CSE 127: Computer Security (Spring 2018) **Due October 23, 10:00 P.M. PDT**

Assignment 2

100 pts

The goal of this assignment is to gain hands-on experience with the effects of buffer overflows and other memory-safety bugs. You will be provided a skeleton for implementing these exploits in C. Your solution is due on October 23, 10:00 P.M. PDT. You may work with *one* other person in your class section on the assignment; if so you should only submit one solution for the two of you. You and your partner must *not* discuss your solution with other students until seven days after the assignment deadline. You may consult any online references you wish. If you use any code in your answer that you or your partner did not write yourselves, you *must* document that fact. Failure to do so will be considered a violation of the academic integrity policy.

1 Getting Started

To complete this assignment you will be provided with a VirtualBox VM and a set of files including a turn-in script.

1.1 VM Image

In order to match the environment in which your submission will be graded, all work for this assignment must be done on the VirtualBox VM we have provided named `cfbox`. Do not create a VM in the Virtualbox software and then load the image from the link below. Instead, double click on the file provided below to initialize a VM with the proper configuration. You can download the VM image from:

https://drive.google.com/file/d/1IGTFd60VijSqUw3r_pOSBBSj5-qt02-7/view?usp=sharing

The VM is configured with two users: `student`, with password "`hacktheplanet`", and `root` with password "`hackallthethings`". The VM is configured with SSH on port 2222. Please note that SSH is disabled for `root`, so you can only SSH in as the student user. You can still log in as `root` using `su` or by logging into the VM directly.

To SSH into the VM:

```
ssh -p 2222 student@127.0.0.1
```

To copy files from your computer to the VM:

```
scp -P 2222 -r /path/to/files/ student@127.0.0.1:/home/student
```

To copy files from the VM to your computer:

```
scp -P 2222 student@127.0.0.1:/path/to/files/ /destination/path
```

1.2 Submission Script

To submit your solution, use the turn-in script provided with the starter files. It will archive your submission and submit it to our server to be graded. You may submit multiple times, but only your latest submission will be graded. Fill out the `PID` file with your PID on the first line. If you're working with a partner, include both PIDs on the first line separated by a space.

1.3 Assignment Files

Starter files are provided in an archive on the class webpage. It contains exploit starter code (in the `sploits` directory) for each of the 4 vulnerable programs (in the `targets` directory). Additionally included with the exploits is `shellcode.h`, which gives Aleph One's shellcode. The `sploits` directory contains a Makefile to build the expoits. The `targets` directory contains a Makefile to generate targets specific to your PID, and a folder called `base` that you should not modify as these are what are used to generate your targets.

2 Assignment Instructions

You will be writing an exploit for each of the 4 vulnerable programs provided in the assignment. Each exploit, when run in the VM with its target installed setuid-root in `/tmp`, should yield a root shell (`/bin/sh`). You can verify this by typing `whoami` in the root shell, to which you should see the response `root`. You must use Aleph One's shellcode in `shellcode.h`, as this will be used in the grading scripts. Additionally, for each exploit, provide a brief description of how it works in the `writeup.txt` file.

3 Exploit Construction

To complete the assignment, you will first need to generate targets specific to your PID, then use GDB to find vulnerabilities in the targets, then build your exploits.

3.1 Generating the Targets

The first step to generate your targets is to fill in the `PID` file as specified in Section 1.2. Then run `make generate` in the `targets` directory to create the 4 target source files specific to you: `target1.c` through `target4.c`. Then run `make` to build the target binaries: `target1` through `target4`. Then run `make install` to copy the binaries into the `/tmp` directory.

To make the binaries as setuid-root, use `su` to launch a root shell, and then `make install` and `make setuid`. Don't forget to `exit` your root shell when you're done, returning you to the user shell.

3.2 Using GDB

To run an exploit in GDB, run e.g., `gdb -e sploit1 -s /tmp/target1` to execute `sploit1` and use the symbol file `target1`. I recommend the following workflow in GDB:

1. **Starting**. Set breakpoints that you can later use for analysis:
   - `b foo` — break at function `foo`
   - `b *0x08048489` — break at the instruction at address 0x08048489
   - `r` — run the executable
2. **Analyzing**. Examine memory, registers, etc.; disassemble code; show stackframe, backtrace, etc; and more:
   - `disas foo` — disassemble function `foo`
   - `i r` — view registers
   - `x <loc>` — examine memory
   - `x $eip` — examine current instruction pointer
   - `x $ebp+4` — examine return address
   - `x /10x $esp` — examine 10 words at top of stack
   - `x /10x buf` — examine 10 words in `buf`
   - `x /10i $eip` — examine 10 instructions starting at instruction pointer

- `x /10i buf` — examine 10 instructions starting at `buf`

3. **Continuing**. Continue analysis:
   - `c` — continue execution until next breakpoint/watchpoint
   - `si` — step to the next instruction
   - `s` — step to the next line of source code

Note that this is only a cursory overview of GDB; much more info is available from online resources.

If you wish, you can instrument the target code with arbitrary assembly using the "`__asm__()`" pseudo-function, to help with debugging. Be sure, however, that your final exploits work against the unmodified targets, since we will use these in grading.

3.3 Exploit Notes

For this assignment you should read and have a solid understanding of Aleph One's "Smashing the Stack for Fun and Profit".

Aleph One gives code that calculates addresses on the target's stack based on addresses on the exploit's stack. However, addresses on the exploit's stack can change based on how the exploit is executed (working directory, arguments, environment, etc.); in my testing, I do not guarantee to execute your exploits exactly the same way bash does. You must therefore hard-code target stack locations in your exploits. You should *not* use a function such as `get_sp()` in the exploits you hand in. You should only modify the sploit C files; i.e. your sploits should work with unmodified Makefiles, targets, etc.

In grading, the exploits may be run with a different environment and different working directory. Your exploits must work in these cases also. Your exploit programs should not take any command-line arguments.

Shutting down the VM removes the files in `/tmp`, and if you want to keep them then pause the VM and save its state!