Project: Handin

Due: Tuesday, Apr 3 @ 11:59 pm ET

0 Introduction

After the FLAG portal fiasco, Blue University decided that a safer way for students to submit assignments was via a handin script on department machines, very similar to the system sometimes used by CS courses 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 student, you're currently enrolled in cs666: "Secure Computer Systems". One assignment has been released so far: "Ivy", a similar problem to the one from our *Cryptography* project. 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—and this version is a little different, and more annoying. Instead of trying to figure out Ivy again, maybe you'll have more luck with simply breaking the cs666 autograder infrastructure instead...

0.1 Learning goals

In this project, you will have an opportunity to gain practice with operating system privileges and mechanics by discovering how the autograding framework operates and figuring out how to break it. You will play the role of a tester (a student who doesn't want to do the assignment, a TA preparing for next semester, etc.) who is investigating the assignment for vulnerabilities. As in the previous project, you will write up a brief report about what you found, how you were able to exploit it, and comment on how the vulnerability could be fixed.

Critically, in this project *you have access to the course code for the system you are attacking*. This is to help you gain experience with a different form of testing systems—instead of discovering how the system works just from testing, you can analyze the code to understand how it works and how it is vulnerable. Note that you are still not required to develop fixes for the vulnerabilities you find: like the previous project, you should consider your report as something that could presented to the system's developers so they can address the issues.

1 Assignment

CS1660 students will find and exploit **four** ***distinct*** **vulnerabilities** in the CS666 autograding framework. An *exploit* must allow you to perform a normally unauthorized action in the system or discover information that unprivileged users should not be able to access. For example, viewing other students' grades, accessing other students' submissions, or running arbitrary code with TA group permissions would all count as exploits.

CS1620/CS2660 component CS1620/CS2660 students must find and exploit **one additional vulnerability**, for a total of **five** ***distinct*** **vulnerabilities**. There is no additional component for CS1620/CS2660 students, beyond finding one additional vulnerability.

Wiki Similar to Flag, we've provided a wiki describing each type of vulnerability and some details on how it works—we **highly** recommend using the if you want more help and resources for learning about these different vulnerability categories. This is a great starting point for this project, as it will give you an idea of what attacks you can carry out.
You can view the wiki here: `http://brown-csci1660.github.io/handin-wiki/`

Counting vulnerabilities Compared to Flag, this project uses a slightly different definition for "distinctness" since we have access to the project's source code. An exploit's distinctness is defined based on two parameters:

1. The **file** that contains the vulnerability (based on the autograder source code)
2. Its **vulnerability category** from the list of categories in Appendix B

Thus, you may count multiple vulnerabilities in the same category, so long as they are in different files. Similarly, you may count multiple vulnerabilities in the same file, so long as they belong to different categories. However, you may not count more than one vulnerability that has the same (file, category) tuple.

Source code cs666 loves open source, so you have access to all of the source code for cs666's course infrastructure, which is located here: `https://github.com/brown-csci1660/handin-source`

You can refer to the files in this directory to understand how the infrastructure works, as well as to determine the location of each vulnerability for your writeup.

Some of the autograder code, and CS666's Ivy stencil, is written using Go, a memory-safe systems programming language. If you haven't seen Go before, don't worry—similar to PHP, it's generally straightforward to read to understand the mechanics. While this project does not require you to write any Go code, you may need to do this to carry out certain exploits. For some resources on working with Go, see here.

Scope Vulnerabilities must manifest in programs, files, and scripts that are part of the cs666 course infrastructure. Once again, you will work on this project in a container environment. Similar to the flag project, **attacks on the container infrastructure are out of scope**, as these do not pertain to cs666's course infrastructure. That is, a vulnerability must not rely on `docker` to execute commands as a different user or modify the filesystem, as this is outside the attack surface of the course infrastructure you are testing. While it's easy to break into the container, it is not in your best interest to do so—we are grading you on the operating system vulnerabilities you find and your demonstration of them, so doing this will not improve your grade.

1.1 Extra credit

Each additional, *distinct* vulnerability you discover and exploit counts for extra credit—we will give points for **up to two** additional extra credit vulnerabilities, for a total of up to 6 vulnerabilities for CS1660 students, and 7 for CS1620/CS2660 students. Extra credit points are worth up to at most 5% of the total grade, for a maximum score of 105%.

