EECS 765 Project 1

Author: Jace Kline 2881618

Introduction

The purpose of this programming assignment is to practice the development and execution of a remote buffer overflow attack on a vulnerable webserver deployed on RedHat 8 Linux. We also develop an additional attack on RedHat 9 Linux running the same webserver as an exercise in bypassing stack address randomization.

Environment Setup

The first step in developing an exploit is properly configuring an environment that accurately portrays the environment used in a live attack scenario. We first set up an environment following the constraints given in the programming assignment description. We create a NAT subnet 192.168.180.0 in VMWare and assign the static IP addresses of 192.168.180.10, 192.168.180.40, and 192.168.180.50 to the Kali, RedHat 8, and RedHat 9 VMs, respectively. Next, we ensure that each machine can reach all the others via netcat. We also start the nweb servers on the RedHat 8 and RedHat 9 machines by navigating to the /root directory and running `./nweb 8888 .`, and then we ensure that the webservers are accessible from the Kali VM. For efficiency of attack development, we also set up SSH for each of these machines.

Running the Exploit

The exploits work in both the RedHat 8 and RedHat 9 attack scenarios. The IP addresses used in the exploits were identical to those specified in the PA1 description (shown in the environment setup above). We use port 8228 on the Kali VM to listen for the reverse shell TCP connection. The steps for running each of the exploits are below. We assume that for each attack, the environment is properly configured and the victim VM is running the nweb webserver on port 8888. We assume that the working directory on the Kali VM is at the root of the unzipped project code folder.

First, navigate to the target exploit folder in the current shell. For the RedHat 8 exploit, run `cd redhat8`. For the RedHat 9 exploit, run `cd redhat9`. Each of these directories has the same structure and includes `exploit.sh` and `generate_shellcode.py` files specific to each of these attacks.

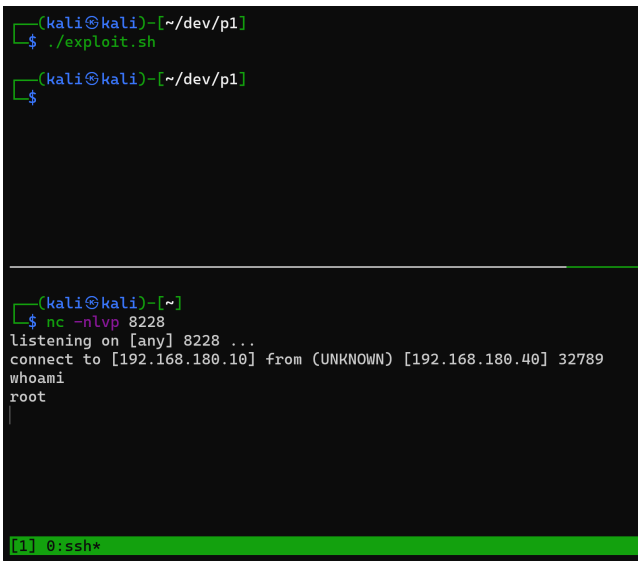
Next, open another shell on the Kali machine. In this shell, we want to set up a listener to receive the callback TCP connection from the reverse shell that will be spawned on the victim during the attack. We use netcat to achieve this, and we specify port 8228 to listen on.

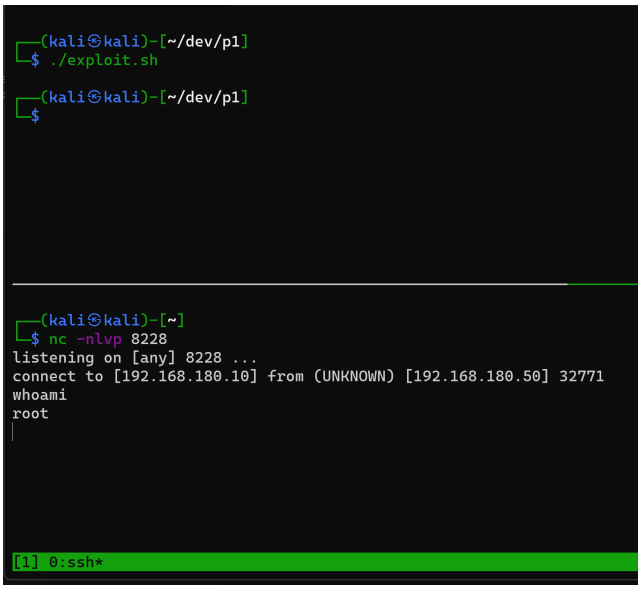
```
nc -nlvp 8228
```

Next, we return back to the other shell to run the exploit. Run the `exploit.sh` shell script to generate and send the malicious input to the webserver on the victim machine. The contents of `exploit.sh` are shown in the 'Developing the Exploit' section below. Specifically, we run the following command:

```
./exploit.sh
```

After sending the malicious input above, return attention to the listener shell. After a slight delay, there should be a message that indicates a connection initiated from the victim machine. The RedHat 8 exploit should result in a TCP connection from IP address 192.168.180.40. The RedHat 9 exploit should result in a TCP connection from IP address 192.168.180.50. In this listener shell, run some test commands such as `whoami` and `ls` to verify that the reverse shell (as root user) has been established successfully. Below, we show the results from successful exploits targeted at both the RedHat 8 and RedHat 9 victim machines.





Developing the Exploit

In this section, we talk about the general thought process used when developing the exploits. We first outline the exploit structures and then specify the specific parameters prevalent to each attack.

