**Jigyas Sharma**

**Dr. Alexandru Bardas**

**EECS 765**

**14 September 2023**

**Programming Assignment 1: Report**

**Introduction**

The aim of this programming assignment is to create an exploit that will take advantage of the buffer overflow in a program. The targeted operating system is RedHat8 and RedHat9(tentative) and the attack is going to be launched remotely. The exploit is going to target the buffer overflow segmentation faults in nweb. The exploit is going to be designed in a way that the EIP is going to be filled with an address to point to a location in memory which is going to contain a shellcode, and assuming that the nweb server is running with root privileges, the access to the shell provided by the shell code would also have root privileges.

**Running the Exploit**

**Developing the Exploit**

The initial step to creating the exploit is to check for vulnerabilities. The aim of the attack is to obtain a root shell by calling back from the nweb server. The first part of the process was to recreate the environment in which we would be attacking. This includes starting an nweb server on a Redhat8 machine with some defined parameters(ip address and port). The first step was to check for a buffer overflow vulnerability. To check for the buffer overflow vulnerability, use the metasploit frameworks “pattern\_create” and “pattern\_offset” scripts. The length for pattern create was 2000 and the offset received was 1032. However, upon testing the EIP overflows at 1030 instead of 1032. The next part is to develop a shell script that will return a shell to the attackers ip address who will be listening on some port. To create the shell script, use the metasploit framework’s, msfconsole which will create a shell script with LHOST and LPORT, the shell code can also be specified for multiple languages, we used python. Now with the above information we try to create the structure of the malicious input which we will send to the nweb server.

Now, essentially when I was testing my work, I was running a shell script inside my python program which would create core dumps, however, after talking to the instructors, I think it is essential to understand that the first part of development is to see if your payload works or not, in our case would return a root shell. So, after about a week of using a script, I switched to a process of having my python program print which is then filled into an environment variable “EGG”(Shawn likes eggs) and then using the command “echo GET /$EGG HTTP/1.0 | nc 192.168.32.40 8888” to send the malicious input. (Note: after I switched to this method, I had my exploit working within one evening).

**Structure of the Input**

**ESSENTIALLY:**

overflow\_string = b"\x90"\*785+buf+b"\x90"\*30+b"\xc1\xf7\xff\xbf"

send\_string = b'helloworld' + overflow\_string

In the overflow string, first 785 bytes are NOP sleds followed by buf which is your shell code and then some more NOP sleds and then lastly the redirection address but backwards. Then because you cannot start your string with a NOP sled(it corrupted my input), you create another string, send\_string” this string is then put in the environment variable which is then sent over using the GET instruction.

We look at the NOPs as padding, however, the starting string in send\_string is also considered padding so that it doesn’t corrupt your sent data. The NOPs are basically instructions that keep passing the baton to the next address.

**Determining the Parameters used in the Malicious Input**

Firstly, to figure out the buffer length we just keep feeding the program more and more data or you could create an absurdly large pattern using “pattern\_create” in the metasploit framework. I used a pattern of size 3000 and sent it to the nweb server through ncat. This created a core dump on our recreation of the environment which told us that there was a segmentation fault with signal 11. Using the pattern\_offset, we figured out that the saved EIP starts at a distance of 1032. Now, to figure out the architecture, I filled a string with 1032 “A”s and the last 4 bytes with “BCDE”. Upon inspecting the core dump file, the saved EIP was overflowed with “0x45444342”. This step helps us understand that we are working with a Little Endian Architecture and that we have control over what can be written over the EIP. The next part was to figure out what can be a suitable address to overwrite our EIP to and fill those addresses with our shell code. To figure out a suitable memory location we use the command “x/64 $esp-0x\_\_” and then try to work our way upwards to at least 205 bytes, cause that’s how long the shell code is. I worked my way up 0x33F where the memory was filled with As. The memory address I determined to redirect the flow of the code was “0xbffff7c1”.

**Generating Malicious Input**

The language I used to generate and test my exploit was python for a week which frustrated me and then I switched to C(attached the half developed exploit) for a brief evening before going back to my work in python.

After figuring out the memory address, where it would be suitable and could fit the entirety of the exploit, you try to reach that address by trying to fill it with characters you can recognize. The input that I sent over after knowing the above information was to create a pattern where I send 827 As, these would represent the NOP sleds, then 205 Fs which would represent the shell code and then 4 Bs, which would be the address to where the redirection of the program would go. Now, in practice those parameters should have worked however, during testing I figured out that it is easier to create a padding before the shell program and after the shell program. Now this process took sometime as when you filled up with the stack with characters, the behaviour wouldn’t change. However, when you replace characters with shell code and NOP sleds, you understand if your code runs or not. Essentially, there should be a better way to figure out how to create a perfect padding however, I did it using a trial and error method of wrapping my shell code with a padding of NOP sleds. The formula that essentially worked for me was putting close to enough padding of NOP sleds in the front that it reaches the redirection address, then putting the shell code and then putting padding of NOP sleds till you reach the start of the EIP at which point you put the address in reverse order(cause Little Endian). After you figure out your padding, it took me close to a weekend to realize that when you send your exploit, if a program just receives NOP sleds, it will corrupt your input, or at least that is what happened in my case. The structure for my exploit which is overflow\_string = NOP + shell code + NOP+Redirection address. However, when sending the exploit to the server, I created another string, called “send\_string”, this string contained some non-trivial string, in my case helloworld. Since, we added helloworld to the start of the send\_string you have to subtract that many bytes from your padding. Try to remove it from the padding after the shell code so that you do not have to figure out a new starting NOP padding for your exploit. Now, after similar to the way you were testing while understanding the padding needed, you test to see if your exploit works.

**Collaborations and References**

[**https://www.fortinet.com/resources/cyberglossary/buffer-overflow**](https://www.fortinet.com/resources/cyberglossary/buffer-overflow)

[**http://users.umiacs.umd.edu/~tdumitra/courses/ENEE657/Fall17/homeworks/enee657\_buffer\_overflow.pdf**](http://users.umiacs.umd.edu/~tdumitra/courses/ENEE657/Fall17/homeworks/enee657_buffer_overflow.pdf)

[**https://web.ecs.syr.edu/~wedu/seed/Book/book\_sample\_buffer.pdf**](https://web.ecs.syr.edu/~wedu/seed/Book/book_sample_buffer.pdf)

[**https://forums.kali.org/archive/index.php/t-28940.html**](https://forums.kali.org/archive/index.php/t-28940.html)

[**https://security.stackexchange.com/questions/157478/why-jmp-esp-instead-of-directly-jumping-into-the-stack**](https://security.stackexchange.com/questions/157478/why-jmp-esp-instead-of-directly-jumping-into-the-stack)

**Ina – Helped me understand that the NOP sleds are being corrupted and suggested that I add some string before the exploit string.**