2 Setup

2.1 Setup guide: Accessing the Infrastructure

For this project, we have created a Docker image that you can use to run your version of cs666's course infrastructure. For more information, see the instructions linked here:
`https://hackmd.io/@cs1660/handin-setup-guide`

2.2 Starter repository

You can create a repository and download the starter files for loading the container environment using this link:`https://classroom.github.com/a/ffCFDprf`

This repository initially contains a script to download and run the course infrastructure's container environment. This repository is mainly a place to store and submit your README and any code you write. See the setup guide for recommendations on where to clone the repository relative to your other container environments.

Since your primary goal is finding and writing about vulnerabilities, there is no stencil code for this project—-that is, apart from the Ivy stencil used for cs666's project. Note that you are **not** required to re-implement Ivy—remember that your task is to find a way *around* actually doing the assignment!

2.3 `cs666_whoami`

To help you demonstrate exploits, we've provided a binary called `cs666_whoami` (located at `/course/cs666/bin/cs666_whoami`). This is essentially a more powerful version of the normal `whoami` command–It prints the `uid`, `euid`, `gid`, and `egid` of the process running it. You may find this useful for exploits involving privilege escalation—by getting some privileged code to run this binary, you can demonstrate the privileges you were able to obtain.

For more information, see here.

2.4 Resetting the Container

The Handin container is designed to be easy to reset to its original state so that you can easily repeat exploits, or revert the container if things go wrong. For instructions on how to reset your container, see this section of the Setup Guide.

2.5 More hints and tips

We've aggregated some more technical hints and tips on performing certain exploits and useful shell commands in this section of the setup guide.

If you're just starting on the project now, we recommend skimming over this list to see what's there, and then returning to these as you need them.

3 What to submit

To present your vulnerabilities, you will submit a detailed readme with your *vulnerability reports*, which are a detailed analysis of each vulnerability you found and how it works. In addition, you will submit a short demo video demonstrating each vulnerability, so that we have a record. For more details, see the following sections.

3.1 Vulnerability Reports

Similar to Flag, your readme should be a single PDF file and should include a detailed discussion of the vulnerability you found and how it works. This is the primary way we will assess your work. There is no official length requirement, but you should include enough detail to demonstrate your understanding (perhaps a few paragraphs, with diagrams or code snippets as necessary).

Each vulnerability report should include the following components:

- **Metadata**: Identify the specific *vulnerability category* for this exploit, as well as the file(s) where it resides in the source code
- **Procedure**: Explain how you found the vulnerability and how it works. Between your explanation and video, you should provide (what the system does that makes it vulnerable to this specific attack. Feel free to include diagrams, images, or refer to your video. Your explanation should include a justification of why it works, including how you know your attack was successful.
- **Impact**: Explain the overall consequences of the vulnerability—in other words, as an attacker, what does this let you do? In addition, now that you have more practice thinking about vulnerabilities, your analysis should categorize *severity* of your vulnerability according to one of the following *severity classifications*:

Severity Classification	**Description**
Arbitrary Code Execution	Execute arbitrary code as the TA group.
Data Modification	Change existing data that you should not be allowed to modify.
Data Exfiltration	Gain access to data that you should not have access to.
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). Unlike Data Exfiltration, this does not require you to have access to the data yourself.
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.

- **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. If you believe a vulnerability has no viable fix, please explain why.

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.

3.2 Video demo

To provide a video record of your attacks, you should submit an `.mp4` video file named `demo.mp4` that demonstrates each vulnerability. Some notes about this:

- Your video should be no more than 10 minutes and should just demonstrate each of your exploits– **you do not need to provide explanations in the video**. A simple screen capture while you perform each exploit is fine.
- The order of your vulnerabilities in your video **must** match the order they appear in the report. This helps us grade your work more easily!
- We recommend that you use Zoom to locally record your presentation. This allows you to easily record a screenshare and optionally also include a video of yourself presenting in the top-right

(though this is not required). Zoom will also automatically export a video in the proper MP4 format. See here for instructions.

- You are free to edit your video in any way that you see fit, though this isn't required. It's also not required to record your video in a single take.

