Legolas Assessment

Executive Summary:

This report identified vulnerabilities in the program vulnFileCopy2 and provided recommendations for securing it against potential attacks. The report identified five vulnerabilities: buffer overflow, insecure file handling, lack of input validation, incomplete setuid usage, and memory leak. Recommendations included explicitly setting the size of the data buffer, usage of more restrictive modes when opening files, validated command-line arguments, with the usage of setreuid function with appropriate parameters, and implementation of memory management techniques.

The report provided a detailed account of how the vulnerability was exploited, included the steps taken to reproduce the attack and retrieve the flag. We were able to bypass DEP (Data Execution Prevention), stack smashing protection and ASLR (Address Space Layout Randomization) to overwrite the canary value and return address to execute arbitrary code; in this case it was to view the contents of the flag in `legolasflag.txt`.

Vulnerabilities Identified:

Memory Leak: The program used the `printf(fileName)` in `vulnFileCopy()` to take input. This was a poor usage because the first argument should be format string that specified how the following arguments were formatted and printed. Since there was no format string here we were able to control how the function worked.

Buffer overflow: The function `vulnFileCopy` created buffer `data` without explicitly setting its size. The subsequent loop that copied data from the input file to the buffer used the size of the file, but if the file is larger than the buffer, a buffer overflow could occur, which could crash or allow arbitrary code execution.

Insecure file handling: The function `vulnFileCopy` used the `fopen` function to open the input file, but it does not specify a mode, which defaulted to "r" (read). This can be problematic if an attacker was able to control the file name and can supply a malicious file. Additionally, the function does not check the file's permissions or ownership, which could allow an attacker to read or modify sensitive files.

Lack of input validation: The program assumed that the input file name was the first command-line argument, but it did not verify if the argument is valid. An attacker could supply a malformed file name or no file name at all, which could crash the program or make it behave unpredictably.

Incomplete setuid usage: The program used the `setreuid` function to switch the effective user ID, but the actual values for the `ruid` and `euid` parameters are not specified. This could potentially allow an attacker to elevate privileges if the program ran as a privileged user.

Recommendations:

Always use a format string in `print()` function like `printf("%s", filename)` instead of `print(filename)`. This ensures that the input is printed as a string and any special characters in the input are properly handled by `printf()`.

Explicitly set the size of the data buffer to a safe value that is at least as large as the largest expected file. Use the `fopen` function with a more restrictive mode, such as "`rb`" (read binary), and check the file's permissions and ownership before opening it.

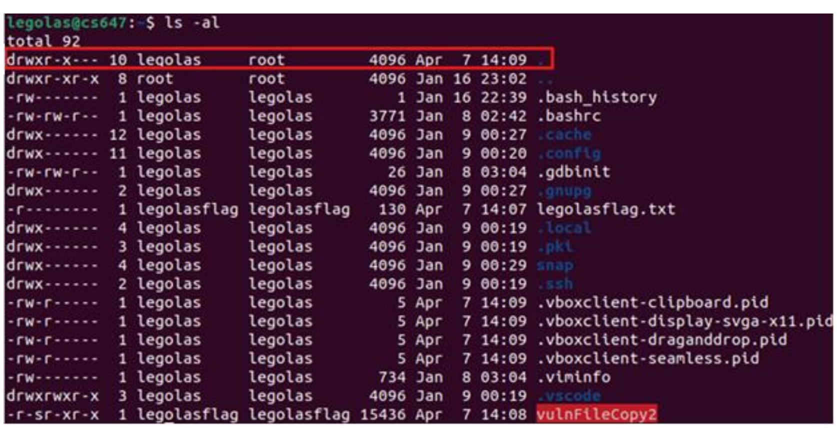
Validate the command-line arguments and provide appropriate error messages for invalid input.
Use the `setreuid` function with appropriate values for the `ruid` and `euid` parameters and ensure that the program is running with minimal privileges.

Assumptions:

We have made multiple assumptions for the exploit to work. The first one being, "`/bin/sh`" (Bourne Shell) exists in the system. Secondly, we assumed that we can create and write files in the system. Finally, when we ran the exploit, we assumed that the program ran with the same `libc` version we ran the exploit with.

