CS155: Computer Security **Spring 2021**

Project #1

Due: Part 1: Thursday, April 8 - 11:59pm,
Parts 2 and 3: Thursday, April 15 - 11:59pm.

Submit by Gradescope (each answer on a separate page)

The goal of this assignment is to gain hands-on experience finding vulnerabilities in code and mounting buffer overflow attacks. In Parts 1 and 2, you are given the source code for six exploitable programs which are to be installed with `setuid` root in a virtual machine we provide. You'll have to identify a vulnerability (buffer overflow, double free, format string vulnerability, etc.) in each program. You'll write an exploit for each that executes the vulnerable program with crafted argument, causing it to jump to an exploit string. In each instance, the result will yield a root shell even though the attack was run by an unprivileged user. In Part 3, you will use a fuzzer to find a vulnerabiilty in a program called bsdtar, part of a widely used library called libarchive. You'll download and build the vulnerable version of libarchive in your VM and then run the fuzzer to find an input that causes the program to crash.

The Environment

You'll run your exploits in a virtual machine (VM) provided for the assignment. This serves two purposes. First, the vulnerable programs contain real, exploitable vulnerabilities and we strongly advise against installing them with `setuid` root on your machine. Second, everything from the particular compiler version, to the operating system and installed library versions will affect the exact location of code on the stack. The VM provides an identical environment to the one in which the assignment will be tested for grading.

The VM is configured with Ubuntu Linux 16.04 LTS, with ASLR (address randomization) turned off. It has a single user account "user" with password "cs155", but you can temporarily become the root user using `sudo`. The exploits will be run as "user" and should yield a command line shell (`/bin/sh`) running as "root". The VM comes with a set of tools pre-installed (curl, wget, openssh, gcc, vim etc), but feel free to install additional software. For example, to install the emacs editor, you can run as root:

```
$ apt-get install emacs
```

When you first run the VM, it will have an OpenSSH server running so you can login from your host machine as well as transfer files using, e.g., `ssh` and `scp`. You can login to the VM from your host machine using the command:

```
$ ssh user@192.168.56.155
```

The VM already contains the starter code in the `proj1` directory.

Parts 1 and 2

Parts 1 and 2 ask you to develop exploits for six different vulnerable target programs.

Targets

The `targets/` directory in the assignment tarball (which has already been copied to the VM for you) contains the source code for the vulnerable targets as well as a Makefile for building and installing them on the VM. Specifically, to install the target programs, as the non-root "user":

```
$ cd targets
$ make
$ sudo make install
```

This will compile all of the target programs, set the executable stack flag on each of the resulting executables, and install them with `setuid` root in `/tmp`.

Your exploits **must** assume that the target programs are installed in `/tmp/` such as `/tmp/target1`, `/tmp/target2`, etc.

Exploit Skeleton Code

The `sploits/` directory in the assignment tarball contains skeleton code for the exploits you'll write, named `sploit1.c`, `sploit2.c`, etc., to correspond with the targets. Also included is the header file `shellcode.h`, which provides Aleph One's shellcode in the static variable `static const char* shellcode`.

Part 3

In Part 3 you'll learn how to find security vulnerabilities using a fuzzer called American Fuzzy Lop. Fuzzing is a technique for finding vulnerabilities in a program by running the program on random data until it crashes. We will be using afl-fuzz, one of the most successful and widely-used fuzzers currently available. afl-fuzz has already been installed on the VM. You can read more about afl-fuzz and how it works at `http://lcamtuf.coredump.cx/afl/`.

