CS 647: Counter Hacking Techniques

Legolas Assessment Report

Aleyna Aydin
Fall 2024
11/17

Executive Summary

The goal of this assessment was to analyze the `vulnFileCopy2` program on the 'Legolas' terminal for vulnerabilities and exploit them to gain shell access to the user 'legolasflag' and obtain its privileges. The objective was to retrieve the contents of the hidden `legolasflag.txt` file and ensure the shell exited gracefully without crashing. This required bypassing protections such as Address Space Layout Randomization (ASLR), Stack Smashing (SS), and Data Execution Protection (DEP).

The vulnerabilities identified included a buffer overflow vulnerability, which allowed overwriting memory and altering the program's execution flow, as well as a format string vulnerability, which improperly handled input and allowed arbitrary memory access. These vulnerabilities presented a high risk, as they allow an attacker to bypass security mechanisms, gain unauthorized access, alter program execution, and compromise system integrity and confidentiality.

To mitigate the vulnerabilities exploited in this attack, input validation should be enforced to prevent malicious format specifiers from being passed into functions like `printf`, and safer alternatives such as `snprintf` should be used. Proper buffer size management is also essential by allocating memory based on the exact input size and using secure functions like `strncpy` to prevent buffer overflows.

The exploit was carried out in several stages. First, the `chmod 777` command was used to modify the program's permissions, enabling it to be executed. To bypass ASLR, the GNU Debugger (GDB) was employed to locate key memory offsets, including the canary value, main address, and system function addresses, all of which were essential for constructing the buffer overflow exploit. A return-to-libc attack was utilized to bypass DEP, allowing for the execution of shellcode. Additionally, the stack protection mechanism was circumvented by leaking the canary value and overwriting it with itself, effectively bypassing stack smashing detection. The format string vulnerability was exploited to leak the necessary addresses for the attack and acted as the file name for the input to be short enough to pass the size check imposed by the program. Once these addresses were retrieved, a payload was carefully crafted and written to a file with the same name as the program input. By pausing the program with 'Ctrl + Z', the payload was executed once the program was resumed, successfully completing the exploit.

Post exploitation, the `legolasflag.txt` file was discovered by gaining access to the 'legolasflag' user through the shell, which was able to exit without crashing. The exploit successfully bypassed all protections, achieving the intended results.

1 Objectives

1.1 Identify Program Vulnerabilites

The primary objective of this assessment is to analyze the `vulnFileCopy2` program to identify security vulnerabilities that could be exploited to gain unauthorized privileges. Specifically, this assessment focuses on uncovering weaknesses that may provide access to the 'legolasflag' user, enabling retrieval of the `legolasflag.txt` file, which is accessible only to this user.

1.2 Gain Shell

To successfully gain access to the 'legolasflag' user, the second objective is to craft an exploit that can achieve shell access on the system with the required privileges. This shell is essential to interact with and retrieve the `legolasflag.txt` file.

1.3 Exit Shell Gracefully

Upon achieving shell access, the final objective is to ensure that the shell exits gracefully, preventing the `vulnFileCopy2` program from crashing or encountering segmentation faults. This involves carefully managing program execution so that it resumes normally after the shell is closed, directing it to a valid address to avoid any program disruption.

2 Attacks

2.1 Buffer Overflow

Exploit #1	Buffer Overflow exploit on vulnFileCopy2
Description	A buffer overflow is a memory corruption attack where an attacker inputs more data than a program's buffer can hold, causing overflow into adjacent memory regions. This overflow can overwrite critical information like return addresses, enabling the attacker to redirect program execution. By carefully crafting the overflowed data, an attacker can manipulate the program to execute unauthorized code or access restricted functions and data. The impact varies but can include unauthorized data access, code execution, or program crashes, leading to service disruption. Buffer overflow attacks are severe, as they exploit vulnerabilities to gain control over system functions or data, and can potentially escalate privileges, depending on the affected program's permissions and system role. This relies on exploit 2.2 due to a length check done on the file's name inputted to the program.
Objectives	This attack relates to objectives 1.1, 1.2, and 1.3, as it requires spawning a shell on the system and exiting gracefully to gain access to the `legolasflag` user. A buffer overflow can take control of a program by forcing it to execute arbitrary code—in this case, the shellcode that we aim to direct the program to within memory.
Assumptions	This attack assumes that the buffer is large enough to allow for the injection of arbitrary code since a properly limited buffer size would prevent overflow. Additionally, this attack requires the bypassing of input length checks, relying on exploit 2.2 to circumvent restrictions that would otherwise block execution if the input is too lengthy.

