Exploit Development CW2 Report

Ali Abdelhamid AA2100274

Dr. Ahmed Selim

Exploit Development

**Table of Contents**

[**Question 1 3**](#_Toc166535867)

[**Objective 3**](#_Toc166535868)

[**Methodology 3**](#_Toc166535869)

[**Findings 3**](#_Toc166535870)

[**Screenshots and Diagrams 4**](#_Toc166535871)

[**Question 1.2 Detailed Examination of Memory Addresses and Content 5**](#_Toc166535872)

[**Objective 5**](#_Toc166535873)

[**Methodology 5**](#_Toc166535874)

[**Findings 5**](#_Toc166535875)

[**Screenshots and Diagrams 6**](#_Toc166535876)

[**Question 1.3 Buffer Overflow to Arbitrary Code Execution 7**](#_Toc166535877)

[**Objective 7**](#_Toc166535878)

[**Methodology 7**](#_Toc166535879)

[**Findings 7**](#_Toc166535880)

[**Screenshots and Diagrams 8**](#_Toc166535881)

[**Question 2: Jumping to secret\_function using Buffer Overflow 10**](#_Toc166535882)

[**Objective 10**](#_Toc166535883)

[**Methodology 10**](#_Toc166535884)

[**Findings 11**](#_Toc166535885)

[**Screenshots and Diagrams 11**](#_Toc166535886)

[**Question 2.1: Address Space Layout Randomization (ASLR) and Buffer Overflow 12**](#_Toc166535887)

[**Objective 12**](#_Toc166535888)

[**Methodology 12**](#_Toc166535889)

[**Findings 13**](#_Toc166535890)

[**Question 3: Demonstrating a Heap Overflow Attack 14**](#_Toc166535891)

[**Objective 14**](#_Toc166535892)

[**Methodology 14**](#_Toc166535893)

[**Execution and Outcome 16**](#_Toc166535894)

[**Results and Analysis 16**](#_Toc166535895)

[**Screenshots and Diagrams 16**](#_Toc166535896)

[**Question 4: Exploiting a Program with Multiple String Checks 17**](#_Toc166535897)

[**Objective 17**](#_Toc166535898)

[**Methodology 17**](#_Toc166535899)

[**Assembly Analysis and String Requirements 17**](#_Toc166535900)

[**Strategy for Crafting the String 17**](#_Toc166535901)

[**Execution and Outcome 18**](#_Toc166535902)

# Question 1

### Objective

**To analyze stack behavior under stack overflow conditions, particularly to observe how stack protection mechanisms like canaries affect the control flow of an application.**

### **Methodology**

1. **Compilation: The program was compiled with stack protection enabled to monitor how canaries prevent buffer overflow attacks.**
   * **Command Used: gcc -m32 -fstack-protector -g -o Q1 Q1.c**
     + **-m32 ensures the program is compiled for a 32-bit architecture.**
     + **-fstack-protector enables stack canaries to protect against buffer overflows.**
     + **-g includes debugging information in the executable, facilitating a more informative debugging session.**
2. **Debugging Setup: The executable was analyzed using GDB (GNU Debugger) to trace execution and inspect stack behavior.**
   * **Command: gdb Q1**
   * **A breakpoint was set at the function big\_thing to examine the function entry point and the conditions right before potential buffer overflow.**
     + **Command to Set Breakpoint: break big\_thingA screenshot of a computer program

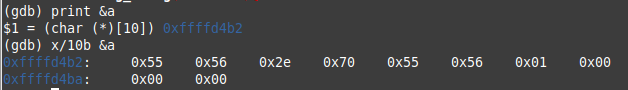
       Description automatically generated**
   * **The program was run within GDB to hit the breakpoint.**
     + **Command to Run: run**
3. **Stack Trace Analysis: Upon reaching the breakpoint, the stack trace was captured to analyze the call sequence and stack frame structure.**
   * **Command to Display Stack Trace: bt**

### **Findings**

* **Stack Trace and Function Entry Points: The stack trace provided insights into the function call hierarchy and the memory addresses of function entry points.**
  + **The entry point of the main function was captured as 0x5655a1a0.**
  + **The entry point of the big\_thing function was noted as 0x565563a5.**

