Project: Handin

Due: Saturday, March 26 @ 11:59 pm ET

0 Introduction

After the FLAG portal fiasco, the Academy decided that a safer way for students to submit assignments was via a handin script, one very similar to the system used at Brown. Each course has a directory in a shared filesystem, and running `csXXX_handin` (where `csXXX` is a course number) invokes a `setgid` binary that saves all files in the current working directory in a `.tar` archive in the course's handin directory. Additionally, every course has an autograder which can extract a student's handin and automatically grade it by running test suites against their code. These grades are automatically collected in a course-wide grades database.

As a new crewmate, you're currently enrolled in `cs666`: "Secure Computer Systems". Presently, they've released one assignment, "Ivy", a similar problem to the one from the *Cryptography* project from Brown University's CS166. Appendix A contains the handout for their version of "Ivy". One thing you know after talking to your friends at Brown University is that solving "Ivy" wasn't easy—maybe you'll have more luck with simply breaking the `cs666` infrastructure instead. . .

0.1 Requirements

CS166 and CS162 students must both complete the assignment described in Part I.

0.2 Accessing the Infrastructure

Each student gets their own instance of the Academy's infrastructure to attack. You can access your Academy machine by running `/course/cs1660/bin/cs166_ssh_handin` from a department machine.

Your username on the Academy infrastructure is `alice` and your password is `iamalice`. We've also provided another user, `bob` (password: `iambob`), whose account you can use to test attacks against other students. (You can use the `su <user>` command to switch between accounts on the VM.)

Part I

The cs666_handin System

In this project, you'll exercise your operating systems security knowledge and gain practice in *white box testing* to analyze the source code of a complex handin system to discover and exploit vulnerabilities of increasing impact. Additionally, you'll gain practice in scripting automated attacks against systems.

1 Assignment

You will break `cs666`'s course infrastructure by creating *exploits* that take advantage of *distinct vulnerabilities*. *Exploits* must allow you to perform a normally unauthorized action in the system or discover information that unprivileged users should not have access to. For example, viewing other students' grades, accessing other students' handins, or running arbitrary code with TA group permissions would all count as exploits.

An exploit's "distinct-ness" is defined as a tuple over a file from the source code of the `cs666` course infrastructure *and* one or more *vulnerability categories*. **Two exploits take advantage of distinct vulnerabilities if (1) both exploits take advantage of vulnerabilities in *different files or* (2) both exploits take advantage of vulnerabilities in the *same* file, but rely on non-overlapping sets of *vulnerability categories*.**

The source code of `cs666`'s course infrastructure can be found at `/course/cs1660/pub/handin` on the department machines—you should refer to the files in this directory when determining if two exploits satisfy the *distinct vulnerability* definition above. You should also consult Appendix B, which contains the list of possible vulnerability categories that we'll accept on this project.

1.1 Severity Categories

Each exploit receives points for its *severity category*, which describes the impact of the exploit on the system. The values of each category are outlined in the following table:

Severity Category	Description	Points
Arbitrary Code Execution	Execute arbitrary code as the TA group.	10
Data Modification	Change existing data that you should not be allowed to modify.	7
Data Exfiltration	Get access to data that you should not have access to.	6
Data Theft	Trick the infrastructure into believing that somebody else's data is your own (for example, use another student's handin as your own). If you manage to also get access to the data yourself, that counts as *Data Exfiltration* (not just *Data Theft*).	4
Metadata Exfiltration	Get access to metadata that you should not have access to. Metadata includes whether or not other students have handed in, the names (but not contents) of files in restricted parts of the file tree (under `/course/cs666`), etc.	2

Both CS166 students are tasked with finding 26 points worth of exploits, with an extra credit cap of 43 points. CS162 students are tasked with finding 32 points worth of exploits, with an extra credit cap of 49 points. Raw scores are scaled to a *final project score* using the formulas in Appendix D.

