Smashing the Stack for Fun and Credit[1]

1 Overview

In this assignment you will learn about security vulnerabilities in software. In particular, you will find and exploit memory corruption vulnerabilities in order to better understand how simple programming errors can lead to whole system compromise. You will also examine some common mitigation techniques and analyze exploit code (i.e., shellcode).

2 Getting started

The exercises in this lab will be conducted inside a virtual machine environment to promote a consistent experience for everyone in the class and facilitate the grading process. To get started, first download and install a copy of *Oracle VirtualBox* for Windows, GNU/Linux, or Mac OS X.[2] Next, download the GNU/Linux virtual machine image that we will use for this project.

3 Virtual Machine

We will be using a special virtual machine image to carry out this assignment.

The login information for the two available users is:
Username: user
Password: user

Username: root
Password: root

4 Vulnerability identification and exploitation

Your first objective is to analyze the source code of four programs written in C. Each program has at least one exploitable vulnerability. It will be your task to identify the exploitable vulnerability in each program and construct an attack against each compiled program that will hijack program execution in a meaningful way (as instructed). We encourage you to compile the vulnerable programs using the makefile included

[1]Thanks to Sam Small for creating course materials for this class, including this assignment

[2]The software and license information is available via the Oracle VirtualBox website.

[3]I recommend using `wget` or `scp` to transfer the Project 1 materials to your virtual machine. To use `wget` for example, use the command `wget`

We will use this same makefile when testing your attacks. For more information about the project makefile, see Section 6.1.

4.1 Exploitation objectives

Once you have identified an exploitable vulnerability in each program, your goal is to construct an attack that provides you with a root shell.[4] For programs generated by `vulnerable1.c`, `vulnerable2.c`, and `vulnerable3.c`, you will accomplish this by injecting shellcode into the vulnerable process. You may construct your attack using C, python, or perl. We have provided Aleph One's shellcode[5] for you to use. It is included in the file `shellcode.h` and launches a command shell when executed. Using the makefile we have provided, you may notice that `vulnerable4.c` is compiled with a stack canary and a non-executable stack. Since the stack is non-executable, you will be unable to execute shellcode from the stack. However, you must still find a way to launch a command shell using techniques we have discussed in class.

For each program that we have provided, we ask that you explicitly:

1. Briefly describe the behavior of the program.
2. Identify and describe the vulnerability as well as its implications.
3. Discuss how your program or script exploits the vulnerability and describe the structure of your attack.
4. Provide your attack as a self-contained program written in C, perl, or python.
5. Suggest a fix for the vulnerability. How might you systematically eliminate vulnerabilities of this type?

For `vulnerable4.c`, provide answers to the following questions in addition to those listed above.

1. What is the value of the stack canary? How did you determine this value?
2. Does the value change between executions? Does the value change after rebooting your virtual machine?
3. How does the stack canary contribute to the security of `vulnerable4`?

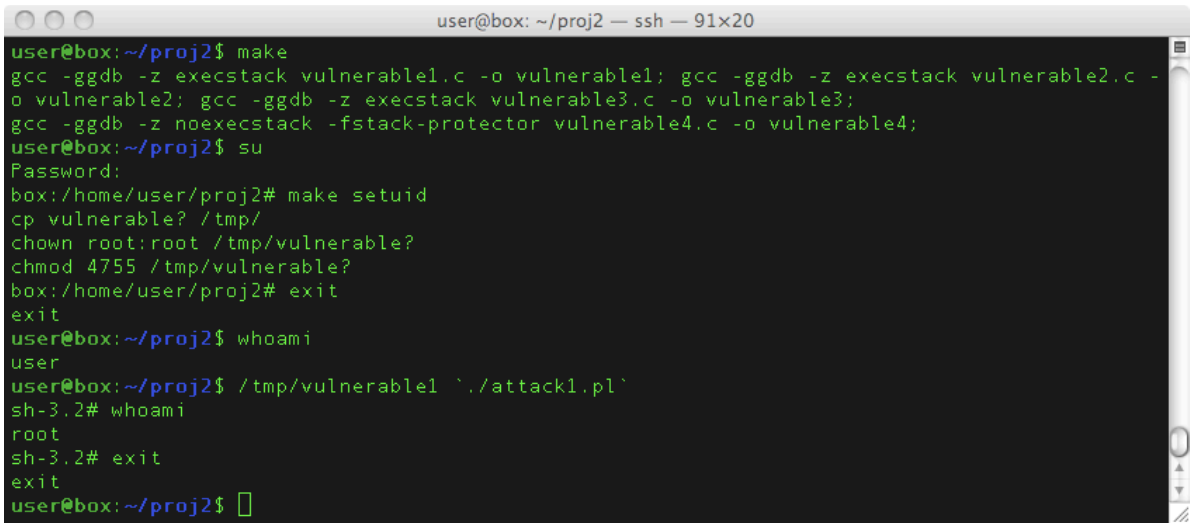
5 Shellcode analysis

The file `mystery_shellcode.h` defines a character buffer that contains shellcode. You must analyze this shellcode and comment on its design, what it does, and how it works. To do this, we suggest that you disassemble the shellcode and annotate its disassembly with your own comments. Include the code and your comments in `answers.pdf`. In completing this task, you may find that `http://syscalls.kernelgrok.com/` is a helpful resource.[6]

[4] You will run each program's setuid executable as the non-privileged user named `user`.