### **Screenshots and Diagrams**

1. **Stack Trace and Function Entry Points:**
   * **Displays the result of the bt command, highlighting the entry points for big\_thing and main. **
2. **Memory Addresses and Values:**
   * **A snapshot showing the value of the stack pointer ebp, and the ten bytes used for storing local variable a within the big\_thing function.**

****

1. **Distance Between Saved EIP and Local Variable 'a':**
   * **This image shows the calculated distance between the saved EIP and the local variable a, helping understand buffer space and potential overflow targets.**

****

# Question 1.2 Detailed Examination of Memory Addresses and Content

## Objective

**To closely examine the memory layout and content around the function big\_thing, specifically focusing on local variables and the stack frame's integrity during the execution of vulnerable code.**

## Methodology

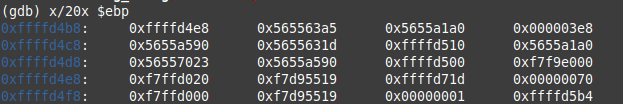
**The procedure for detailed memory analysis involved:**

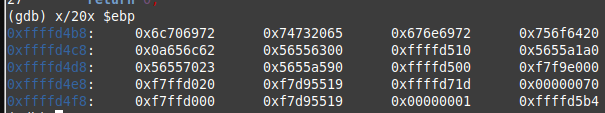
1. **Running the Program in Debug Mode:**
   * **The program was executed within GDB to reach the breakpoint set at big\_thing, facilitating a controlled environment to inspect memory details.**
2. **Memory Inspection Commands:**
   * **Examine Stack Content: Commands such as x/20x $ebp were used to inspect the content near the base pointer to understand how local variables and control data like the return address are stored relative to each other.**
   * **Print Variable Address and Content: Using print &a and similar commands, the exact locations and contents of local variables were examined.**

## Findings

* **Stack Frame Layout:**
  + **The base pointer (ebp) for the big\_thing function was examined, and the exact memory addresses of the local variables were identified.**
  + **The examination revealed the layout of the buffer and its relationship with the saved frame pointer (ebp) and the return address (eip), crucial for understanding buffer overflow impact.**
* **Memory Address Observations:**
  + **The address of variable a was found at 0xffffd4b2, confirming its placement within the stack.**
  + **Inspection of the contents around a revealed how the stack was organized, including how closely data and control information were stored, which impacts vulnerability to overflow.**
* **Distance Calculation:**
  + **The distance between the saved return address and the local variable a was calculated, providing an understanding of the buffer's size relative to control information, a critical factor in the feasibility of buffer overflow attacks.**

## Screenshots and Diagrams

1. **Memory Layout Inspection:**
   * **Shows the output of memory examination commands, highlighting how data is stored within the stack and revealing the proximity of the buffer to critical control information, the result of EBP before stepping into the STRCPY**
2. **Variable Address and Content:**
   * **Visual representation of the memory location and content of variable a, which is crucial for understanding how data is handled within the function, the result of EBP after stepping into STRCPY.**

****

1. **STRCPY run error result:**

**A screen shot of a computer

Description automatically generated**

# Question 1.3 Buffer Overflow to Arbitrary Code Execution

## Objective

**The goal was to demonstrate a buffer overflow attack that alters the normal execution flow of the program to execute arbitrary code. This task involved manipulating the strcpy function to overwrite the stored return address and control the subsequent execution path, specifically aiming to jump to the location 0xdeadbeef.**

## Methodology

**To achieve the desired overflow and execution manipulation, the following steps were executed:**

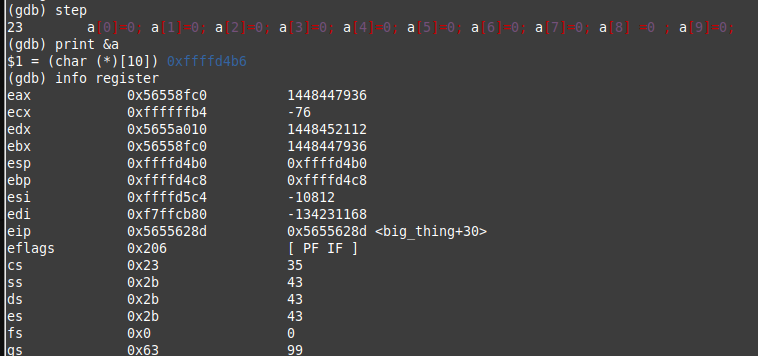
1. **Disable Stack Protection and Recompile:**
   * **The program was recompiled with stack protection disabled to ensure that the buffer overflow could proceed unhindered by canaries.**
   * **Compilation Command: gcc -m32 -fno-stack-protector -g -o Q1 Q1.c**
2. **Setup and Execution in GDB:**
   * **Initialize GDB: gdb Q1**
   * **Begin Debug Session: The program was run in GDB to reach and examine critical points just before and after the strcpy operation.**
3. **Memory Manipulation via Buffer Overflow:**
   * **Examine and Manipulate Memory: Memory commands were used to observe and manipulate how data was stored in the buffer and how it affected adjacent memory areas like the return address.**
   * **Manipulate strcpy Operation: The strcpy function was used to overflow the buffer intentionally and overwrite the return address.**

