CS155: Computer Security **Spring 2023**

Project #1

Due: Part 1: Thursday, April 13 - 11:59pm,
Parts 2 and 3: Thursday, April 20 - 11:59pm.

Submit by Gradescope

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 five 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 vulnerability 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) emulated using QEMU. 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.

Launching your VM

To launch your VM, you will first need to install QEMU. Follow the instructions on QEMU's website here (https://www.qemu.org/download/). MacOS users will need to first install Homebrew. You can find instructions on doing that here (https://brew.sh/).

You can then launch your VM by running the included `run_qemu.sh` script. It may take a minute or two for your VM to launch. Once it does, you should see a login prompt in the terminal where you launched the VM. You can login and use your VM from that terminal. Your VM has a single user with the username "user" and the password "cs155". Alternatively, you can use `ssh` to log into your VM. You can use the included `./ssh_to_qemu.sh` script or run

```
$ ssh -p 5555 user@localhost
```

You may have a smoother terminal experience using your VM over ssh because QEMU doesn't use your entire terminal window.

Using your VM

The VM is configured with Debian GNU/Linux 11 (bullseye), 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:

```
$ sudo 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`, `scp`, and `sshfs`. The VM already contains the starter code in the `proj1` directory.

To shutdown your VM, you can run:

```
$ sudo systemctl poweroff
```

Parts 1 and 2

Parts 1 and 2 ask you to develop exploits for five 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.

Note: When you reboot your machine, files in the `/tmp` directory will be automatically deleted. This means you need to run `sudo make install` in the `proj1/targets` directory to reinstall the targets.

Exploit Skeleton Code

The `xploits/` directory in the assignment tarball contains skeleton code for the exploits you'll write, named `xploit1.c`, `xploit2.c`, etc., to correspond with the targets. Also included is the header file `shellcode.h`, which provides a shellcode in the static variable
`static const char* shellcode`.

target5

The fifth target has had its stack marked as non-executable. This means that you will not be able to jump to a shellcode that you've placed in memory. You will need to use Return Oriented Programming (ROP) to attack this target. For your convenience, you can use the `./find_gadgets.py` script to locate gadgets in this target. The script will search through a given executable and find every potential gadget in it. It's your job to use those gadgets to exploit `target5`.

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-clang ./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 three 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 `xploits/` directory, running `make` with no arguments should yield `xploit1` through `xploit5` 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 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.

Hints

1. `gdb` is an incredibly helpful tool for this assignment. CS 107's guide to `gdb` is a great resource. https://web.stanford.edu/class/cs107/resources/gdb
2. If you run `gdb`, and the `target` crashes, you will need to relaunch `gdb` to run the `xploit` again. This is because the `gdb`'s `run` command will attempt to restart the `target` rather than the `xploit`.
3. The `gdb` command `info frame` gives you useful information about the current stack frame.

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

Download the VM from https://crypto.stanford.edu/cs155/proj1-vm.tar.gz and extract. The tarball contains a file `disk.img`, which is an QEMU QCOW Disk Image.

Launch the machine using the included `run_qemu.sh` script. The script just invokes the following line

```
qemu-system-x86_64 disk.img -m 2G -nic user,hostfwd=tcp::5555-:22 -nographic
```

The `-m 2G` option launches the VM with 2 Gigabytes of memory. The `-nic user,hostfwd=tcp::5555-:22` option forwards internet traffic on your computer's port 5555 to the VM's port 22. This allows you to ssh into your VM. The `-nographic` option make QEMU forward the output of your VM to your terminal. You can launch your VM without this option. If you do, QEMu will create a new window for your VM. See QEMU's documentation for more information on this. https://www.qemu.org/docs/master/

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 to forward traffic on your machine's port 5555 to the VM's port 22. This lets you ssh into your VM with the following

```
    $ ssh -p 5555 user@localhost
```

Once logged in, build and install the targets:

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

Write, build and test your exploits:

```
    $ cd ../xploits
    ...edit,test...
    $ make
    $ ./xploit1
```

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.