System Security, Spring 2010

Exercise 5

Distribution: 23.04.2010
Hand in: 29.04.2010

Introduction

In this assignment you will build exploits for the binary program `vulnapp`. There are two buffer overflow attacks to perform on the same binary program (`vulnapp_ex4.tar`). You have already used `vulnapp` in the previous assignment. All binaries are compiled to run on Linux machines. For this exercise, you **must** use a linux system which does not have stack protection mechanisms enabled by default, e.g. Ubuntu 8.10 (or earlier)[1]. As in the last exercise, you can use a Linux live CD (e.g. Ubuntu) to solve this exercise if you do not have a suitable operating system installed.

The type of buffer overflow attacks that we propose to do are generally known as return-to-libc attacks. In such an attack, the main idea is to overwrite the return address with the address of some function in the libc library (any C program dynamically links to this library upon execution), pass the correct parameters to that function and make it execute. The advantage of this attack is that it can be successfully executed on operating systems with non-executable stack.

Executing the program

The `vulnapp` execution looks like the following:

```
./vulnapp SYSSEC
Type some text:
FOO

You typed: [FOO]
You provided: [SYSSEC]
```

As you may expect, operating systems already provide a number of measures to prevent such an attack from being 100% successful. One such measure is library address randomization. The method randomizes the stack addresses that are used for dynamically linkable libraries such as libc. This brings an additional protection given that the attacker must predict the function address on the stack. In order to make this exercise easier for you, we suggest to disable the virtual address randomization on Ubuntu. This will ensure a stable virtual address space across multiple executions. Remember that doing so makes your system

[1]The latest version of Ubuntu, Ubuntu 9.10 cannot be used due to default NX emulation.

more vulnerable to outsider attacks, so you should turn on the randomization at the end of the exercise. Turning on/off the randomization is as follows (off=0, on=1):

```
sudo echo 0 > /proc/sys/kernel/randomize_va_space

sudo echo 1 >/proc/sys/kernel/randomize_va_space
```

Building & executing exploits

Your exploit user input will typically contain byte values that do not correspond to the ASCII values for printing characters. We recommend using PERL to supply your exploits on the command line argument. Here is an example on how call vulnapp with the ASCII characters "AAA" followed by 0xf05effbf as argument:
bash> ./vulnapp `perl -e 'printf "A" x 3 . "\xf0 \x5e \xff\xbf"'`

This way, you can use the PRINTF function to fill the buffer with arbitrary characters and then add your actual exploit input that contain specific addresses in hexadecimal format.

Attack 1: Execute your favorite shell

We have collected hints for both exercises at the end of this document!

Your task is to get VULNAPP to execute a favorite shell of yours (e.g., /bin/bash, /bin/sh, /bin/tcsh).

To perform the attack you have to supply an exploit string that overwrites the stored return pointer in the stack frame for `cpybuf` with the address of the system() function from the libc library and provide the necessary argument. This function allows you to execute any program (e.g., system("/bin/sh")). To provide arguments to the system function, you have to prepare the stack to look like before any function call — on your Linux box, this means that the arguments should follow directly the address of the function. As the system call expects a pointer to a string as argument, you will have to find a string, which matches the path of the file you want to execute. Luckily for you, the vulnapp application allows you to place a string of your choice on the stack as follows.

```
bash> ./vulnapp Hello
bash>  Type some text:
bash>  /bin/sh
```

Note that your exploit string may also corrupt other parts of the stack state. In order to make the program terminate safely you will need to put on the stack the address of the function exit() to properly terminate the execution. Here is a list of steps you may consider following to perform the attack:

1. Find the vulnerable buffer length by using gdb and/or testing with various user inputs as a command line argument.
2. Find the addresses of system() and exit() on the stack.
3. Find the address that points to your typed text containing the shell to execute.
4. Understand how to position these addresses in your exploit.
5. Build and test the exploit.

Attack 2: Execute the shell from environmental variables

This attack is similar to the previous one with that difference that you will need to find the path to the shell i.e., "/bin/bash" in the environmental variables. These variables are automatically loaded in the program space upon execution.

For this attack, you do not need to supply any information when the program asks you to type some text as it will not be needed. In difference to the previous attack, you just need to find the path to the shell program already included in the program space after loading the vulnapp program.

Expected Deliverables and Some Advice/Help

First, we point out that this exercise is individual and we expect individual submissions. Your submission should contain the gdb commands used and their output for each step towards successfully performing the attack. It should also include a brief explanation after some major steps to summarize your findings.

Some Advice

- In GDB, you can disassemble the current function using `disassemble` or any function using `disassemble functionname`
- All the information you need to devise your exploit can be determined by debugging VULNAPP in gdb.
- Be careful about byte ordering.
- You might want to use GDB to step the program through the last few instructions of `cpybuf` to make sure it is doing the right thing.
- You will need to pad the beginning of your exploit string with the proper number of bytes to overwrite the return pointer. The values of these bytes can be arbitrary (0x00 is not recommended though — why not?).
- As a consequence of the last hint, you may run into problems if the address of `system` or `exit` contains zeros — why? In such a case, either consider calling a different function from the libc (if the address of system contains zeros), or find a different address to return to — be creative, there is at least one easy solution to this.

Additional Help

You can find additional resources on the web. Here we point out to a tutorial which performs some of the steps that you need to perform in this exercise. The tutorial can be found at: http://www.infosecwriters.com/text_resources/pdf/return-to-libc.pdf