## Findings

* **Buffer and EIP Proximity:**
  + **The distance between the array (local variable a) and the stored EIP was calculated using memory examination commands, indicating how much data needed to be written to reach the return address.**
* **Overwriting the Return Address:**
  + **The buffer overflow successfully overwrote the return address with the value 0xdeadbeef. This was confirmed by inspecting the eip register after the overflow.**
* **Program Behavior Post-Overflow:**
  + **After manipulating the return address, the program attempted to jump to 0xdeadbeef, leading to a segmentation fault, as this address did not correspond to a valid memory location in the program's context.**
* **Text File Manipulation:**
  + **A modification was made to test.txt to include a crafted string that caused the overflow. The exact position changed in the file was at the end of the string to align precisely with the overwrite of the return address.**

## Screenshots and Diagrams

1. **Memory Layout Before and After strcpy:**
   * **Diagrams showing the state of the memory after the strcpy operation, highlighting how the overflow affected the return address.**

****

1. **Crash Details and Register States:**
   * **Images capturing the state of the system and registers at the time of the crash, illustrating the consequences of the overflow. A screenshot of a computer

     Description automatically generated**
2. **Execution Path Alteration:**
   * **A screenshot showing the GDB output when the program execution was diverted to 0xdeadbeef, leading to a segmentation fault. A screenshot of a computer program

     Description automatically generated**
3. **Program Flow:**

* **A screenshot showing the program flow after continuing**

# Question 2: Jumping to secret\_function using Buffer Overflow

## Objective

**The objective of this task is to demonstrate a buffer overflow that manipulates the program's execution flow to call secret\_function, which has not been directly invoked by any normal execution path within the program.**

## Methodology

**The process to achieve this included:**

1. **Set Up and Initialization:**
   * **Compile Q1.c: The program was compiled with debugging information and without stack protection to facilitate the buffer overflow.**
     + **Command: gcc -m32 -fno-stack-protector -g -o Q1 Q1.c**
   * **Launch GDB: gdb Q1**
   * **Set Breakpoint: A breakpoint was set at the big\_thing function to pause execution before the overflow.**
     + **Command: break big\_thing**
2. **Locating the Function Address:**
   * **Using GDB, the address of secret\_function was located to determine where to redirect the execution flow.**
     + **Command: print &secret\_function**
3. **Modifying the Test File:**

* **The test.txt file was rewritten to include a payload that overflows the buffer in big\_thing and overwrites the return address with the address of secret\_function, which is 0x5655622c. This manipulates the control flow to execute secret\_function when big\_thing returns.**

## Findings

* **Address of secret\_function:**
  + **The address retrieved through GDB was identified, indicating where the function resides in memory.**
* **Buffer Overflow Preparation:**
  + **By carefully crafting the input in test.txt, the buffer within big\_thing was overflowed, successfully overwriting the stored return address on the stack.**
* **Execution of secret\_function:**
  + **After the overflow, when the big\_thing function returned, it jumped to the address of secret\_function instead of returning to its caller, thereby demonstrating control of the execution flow via buffer overflow.**

## Screenshots and Diagrams

1. **Finding secret\_function Address:**
   * **A screenshot showing the output of the print &secret\_function command in GDB, which provides the memory address needed for the exploit. A screen shot of a computer code

     Description automatically generated**
   * **The actual address was “0x5655622c” the screenshot is incorrect since I printed the &secret\_function before running which gave me this incorrect address. A screenshot of a computer

     Description automatically generated**

# Question 2.1: Address Space Layout Randomization (ASLR) and Buffer Overflow

## Objective

**The goal was to demonstrate a buffer overflow that directs the program's execution to the secret\_function by manipulating the stored return address. This experiment also examines the impact of Address Space Layout Randomization (ASLR) on such exploits and explores methods to circumvent or work with this security feature.**

