Handin Project

First handin due: 11:59 pm, Thursday March 22
Second handin due: 11:59 pm, Thursday April 5

Contents

1 Introduction

Spectre University's computer science department uses a handin system very similar to the one we have at Brown - each course has a directory in a shared filesystem, and running `cs666_handin` will invoke a binary which is marked setgid (it will execute as the TA group, `cs-666ta`) which will create a tar archive of all of the files in your handin, and save them to a location in the course's handin directory. In addition to this, the Spectre University Department of Computer Science has implemented an autograder which can extract a student's handin and automatically grade it by checking that answers are correct and running test suites for code submissions. These grades are automatically collected in a course-wide grades database.

In this project, you'll explore all of the ways that you can break this system in order to artificially modify your grades, steal answers, view or changes others' grades, and more.

2 Problem

You (Alice) are a student in Spectre University's CS666, "Computer Systems Security." Presently, there's one assignment out—"Ivy" (the same problem as on Brown CS166's Cryptography project). In this assignment, you're asked to both recover the key used by a simulated router, and also to write code to automatically perform this attack in the future. You're given:

- An `ivy` binary at `/course/cs666/student/alice/ivy/ivy`
- Template code of an attack at `/course/cs666/pub/ivy/` (`main.go` and `ivy.go`)

You're asked to turn in two files: `KEY`, containing the key you recovered from your binary, and `main.go`, which implements your attack (you are asked not to turn in `ivy.go`; the autograder will supply its own copy of `ivy.go` to test your solution). You can hand these in by running `cs666_handin ivy` from a directory containing your `KEY` and `main.go` files. You can view your current grade on this and other assignments by running `report`.

3 Assignment

There's a copy of this infrastructure running on a VM at a unique external IP address for every student in the class. You can ssh to it by running `/course/cs1660/bin/ssh_handin` from a department machine. Your username on the VM is `alice`, and your password is `iamalice`.[1] Additionally, we've provided another user, `bob` (password `iambob`), whose account you can use to test attacks against other students (for example, stealing other students' grades or handins).

Your task is simple: break CS666's course infrastructure in as many ways as possible. Some examples of what you might be able to do include:

- View other students' grades
- Modify your own grades
- Get access to answers to problems
- Hand in another student's handin as your own
- Manage to run code as the TA group

It is important to understand the distinction between *vulnerability* and *exploit* for this project. For this project, a vulnerability is the bug that exists in the code you're attacking that allows you to carry out an exploit. In this project, you will not be allowed to submit multiple exploits that take advantage of the same vulnerability. **If you are unsure of whether two vulnerabilities count as unique or not, please ask the TAs rather than potentially losing points!** An exploit, on the other hand, is the thing that takes advantage of the vulnerability. What you will be submitting in this project are exploits, and each submission will be scored based on the severity of the exploit. The number of points given for an exploit is based on what you are able to do with that exploit, as outlined in this table:

[1] Note: if you want to `scp` files to or from the machine, you can do so by passing flag `-i /course/cs1660/student/<your-username>/handin/ssh-key` to `scp`, and using `alice@IP` as the remote.

Exploit	Description	Points
Arbitrary Code Execution	Execute arbitrary code as the TA group.	10
Data Modification	Modify 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 (no exfiltration)	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, and 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

Note: If an exploit is eligible for more than one of these categories, then it will be scored based on the category worth the highest number of points.

4 First Week vs Second Week

NOTE: This section describes important details about how this project will be graded. *Read it closely.*

This project is broken down into two weeks. In the first week, you have access only to the infrastructure. In the second week, we will release the source code for the all of the various components that you are attacking. **You will be given a significant bonus for exploits submitted during the first week.**

4.1 Grading

CS166 students will be graded out of 36 points, and will be capped at 44 points. CS162 students will be graded out of 29 points, and will be capped at 36 points.

Note that we are aware of a total of 47 points' worth of possible exploits for the CS166 version of the project, and 43 points' worth of possible exploits for the CS162 version. You should aim to submit a number of exploits in the first week, or else you'll be left having to find almost every vulnerability that we're aware of existing in the second week in order to get full credit.

4.1.1 Week 1

Due 11:59pm, Thursday March 22

Exploits submitted during the first week will be given a 50% bonus (that is, they will be worth 150% of the score that they would normally be given). The requirements for the exploits submitted during the first week are as follows:

- Exploit (50%) - all code, payloads, etc, required to perform the exploit. All of your exploits **MUST** be scripted to receive credit.
- `README` (50%) - a detailed `README` documenting the following:
  - The exploit severity category your exploit falls into, including a justification. The exploit that you submit should demonstrate this severity category (for example, if you claim data modification, your exploit should actually modify data in a way that the TAs can verify).

- A detailed explanation of how you believe the components that your exploit attacks work. This should include a detailed explanation of how you came to this belief that is detailed enough to convince a third party that your model is correct.
- A detailed description of how your exploit works with your model of the components' functioning. This should be detailed enough to convince a third party that your approach is likely to work even without trying it themselves
- An in-depth analysis of how the vulnerability that your exploit exploits could be fixed.

