CS 647: Counter Hacking Techniques

Sam Assessment Report

Fall 2024

Aleyna Aydin

10/27

Executive Summary:

The objective of this assessment was to analyze the `helloVuln5` program in the Sam terminal for potential buffer overflow vulnerabilities. The goal was to exploit this weakness and gain shell access with the privileges of the samflag user. After identifying a buffer overflow in the program's `vulnFunction`, I crafted and executed an exploit to manipulate the program's control flow and inject shellcode into memory. This exploit successfully provided access to the samflag shell and the contents of the `samflag.txt` file.

The vulnerability identified was a buffer overflow that allowed data to overwrite memory beyond the buffer's allocated space. By taking control of the Extended Instruction Pointer (EIP), I was able to redirect the program's execution to the injected shellcode. Using tools like the GNU Debugger (GDB), I calculated the precise offset between the start of the input buffer and the return address in memory. This enabled the creation of a payload that not only filled the buffer but also altered the EIP to jump to the memory location where the shellcode was injected. After finalizing the payload, the shellcode was appended to the input. Finally, the return address was found by analyzing the stack to determine the address of where the shellcode is located.

The risks posed by this attack were significant, as it allowed unauthorized control of the system, access to confidential data, and the ability to execute arbitrary commands. Although the program had some mitigations in place—such as input length checks and environmental variable sanitization—these were insufficient and failed to prevent the overflow.

To mitigate this vulnerability, it is recommended that the user enable buffer overflow defenses, including Address Space Layout Randomization (ASLR), Data Execution Prevention (DEP), and Stack Smashing Protection (SS). Additionally, safer input-handling functions like `strncpy()` should be used to ensure consistent buffer size management. These mitigations would help prevent attackers from exploiting similar vulnerabilities in the future.

In conclusion, I successfully achieved the objective of this assessment by gaining shell access through a buffer overflow exploit and displaying the contents of the `samflag.txt` file. This result demonstrates the importance of implementing robust security measures to prevent unauthorized access and ensure the integrity of the system.

1 Objectives

The assessment's objective was to analyze the `helloVuln5` program in the Sam terminal for potential buffer overflow vulnerabilities, specifically focusing on `vulnFunction`. I aimed to exploit these vulnerabilities to manipulate the program's control flow and inject shellcode into memory, gaining shell access with the privileges of the samflag user. Through a successful exploitation, I gained access to the samflag shell and recovered the contents of the `samflag.txt` file to validate the effectiveness of the exploit.

2 Attacks

2.1 Buffer Overflow Exploit

Exploit #1	*Buffer Overflow Exploit on the helloVuln5 program*	
Description	The buffer overflow exploit occurs in the `helloVuln5` program. It overflows the buffer and overwrites existing data, including the return address of the Extended Instruction Pointer (EIP), to execute arbitrary shellcode injected into the buffer, allowing the manipulation of the program's execution flow. The attack relies on determining the exact buffer size and injecting shellcode into an area of memory in which it can be pointed to and executed, gaining a shell on the system. The impact of this exploit is gaining unauthorized access to the samflag user shell and uncovering the contents of the `samflag.txt` file on the system, compromising system confidentiality and integrity.	
Objectives	The objective is to craft a buffer overflow exploit to gain control of EIP and achieve a shell to gain access to the samflag user and uncover the contents of the `samflag.txt` file	
Assumptions	This exploit depends on buffer overflow defenses like Address Space Layout Randomization (ASLR), Data Execution Protection (DEP), and Stack Smashing Protection (SS) being disabled.	
Findings	The exploit was successful due to a buffer overflow vulnerability found in `vulnFunction` leading to the injection of arbitrary shellcode to gain shell access to the samflag user. Post-exploitation, I uncovered the contents of the `samflag.txt` file.	
Mitigations	To prevent this vulnerability, follow these practices: ● Enable ASLR to create randomized memory addresses making it more difficult for attackers to predict exploit specific memory addresses. ● Enable DEP to mark specific areas of memory as non-executable, preventing code written to these areas from being executed. ● Enable SS protection to create a canary value before the intended return address to verify that the memory has not been overwritten, aborting the program if tampering is detected. ● Use safer input functions such as `strncpy()` to limit input size and verify the buffer size matches. Through these mitigations, the user can achieve a safer system.	
Tools Used	`gdb`	This tool is the GNU Debugger which allows users to analyze and troubleshoot programs, more specifically, to inspect

