0 Introduction

After the FLAG portal fiasco, The Hotel CIT 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 \texttt{csXXX_handin} (where \texttt{csXXX} is a course number) invokes a \texttt{setgid} binary that saves all files in the current working directory in a .\texttt{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 hotel trainee, you’re currently enrolled in \texttt{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 \texttt{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 \texttt{cs666} infrastructure instead...

0.1 Requirements

CS166 students must complete the assignment described in Part \texttt{I}. CS162 students must complete the CS166 requirements as well as an additional component described in Part \texttt{II}. For CS162 students, Part \texttt{I} is worth 80\% of the project credit, and Part \texttt{II} is worth 20\% of the project credit.

0.2 Accessing the Infrastructure

Each student gets their own instance of The Hotel CIT’s infrastructure to attack. You can SSH to your unique machine by running the \texttt{cs166_ssh_handin} command from a department machine.

Your username on the VM is \texttt{alice} and your password is \texttt{iamaalice}. We’ve also provided another user, \texttt{bob} (password: \texttt{iambob}), whose account you can use to test attacks against other students. (You can use the \texttt{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.

Having distinct vulnerabilities means all of your discovered vulnerabilities belong to different vulnerability categories. Appendix B contains a list of OS vulnerability categories that we’ll accept on this project. