CS 647: Counter Hacking Techniques

Merry Assessment Report

Fall 2024

Aleyna Aydin
Nicholas Richmond
Shraddha Sawarna

10/13/24

Executive Summary:

The primary objective of this assessment was to identify and exploit a buffer overflow vulnerability in the program located at `home/merry/retAddr3`. The team aimed to manipulate the program's execution flow to call the `getFlag` function by injecting a specific address to control the Extended Instruction Pointer (EIP) register, which holds the next instruction to be executed. This involved calculating the necessary padding to control the contents of EIP and finding the correct injection point.

The identified vulnerability stemmed from the use of the `strcpy()` function, which accepts user input from the Merry terminal. This function is susceptible to buffer overflow attacks due to its lack of input validation, allowing input to overwrite memory beyond its allocated buffer. In C, the absence of built-in safeguards increases this risk, enabling attackers to alter program flow and potentially execute arbitrary code.

The team executed an attack by creating a buffer overflow that exploited the `strcpy()` vulnerability to overwrite the EIP register. Using the GNU Debugger (GDB), the correct offset and return address were calculated through disassembly of the program's `main` and `vuln` functions and examination of the stack. The overflow was achieved by inputting the necessary padding and the address `0x565563b2`, successfully redirecting execution to the `getFlag` function and retrieving the contents of 'merryflag.txt'. The results demonstrated effective modification of the program's execution flow through this overflow.

To mitigate the vulnerability, several strategies can be implemented. For instance, using stack canaries—random values placed in memory to detect corruption—can terminate execution upon detecting an overflow. Additionally, implementing Data Execution Prevention (DEP) and Address Space Layout Randomization (ASLR) can restrict executable code on the stack and randomize memory addresses, respectively. Specifically, instead of using `strcpy()`, the `strncpy()` function can be employed to enforce size constraints on input, preventing memory overwrites and protecting against buffer overflows.

1 Objective

The team was tasked with exploiting a buffer overflow vulnerability in the ‘retAddr3’ program. They aimed to overwrite the Extended Instruction Pointer (EIP) to force the execution of the bypassed function, `getFlag`, which holds the contents of the ‘merryflag.txt’ file. The scope involved analyzing the program’s input handling to identify buffer management weaknesses, determining the buffer size required to craft an effective payload, and using memory inspection tools like GNU debugger (GDB) to analyze x86 assembly code and the stack to gain control over the program’s execution flow.

2 Attack

2.1 Buffer Overflow Exploit

Exploit #1	**Buffer Overflow Exploit on retAddr3**	
Description	The attack leverages a buffer overflow vulnerability in the ‘retAddr3’ program. It overflows the input buffer, allowing the attacker to overwrite the return address and redirect program execution to the `getFlag` function. The attack relies on finding the exact buffer size, overwriting the return address with a specific function’s address, and controlling program execution. It is a memory corruption attack in which user supplied input manipulates memory behavior.	
Objectives	The team was asked to exploit a buffer overflow vulnerability in the ‘retAddr3’ program. The objective is to overflow the buffer and overwrite the return address with an address allowing the execution of the `getFlag` function, enabling arbitrary code execution. The final goal is to retrieve the contents of ‘merryflag.txt’ stored in this function.	
Assumptions	This exploit depends on the disabling of Address Space Layout Randomization (ASLR), Stack Smashing protection (SS), and Data Execution Prevention (DEP) to correctly determine the offset of the attack.	
Findings	The contents of the `merryflag.txt` file were successfully retrieved by executing the `getFlag` function. This was made possible due to a buffer overflow vulnerability in the `vuln` function, specifically at the `strcpy()` instruction. Exploiting this vulnerability allowed for control over the program’s execution, leading to command execution under the `merry` user. Post-exploitation, the team accessed the contents of the `merryflag.txt` file.	
Mitigations	To ensure the safety of a program against Buffer Overflow Exploits, follow these practices: ● Enabling ASLR to randomize the memory addresses of key programming components, making it difficult to predict and exploit specific memory locations. ● Enabling SS protection to abort program execution if program tampering is detected. ● Enabling DEP to mark certain areas of memory as non-executable. ● Use the `strncpy()` function rather than `strcpy()` to constrict the input size of the program. These measures can help maintain security.	
Tools Used	`gdb`	This tool is the GNU Debugger which allows the team to analyze and troubleshoot programs, more specifically, to inspect registers, the stack, and disassembled C code.

