Cristian Avina Lopez

Exercise 8 Report

4. Problem 1: Launching a Shell

* The shellcode1.c script’s function is to execute a shell, after compilation and running the executable the command line just prints out $ separator, and allowed me very limited access to the terminal. The shell enters interactive mode, which reads and writes to a user’s terminal.
* Step 2 of this process involves injecting compiled C code into the running program, in other words, compiled machine code. A script is created to make a buffer in memory containing the contents of shellcode1.c in translated assembly memory addresses.

The script first creates a buffer array of chars of size of the array shellcode. The shellcode array contains the machine code we are injecting. Shellcode gets copied into the buffer

array using the strcpy() function and gets casted in a way that it will allow us to call it.

I compiled shellcode2.c inside shellcode1 and executed shellcode2, and the result was

another shell appearing on the terminal, a shell within a shell, further showing that it is

possible to place shellcode on the running stack and execute.

5. Problem 2: A Vulnerable Program

* The created stack.c script contains a buffer overflow vulnerability in function bof().

The program reads in 517 bytes from badfile and passes the information to the bof

function as an array str. This is where the issue arises, the buffer that is being used to

store all these elements is only allocated to 24 bytes, on top of that, strcpy() does not

restrict or check for boundaries in the buffer, thus leading to overflow. This shows that

any user controlled vulnerable file can be passed in and cause an overflow of the buffer.

Not only is there an overflow vulnerability in this case, but since there are certain privileges given to the user in this instance, this can also lead to potential root shell access.

6. Problem 3: The Exploit

* The purpose of exploit.c is to exploit the vulnerability setup in problem 2. The program is set up similarly to the vulnerable program. The programs task is to successfully create a bad file and insert NOP instructions, shellcode and target addresses. The program starts by initializing a buffer array of size 517, an allocated pointer to a badfile, two longs for address and pointer\* storage, and an integer that will hold the offset. The first function of the script starts by filling the buffer with NOP instructions, the purpose of this is for memory alignment, memory hazard preventions, and for a better chance of successful memory address targets. memset() is the C function used to fill in 517 values of NOP instruction (0x90) into the buffer array. The memset(void\* str, int c, size\_t n) function works by copying the character c to the first n amount of characters of the string it is pointing to (str). Next, strcpy(buffer + 400, shellcode) is used to copy the memory addresses contained in shell code near the end of the buffer. Notice the +400 being added to the pointer, since the pointer by default is at position 0 of the array, it’s setting the destination inside the parameter to 400, meaning it will start filling in the shellcode memory addresses from position 400 of the array. The new return address is stored in the long addr variable, and the offset is stored inside the the offset integer variable. The offset is the number of bytes offset from the start of the buffer where the the return address is stored. The ptr will be the new return address on the stack, buffer + offset will give us the new location on the stack.

7. Problem 4: Using the Debugger

* In order to find the appropriate values for addr and offset, a program called debugger will be used to check each statement in the source code and check things such as values and addresses of variables. For this task an open source tool called gdb will be used to check through the program. The tool cannot just parse through the C code itself, so gdb stores variable and functions that are usually thrown out during the compiling process by reading through translated debugging symbols instead of C syntax. Using the gdb function next, I skip each line I want to ignore until we get to bof(str), this line of code is the statement that will cause the overflow in the buffer array, after execution this will write 517 characters inside the 24 character buffer array. The goal is to track the new return address after the buffer. To do this I take the address given by the (gdb) step function which gives the location address of the buffer array, which is the address I use to find the similar return address. After I printed the stack and found the similar return address, I calculated my offset by counting the addresses from the beginning to the new address and multiplying that number n by 4 bytes. The return address needs to land in one of the NOP addresses in between the buffer array and the shell code, so for this I did trial and error to figure out which NOP the return address lands on. In conclusion, my new address came out to 0x080484ff, my offset to 36 (9 locations x 4bytes), and my NOP: 0xbffff178.