Steps to Reproduce the Attack:

After we logged into the user Legolas, we found the file named `vulnFileCopy2`. We noticed that the `vulnFileCopy2` took a file as input. So, we created a random file called `input` with the command `touch input` and ran `./vulnFileCopy2 input`. The program printed an output that said `Permission denied`. I used the command `ls -al`. This command displayed a long list of files (because of the flag `-l`) and directories in the current directory, along with hidden files and directories (because of the use of the flag `-a`) as shown in screenshot 1.



Screenshot 1: Setuid permissions of the current working directory marked in red.

From screenshot 1, we inferred that in the current working directory only the owner or group of that specific file was able to run it. The owner and group for `vulnFileCopy2` was `legolasflag`. We changed the permission of the current directory with the command "`chmod 777 .`" and it gave permission for users to run a file in the current directory.

In the next step, we used the command `readelf -l vulnFileCopy2 | grep GNU_STACK` to check if the defense mechanism DEP(Data Execution Prevention) was enabled. This command displayed the information of the program header GNU_STACK as shown in screenshot 2. It had the flag set to RW which meant read and write instead of RWX which meant read write and execute. From this, we concluded that DEP was enabled.

```
legolas@cs647:~$ readelf -l vulnFileCopy2 | grep GNU_STACK
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x10
```

Screenshot 2: DEP check.

In the next step, we used the command `readelf -s vulnFileCopy2 | grep stack` to check if stack smashing protection was enabled. This command displayed the symbol table of `vulnFileCopy2`. The `grep stack` command was used to filter the symbol table data to check if there were any symbols related to stack smashing protection as shown in screenshot 3. From that, we deduced that stack smashing protection was enabled.

```
legolas@cs647:~$ readelf -s vulnFileCopy2 | grep stack
    28: 00000000     0 FUNC    GLOBAL DEFAULT  UND __stack_chk_fail[...]
```

Screenshot 3: Stack smashing protection check.

After that we used the command `sysctl kernel.randomize_va_space` to check if the defense mechanism ASLR(Address Space Layout Randomization) was disabled. This command printed the output as shown in screenshot 4. `kernel.randomize_va_space = 2` meant that ASLR was enabled. If ASLR was disabled, it would have printed `kernel.randomize_va_space = 0`.

```
legolas@cs647:~$ sysctl kernel.randomize_va_space
kernel.randomize_va_space = 2
```

Screenshot 4: ASLR check.

In the next step, we built a format for the payload that was used for the exploit. The payload format is shown in screenshot 5 where `buffer_size` represents the total buffer size of the program before we overwrote `canary_value`. `"A"x 12` was the two four-byte sized byte alignments plus four bytes of the `previous ebp`. Then, we entered the `canary_value` between `"A"x buffer_size` and `"A"x 12` to overwrite it with the same value to bypass stack smashing protection. The `system_addr` was the address of the `system()` function present in C-shared libraries(`libc`).

```
legolas@cs647:~$ perl -e 'print "A"x buffer_size . "canary_value" . "A"x 12
. "system_addr" . "system_return_addr" . "command_string_addr"' > payload
```

Screenshot 5: Creation of payload format.

Then, we used the `system()`address in the place of return address to bypass the defense mechanism DEP (Data Execution Prevention). Since DEP removed the possibility of injection and execution of our shell code, we used the `system()` address present in `libc` as it had executable code and was not protected by DEP. `system_return_addr` was the `system()` return address which was the address of `exit()` function needed for clean exit of program execution and `command_string_addr` was the address of the command string which was required to execute and spawn a shell. We stored the payload format in a file named `payload`.