1.2 Vulnerability Reports

In a `README.pdf` file, you should document the following for each of the exploits you discover:

- **Metadata**: The *severity category* of the exploit (see Section 1.1), the *vulnerability categories* that the exploit takes advantage of (see Appendix B), and the name(s) of the file(s) that these vulnerabilities manifest in (see the source code).
- **Discovery**: An explanation of how you came to this plan of attack (what the system does that makes it vulnerable to this specific attack; references to relevant sections of the handin system's source code; any tools you used to make these findings; etc.).
- **Impact**: An explanation of how and why your attack works (what it does and why; references to portions of your exploit script, etc.) and a justification for why it works (including how the output of the script makes it clear that the attack was successful).
- **Mitigation**: Explain (from a technical perspective) how to repair the vulnerability without compromising intended functionality and justify why this fix blocks your exploit (and exploits similar to it). You should include *specific references to the source code* as to where fixes should be applied.

You should also include any additional files needed to perform your exploit (code, payloads, etc.) in your final handin. Your report should allow us to *easily* recreate your attack from only your verbal (and written) explanations and submitted files.

2 Hints, Tips, and Tricks

2.1 `whoami`

There's a binary in your VM at `/home/whoami` which is essentially a more powerful version of the normal `whoami` command. It prints the `uid`, `euid`, `gid`, and `egid` of the process that it runs as (and thus, by default, that its parent process runs as). This may be useful in testing some of your exploits.

2.2 Tools and `man` Pages

You may find the `environ(5)`, `proc(5)`, `credentials(7)`, and `symlink(7) man` pages helpful for this assignment. In addition to those resources, you may find the following tools on your VMs useful (refer to their `man` pages for usage information):

- `stat` — get detailed information about a given file or directory
- `ps` — lists running processes and basic info like the commands that spawned them
- `htop` — live process viewer
- `watch` — execute a program periodically
- `strace` — traces system calls
- `strings` — prints strings in binary
- `id -u <user>` — gets user id
- `getent group <group>` — gets group id

2.3 Transferring Files Using `sftp`

Using `sftp`, you can access files in your VM to download and edit them on your local machine.If you want to transfer files between the department machines and your VM using `sftp`, you can run the `cs166_sftp_handin` command from a department machine and a `sftp` instance will automatically be launched for you.[1] After `sftp` is launched, you can use the `get /path/to/VM/file /path/to/local/directory` command to copy a file from your VM to a department machine.

[1] If you want to run the `sftp` command yourself or want to access the Academy's infrastructure via other tools, you can find the IP address and SSH key needed to access your instance in `/course/cs1660/student/<your-login>/handin`.

To copy a file to your VM, you can use the `put /path/to/local/file /path/to/VM/directory` command. (To copy an entire directory, you can specify the recursive flag `-r` with either of the commands.)

2.4 Resetting the VM

If you would like to refresh the `/course/cs666` directory to its original state, you can do so by running the command `reset-cs666` (located at `/bin/reset-cs666`) on your VM. This will delete the `/course/cs666` directory and recreate it. You are not allowed to use the `reset-cs666` command in your actual exploits, though feel free to use it to verify that your exploit scripts work on unmodified versions of the `/course/cs666` directory after you've made some progress.

If you find that you have have broken the infrastructure to the point where you think you need a full reset of your VM, please email the TA list and we'll create a new VM for you. This will change your VM's IP address and credentials needed to access the VM *and* will delete all files you've stored on the machine, so make sure to save all of the work you want to keep elsewhere before asking for a hard reset.

3 Deliverables

In the security world, attacks are only taken seriously when one can demonstrate that their attack actually allows one to perform unauthorized tasks in a clear and convincing manner. For this project, you must prepare a *recorded video demonstration* of all of your attacks as part of your final handin, alongside a README containing your vulnerability reports.