<table>
<tr><td></td><td>disassemble</td><td>This command disassembles the specified code segment input to be able to analyze its x86 assembly code. This is used with the functions <code>main</code> and <code>vuln</code>.</td></tr>
<tr><td></td><td>x</td><td>This command examines the stack. It can be used with a number followed by <code>xw</code> to examine a specific set of hex words in the stack. If followed by <code>$esp</code>, the stack will be displayed starting with the address at register <code>esp</code> which points to the top of the stack.</td></tr>
<tr><td></td><td>perl</td><td>This command is used to generate specific input strings including a certain number of bytes followed by a target address. When used with the <code>-e</code> flag, this command allows the execution of a Perl command directly from the command line.</td></tr>
<tr><td></td><td>break</td><td>This command is used to set a breakpoint at a specified location in the program.</td></tr>
<tr><td>Commands Used with Syntax</td><td colspan="2">1. gdb retAddr3
2. disassemble main
3. disassemble vuln
4. break *0x5698232c
5. run AAAAAAAAAAAAA
6. x/400xw $esp
7. ./retAddr3 $(perl -e ‘print “A”x1031 .
“\xb2\x63\x55\x56”’)</td></tr>
<tr><td>Details</td><td colspan="2">In this exploit, a buffer overflow allowed the team to overwrite the return address in the program’s memory. The payload consisted of 1031 “A” characters, which were used to fill the buffer and reach the return address. After the buffer was filled, the address ensuring the execution of the <code>getFlag</code> function, <code>0x565563b2</code>, was inserted in little-endian format (<code>\xb2\x63\x55\x56</code>). This address overwrote the original return address, forcing the <code>getFlag</code> function to execute and revealing the contents of the ‘merryflag.txt’ file. This vulnerability arose due to the lack of input validation which allowed the team to provide more input than the buffer was meant to accommodate, leading to a buffer overflow and gaining control of program execution. The tool GDB played a crucial role in this process by helping analyze the stack of the <code>vuln</code> function to determine the buffer payload. Crafting the payload through GDB ensured that the exploit effectively overwrote the contents of EIP, triggering the execution of the <code>getFlag</code> function and revealing the contents of the flag file.

Below is a stack frame diagram of the function <code>vuln:</code>
<table>
<tr><td>Return Address</td><td>Saved return address</td></tr>
<tr><td>%<code>ebp</code> (Extended Base Pointer)</td><td>The old base pointer</td></tr>
</table>
</td></tr>
</table>

	%edi (Extended Destination Index)	Callee-saved register	
	%edx (Extended Data Register)	Callee-saved register	
	Local Variables: -0x4c4(%ebp) -0x4c0(%ebp) -0x4bc(%ebp) local buffer	Buffer and variables	
	Arguments: 0x10(%ebp) → passed to vuln, loaded into %eax 0xc(%ebp) → loaded into %edx 0x8(%ebp) → loaded into %ecx	Arguments of inner function calls	

3 Flags

3.1 Aleyna Aydin

Flag Contents
5dba760944e51261a450ee5534ddd93dc418661cd9937a11cd70b6ec37d621a2 **9a52a520777d871c7d906a0b6599a876ea6a2e7a2762b7133bba2ffa00329bb9**

Screenshot

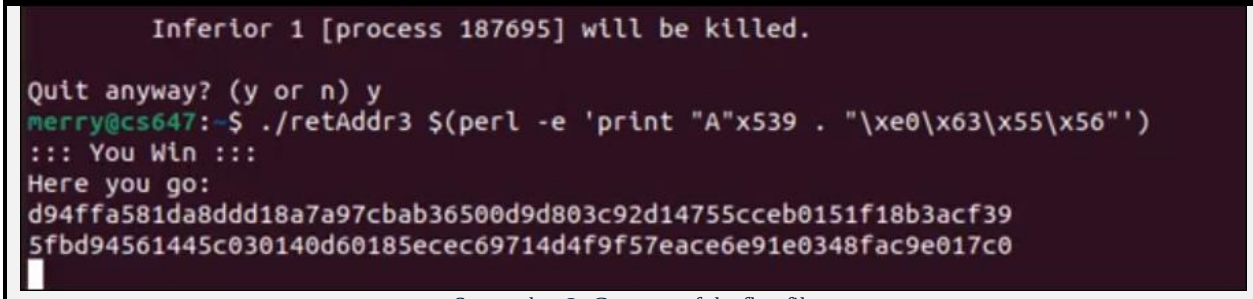
```
merry@cs647:~$ ./retAddr3 $(perl -e 'print "A"x1031 . "\xb2\x63\x55\x56"')
::: You Win :::
Here you go:
5dba760944e51261a450ee5534ddd93dc418661cd9937a11cd70b6ec37d621a2
9a52a520777d871c7d906a0b6599a876ea6a2e7a2762b7133bba2ffa00329bb9
Segmentation fault (core dumped)
```

Screenshot 1: Contents of the flag file

3.2 Nicholas Richmond

Flag Contents
d94ffa581da8ddd18a7a97cbab36500d9d803c92d14755cceb0151f18b3acf39 **5fbd94561445c030140d60185ecec69714d4f9f57eace6e91e0348fac9e017c0**

Screenshot



Screenshot 2: Contents of the flag file

3.3 Shraddha Sawarna

Flag Contents

516ddaf9467c8f155cf6782e5e9fe247402145e6eacf6d7b24f0fcf399f9d746

5b5c772913a66b0513b95753c9d68df87da904dcf87a5d09b8a33e0f15ed40f7

Screenshot

Screenshot 3: Contents of the flag file and whoami command