[5] See Aleph One's seminal article *Smashing the Stack For Fun and Profit*, which is posted on the course webpage and all over the Internet.

[6] Your class notes will be helpful too.



```
user@box:~/proj2$ make
gcc -ggdb -z execstack vulnerable1.c -o vulnerable1; gcc -ggdb -z execstack vulnerable2.c -
o vulnerable2; gcc -ggdb -z execstack vulnerable3.c -o vulnerable3;
gcc -ggdb -z noexecstack -fstack-protector vulnerable4.c -o vulnerable4;
user@box:~/proj2$ su
Password:
box:/home/user/proj2# make setuid
cp vulnerable? /tmp/
chown root:root /tmp/vulnerable?
chmod 4755 /tmp/vulnerable?
box:/home/user/proj2# exit
exit
user@box:~/proj2$ whoami
user
user@box:~/proj2$ /tmp/vulnerable1 `./attack1.pl`
sh-3.2# whoami
root
sh-3.2# exit
exit
user@box:~/proj2$
```

Figure 1: An example that demonstrates the use of the project makefile and the successful exploitation of `vulnerable1.c`.

6 Helpful information

6.1 Using the makefile

The makefile provided with the Project 1 materials is called `Makefile`. It has 3 targets of interest: *all* (the default target), *setuid*, and *clean*. The default target will compile the source code we have provided. The *setuid* target must be executed as superuser or root. It copies the compiled programs into `/tmp/` and marks the executables as setuid root. When a user launches a program marked as setuid root, it runs with the same privileges as the root user. The *clean* target removes any existing Project 1 executables from your working directory (but not from `/tmp/`).[7] For an example demonstrating the use of the project makefile, see Figure 1.

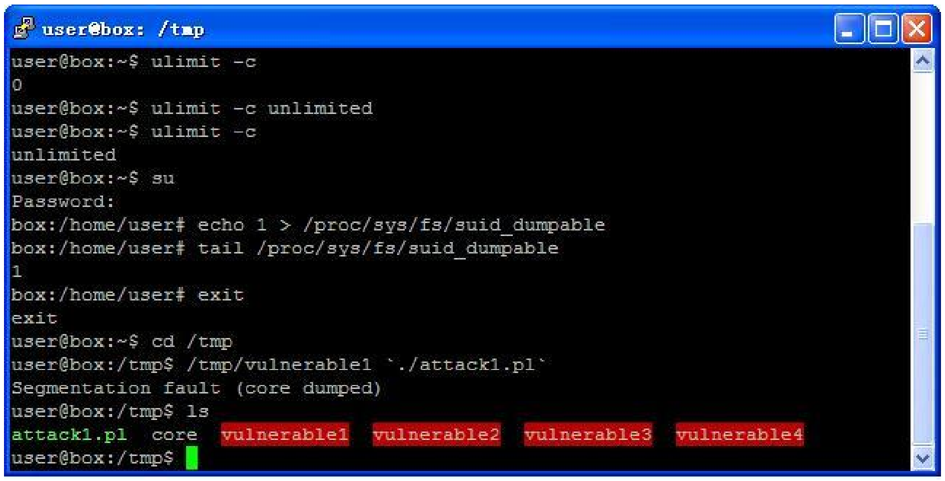
6.2 Using ssh

To successfully exploit, use an ssh client such as Putty to connect to the virtual environment and execute your code that way. Please also don't change/customize the user environment (.bash_profile), or else your exploit program may have problems when grading.

6.3 Enabling core dump files

Core dump files can be useful for analyzing the cause of an execution crash. Once enabled, the operating system will create a file named `core` in your current working directory when a process crashes (e.g., due to a segmentation fault). Used in conjunction with `gdb`, the core file can give you a complete view of the process state when it crashed. To analyze a crashed process, simply launch `gdb` with the names of the executable file and its core file as arguments.

[7]**Warning:** Files left in `/tmp/` may not survive a reboot of your virtual machine.



```
user@box:~$ ulimit -c
0
user@box:~$ ulimit -c unlimited
user@box:~$ ulimit -c
unlimited
user@box:~$ su
Password:
box:/home/user# echo 1 > /proc/sys/fs/suid_dumpable
box:/home/user# tail /proc/sys/fs/suid_dumpable
1
box:/home/user# exit
exit
user@box:~$ cd /tmp
user@box:/tmp$ /tmp/vulnerable1 `./attack1.pl`
Segmentation fault (core dumped)
user@box:/tmp$ ls
attack1.pl  core  vulnerable1  vulnerable2  vulnerable3  vulnerable4
user@box:/tmp$
```

Figure 2: If you use `ssh` to connect to your virtual machine and want to enable core dump files, you must execute `ulimit -c unlimited` and for setuid executables execute `echo 1 > /proc/sys/fs/suid dumpable` as root.

You can enable core dump files for processes that crash by using the command `u limit - c unlimited` from the command prompt. However, to enable core dump for setuid executables, you should use an additional command (`echo 1 > /proc/sys/fs/suid dumpable`). If you need assistance enabling core files, see Figure 2 or ask for help via the course discussion board.

[8]See the *Project Submission* link on Blackboard.