3.1 Video Demo

Your final handin must include an MP4 video file named `demo.mp4` in which you demonstrate each of your exploits. The logistical requirements for this video are as follows:

- Your video must be *at most 10 minutes* in length. In your video, you should only demonstrate each of your exploits, **not** provide explanations—your README is the only place you need to include your vulnerability reports.
- We recommend that you use Zoom to locally record your presentation. Zoom will also automatically export a video in the proper MP4 format. See `https://support.zoom.us/hc/en-us/articles/201362473` for instructions.
- You are free to edit your video in any way that you see fit, though you aren't required to. Similarly, you don't have to record your presentation in a single take, though you can if you want.

You should aim to convince your grader that each of your exploits would work against a *clean* instance of the handin system just from your presentation of that exploit. By "clean" instance, we mean an instance of the `/course/cs666` that has been reset (see Section 2.4). This means that if your exploits may potentially interfere with each other, you should reset the application in between the presentation of your exploits.

3.2 Handing In

Your handin should consist of `demo.mp4` and a single PDF file `README.pdf` that contains written forms of your vulnerability reports *in the order you are presenting each vulnerability in your demo video*. You should also submit any code, files, or payloads needed to execute each of your exploits. Your additional files do not need to be named in any particular way as long as you make clear in your recorded demo and in your `README.pdf` which files are relevant to each vulnerability report.

Once you're ready to submit, please upload your files to the appropriately named upload point on Gradescope—do *not* upload your files as a `.zip`.

Part II

Linking

CS162 students can *optionally* complete this part of the project for up to 15 points extra credit on the whole assignment.

In this problem, you will explore another security hole in the Crewmate Academy's infrastructure that works regardless of whether or not the `setgid` handin script exists.

4 Setup

While you've discovered exploits that allow you to poke around the `/course/cs666` directory, you've heard rumors that the `cs666` course staff left a *honeypot*[2] within their course directory to detect security breaches. While the directory seems to contain the "correct" `KEY` value for your `ivy` binary, the `cs666` staff supposedly put an incorrect key there as a way to detect if a student broke their permissions system! The staff can then check to see if anyone used the honeypot value as their `KEY`.

While you haven't been able to confirm the veracity of these remarks, you're not willing to take any chances. However, you're still a little lazy—it's too much work to actually solve the assignment as intended. Directly extracting the key from the `ivy` binary's data is more up your alley—and that way, you'd know for sure what the correct `KEY` is.

Unfortunately, the `KEY` is compiled directly into the binary, so the only way to extract the `KEY` would involve examining the memory of the binary (or the process that runs it), and non-root processes cannot examine other processes' memory. Additionally, the `ivy` binary doesn't have world read permissions on it, so you can't simply run `strings(1)` on the binary or use `gdb(1)` to extract the `KEY`. Luckily for you, `cs666` made a subtle mistake in creating their binary that might still allow you to do what you want...

4.1 Dynamic Linking

Dynamic loading[3] is a mechanism by which a computer program can, at run time, load a library into memory and execute the addresses of functions and variables contained in the library; a *dynamically linked* program is a program that takes advantage of dynamic loading. In comparison, a *statically linked* program has all library code included directly in the binary. Usually, there are many benefits to dynamic linking—for example, multiple processes that load the same library share a single copy of the library in physical memory, which reduces memory load—but, as we'll see in this assignment, it can also pose a massive security risk.

4.1.1 LD_PRELOAD

Normally, the Linux dynamic loader, `ld-linux(8)`, finds and loads the shared libraries needed by a program (for example, the C standard library, `libc.so`), prepares the program to run, and then runs it. However, `LD_PRELOAD` is an environment variable which can contain one or more paths to shared libraries, or shared objects, that the loader will load before any of the default libraries. This is called *preloading* a library.

When we run a dynamically linked program, the dynamic loader tries to match the names of all library functions referenced in our program to an identically named function from a loaded library, checking in each library *in the order in which the libraries were loaded.* This means that preloaded libraries are matched against before standard libraries.