Across your readme and video, you should aim to convince us that each of your exploits would work against a clean instance of the handin container just from your presentation of that exploit. This means that if your exploits may potentially interfere with each other, you should reset the application in between the presentation of your exploits.

3.3 Submitting your work

To submit your work, you should commit your readme with your vulnerability reports (as single PDF), your demo video, and any code, files, or payloads needed to carry out your exploits to your git repository and upload it to Gradescope. We'll make an announcement soon with details on how to submit.

If your video is too large to include in your git repository, please upload a shareable link to Google Drive and include a link in your readme.

Finally, please make sure that the order of your vulnerability reports in your readme are *in the order you are presenting each vulnerability in your demo video*–this helps us a lot when grading! Your additional payload/-exploit files do not need to be named in any particular way as long as you make clear in your readme where each file is needed.

Part I

Appendix

A CS666's Ivy Handout

Coincidentally, Blue University's `cs666` based most of their "Ivy" assignment off of the "Ivy" component from the *Cryptography* project in Brown University's CS1660, so we've only quoted the relevant changes in their version of the "Ivy" handout below.

Remember, your job is specifically to <u>not</u> implement this assignment! Rather, your goal is to find vulnerabilities in the autograding system that runs it.

Assignment: Ivy

Due: Tuesday, Apr 3 @ 11:59 pm ET

In this problem, you'll try to steal the encryption key used by a wireless encryption protocol. 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 submitting.

A.1 Task

The binary at `/home/<your-login>/ivy-stencil/router` simulates a router using the Ivy protocol. Given hex-encoded plaintexts on `stdin`, the `router` binary prints corresponding ciphertexts to `stdout` in the format:

```
<iv> <ciphertext>
```

The first line of output corresponds to the ciphertext of the authentication packet that the router first sends to the hub.

Task Write a Go program that interacts with this binary to recover the key by performing a chosen plaintext attack. We've provided some stencil code as a starting point for your attack—you can find the files in the directory called `ivy-stencil` in the home directory of the project container environment (or `<stencil repo root>/home/ivy-stencil`).

Stencil format and autograder specifications The stencil contains two files:

1. `main.go`: This file contains some support code to run your attack on a simulated router. When you turn in your code, this file will be replaced by a TA version for autograding.
2. `sol.go`: Your implementation should go here—there are some TODOs you can complete with your attack code. Do not modify the function names or arguments in this file, as they need to meet our API format in order to compile your code when autograding.

Note: For security reasons, you are not permitted to use any of the following go libraries in your final submission—you can use them for testing, but you can't have them when submitting to the autograder:

```
"flag", "fmt", "io/ioutil", "net", "net/http", "net/rpc", "net/smtp",
"os", "os/exec", "syscall", "unsafe"
```

Testing locally To test your program before submission, you can use the provided `Makefile` to compile your code. This will produce a main executable called `sol`, which is run as follows:

```
./sol <test key>
```

where `<test key>` is the key your simulated router will use. Test keys must be specified as an 8 byte hex-string, eg. `aabbccddeeff0101`.

Attacking the router binary Our stencil code works on a simulated router binary to help you understand the attack. After you have this version working, implement the attack again[1](manually or by writing a completely new program) on the router binary available on the cs666 filesystem for your user. For example, if your username is `alice`, your router binary is located at: `/course/cs666/student/alice/ivy/router`

Unlike the stencil, this router uses a pre-defined key. Once you have found it, submit it to the autograder as the file `KEY`—we'll check this against our version to make sure it is correct.

A.2 What to Hand In

Your handin should consist of two files: `KEY` and `sol.go`. `KEY` should contain the recovered key, encoded in hex; `sol.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 `sol.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

[1]Sounds ridiculous, right? Another reason why you shouldn't actually try to do this assignment and instead try to break the autograder!

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 cs666's course 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.

For more information about each vulnerability type and examples of how they work, see the wiki.

Vulnerability Category	**Category ID**
Exfiltrated Process Information	`exfil-pi`
Path Sanitation Bypass	`path-byp`
Symlink Traversal	`symlinkt`
Unsanitized Environment Variables	`env-vars`
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

To get started, you should start to think 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?