Assignment 7
Assigned: 9/1/2024
Due: 19/1/2024

Buffer overflow exploitation

“Smashing the stack for fun and profit.”

In this assignment you are going to exploit a buffer overflow vulnerability in a very simple but badly written program. This assignment assumes background knowledge of Linux process memory layout, Stack ABI and familiarity with x86 architecture and GDB.

The vulnerable program of this assignment is Greeter. It simply asks for your name and kindly greets you. Initially, it asks the name of the user and calls the ***readString*** function, readString uses ***gets*** function in order to place the name of the user in a local buffer (placed in the stack). Then, the local buffer is copied in a global buffer and the readString function returns. Finally, the program calls ***printf*** function with the global buffer as an argument in order to print “Hello <user>, have a nice day.”.

However, the developer of the greeter program did not take into account that ***gets*** function does not check if the size of the input string is larger than the size of the buffer. Thus, if a large string is provided, ***gets*** will write past the local buffer and overwrite adjacent memory areas. This will likely result in a Segmentation fault.

Given the above situation, a sophisticated user with malicious intent will be able to provide specially crafted input that will overwrite the return address of the ***readString*** function and divert the execution, anywhere in the Greeter program. The malicious user, can also introduce new functionality in the greeter program by providing machine instructions as input while overwriting the return address to point at the address the provided input is stored by the program. This type of attack is called Arbitrary Code Execution. For more information visit: https://en.wikipedia.org/wiki/Arbitrary_code_execution

For this assignment you have to force the greeter program to spawn a terminal shell. In order
to accomplish this, you have to provide a specially crafted input that will make the greeter program execute arbitrary code.

Proposed steps for the assignment

1. Finding the buffer and the return address location.

For this step you can use GDB (https://www.cprogramming.com/gdbtutorial.html). Run the greeter program using the debugger, the size of the buffer is 32 bytes. What happens when you provide a long string of A characters? (Hint: look at the segmentation fault address, what is the ascii hex number of character 'A'). For simplicity, the provided binary is compiled with debug symbols, you can easily find the address of buffers using GDB.

2. Payload generation and test.

Your goal for this assignment is to make the Greeter program spawn a terminal shell. A simple C code snippet that would return such a shell is:

```
char *args[2];
args[0] = "/bin/bash";
args[1] = NULL;
execve(args[0], args, NULL);
```

You can generate the shellcode by compiling the above example in order to obtain the machine code. You can find numerous examples of shellcode online (binary form), it is not required to generate your own from source. Find a way to test the shellcode before trying to exploit the greeter program. Create a simple test program to try this (Hint: Look at ***"-z execstack"*** compilation flag in gcc, how can you make a memory page executable in Linux?, how can you execute machine code using C programming language). A successful run of the test program will spawn a terminal shell.

3. Input file generation.

It is not advisable to type characters in hex form as input. Thus, you must create a simple script (preferably in python), that will create the exploit (i.e. input file that results in arbitrary code execution). The input shall contain the payload (i.e. machine code for shell spawning) and the address of the buffer the input will be stored. The address must be accurately placed in the input file in order to overwrite the return address when the stack buffer overflows. You may need some padding between the shellcode and the address (Hint: check what "0x90" is in x86 architecture). Do not forget that the input file must contain bytes, not characters.

4. Testing the exploit.

Run the greeter program with the file containing the exploit as input. When the shell is spawned run some commands (e.g. ls, whoami, etc.). By redirecting the input however you will not be able to enter any commands in the spawned shell. A very nice solution is described in:
https://reverseengineering.stackexchange.com/questions/13928/managing-inputs-for-payload-injection