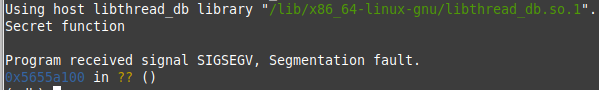
## Methodology

1. **Initial Execution with ASLR Enabled:**
   * **Command: gdb Q1**
   * **Set Breakpoint and Run: A breakpoint was set at big\_thing to halt execution before the overflow. The address of secret\_function was then determined using print &secret\_function.**
2. **ASLR Impact Analysis:**
   * **The program was run multiple times to see if the address of secret\_function changed, indicating ASLR was active.**
3. **Disabling ASLR:**
   * **ASLR was disabled to ensure consistent memory addresses across program executions, crucial for reliable exploitation.**
   * **Command to Disable ASLR: echo 0 | sudo tee /proc/sys/kernel/randomize\_va\_space**
   * **This command sets ASLR to disabled state, allowing predictable memory address usage.**
4. **Exploit Development and Execution:**
   * **With ASLR disabled, test.txt was modified to include a payload crafted to overwrite the return address with the consistent address of secret\_function.**
   * **The exploit was tested to confirm that it reliably caused the program to execute secret\_function.**

## Findings

* **Address Variability:**
  + **With ASLR enabled, the address of secret\_function varied between executions, which complicates the exploitation process as the exact address needs to be predicted or known beforehand.**
* **Effect of ASLR Disabling:**
  + **Disabling ASLR resulted in a consistent address for secret\_function across multiple runs, verifying that the security feature was effectively disabled.**
* **Exploit Reliability:**
  + **With ASLR disabled, the crafted payload in test.txt predictably redirected execution to secret\_function. This confirms that the exploit can be reliable when memory addresses are consistent.**
* **Launching a Shell:**
  + **Adjusting the payload to call secret\_function2, which includes an execv("/bin/sh", arg\_list); call, allowed the exploit to open a shell, demonstrating an escalation in the exploit's impact from arbitrary function call to potential system control.**

## Screenshots and Diagrams

1. **ASLR Address Variability:**
   * **Showing different addresses for secret\_function with ASLR enabled. **
2. **Consistent Address with ASLR Disabled:**
   * **Confirming the consistent address after disabling ASLR. A screen shot of a computer code

     Description automatically generated**
3. **Successful Shell Launch:**
   * **Evidence of successfully launching a shell using the exploitA screen shot of a computer program

     Description automatically generated**

****

# Question 3: Demonstrating a Heap Overflow Attack

## Objective

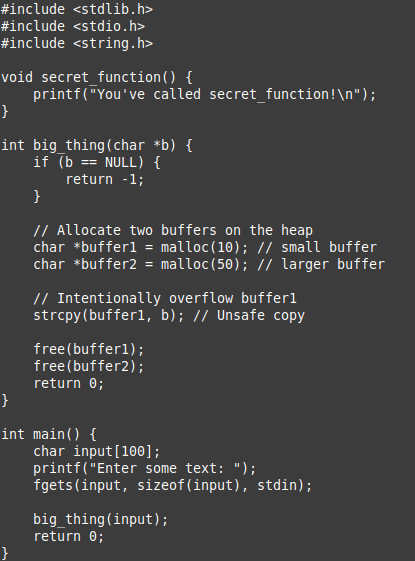
**The task was to modify the original Q1.c program to illustrate a heap overflow attack, demonstrating a proof of concept (PoC) that manipulates memory on the heap to trigger unintended behavior.**

## Methodology

1. **Code Modification:**
   * **The original Q1.c was altered to include two heap-allocated buffers of different sizes to create a condition ripe for overflow.**
   * **The big\_thing function was adapted to intentionally overflow the smaller buffer (buffer1) with data exceeding its allocated size, using data from user input.**
2. **Experiment Setup:**
   * **The modified program prompts the user to enter text, which is then used to overflow buffer1.**

**Modified Code Explanation**

**Here's the critical portion of the modified code that introduces the heap overflow vulnerability:**

****

**This function fails to check the length of the input before copying it into buffer1, which only has space for 10 bytes, thus any input longer than this will overflow into the adjacent buffer2 or other critical heap structures.**

