Assignment 1 Documentation

Written by Chuck Zumbaugh

Collaborators: None

Running the exploits

There are two exploits: 1) `redhat8_exploit.sh` that attacks the RedHat 8 machine and 2) `redhat9_exploit.sh` that attacks the RedHat 9 machine. Both exploits generate a reverse shell for a listener on port 2222 at 192.168.219.134, and attack the machine at port 8888. The IP addresses for the victim machines are shown in the table below. To run the exploit, ensure the attacking machine is listening on the appropriate port:

```
nc -l -p 2222
```

Then, the exploit can be run by executing the script: `./[machine]_exploit.sh`. At this point, the listener should have received a shell to the victim machine.

Machine	IP Address
RedHat 8	192.168.219.131
RedHat 9	192.168.219.133

Objective and description of exploits

The objective of these exploits is to execute a buffer overflow attack on machines running the nweb server and obtain a shell to the machine. While both the RedHat 8 and RedHat 9 exploits are similar in principal, RedHat 9 randomizes the stack addresses so the exploit must be modified so the EIP is overwritten with a known address. This is discussed in greater detail later in this document.

Determining the length of the buffer

The length of the buffer was determined by sending a string of sufficient length to the server to crash it and gather debug data. Metasploit was used to generate a string of 1,500 bytes using the `msf-pattern_create -l 1500` command. This pattern was then sent to the server, which resulted in a segmentation fault. The core file was then examined to determine the value of the EIP at the crash, and `msf-pattern_offset -l 1500 -q <EIP value>` was used to determine the offset. Analysis of the EIP in the core file revealed an offset of 1,033 bytes. Thus, if we send a string of length 1,037 bytes we can overwrite the EIP with the last 4 bytes.

Generating the payload for RedHat 8

Metasploit was used to generate shellcode for a reverse shell using `payload/linux/x86/shell_reverse_tcp` with the `x86/alpha_mixed` en-

coder, `LHOST=192.168.219.134`, and `LPORT=8888`. The resulting code was 198 bytes in length. Thus, offset - len(SHC) = 835 bytes. Since allowing the shellcode to reside just above the EIP can result in possible issues, an additional 100 byte buffer is allowed between the shellcode and the EIP. To accomplish this, a NOP sled of length 735 bytes is used. As 100 is divisible by 4, this should maintain the alignment needed when overwriting the EIP. To ensure success, and address roughly in the middle of the NOP sled was selected to be the return address. For simplicity, the return address is replicated a sufficient number of times to cover the address space between the end of the shellcode and the EIP. Thus, the stack is set up in a manner similar to the below figure.

Lower memory addresses	
...	NOP sled
`0xbffff720`	NOP Sled
...	NOP Sled
...	Shellcode (198 bytes)
...	`\x20\xf7\xff\xbf`
`0xbffffafc` (EIP)	`\x20\xf7\xff\xbf` (Overwrite EIP)
...	`\x20\xf7\xff\xbf` (remaining replications)
Higher memory addresses	

Generating the payload for RedHat 9

Finding the address of `jmp esp` A search was performed on the libc shared object for the instruction `\xFF\xE4` (`jmp esp`) and `\xFF\xD4` (`call esp`). Both of these will, when executed, jump to the esp (top of the stack) and continue execution. The library can be searched by executing `objdump -d /path/to/libc/so | grep "ff e4"` and `objdump -d /path/to/libc/so | grep "ff d4"` where `/path/to/libc/so` can be found by `ldd prog_name` with `prog_name` a program using libc. In this case, there were several matches to `call esp`. For simplicity, the address was found using the `./searchJmpCode` program. However, this effectively performs the above search and adds the additional offset needed to jump to the start of `FF D4`.

```
# Example output of objdump -d /path/to/libc/so | grep "ff d4"
4202f646:       83 bd a8 fe ff ff d4    cmpl $0xffffffd4, 0xfffffea8(%ebp)
...
```

Generating the payload Similar shellcode was generated using Metasploit with the same `LHOST` and `LPORT`. While the offset is the same as that of RedHat 8, we cannot rely on jumping to a memory address on the stack due to stack address randomization. Thus, instead of overwriting the EIP with a stack address we use the address of a `jmp esp` instruction in libc (see above). Since only the stack addresses are randomized, we can rely on addresses in libc to remain static. To

exploit this, we place the shellcode below (higher address) the EIP, so when the EIP is popped the ESP points to the top of the shell code. In this case, control will transfer to the address of the `jmp esp` instruction, which jumps to the start of the NOP sled and begins execution of the shellcode. The stack is manipulated to look like the below figure.

Lower memory addresses	
Random stack address	Padding (NOPs)
EIP	`\xa7\x2b\x12\x42` (`jmp esp` address)
EIP + 4	Start of NOP sled
...	NOP sled
...	Shellcode
Higher memory addresses	