Here's the critical insight—if we could figure out the library functions called by a dynamically linked binary, we could preload another library that defines a function with the same name as one of the library functions

[2] `https://en.wikipedia.org/wiki/Honeypot_(computing)`
[3] `https://en.wikipedia.org/wiki/Dynamic_loading`

referenced by that binary. In this way, we could essentially inject code into that binary, which gives us access to normally privileged actions such as inspecting the binary's memory at runtime.

4.1.2 Overriding a Library Function

While we might be able to inject code into the running process, since the code we've injected runs as a standalone library, it doesn't have direct access to the variables defined within the process. However, overriding a library function gives us access to the arguments that are passed into any applications of that function in the target program. Using this technique and a little bit of source code analysis, you should be able to find an attack vector by which you can get (at least indirect) access to the `KEY` value, and eventually recover the `KEY`.

4.1.3 Creating a Library

The last part to all of this is creating our own library. To do this, we can write our library in C, then use `gcc(1)` to compile it using special flags—in particular, the `-shared` and `-fPIC` flags.[4] For example, if we've defined a library in `my-library.c`, we can compile it to a `my-library.o` file with the following:

```
gcc -shared -fPIC my-library.c -o my-library.so
```

You can then run the `ivy` binary in your shell with a particular `LD_PRELOAD` environment value by setting the variable on the same line as you execute the program:

```
LD_PRELOAD=/path/to/my-library.so /course/cs666/student/alice/ivy/ivy
```

4.1.4 Dynamic Linking in `ivy`

Of course, none of this matters to us if we don't have a dynamically linked binary. However, by default, `gcc(1)` will compile C programs such that they dynamically link any referenced standard libraries unless the `-static` flag is passed at compile-time. The source code for the `ivy` binary shows us that this flag isn't present in the `compile.sh` script, so we can safely assume the binary is dynamically linked.

5 Assignment

Your task is to extract the key used in the `/course/cs666/student/alice/ivy/ivy` binary using the `LD_PRELOAD` attack described above. You are not required to follow our attack description exactly, but your final submission *must* involve code injection via preloading a library to receive any credit.

5.1 Handing In

Your handin, which should be uploaded to Gradescope, will consist of two primary files: `KEY`, which contains the key you recovered from the `ivy` binary, and `README.pdf`, a PDF containing:

- A *detailed* account of the steps that you took to recover the key (including how your library works).
- The *simplest* possible fix for this vulnerability you can find. This should be a fix that works immediately, doesn't introduce any new vulnerabilities, and requires the least amount of work to implement. (As a rough guideline for what we mean by "simplest"—we're looking for something that requires the least amount of characters to type.)

Additionally, you should submit any other files that you needed to carry out your attack. Please include documentation on how your code works and how to compile it in your `README.pdf`.

You do not need to demonstrate your Linking exploit during the video demo.

[4] `-shared` compiles our program to a shared object that can be included in other executables; `-fPIC` generates *position-independent code* that ensures that other programs can dynamically load our shared object. These flags are not necessarily what you'll need on other architectures (but they definitely work for the Academy's system).

Part III

Appendix

A The Academy's Ivy Handout

Coincidentally, the Crewmate Academy's cs666 based most of their "Ivy" assignment off of the "Ivy" component from the *Cryptography* project in Brown University's CS166, so we've only quoted the relevant changes in their version of the "Ivy" handout below.

Assignment: Ivy

Due: Saturday, March 26 @ 11:59 pm ET

In this problem, you'll try to steal the encryption key used by a wireless encryption protocol. As usual, this assignment is autograded immediately upon handing in, so please make sure to double-check that your handin matches the specifications described in this handout before handing in.

A.1 Tasks

The binary at `/course/cs666/student/<your-login>/ivy/ivy` simulates your router. Given hex-encoded plaintexts on `stdin`, it prints corresponding ciphertexts to `stdout` in the format:

`<iv> <ciphertext>`

Input plaintexts *must* match the length of the key in order to be accepted by the router. Additionally, the first line of output corresponds to the ciphertext of the authentication packet that the router first sends to the hub. You have two tasks:

1. Recover the shared key, k, by interacting with the binary.
2. Write a Go program that automatically performs your attack in the future. The `/course/cs666/pub/ivy` directory contains two files, `main.go` and `ivy.go`, that you should use as a starting point for your attack.