In the next step, we began the process of calculation of every component in `payload`. First, to find the buffer size of the program, we disassembled `vulnFileCopy2` in GDB(GNU Debugger) with the command `gdb vulnFilecopy2`. After that, we used the command `break vulnFileCopy` to set a breakpoint at `vulnFileCopy()` followed by the usage of the `run` command to start the execution of program. Then we disassembled the `vulnFileCopy()` with the command `disassemble vulnFileCopy`.

```
0x56613368 <+3>:      sub     $0x7c8,%esp
```

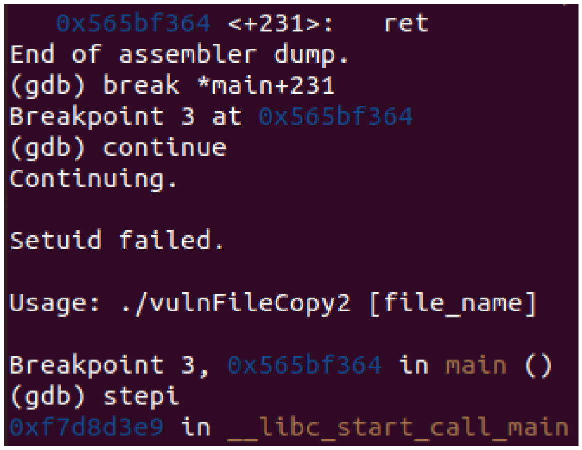
Screenshot 6: Calculation of upper bound of the buffer size with the help of assembly dump of `vulnFileCopy` function.

From the dump of disassembled `vulnFileCopy`, we found the upper bound of the buffer size as the instruction showed in screenshot 6, as it created `0x7c8`(1992 in decimal) amount of space in the stack. Then we exited the GDB shell with `exit` command and started to find the buffer size by the usage of a series of decremented inputs from 1992 till we got the exact buffer size as shown in screenshot 7. We concluded that 1952 was the size of the total buffer before it overwrote the canary value and printed “`***stack smashing detected***`”.

```
legolas@cs647:~$ perl -e 'print "A"x1952' > input
legolas@cs647:~$ ./vulnFileCopy2 input

File to copy: input

Press enter to begin copying...

Done copying.
legolas@cs647:~$ perl -e 'print "A"x1953' > input
legolas@cs647:~$ ./vulnFileCopy2 input

File to copy: input

Press enter to begin copying...

Done copying.
*** stack smashing detected ***: terminated
Aborted (core dumped)
```

Screenshot 7: Calculation of buffer size with a series of inputs

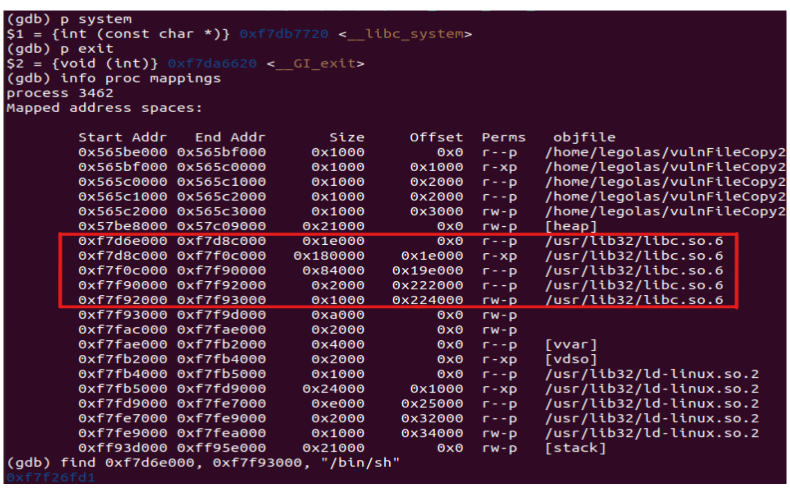
We set a breakpoint 2 at `main()` with the command `break main`. Then, we used `run` command to re-run the program. After that we disassembled the `main()` function with the command `disassemble main`. After that we set breakpoint 3 at the last instruction where it returned with the command `break *main+231` as shown in screenshot 8.



Screenshot 8: `main()` function that was called by `__libc_start_call_main()`.

After that, we used the command `continue` to continue the execution of the program. The program stopped after it reached breakpoint 3. At that point, the program was at the end of the main function, and in the next step the program would return to `start_main()` which called `__libc_start_call_main` in the present `libc` version where we ran the exploit. This was because the `main()` function was called by `__libc_start_call_main`.