Malicious Input Structure

RedHat 8

Since RedHat 8 does not use stack randomization, we can place our shellcode directly in the overflowed buffer since the buffer's start address remains consistent across runs. This consistency of the vulnerable buffer's start address allows us to compute this, and subsequently compute the offset from the buffer start to the saved EIP location. Using these parameters, we construct the malicious input by filling roughly the first third of the buffer with NOP sled bytes (0x90), then appending the shellcode generated by Metasploit, and then filling the remaining bytes of the offset (overwriting the saved EIP) with a return address that exists within the NOP sled. In addition, we ensure that the first byte of the shellcode and the modified return address used to overwrite the saved EIP both align with a word boundary (1 word = 4 bytes).

RedHat 9

The stack randomization used in RedHat 9 requires us to take a different approach to executing our shellcode. Although the start of the buffer in this attack is unknown between runs, we know the offset remains consistent (same as RedHat 8 exploit). The ultimate goal is to place the shellcode somewhere after the $ESP with a NOP sled starting at $ESP that will ultimately lead to execution of the subsequent shellcode. We find a sequence of 2 bytes in shared (static) memory that contains the bytes corresponding the instruction `JMP $ESP`, which is 0xffe4, and then find the corresponding memory address that contains this sequence. We use this address to overwrite the saved EIP in our exploit. Knowing that the nweb binary uses the libc library via the `ldd nweb` command, we use the `./searchJmpCode` program provided by Dr. Bardas to list locations that contain the bytes '0xffe4'. After obtaining a valid address, we construct the malicious input as the of repetition of arbitrary bytes to overflow the buffer + the overwriting of the saved EIP with the address containing '0xffe4' + a NOP sled (starts at $ESP) + the generated shellcode (same as RedHat 8 exploit).

Malicious Input Parameters

For the RedHat 8 exploit, we must find the endianness of the architecture, the word size, the address of the start of the buffer, and the offset from the start of the buffer to the saved EIP location.

First, we find via a quick search that the endianness of the x86 architecture is little endian and the word size is 4 bytes.

To compute the address of the start of the buffer, we simply use an input of 1500 'A' (0x41) bytes such that a segmentation fault occurs and the core is dumped. Using the `gdb` tool, we analyze the memory to find the address of the first occurence of the 0x41 byte. We repeat this using 'B' bytes (0x42) to cross check our results. We find that the start address of the buffer is 0xbffff6a4.

To compute the offset from the start of the buffer to the saved EIP location, we first use the Metasploit's `pattern_create.rb` Ruby script to generate and save input that we use to overwrite the saved EIP value in the buffer on the webserver. We then send the malicious input to the webserver, which causes a segmentation fault and core dump. Using `gdb` on the core file, we record the contents of the value that produced the segmentation fault, which is in fact the value that was in the saved EIP. With this value, we use Metasploit's `pattern_offset.rb` tool to compute the offset (in bytes) from the start of the buffer to the location of the saved EIP. This offset is found to be 1032 bytes.

For the RedHat 9 exploit, we use the endianness of the architecture, the word size, the offset from the start of the buffer to the saved EIP location, and the distance from the saved EIP location to the $ESP pointer address.

Since RedHat 9 also uses the x86 architecture, we conclude that little endian is also used on this machine in addition to a 4 byte word size.

We repeat the same process as above to compute the offset of the location of the saved EIP from the start of the buffer. Once again, we find that the offset is 1032 bytes.

Lastly, to find the distance from the location of the saved EIP to the value of $ESP, we induce and core dump on the target machine and use `gdb` to show the address of the $ESP pointer. We then traverse backwards in memory until we find the start of the long chain of 'A' (0x41) bytes we use for the malicious input and we record this address. We repeat this again with 'B' bytes to ensure we have a consistent value. We then conclude that the `distance between the address of the saved EIP and $ESP = ($ESP - start address of buffer) - offset` where the offset was computed above. We find that $ESP directly follows (is 4 bytes after) the address of the saved EIP.

Generating the Malicious Input

To generate the malicious input byte string that is sent to the victim webservers, we first use the Metasploit Framework's `msfconsole` tool to generate the shellcode portion of the input. Per the assignment's criteria, we use the linux/x86/shell_reverse_tcp payload type and the x86/alpha_mixed encoder. We also specify the LHOST and LPORT options for the shellcode generation as 192.168.180.10 (the Kali VM) and 8228 (listener port), respectively. We output this shellcode in the Python programming language. After generating the base shellcode, we fit the shellcode to the specific attack by adding to the generated Python. The shellcode generation Python scripts for each of the exploits can be found in the `generate_shellcode.py` file in the respective directories.

For each attack, we codify the structure outlined in the 'Malicious Input Structure' section. We attempt to parametrize as much of the construction as possible for easier reuse in the future. We create a function called `mk_shellcode()` for each the RedHat 8 and RedHat 9 exploits that essentially uses the parameters defined in the previous section in conjunction with the generated shellcode bytes to construct a byte string to be used as malicious input. The size and locations of the NOP sleds and overwritten EIP values in the exploited buffer are completely dependent on the provided offset and size of the shellcode, offering flexibility for reuse in future attacks. We note that any instances of overwriting the EIP with a desired address requires us to reverse the bytes of the desired address in the input in order to fit with the little endian architecture on the target machine.

To simplify the attack, we provide a `exploit.sh` shell script in each of the exploit directories. This script acts to create an environment variable to store the generated shellcode and then perform the HTTP GET request to the remote server, ultimately overloading the vulnerable buffer and spawning the attack. The RedHat 8 version is seen below.

```
#!/bin/sh

EGG=`python3 generate_shellcode.py`
echo GET /$EGG HTTP/1.0 | nc 192.168.180.40 8888
```

References and Collaborations

References

- How to Create Reverse Shells with Netcat in Kali Linux
- Examining Memory in GDB

Collaborations

I collaborated with Anjali Pare to set up the environment and VMs. She also offered general high-level tips for constructing the exploit.