		registers, the stack, and disassembled C code.
	`disassemble`	This command disassembles the specified code segment input to be able to analyze its x86 assembly code. This is used with the functions `main` and `vulnFunction`.
	`x`	This command examines the stack. It can be used with a number followed by `xw` to examine a specific set of hex words in the stack. If followed by `$esp`, the stack will be displayed starting with the address at register esp which points to the top of the stack.
	`perl`	This command is used to generate specific input strings including a certain number of bytes followed by a target address. When used with the `-e` flag, this command allows the execution of a Perl command directly from the command line.
	`break`	This command is used to set a breakpoint at a specified location in the program.
Commands Used with Syntax	1. `./helloVuln5 $(perl -e ‘print “A”x748’)$(cat shell.bin)$(perl -e ‘print “\x99\xd2\xff\xff”’)`	
Details	In this exploit, I successfully overwrote the return address of the EIP register to take control of the program’s flow through a buffer overflow attack. To determine the exact payload size, I calculated the buffer length allocated to the input. I achieved this by inputting a distinguishable variable into the program, then locating it in the stack and subtracting its address from the stack address where the return address of the function resides in memory. After calculating the difference and subtracting the shellcode size, I determined that the final payload required was 748 characters to fill the allocated buffer space. Since the environmental variables were sanitized, I had to use a stack diagram of the `vulnFunction` to determine where to insert the shellcode. To bypass this mitigation, I inputted the shellcode as a program argument, loading it into virtual memory. Instead of placing the shellcode after the return address, I inserted it before the return address. To determine the correct address to jump to, I inspected the stack to identify the shellcode’s starting address in memory (`0xffffd299)`. Using this address as the new return address, I was able to exploit the program’s vulnerability. As a result of	

this exploit, I gained access to the samflag shell and successfully displayed the contents of the `samflag.txt` file.

Screenshot 1: Payload and successful output

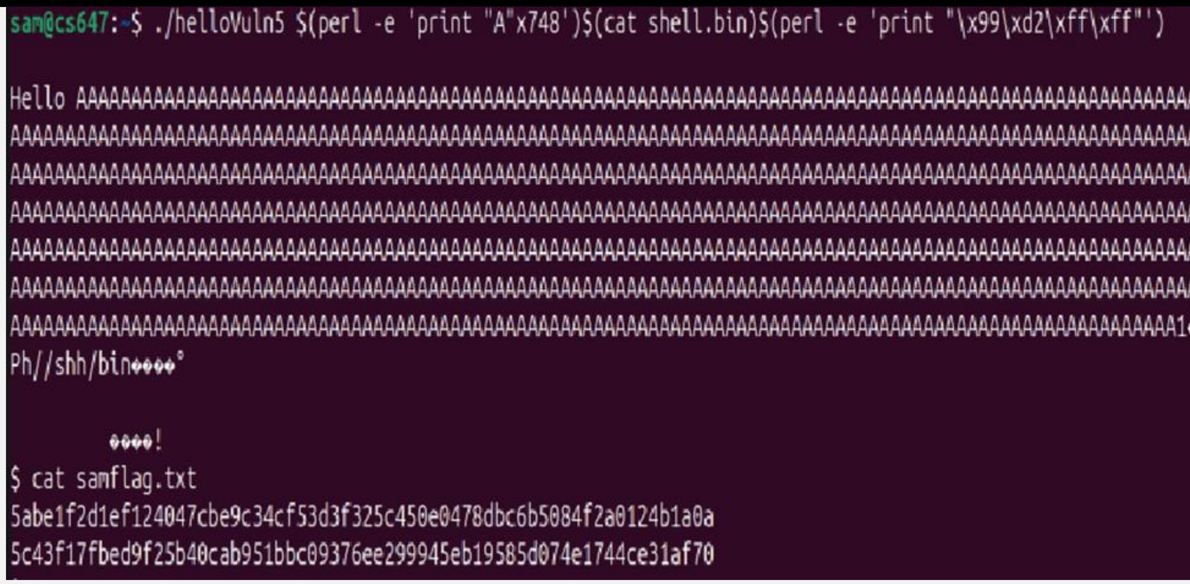
Below is a stack frame diagram of vulnFunction:

0xffffcff8	saved Extended Base Pointer (EBP)
0xffffcffc	saved return address (EIP)
0xffffd000	saved Extended Base Register (EBX)
0xffffd004	saved Extended Destination Index (EDI) Register
0xffffd008	0x300 (768) bytes allocated for local variables
0xffffcff0	current Extended Stack Pointer (ESP)

3 Flag

Flag Contents
5abe1f2d1ef124047cbe9c34cf53d3f325c450e0478dbc6b5084f2a0124b1a0a 5c43f17fbed9f25b40cab951bbc09376ee299945eb19585d074e1744ce31af70

Screenshot



Screenshot 2: Contents of the flag file