After that, we used the command `p system` to get the address of `system()`. Then, command `p exit` to get the address of the `exit()` function that was used as `system()` return address and for the clean exit of program execution. Then, we used the command `info proc mappings` and it displayed the memory regions of current processes as shown in screenshot 9. Then we used the command `find 0xf7d6e000, 0xf7f93000, "/bin/sh"` to find the command string address. That command checked the entire range of `libc`, from `0xf7d6e000` to `0xf7f93000` for the command string `/bin/sh` as shown in screenshot 9.



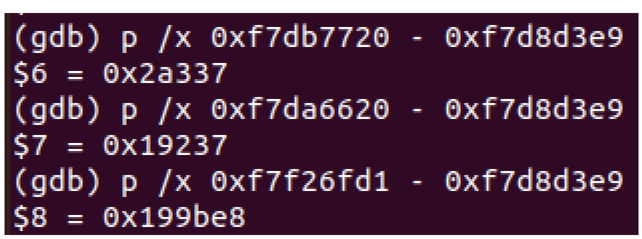
Screenshot 9: Finding the address of command string.

Since ASLR was enabled, the three addresses changed every time we ran the program. We first performed a memory leak from the top of the stack with the command `./vulnFileCopy2 $(perl -e 'print "%08x...."x600')`. %08x printed four bytes of data from the top of the stack and since the buffer size was `1952`, we printed four bytes of data `600` times to leak the data we wanted that was present after the buffer as shown in screenshot 10.

Screenshot 10: Memory leak to find canary and reference point of `main()`. Canary in red and the reference point in blue.

After that, we checked for a reference point where the main() returned to because it was also a function, and it would have been present somewhere in the stack. We were able to find the position of the reference point and it was marked in blue as shown in screenshot 10. We concluded this as the reference point by the observation of libc address spaces and the address spaces almost always started with f7xxxxxx. So, we concluded that the position of the reference point was 523 from the top of the stack.

In the next step, we performed subtraction between `system()` address, `system()` return address, command string address, and `__libc_start_call_main` address one at a time and obtained three new values. These three values respectively were the offsets of system address, system return address, and command string address as shown in screenshot 11. `0x2a337` was the offset for system address. `0x19237` was the offset for system return address. `0x199be8` was the offset for command string address.

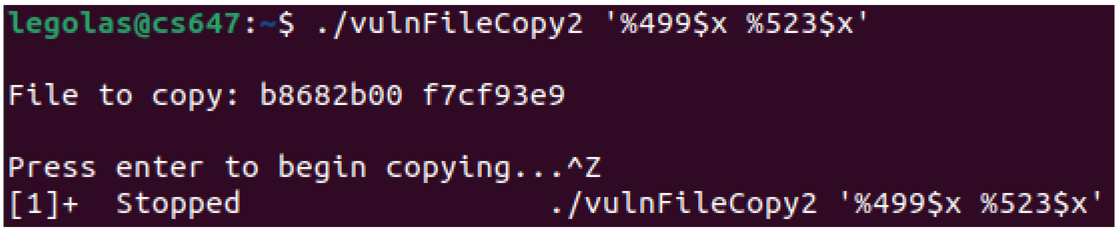


Screenshot 11: Calculation of offset values.

We noticed that `vulnFileCopy2` had a limited input file name size. The execution of `./vulnFileCopy2 $(perl -e 'print "%08x...."x600')` meant that we gave an input size of more than a thousand bytes. To overcome that issue, we used input format `./vulnFileCopy2`

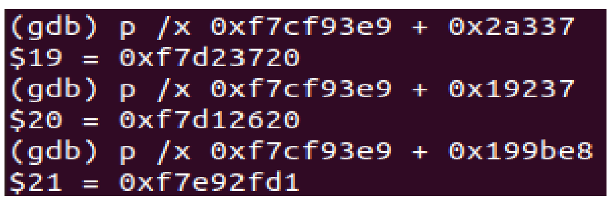
'%a$x' where *a* was the position of the value we wanted to print from the top of the stack and $ indicated to the print value of a'th position. No matter how many times we ran a program the contents of the stack changed but their position from the top of the stack did not change.