<table>
<tr><td>Findings</td><td>The successful exploit resulted in gaining access to the `legolasflag` user and viewing the contents of the <code>legolasflag.txt</code> file. The buffer overflow vulnerability in the <code>vulnFileCopy2</code> function enabled this exploit, allowing shellcode execution with `legolasflag` user privileges. During post-exploitation, I was able to retrieve the contents of the <code>legolasflag.txt</code> file, and the shell exited without causing any program crashes, fulfilling the exploit requirements.</td></tr>
<tr><td>Mitigations</td><td>To mitigate this attack, it is essential to create a buffer with a fixed size that matches the required input size, ensuring that overflow cannot occur. The buffer should be properly sized to handle only the necessary input, preventing data from exceeding allocated memory space. By enforcing strict input validation, ensuring that the data fits within the allocated buffer, and preventing any extra data from being written to adjacent memory regions, the risk of a buffer overflow can be effectively mitigated. This will protect the program from exploitation, even with security protections in place.</td></tr>
<tr><td>Tools Used</td><td>
<table>
<tr><td><code>perl</code></td><td>This command was used to generate specific input strings for the exploit, including a certain number of padding bytes followed by the target address. The <code>-e</code> flag allowed direct execution of the perl command from the command line, enabling the creation of input data that would overflow the buffer and redirect the program's execution flow to the injected shellcode.</td></tr>
<tr><td><code>gdb</code></td><td>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.</td></tr>
<tr><td><code>disassemble</code></td><td>This command in GDB was used to disassemble the specified code segment and analyze its x86 assembly code. It was particularly useful for inspecting the main and <code>vulnFileCopy2</code> functions to understand how the program operates and identify potential vulnerabilities, such as buffer overflows.</td></tr>
<tr><td><code>break</code></td><td>This command in GDB sets a breakpoint at a specified location in the program. This allows the execution of the program to pause at that point, enabling closer inspection of the program’s state, such as the values of registers or the stack. Breakpoints were set at key locations to control program flow and gather relevant data.</td></tr>
</table>
</td></tr>
</table>

	`p`	This command in GDB is used to print the value of an expression, such as variables, memory addresses, or function return values. It was used to print the value of the system function address, which was critical for executing the shellcode and gaining access to the shell.
	`info proc mappings`	This command in GDB displays the memory mappings of the process, including the virtual address ranges for memory segments such as the heap, stack, and other memory regions. It was used to determine the layout of the program's memory and locate areas for injecting shellcode or identifying vulnerable memory regions.
	`find`	This command in GDB was used to search for a specific pattern or value in the program's memory. It was helpful for locating important data, such as the `/bin/sh` string.
	`info address`	This command in GDB is used to display the memory address of a specified symbol, function, or variable in the program's memory space. When you run this command, it provides the address where the symbol (such as a function like __libc_start_main or exit) is located in memory, allowing you to identify its position in the address space.
Commands Used with Syntax	`perl -e 'print "A"x646 . "\x00\x58\xc0\xe2" . "A"x12 . "\x30\xc4\x21\xec" . "\x99\xab\x20\xec" . "\xe8\x0d\x39\xec"' > '%175$x %199$x'`	
Details	In this Buffer Overflow exploit, I inserted 646 "A" characters into the buffer to overwrite the allocated memory space up to the canary value. This number was determined by analyzing the `vulnFileCopy` function, where a loop writes the character "A" to the buffer. The instruction `cmpl $0x285,-0x2a0(%ebp)` compares the value of `ebp - 0x2a0` with `0x285` (645 in decimal), which is the maximum number of characters the buffer can hold without causing an overflow. To trigger the buffer overflow, I added one more character (646) to exceed the buffer size, causing it to overwrite adjacent memory. Since the system has Data Execution Prevention (DEP), Stack-Smashing Protection (SS), and Address Space Layout Randomization (ASLR) enabled, a direct buffer overflow alone is insufficient to exploit the vulnerability. To bypass these protections, I used a return-to-libc attack. I first needed to extract the canary value from memory to correctly construct the exploit's payload, as the addresses shift with each execution due	