## Execution and Outcome

* **Input Preparation:**
  + **A crafted input string was created to significantly exceed the 10-byte limit of buffer1.**
  + **Example of the input: AAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB**
* **Execution and Monitoring:**
  + **The program was executed with the exploit input, causing it to crash due to a segmentation fault, indicating successful overflow and corruption of heap metadata.**

## Results and Analysis

* **Heap Corruption:**
  + **The program crashed with a "Segmentation fault" and an error indicating issues with heap management (munmap\_chunk(): invalid pointer), confirming that the overflow corrupted heap metadata.**
* **Heap Layout Visualization:**
  + **The screenshots capture the state of the memory and the error messages post-crash, illustrating the effects of the heap overflow.**

## Screenshots and Diagrams

1. **Execution Error Due to Overflow:**
   * **Capturing the program's crash and the specific heap management error. A screen shot of a computer code

     Description automatically generated**
2. **Exploit Input Details:**
   * **Displaying the actual input used in input.txt that triggered the overflow. A close up of a logo

     Description automatically generated**

# Question 4: Exploiting a Program with Multiple String Checks

## Objective

**To manipulate an unknown binary (Q4) and cause it to execute a function you\_win, which prints a flag. The binary checks the input string against multiple conditions that must be satisfied.**

## Methodology

1. **Analysis:**
   * **The provided binary included a function check\_string that applied several conditions to the input string to determine if further action should be taken. The assembly code was analyzed to decipher these conditions.**
2. **String Construction:**
   * **Based on the assembly code analysis, a string was crafted that passed all checks, ensuring that the function you\_win would execute.**

## Assembly Analysis and String Requirements

**The assembly code provided checks that the input string meets specific conditions:**

* **It checks if the string is a palindrome.**
* **It performs multiple comparisons between different characters in the string to ensure they meet certain conditions (like being a certain character or the product of other characters).**

## Strategy for Crafting the String

**A Python script was written to generate a string that met all the requirements determined from the assembly code analysis. The script ensured the string would be a palindrome and meet the specific character conditions laid out in the disassembly.**

## Execution and Outcome

* **Execution Steps:**
  + **The crafted string was saved to q6answer.txt.**
  + **The binary was executed with this input, causing the you\_win function to execute and display the flag.**
* **Flag Acquisition:**
  + **The flag, as revealed by the successful execution of the you\_win function, was captured and noted.**

## Results and Analysis

* **Verification of the Flag:**
  + **The correct execution of the you\_win function and the display of the flag confirmed the success of the exploit.**
* **Contents of q6answer.txt:**
  + **The exact content of the input file was 34hackerC!""!Crekcah43, designed to meet the conditions extracted from the disassembly**

## Screenshots and Diagrams

1. **Crafting the Input String:**

****

1. **Python script used to generate the input string. A screenshot of a computer program

   Description automatically generated**
2. **Execution Output Capturing the Flag:**
   * **Screenshot showing the successful capture of the flag**

# Conclusion

Throughout the course of addressing these questions, we have demonstrated a comprehensive understanding of various security and exploitation techniques using the GDB debugger, string manipulation, and memory handling in C programming. Each question presented unique challenges that required a combination of theoretical knowledge and practical skills in computer security.

### Buffer Overflow and Stack Manipulation (Q1):

We effectively utilized GDB to trace function calls, inspect memory addresses, and manipulate the program's execution flow. By examining the stack and manipulating input files, we managed to redirect the execution to desired functions (secret\_function and secret\_function2), which showcased practical buffer overflow exploitation to alter program behavior.

### Heap Overflow and Memory Corruption (Q3):

By modifying the program to create a heap overflow scenario, we were able to demonstrate how unsafe memory operations can lead to executable exploits. The structured overflow allowed us to corrupt heap memory and manipulate pointers, leading to unexpected behaviors and potential for further exploitation.

### Exploit Reliability and ASLR (Address Space Layout Randomization) (Q2):

We explored the implications of ASLR on exploit reliability, learning how to disable this security feature for consistent behavior during testing. This knowledge is critical in both penetration testing and in understanding how modern operating systems defend against exploits.

### Crafting Condition-Specific Payloads (Q4):

The task involved creating a string that passed multiple specific checks by analyzing disassembled code. This not only tested our ability to understand assembly language but also to apply logical reasoning to satisfy complex conditions, leading to successful exploitation.