In the next step, we started the execution of our exploit with the command `./vulnFileCopy2 '%499$x %523$x'` and it printed the canary value and reference position of the `main()` function. After that, we used ctrl + z on the keyboard and paused the program as shown in screenshot 12. We performed this to fill `'%499$x %523$x'` as it was empty at that time and to update the canary value as it changed with every program execution and find the three new addresses with the help of offsets we calculated earlier.



Screenshot 12: First step of exploit to find canary and reference address.

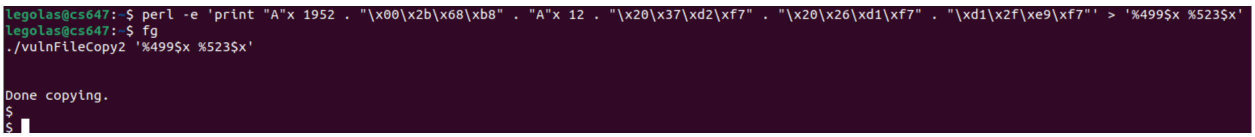
Then, we added the offsets we found earlier to `0xf7cf93e9` and calculated the new address of the system, system return address, and command string address as shown in screenshot 13. The new system address was `0xf7d23720`. The new system return address was `0xf7d12620`. The new command string address was `0xf7e92fd1`.



Screenshot 13: Calculation of new address with the help of offsets.

After that, we entered all the addresses and updated the canary in little Endean format. Our final payload was `perl -e 'print "A"x 1952 . "\x00\x2b\x68\xb8" . "A"x 12 . "\x20\x37\xd2\xf7" . "\x20\x26\xd1\xf7" . "\xd1\x2f\xe9\xf7"'`. We injected this payload into the file `'%499$x %523$x'`, we ran `vulnFileCopy2` with, before we paused the program.

After we resumed the program execution, it took the entire payload from `'%499$x %523$x'` as input and executed our exploit. We then brought back the paused program to the foreground with `fg` command and resumed the execution. We inferred that our exploit worked since we were able to access the shell as shown in screenshot 14. The stack frame diagram after the execution of exploit was shown in figure 1.



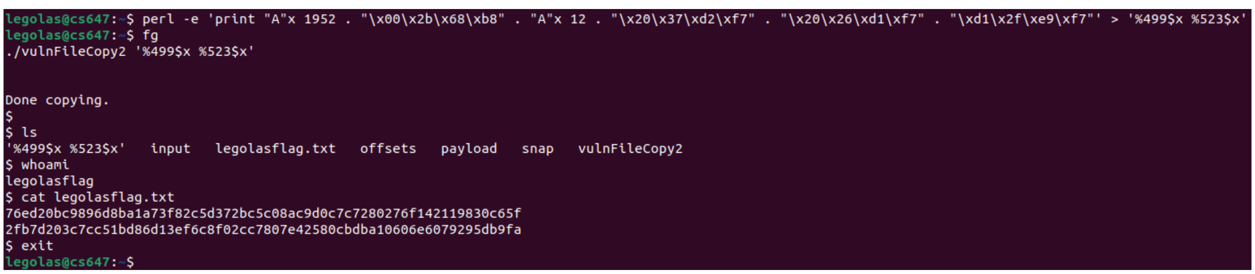
Screenshot 14: Working of the exploit.

Address	Memory
0xfffffffc	Top of memory
…	…
…	…
0xbfff????	Command string address
0xbfff????	System return address
0xbfff????	System address
0xbfff????	Previous ebp
0xbfff????	Byte alignment
0xbfff????	Byte alignment
0xbfff????	Canary
0xbfff????	Buffer(size 1952)
0xbfff????	…
0xbfff????	…
0xbfff????	…
0xbfff????	Buffer
…	…
0x00000000	Bottom of memory

Figure 1: Stack frame diagram after the exploit.

Findings:

Once we executed our exploit, we were able to obtain the following information and the program exited without any error as shown in screenshot 15.



Screenshot 15: Shell access and graceful exit of the program.

Contents of legolasflag.txt as text:

76ed20bc9896d8ba1a73f82c5d372bc5c08ac9d0c7c7280276f142119830c65f
2fb7d203c7cc51bd86d13ef6c8f02cc7807e42580cbdba10606e6079295db9fa