights gained from each step to discuss in class or submit in a written report.

Part2 [30%]: Code Cracking with GDB

Objectives:

This part aims to introduce students to practical reverse engineering skills through the completion of one of the **Crackme** challenges. You will use GDB (GNU Debugger) to dissect and manipulate the program's operation, uncover hidden password, modify program behavior, and reverse engineer the assembly code back into its equivalent C code.

Instructions:
Perform the following tasks:

1. Intercept Program Operation Using GDB
Start the program under GDB:

```
gdb ./crackme0x00
```

Familiarize yourself with the program's flow by setting breakpoints and stepping through the execution to understand how it checks for the correct password.

2. Find the Password inside the GDB
Analyze the program to determine how it verifies the correct password. Look for comparisons or conditional jumps related to input validation.

Hint: *Before the critical comparison operation occurs, check the values stored in the registers and the stack. Pay close attention to any data that is compared directly or indirectly against the user's input. Inspect the state of the program right before these key operations to understand what conditions must be met for successful authentication.*

Use breakpoints, memory inspection, and code analysis to determine the actual password that the program expects.

3. Hijack the Execution to Accept Any Password

Once you identify the condition that checks the password, modify this condition using GDB so that the program accepts any password (inside GDB). This might involve changing a jump condition or directly modifying a register value.

Example:

set $eip = address_of_next_instruction_if_password_checks_pass

Test the modification to ensure that the program now accepts any input as a valid password.

Hint:

A. Replace the Instruction:

Use the set command to write new instruction opcodes directly into the memory. Since **jne** takes two bytes, you would replace it with two nop instructions:

set {char}address of first byte = 0x90

set {char} address of second byte = 0x90

B. Alternatively, you can use GDB's set command with the convenience variable write_memory:

set write_memory address of first byte, "\x90\x90"

This command writes the nop (0x90) opcode directly to the addresses that previously held the jne instruction.

4. Reverse Engineer the Code and Write the C Equivalent

Based on your understanding of the program's assembly code, write the equivalent C code that performs the same function of **crackme0x00**.

Ensure that your C code accurately reflects all logical conditions and operations you observed in the assembly code.

Useful information: https://firmianay.github.io/ioli_crackme_writeup/

Part 3 [30%]: Attacker Bad Input

Objectives:

Use GDB to explore how the program handles various password inputs and identify any potential for bypassing the intended security checks. This will involve examining how the password is stored and checked, and what happens to the stack during this process.

Attacker Bad Input Quiz

What **type of password string** could defeat the **password check code**? (Check all that apply)

```
#include <stdio.h>
#include <strings.h>

int main(int argc, char *argv[]) {
        int allow_login = 0;
        char pwdstr[12];
        char targetpwd[12] = "MyPwd123";
        printf("Password?: ");
        gets(pwdstr);
        if (strncmp(pwdstr,targetpwd, 12) == 0)
                allow_login = 1;

        if (allow_login == 0)
                printf("Login request rejected");
        else
                printf("Login request allowed");
}
```

- ☐ Any password of length greater than **12 bytes that ends in '123'**
- ☐ Any password of length greater than **16 bytes that begins with 'MyPwd123'**
- ☐ Any password of length greater than **8 bytes**

Student Instructions

Part 1: Solve the Quiz [5%]

Task: Carefully examine the provided code and determine which types of password strings could defeat the password check. Consider the implications of the string length and content on the strncmp function and the fixed-length character array.

Quiz: Based on the program shown, which type of password string could defeat the password check code? (Check all that apply)

A. **Any password of length greater than 12 bytes that ends in '123'**
B. **Any password of length greater than 16 bytes that begins with 'MyPwd123'**
C. **Any password of length greater than 8 bytes**

Steps:

- Review the **strncmp** function and its arguments.
- Analyze how the buffer **pwdstr** is being used and the potential for overflow.
- Determine the correct answers to the quiz questions and justify each choice.
- Or, through trial and error ……. after all, you are becoming skilled crackers
-

Part 2: Explore the Program Using GDB [15%]

Task: Use GDB to investigate the program's behavior with different inputs. Validate your answers from Part 1 by observing how the program processes and compares the password strings.

