Assignment 8
Assigned: 20/12/2024
Due: 20/01/2025

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, grades your assignment 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, readString copies the local buffer is copied in a global buffer, calls ***printf*** function with the global buffer as an argument in order to print "Hello <user>, your grade is <Grade> . Have a nice day." and the readString function returns. Finally, the program exits.

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 local variables and/or 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:

1. Grade you with more than 6. Also investigate, why you cannot make Greeter grade you with '10'.
2. 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, Grade 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 grade value after the printf and segmentation fault address, what is the ascii hex number of character 'A'). For simplicity, the provided binary is compiled with debug symbols and you can easily find the address of buffers, variables using GDB.

2. Payload generation and test.

Your second 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; \x90\x90\x90\x90
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

Bypassing Data Execution Prevention

After the catastrophic security flaws in the Greeter program, the developers decided to make their program a bit more secure. One of them proposed to leverage Data Execution Prevention (DEP) in order to prevent attackers from executing the injected payload. The new version of SecGreeter is dynamically linked (they realized that they can ship a smaller binary) and also compiled without the “-z execstack” flag. Little did they know that their efforts are not adequate and their application can still be easily exploited. For the bonus of this assignment, you have to exploit **SecGreeter** with a return-to-libc (https://en.wikipedia.org/wiki/Return-to-libc_attack) attack. In this exploitation technique, you do not have to inject the payload, but rather use existing code in the application.

1. Primitives and setting up the smashed stack.

To accomplish this you need to locate the following primitives in the loaded application. First, you need to locate a function with “**execve**” functionality. In **libc** this function is “**system**” (https://man7.org/linux/man-pages/man3/system.3.html). Then you need to locate the “**/bin/sh**” string in the loaded application in order to set it as a function argument when setting up the stack for “**system**” invocation. Finally, you also need to find an appropriate function for **SecGreeter** (the parent process of the spawned shell) to *exit* peacefully after executing “**system**” (i.e. set up a return address for “**system”**).

2. Making things a bit easier.

Modern systems usually leverage Address Space Layout Randomization (ASLR) (https://en.wikipedia.org/wiki/Address_space_layout_randomization). This mechanism randomizes the addresses of loaded objects in order to make it harder for attackers to locate the primitives discussed in the previous paragraph. To make things easier in this assignment, you can switch off this feature by following the steps discussed here (https://askubuntu.com/questions/318315/how-can-i-temporarily-disable-aslr-address-space-layout-randomization). Make sure to enable back ASLR after finishing the assignment, it will be also re-enabled automatically after rebooting the system.

Notes

1. Preferably use the precompiled version of the Greeter program. It is compiled statically without position independence, thus, your generated exploit input is likely to work in any machine you test it. Moreover, the addresses of the buffers will not change each time you run the program. The binary is compiled with "-m32" (32-bit addressing mode) in order to be easier to reverse engineer Greeter (find addresses of buffers, etc.). For the bonus please remember to disable ASLR and remember that it is compiled dynamically, thus addresses might change a bit on different machines.
2. A lot of compiler features are disabled in order to make Greeter as vulnerable as possible, check the compilation flags in the Makefile.
3. You need to write a generator program that will generate the input that exploits the Greeter program (preferably written in Python).
4. There are numerous online tutorials with techniques and tricks that will help you develop a successful exploit. This point is very important especially for the bonus.
5. You need to create a README with your name, your AM and a description of how you developed the exploit (or exploits) of this assignment.
6. You must submit the following files: README, generator script, shellcode test program, exploit input file, separate generator for bonus (if implemented), separate exploit input file for bonus (if implemented).
7. You should place all these files in a folder named <AM>_assign8 and then compress it as a .zip file. For example, if your login is 2020123456 the folder should be named 2020123456_assign8 you should commit 2020123456_assign8.zip.
8. Use the tab "Συζήτηση" in courses for questions.