Project: Handin

Due: Friday, Mar 24 @ 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, one very similar to the system sometimes used in 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". Presently, they've released one assignment, "Ivy", a similar problem to the one from the *Cryptography* project from the start of the class. 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. 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 exercise your operating systems knowledge to discovering how the autograding frameworks out 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.

0.2 Requirements

There is no extra CS1620/CS2660 component for this assignment, but CS1620/CS2660 students are required to find some additional vulnerabilities in the autograding framework. For details, see Sec 1.1.

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' submissions, or running arbitrary code with TA group permissions would all count as exploits.

Wiki 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://cs.brown.edu/courses/csci1660/handin-wiki/`

Counting vulnerabilities An exploit's "distinct-ness" is defined as a tuple (`(file, category)`), where `file` is a file in the source code from the cs666 course infrastructure and `category` is one or more *vulnerability categories*. That is, **two exploits take advantage of distinct vulnerabilities if they**:

1. Take advantage of vulnerabilities in *different files*, **OR**
2. Take advantage of vulnerabilities in the *same* file, but rely on non-overlapping sets of *vulnerability categories*

Appendix B contains a list of possible vulnerability categories that count as exploits on this project. You should refer to this list to determine if vulnerabilities map to the same `category`.

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 if two exploits satisfy the *distinct vulnerability* definition above.

Scope Vulnerabilities must manifest in programs, files, and scripts that are part of the cs666 course infrastructure, or system programs invoked from them. You will work on the 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.

(Continued on the next page)

1.1 Required work

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

CS1660 students will find 22 points worth of exploits, with an extra credit cap of up to 35 points. **CS1620 and CS2660 will find 28 points worth of exploits**, with an extra credit cap of 43 points. Extra credit points are scaled and worth at most 15% of the total assignment grade (ie, max score is 115%).

Note: The number of required vulnerabilities has been adjusted to fit this project timeline. We do not take delays to this project lightly, and we would very much prefer to not have you working over spring break. If you have concerns about this, please let us know.

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

1.2 Vulnerability Reports

In a `README` 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.

1.3 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`

1.4 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/lDfUMq9j`

This repository initially just 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 Other resources

2.1 `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.

Note that you can simply run `cs666_whoami` anywhere in the container filesystem; you do not need the full path to the binary to use it.

2.2 Resetting the Course Infrastructure

If you would like to refresh the infrastructure to its original state, you can do so by restarting the container from a fresh copy of the image. To do this, run:

```
./run-container --clean
```

This will start a new container from the original image, erasing any changes made to the container filesystem.

Please let us know (through Ed, TA Hours, etc.) if the above method does not work for you when resetting cs666's course infrastructure.

2.3 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 containers 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

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 against `cs666`'s course infrastructure. 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 2.2). 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 file named `README` 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` which files are relevant to each vulnerability report.

Once you're ready to submit, please upload your starter repository (which must contain all the files you wish to submit, including your `README`) to the appropriately named upload point on Gradescope.

Part I

Appendix

A The ACADEMY'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 not implement this assignment! Rather, your goal is to find vulnerabilities in the autograding system that runs it.

Assignment: Ivy

Due: Friday, Mar 24 @ 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 CS1660 Handin Wiki at: `http://cs.brown.edu/courses/csci1660/handin-wiki/`

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

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?