	to ASLR. The canary value was found using the technique described in exploit 2.2 and was inserted back into the payload in little-endian format. This ensures that the canary value is correctly restored, bypassing the stack-smashing protection that would otherwise terminate the program. After the canary value, I inserted 12 additional "A" characters to account for the space between the canary and the return address that needs to be overwritten. This value was determined by analyzing the stack frame in the disassembled code of `vulnFileCopy`. Between the canary and the return address, there is an 8-byte alignment and the previous value of ebp (extended base pointer), which needed to be considered when constructing the payload. The next addresses in the payload correspond to the return-to-libc attack, specifically the addresses of the `system`, `exit`, and command line functions. To locate these addresses, I used GDB to disassemble the code and find the address of the `__libc_start_main` function. The reason this reference point works is that the offsets of functions within the libc library are consistent across executions, as the memory layout of the library itself remains static relative to the `__libc_start_main` function. By calculating the offsets from this known function, I can reliably find the memory addresses of other functions in the libc library, even when ASLR (Address Space Layout Randomization) is enabled, because the relative offsets between functions within libc do not change. From this base address, I subtracted the respective function addresses to calculate the offsets for `system`, `exit`, and the command line function. After these calculations, I identified the following offsets: `system` address: 0x2BB777, `exit` address: 0x19EE0, and the command line address: 0x1A012F. The completed payload was written to a file named `'%175$x %199$x'`, as part of the exploit described in exploit 2.2. This file was then used to trigger the buffer overflow and successfully gain control of the program's execution flow, allowing for the execution of arbitrary code.

2.2 Format String

Exploit #2	Format String exploit on vulnFileCopy2
Description	A format string vulnerability exploit takes advantage of improper handling of input data in software programs that use format string functions without properly validating input. This attack occurs when an attacker injects malicious format specifiers into input fields, causing the program to read or write arbitrary memory locations. By manipulating the format string, attackers can access sensitive data or cause memory leaks. This exploit can severely compromise the security and stability of the system, as it may allow attackers to control program execution or access unauthorized information. This vulnerability can be used in combination with other attacks, such as buffer overflows, to escalate privileges or bypass security protections.
Objectives	This exploit corresponds to objective 1.1, as it identifies a vulnerability that facilitates the buffer overflow exploit. The use of the `printf` function in the program without proper input validation creates the opportunity for a format string vulnerability, which can be leveraged to trigger a buffer overflow.

