CSCI 3330 Assignment 8

Deadline: December 5, 2025, 10:00pm

1 Threat model

In this assignment, you will develop and execute a memory exploit that will circumvent all Debian 13 default security protections as well as additional protections imposed by WU-FTPD, including the following:

- Address space layout randomization (ASLR)
- Non-executable stack/heap
- Stack canaries
- Privilege dropping
- chroot "jails"

You will exploit a historical vulnerability in WU-FTPD, a real-world FTP server that was at one time the most popular FTP server in the world. The vulnerability you will be exploiting is a classic format string vulnerability. Identifying these vulnerabilities is often the easy part. Exploiting one will prove more difficult.

2 Setting up Docker container

Set up the Docker container according to the following steps:

1. Following the official instructions, install Docker on a Linux, macOS, or Windows machine.
2. Download and extract the tarball.
3. Run ./build.sh to build the Docker container.
4. Run ./run.sh to execute it. It will give you a shell.

If you have trouble with this, reach out to me immediately, as getting Docker running is necessary to make further progress on the assignment.

You will already have a shell account on the Debian 13 container. Your username is `secpriv` and your password is `secpr!v22`. In fact, you will already

be a sudoer, so you will be able to use the `sudo` command with your password to become root on the server. However, these accounts are provided so that you can use debugging tools such as gdb and strace to debug your attack. This is no different than how you might practice an attack on your own machine before launching it on another. **Your final attack will make no assumptions about your having accounts on the machine and will give you a root shell on the server completely remotely via an anonymous FTP connection.**

The Docker container is set up to mount the `./home` directory in your extracted tarball as your home directory in the container. Thus, while the container is running, you may edit files inside of the `./home` directory on your host, and those changes will be reflected in `/home/secpriv` inside the container (and vice versa). This directory already contains template code for executing your exploit, and it will be retained between reboots of the Docker container. Note that all changes in the Docker container outside of `/home/secpriv` will not persist between runs. If you want to, e.g., install a package and have it be installed on every boot, you will have to edit the `Dockerfile` to include this package and rebuild the container. If you think a package is missing that would be useful to other students, let me know so that I can include it in future revisions of the assignment.

3 Connecting to the FTP server

The server is a standard FTP server, and so you may connect to it with any FTP client. The service is available via two means: first, *inside* the container, it is listening on port 21 (the default FTP control port). Second, *outside* of the container, on the containing host, it is listening on port 2121, bound to the loopback adapter. Therefore, inside of the container, you can connect to the server on 127.0.0.1:21, and outside of the container, on the containing host, you can connect to the server on 127.0.0.1:2121. The FTP service is *not* available to other machines on your network, and you are the only one who will be able to exploit it.

For your convenience, I have included a `connect.sh` script to automatically connect to the server (if your container is running). It will automatically connect on the correct port depending on whether it is run inside or outside of the container. Inside of the container, it is in the "exploit" directory of your `$HOME` directory. Outside of the container, it is in the "exploit" directory inside of the "home" directory extracted from the tarball. Recall that FTP uses two different connections, one for control and one for data. This script is very low level, and it does not implement the data channel of the FTP protocol, only the control channel, so FTP commands which require the data channel won't work (they will silently do nothing).

You may also ssh into the server (if your container is running) from outside the container. You may ssh on port 2222 to 127.0.0.1:

```
ssh -p2222 secpriv@127.0.0.1
```

This will facilitate having more than one shell open in the container.

4 Discovering a vulnerability

The version of WU-FTPD that you will be attacking (2.6.0) contains a format string vulnerability relating to its handling of the `SITE EXEC` FTP command. This command is used to allow you to run remote commands on the server. This sounds at first blush very useful! But, unfortunately, in this instance, and, by default, no such commands are configured in WU-FTPD's configuration file, so this ability won't be useful to us per se.

- Under your home directory, in the exploit directory, run the `./connect.sh` script. This will connect to your local FTP server. What happens if you run the FTP command `SITE EXEC ls -l` or `SITE EXEC whoami`? Do those commands run?

Even though `SITE EXEC` is not configured to run any commands, it does contain a format string vulnerability!

- What happens if you run `SITE EXEC %s` or `SITE EXEC %lx`? What do these FTP commands print, and what does the latter print in relation to the former?

5 Probing for parameters

Recall that writing a sequence of four bytes $b_1b_2b_3b_4$ to memory using a format string exploit requires the following three steps:

1. Determining the *address* of your buffer on the stack relative to `%ebp+8` (the first stack argument on 32-bit x86 with a frame pointer), as measured by the number of 32-bit words.

2. If the sequence of bytes b_1b_2 is $\leq$ to b_3b_4 (when interpreted little-endian), write *address* and *address* + 2 in your buffer, else write *address* + 2 and *address*.

3. Craft a `%hn` format conversion to write b_1b_2 and b_3b_4 to the addresses written in (2) by using the `%ebp+8`-relative offset determined in (1).

In this section, you will determine the `%ebp+8`-relative offset in step (1) by implementing `probe.py`. Since for debugging purposes you already have access to gdb and root on the server, you can obtain this offset using gdb and performing arithmetic on `%ebp+8` and the address of the stack buffer. However, even if you do, it is important to understand how to do this completely remotely, as would be required in a realistic attack scenario.