A.2 What to Hand In

Your handin should consist of two files: `KEY` and `main.go`. `KEY` should contain the recovered key, encoded in hex; `main.go` should implement your attack. You should *not* turn in the `ivy.go` file from the stencil code, as our autograder will supply its own copy of `ivy.go` to test your solution. (You'll get an error if you try to turn in `ivy.go`.)

You can hand in your files by running `cs666_handin ivy` from a directory containing your `KEY` and `main.go` files. As usual, you can view your current grade on this assignment (and other course assignments) by running the `report` command—if you think you can improve your grade, you are welcome to hand in the assignment as many times as you'd like until the project due date.

– the cs666 course staff

B Vulnerability Categories

Below, we've listed every vulnerability category we could imagine coming up in a project like this. This means some categories may not necessarily have a corresponding vulnerability in the Crewmate Academy's infrastructure.

While we've discussed some of these vulnerabilities in lecture, some are probably new to you (or might not appear in the same way you've seen before). Much of security involves learning about previously unknown systems, so we expect that you'll need to do your own research into some concepts covered in this project. If you find yourself at a point where you feel that you haven't been taught how to do something, that's okay! You should feel confident that you can do it if you set your mind to it.

We recommend using the CS166 Handin Wiki at `http://cs.brown.edu/courses/csci1660/handin-wiki/` if you want more help and resources for learning about these different vulnerability categories.

Vulnerability Category	**Category ID**
Exfiltrated Process Information	`exfil-pi`
Buffer Overflow / Memory Corruption	`mem-crpt`
Path Sanitation Bypass	`path-byp`
Symlink Traversal	`symlinkt`
Unsanitized Environment Variables	`env-vars`
Outdated System Components with Known Vulnerabilities	`sys-vuln`
Misconfigured Blocklists / Safelists	`listconf`
TOCTOU (Race Condition)	`racecond`
Misconfigured File / Directory Permissions	`permconf`
Escaping `chroot` or Sandbox	`breakout`

You may also find vulnerabilities that do not necessarily fall into any one of these categories. They're rare, but if you find them, feel free to check in with the TAs to see if it will be accepted under a distinct vulnerability category.

C Some Hints For Getting Started

In general, always be thinking about places where the system could be failing to take into account or making false assumptions about the integrity, permissions, or format of the data/code it is operating on. Here are some ideas for things to think about as you start analyzing the system:

1. How does the hand in system make sure you only turn in code it considers needed for the assignment? What other features/libraries/methods might Go have that are unaccounted for?
2. What ends up getting included in a submission when a student runs the hand in script? What does the archive file extraction code accept?
3. The autograde system creates temporary files at several steps when it runs, where/how are those files created and what actions are allowed on those files?
4. How is data passed between each component of the autograding pipeline? What kind of information about each step might be included in process data?

D Grade Scaling Formulas

The following formulas will convert your *raw score*, the sum of points you earn from your submitted exploits, and the *final project score*, a scaled value computed from your raw score. The *final project score* will be your final grade on the Handin project. These formulas guarantee that reaching the baseline score will result in a final project score of 100% on the Handin project and that earning the maximum amount of extra credit will result in a final project score of 105%.

Formula for CS166 students:

$$\textsf{final-score} = \left(\frac{\min(\texttt{rawScore}, 26)}{26}\right) + \left(0.05 \cdot \frac{\min(\max(\texttt{rawScore} - 26, 0), 17)}{17}\right)$$

Formula for CS162 students:

$$\textsf{final-score} = \left(\frac{\min(\texttt{rawScore}, 32)}{32}\right) + \left(0.05 \cdot \frac{\min(\max(\texttt{rawScore} - 32, 0), 17)}{17}\right)$$