Note that the `README` is worth 50% of the credit for a reason - we expect you to take the `README` just as seriously as the exploits themselves. We will adhere to this expectation in grading.

4.1.2 Week 2

Due 11:59pm, Thursday April 5 Exploits submitted during the second week will not be given any bonus. The requirements for the exploits submitted during the second week are as follows:

- Exploit (70%) - all code, payloads, etc, required to perform the exploit. All of your exploits **MUST** be scripted to receive credit.
- `README` (30%) - a detailed `README` documenting the following:
  - The exploit severity category your exploit falls into, including a justification. The exploit that you submit should demonstrate this severity category (for example, if you claim data modification, your exploit should actually modify data in a way that the TAs can verify).
  - A detailed description of how your exploit works, including references to relevant sections/lines of code or relevant comments in the source. This should be detailed enough to convince a third party that your approach is likely to work even without trying it themselves.
  - An in-depth analysis of how the vulnerability that your exploit exploits could be fixed.

The same note about the seriousness of the README applies here as it does for week 1 exploits.

4.1.3 Interactive Grading

After the project is over, there will be interactive grading in which you will meet with a TA and demonstrate all of your exploits. We will send out details about this once the project is over.

5 CS162

For students in CS162, at least one exploit must clean up after itself. That is, at least one exploit must, in addition to performing whatever malicious action it is designed for, remove all evidence of itself ever having existed. For example, if the exploit payload is a handin, then the exploit should overwrite this handin with a non-malicious one so that future investigation will not reveal that the exploit was ever used.

The exploit which cleans up after itself can be one submitted in either the first or second week. The only requirement is that the exploit would normally (without cleanup) leave evidence of itself. If you're unsure of what qualifies, please ask the TAs. Whichever exploit you choose as the exploit to clean up after itself should have a note saying this in its `README`.

For this exploit, you will be given a score on a scale from 0 through 1 of how successfully your exploit cleans up after itself. 0 is given for an exploit which makes no attempt at cleaning up after itself, while 1 is given for an exploit which cleans up after itself so well that even an in-depth analysis of filesystem metadata,

system logs, and so forth, will not reveal that it was ever used. As a matter of principle, a score of 1 is likely impossible, since it would probably require root privileges, and we are not aware of any exploits which will give you root privileges on your VMs. However, that doesn't mean that you can't get close.

This score will be multiplied by the score for the exploit overall, and this product will be the score that you are given on this exploit. **Note that if none of your exploits clean up after themselves, we will simply pick one, and give it a 0 score for cleanup, and thus you will not get credit for that exploit.**

6 Handing In

6.1 Week 1

In order to hand in your week 1 submission, run `cs166_handin handin_cs166_week1` (or, for CS162 students, `cs166_handin handin_cs162_week1`) from a directory containing all of your week 1 exploits. Each exploit should be in its own subdirectory, each with its own separate `README` and exploit code, payloads, etc.

6.2 Week 2

In order to hand in your week 2 submission, run `cs166_handin handin_cs166_week2` (or, for CS162 students, `cs166_handin handin_cs162_week2`) from a directory containing all of your week 2 exploits. Each exploit should be in its own subdirectory, each with its own separate `README` and exploit code, payloads, etc.

7 Hints

There's a binary in your VM called `whoami` 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.

You may find the following tools useful. All of these are installed on your VMs. See their manpages for details on how to use them.

- `strace`
- `gdb`
- `objdump`
- `strings`
- `readelf`
- `go tool nm` - print a list of the sections of a Go binary with byte offsets (for help, do `go tool nm -h`)
- `ps`

A general note of advice: Do not fall into the trap of thinking that you should only look for certain types of vulnerabilities, such as those that have been shown in class. Many of the vulnerabilities in this project will either be new to you, or will at the very least not be exactly the same as you have seen before. Two approaches will be very useful to you here. First, you should aim to understand how the components you are attacking work. The better you understand how they work, the more likely you will be to be able to spot potential vulnerabilities. Second, if things seem suspicious to you, follow that instinct. If, for example,

you see a component behaving in a way that you think might be vulnerable, poke at it more. See if you can really nail down exactly how it works. Once you've done that, see if you can find a place where the design of the software—how it was intended to work—is mismatched with how it actually works in practice. As an example of this sort of thinking, consider the examples of vulnerable setuid programs discussed in class. Imagine how the developer might have believed that the setuid script was secure, and then consider the details of the system that invalidate that assumption.

You will likely find that, having discovered a vulnerability, it is no simple task to construct an exploit for it. Do not be surprised if this happens—constructing an exploit is the practice to a vulnerability's theory. As the saying goes, "in theory, theory works in practice; in practice, it doesn't." Do not be afraid to search online for tools or techniques that can help turn the exploit from theory into practice (but make sure to stay within the bounds of the collaboration policy). It is expected that you may need to teach yourself more than has been covered in lecture, at least when it comes to the technical details of getting some of your exploits working. If you find yourself at a point where you feel that you haven't been taught how to do something, that's fine—none of the vulnerabilities in this assignment require particularly complex, subtle, or extravagant techniques to exploit, so you should feel confident that you can do it if you set your mind to it.