Instructions:

- Compile the program with debugging information.
- Start a GDB session and set breakpoints at strategic points.
- Run the program with different password inputs and observe how the buffer is filled and compared.
- Use GDB to inspect memory, examine registers, and follow the program's logic to see if and how a buffer overflow occurs.

Hints for GDB:

- Pay attention to the size of pwdstr when providing input.
- Look at the content of memory before and after filling pwdstr.
- Notice how strncmp behaves when comparing the user input to targetpwd.
- Use the info variables, info functions, and list commands to get more information about the code.
- The x/ command can be used to examine memory and the p command to print variable values.

Document your findings and be prepared to discuss how your observations with GDB support your quiz answers.

Part 4 Bonus: [20%] Smash the Stack for fun and profit

Objective

In this part, you will dive into the world of software vulnerabilities by examining and exploiting a buffer overflow in a sample C program (**vul.c**). You will gain hands-on experience with the GNU Debugger (GDB) to manipulate the flow of execution and develop a Python script to stage an attack from outside the debugging environment

Your mission is to manipulate the execution flow of a vulnerable program to invoke a hidden function, **secretFunction**, which is not called during the program's normal operation.

Background

A buffer overflow is a classic security vulnerability that occurs when more data is written to a buffer than it can hold, resulting in adjacent memory being overwritten. This could lead to the alteration of the program's control flow, particularly by overwriting the stack's return address.

Tasks Description

Part 1: GDB Exploration [15%]

A. Compile the provided C program for a 32-bit architecture to make the debugging process more straightforward.
```
gcc -m32 -g -o vuln_program vul.c
```
B. Initiate a GDB session and insert breakpoints at critical points within the sayHello function to intercept the execution flow.
C. Conduct a meticulous stack analysis at each breakpoint to decrypt the data allocation and explore avenues for manipulation.
D. Acquire the memory address of **secretFunction**. Subsequently, engineer the stack such that the return address of **sayHello** is redirected to execute **secretFunction**, circumventing its normal execution return to main.
E. Ascertain the exact payload size (in Bytes) required to induce a buffer overflow in the **name** array (buffer), allowing you to overwrite the return pointer on the stack.
F. Devise and deploy a crafted payload within GDB that strategically overwrites the stack's return pointer with the address of **secretFunction**, seizing control of the program's execution path. (you might use python within the GDB)

Part 2: Python Exploit Development [5%]

- Develop a Python script designed to craft and write a binary payload to a file, named **payload.bin**, which will be used to trigger the buffer overflow.
- Your script should build a binary payload that precisely fills the name buffer and supersedes the stack's return address with that of secretFunction.
- Execute the target program with **payload.bin** as the command-line argument**, effectively launching the attack without the aid of GDB**. This action should result in the activation of **secretFunction**, confirming the exploit's success.

"Us vs. the CAT" round 4

Show the Cat who is in control!

You have already done so brilliantly with HW03, and now you're ready to tackle even more complex topics.

Advanced Dynamic Control Flow and Data Flow, typically explored in senior and graduate-level courses, are within your reach. Having observed what the CAT is capable of, you're now well-prepared to manage and counter various forms of malware. Keep up the great work.

Best of luck on your brilliant future in cybersecurity!

Yours

Emad AbuKhousa

Download Instructions:

Cloning the Repository

- Open a terminal window or command prompt on your system.
- Navigate to the directory where you wish to clone the repository using the cd command.
- Enter the following command to clone the repository:
  git clone https://github.com/RITDubaiCSEC202/HW4.git
- Wait for the cloning process to complete. This command creates a local copy of the repository in your specified directory.