Fuzzing libarchive

You will be fuzzing libarchive (`https://www.libarchive.org/`), a widely used archive and compression library. It provides a program called bsdtar that offers similar functionality to the more common GNU tar program. For example, you can use bsdtar to extract a `.tar.gz` file in the same way as regular tar:

```
$ bsdtar -xf <some-file>.tar.gz
```

In the `proj1/fuzz/` directory, download and extract the source code for libarchive version 3.1.2:

```
$ curl -O http://www.libarchive.org/downloads/libarchive-3.1.2.tar.gz
$ tar -xf libarchive-3.1.2.tar.gz
```

This should give you a directory called `libarchive-3.1.2/`. In that directory, run:

```
$ CC=afl-gcc ./configure --prefix=$HOME/proj1/fuzz/install
```

This will configure libarchive so that it will be built using the afl-fuzz compiler, `afl-gcc`, and so that it will install itself in the `install/` directory rather than system-wide. You can then build and install libarchive, including bsdtar:

```
$ make
$ make install
```

(Note that we are not using sudo.) After this, bsdtar should be installed under `install/bin/`.

The `fuzz/testcases/` directory contains a seed testcase that afl-fuzz will modify to try to crash bsdtar. Run afl-fuzz on this testcase by running the following command from the `fuzz/` directory:

```
$ afl-fuzz -i testcases -o results install/bin/bsdtar -O -xf @@
```

This command instructs afl-fuzz to run bsdtar and supply the arguments `-O -xf @@`. The `-O` (capital-O, not numeral-0) option tells bsdtar to not write any files to disk. afl-fuzz will replace `@@` with the name of the input to test for a crash, so the `-xf @@` part will cause bsdtar to try to extract the input file generated by afl-fuzz. The result is that afl-fuzz will generate a bunch of test cases based on the seeds in the `testcases/` directory and then run bsdtar on each generated file until bsdtar crashes.

The fuzzer may run for several minutes before finding a crash. Once it does, hit Ctrl-C to stop AFL.

Write-up

Spend a few minutes investigating the crash. Use GDB to get a backtrace at the time of the crash. Try to figure out what the vulnerability is in the source code.

In the `fuzz/README` file, include your backtrace from GDB and briefly describe the vulnerability (two or three sentences; no more than 200 words).

Deliverables

The assignment is divided into three parts:

- Part 1 consists of targets 1 and 2.
- Part 2 consists of the other four targets.
- Part 3 (which you will submit together with Part 2) consists of fuzzing a real-world program (bsdtar) to find a vulnerability.

For each submission, you'll need to provide a gzipped tarball (`.tar.gz`) generated by running `make submission` from the top-level directory of the assignment source (`proj1/`). This tarball will contain the contents of the `sploits/` directory, the crashes found during fuzzing, the README in the `fuzz/` directory, and `ID.csv`. Make sure that if you extract your submission tarball:

1. In the extracted `sploits/` directory, running `make` with no arguments should yield `sploit1` through `sploit6` executables in the same directory.

2. In the extracted `fuzz/` directory, running the vulnerable version of `bsdtar` on any of the crashing testcases should reproduce the crash.

3. In the extracted `fuzz/` directory, there should be a README file that describes the vulnerability found via fuzzing.

4. The tarball must include the file `ID.csv` which conatins a comma-separated line for each group member with your SUID number, SUNet ID, last name, first name (order matters). The top-level directory already contains such a file, so you just need to modify it.

NOTE: *Due to the size of the class, the correctness of your submission will be graded primarily by script. As a result, following the the submission format is important. We really, really want to give you full credit! Help us help you!*

Instructions for submitting the tarball will be posted on Piazza.

Extra Credit

The extra credit for this assignment is quite different from the rest of the assignment. Most significantly, you'll mount an exploit on a binary that has stack canaries turned on (specifically, GCC's stack protector). Because of the nature of exploits on canaries, you'll be attacking the program over the network rather than by invoking it through a call to `exec`. This will also require different shell code.

The Vulnerable Program

In the `targets/` directory, `extra-credit.c` is a forking network echo server. It listens for TCP connections on port 5555, for each connection it forks and, in the child process, reads in lines (separated by a carriage return and newline characters, `\r\n`) from the client, echoes them back until it reads an empty line. You're goal is to construct a payload to send over the network that redirects execution of the child process handling your connection to `unlink` the file `/tmp/passwd`. To achieve this you'll have to do two things:

1. Write new shell code that `unlink`s the file `/tmp/passwd` instead of `exec`ing `/bin/sh`. We've provided a file `shellcode.S` with the latter and a Makefile target (`shellcode.bin`) that compiles the shellcode into binary format such that it can be used in your exploit. **Note:** be sure that your exploit works when the extra-credit server is running as root, i.e., `sudo ./extra-credit` – your exploit should also work even if `/tmp/passwd` was created using `sudo`.