<table>
<tr><td>Assumptions</td><td>This exploit depends on the assumption that the program uses a format string function, like <code>printf</code>, without properly validating its input fields. This vulnerability allows the program to read from arbitrary memory locations, which is crucial for the exploit to succeed. Additionally, the attack assumes that the program does not implement adequate input sanitization or defensive measures, such as bounds checking.</td></tr>
<tr><td>Findings</td><td>The successful result of this attack was the ability to perform a buffer overflow and bypass the input length check. The format string vulnerability in <code>vulnFileCopy</code> allowed the attacker to print the canary value and __libc_start_main system address, aiding in the exploit. Post-exploitation, the buffer overflow was used to gain control of program execution. The vulnerabilities enabling this attack were the format string issue and buffer overflow, which allowed manipulation of memory and bypassed protections.</td></tr>
<tr><td>Mitigations</td><td>To address format string vulnerabilities, implement robust input validation and sanitization practices. Ensure that user inputs are thoroughly vetted before being used in format string functions, preventing the injection of malicious format specifiers. Avoid directly passing user inputs to functions like <code>printf</code>; instead, use static format strings or safely encapsulate inputs. For situations where user input must be included in formatted output, utilize functions such as <code>snprintf</code>, which allow explicit definition of the format, offering an added layer of security.</td></tr>
<tr><td>Tools Used</td><td><table>
<tr><td><code>perl</code></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 -e flag, this command allows the execution of a Perl command directly from the command line.</td></tr>
<tr><td><code>$x</code></td><td>This command is used as a format specifier to read values from the stack. It instructs the program to treat the corresponding argument as a pointer to a memory address and display the contents of that memory address in hexadecimal format.</td></tr>
<tr><td><code>%i</code></td><td>This command is a format specifier is used to print an integer in decimal format. When used as input, the program interprets it as a request to print the value of the next integer on the stack and leak its value.</td></tr>
<tr><td><code>%08x</code></td><td>This command is used to read and leak memory addresses by printing them as 8-digit hexadecimal values.</td></tr>
</table></td></tr>
<tr><td>Commands Used with Syntax</td><td><code>./vulnFileCopy2 ‘%175$x %199$x’</code></td></tr>
<tr><td>Details</td><td>In this attack, I used a format string vulnerability to bypass input length checks and gather essential memory information for a successful exploit. The program immediately requests a file name and checks its length, but by injecting a format string, I could control the input and extract critical data from memory.</td></tr>
</table>

	To begin, I needed to identify the locations of the canary value and the __libc_start_main address within the memory. To achieve this, I used the command `perl -e 'print "%08x...." x 250'` which allows me to leak memory contents by repeatedly printing hexadecimal representations of the stack's values. This leaked memory revealed the canary value, which appeared right after the `A` characters inputted by the program, making it identifiable as the 175th value in the leak. Similarly, the __libc_start_main address was located by analyzing the program's disassembly, as described in exploit 2.1, which provided the offsets of the relevant libc addresses and was found to be the 199th value in the memory leak. Once these key addresses were identified, they were essential for crafting the buffer overflow exploit. By constructing the format string payload as `'%i$x %i$x'`, with '175' for the canary value and '199' for the __libc_start_main address, I was able to print these memory addresses from within the constraints of the program's input size, which ensured it would not trigger the length check. After executing the format string exploit, I used 'Ctrl+Z' to suspend the program and keep it in the background. This prevented ASLR from changing the addresses between the leak and the execution phase. I could then craft the buffer overflow payload and insert it into a file with the same name as my format string input, enabling the exploit to execute once I resumed the program.

3 Flag

Flag Contents
615e26bef61d303fb3f94926e98b2922c895a6c0c0bd358db6ec7cb89511ac89 74e078899a5f4794e7979a3b39eeb6cd4e9cf8ae45cd95417e20ec9e0bcc5073

Screenshot

```
legolas@cs647:~$ ./vulnFileCopy2 '%175$x %199$x'

File to copy: e2c05800 ec1f0cb9

Press enter to begin copying...^Z
[1]+  Stopped                 ./vulnFileCopy2 '%175$x %199$x'
legolas@cs647:~$ perl -e 'print "A"x646 . "\x00\x58\xc0\xe2" . "A"x12 . "\x30\xc4\x21\xec" . "\x99\xab\x20\xec" . "\xe8\x0d\x39\xec"' > '%175$x %199$x'
legolas@cs647:~$ fg
./vulnFileCopy2 '%175$x %199$x'


Done copying.
$ whoami
legolasflag
$ cat legolasflag.txt
615e26bef61d303fb3f94926e98b2922c895a6c0c0bd358db6ec7cb89511ac89
74e078899a5f4794e7979a3b39eeb6cd4e9cf8ae45cd95417e20ec9e0bcc5073
$ exit
legolas@cs647:~$
```

Screenshot 1: Successful exploit with contents of legolasflag.txt and whoami command