Thus, you will implement this by planting a special canary ("AAAA") and outputting a sequence of format strings that read increasingly large offsets from

`%ebp+8` looking for the canary. When you find the offset containing the canary, you can use that offset in future attacks.

To instrument this attack, in a loop, output strings of the form

```
sys.stdout.buffer.write(b'SITE EXEC xxx AAAA ' + ...)
```

to print 32-bit values at increasingly large argument indices (corresponding, in WU-FTPD, to offsets from `%ebp+8`) until you find the offset corresponding to "AAAA". While there is no single way to do this, the `%x` or even a carefully crafted `%s` conversion may be useful here. When you find the `%ebp+8`-relative offset corresponding to "AAAA", make a note of this, as you will use it to further develop the final attack. The `xxx` bytes are padding to align AAAA to a 32-bit boundary — they do not need to be modified.

You will find both `probe.py`, the file you will be modifying, and `probe.sh`, a convenience script for piping the output of `probe.py` to the FTP server, located in the same directory as `connect.sh`. Like `connect.sh`, `probe.sh` can be run from either inside or outside of the container.

6 Overwriting memory

In this section, you will implement `exploit.py` to selectively overwrite addresses in the GOT to point to PLT entries to take over execution of the program. The `exploit.py` file is located in the same directory as the `probe.py` script you implemented in the previous section and also includes a convenience `exploit.sh` script to automatically pipe the output into an FTP connection. Look over the provided template, including the code comments, as much of the scaffolding of the exploit has already been completed for you.

In addition to Debian's default security protections, this attack will overcome two protections that WU-FTPD implements. First, after logging in, WU-FTPD drops the effective UID from root to that of the logged in user or, in the case of a guest account, a guest user. Thus, to get a root shell, you will have to restore your root privileges. Thankfully, because WU-FTPD occasionally needs to become root again to perform miscellaneous tasks as an FTP server, it keeps its real UID as zero and only drops root in the effective UID, opening a means for us to reacquire root privileges.

Second, after logging in, WU-FTPD puts the user in a chroot "jail" inside of their FTP directory. After acquiring root privileges, you will use the traditional escape technique to get out.

To complete the exploit, you will need to implement and perform five format string attacks:

1. Replace the GOT entry of `alarm()` with `seteuid()`'s PLT entry so that every call to `alarm(0)` becomes a call to `seteuid(0)`. This will restore your effective UID to zero.

2. Replace the GOT entry of `rename()` with `chroot()`'s PLT entry so that every call to `rename(x, ...)` becomes a call to `chroot(x)`. You will use this to chroot to another directory inside of your chroot jail.

3. Replace the GOT entry of `rename()` with `chdir()`'s PLT entry so that every call to `rename(x, ...)` becomes a call to `chdir(x)`. You will use this to traverse out of the chroot jail.

4. Replace the GOT entry of `rename()` with `chroot()`'s PLT entry so that every call to `rename(x, ...)` becomes a call to `chroot(x)`. You will use this to chroot to the real root, completing your chroot escape.

5. Replace the GOT entry of `rename()` with `execv()`'s PLT entry so that every call to `rename(x, y)` becomes a call to `execv(x, y)`. You will use this to execv() a shell.

If your exploit works correctly, you will have a root shell! Try typing in `whoami`. This attack can be run remotely from outside of the container and will get you a root shell inside. If this FTP server were configured to listen to other machines on your network, anyone on your network could use this to get a root shell on the Docker container!

Questions:

1. What is the significance of UID zero in this attack?

2. What are the steps to escaping a chroot jail? Map the steps you identified here to steps in the attack above.

3. Why can't you use null (zero) bytes in the `argv` vector you pass to `execv()`? Why does that make the attack more difficult?

Some helpful commands may include the following:

1. `ps -e | grep ftp[d]` — show pid(s) of running ftpd process(es); ftpd will only be running if you have a connection to it, and there will be one process per connection

2. `sudo gdb -p PID` — attach with gdb to the process with pid PID; for example, you can use gdb to set a breakpoint on a function you are trying to call in your attack to debug if you are successfully calling it

3. `strace -p PID` — trace system calls of the process with pid PID; another way to determine if the functions you are trying to call are indeed being called

4. `objdump -d /usr/sbin/ftpd` — disassemble ftpd (including the `.plt` section; disassembling the `.plt` section is useful for getting the PLT and GOT addresses that will be useful in your attack

Note that, unlike the `*.sh` files in your home directory, all of the above commands must be run from *inside* of the container.

For reference, you may also consult the WU-FTPD source code, although the assignment can be completed without doing so.

7 Position-independent executables

A lot of older software, including WU-FTPD, is not configured to compile as a position-independent executable (PIE). Thus, when using ASLR, the positions of libraries, the stack, and the heap are randomized, but the actual executable itself, including the GOT and PLT sections, are not.

1. If WU-FTPD were compiled as a PIE, would that break the attack that you implemented?
2. If so, at a high level, what could you add to the attack so that it would work with PIE? If not, at a high level, why does the original attack not assume the position of the executable in memory?

8 Submission

Email your code for this assignment and your answers to each question in a single `*.txt` or `*.pdf` file attachment to my email address in the Syllabus.