2. In the sploits directory, we've included a python script `extra-credit.py` that can serve as skeleton code for your exploit. It reads in `shellcode.bin` (which is compiled from `shellcode.S`), as well as connects to the vulnerable server. Modify `extra-credit.py` to mount your exploit. Note that the function `try_exploit` takes an exploit string and the connection socket, sends the exploit string across and returns `True` if the connection died and `False` if it is still alive (how can you use this signal?).

Extra Credit Deliverables

For the extra credit, you should include two files in your `submission.tar.gz`:

1. A text file `extra-credit.txt` in the `sploits/` directory with no more than 500 words answering the questions:
   - How does your exploit work?
   - What would you modify GCC's stack canaries mechanism, or Linux's fork syscall to protect against your attack?
2. The modified file `extra-credit.py`.

Hints

- Suppose you overflowed a buffer such that only a single byte of the canary is overwritten. If the process crashes and the connection is closed, what does that tell you? Conversely, if it doesn't crash, what does that tell you?
- The syscall number for `unlink` is 0x0a (10 in decimal), which is also the newline character in ASCII. Does this matter? How might you get around this?
- You can debug a running process with `gdb` by passing the `-p` argument with process ID of the target process. Note that each child process (the result of fork will have a different process ID).
- We strongly encourage you to use a NOP slide for this problem, even if you are able to find an exact address for the shellcode. In previous years we've noticed some variability in the exact position of things in memory that resulted in some student's solutions failing in the grading VM. Using a NOP slide will help prevent this problem.

Late Policy

The general course policy on late submissions applies to this assignment. The policy is details in the course overview on the website: `https://cs155.stanford.edu/info.html`

Setup: Set-by-step

> **Note:** The VM will not work on the new ARM MacBooks. If you have an ARM MacBook, please contact the TAs for instructions.

Download the VM from `https://crypto.stanford.edu/cs155/vm-cs155.tar.gz` and extract. The tarball contains a file `CS155.ova`, which is an Open Virtualization Format archive of the virtual machine.

Import the virtual machine using VirtualBox. We **strongly recommend** using VirtualBox. VirtualBox is free for Windows, macOS, and Linux, and the virtual machine was developed and tested using VirtualBox. To import the virtual machine, choose "File", "Import Appliance", and select the `CS155.ova` file. This will set up a new virtual machine called CS155.

Note: You might need to set network adapter 2 to use a host-only network if SSH doesn't work out of the box. On macOS and Linux, you can add a host-only network by selecting "File", "Host Network Manager", clicking the "Create" button and creating a virtual network interface named "`vboxnet0`". On Windows 10, the host-only network should be auto-configured when you open and close the "Settings" for the VM.

Once the VM has booted, login with username "user" and password "cs155". Your home directory will now contain the folder "proj1" which contains the targets and starter code for the project. The VM should be configured with a host-only network adapter and static IP address 192.168.56.155, so you should also be able to log in using SSH from your host:

```
$ ssh user@192.168.56.155
```

Once logged in, build and install the targets:

```
$ cd proj1/targets
$ make && sudo make install
Password: cs155
```

Write, build and test your exploits:

```
$ cd ../sploits
...edit,test...
$ make
$ ./sploit1
```

If at any point you'd like a fresh copy of the starter code, you can download the assignment tarball from `https://cs155.stanford.edu/hw_and_proj/proj1.tar.gz` and extract it:

```
$ wget https://cs155.stanford.edu/hw_and_proj/proj1.tar.gz
$ tar xzf proj1.tar.gz
```

Then repeat the build and installation steps above.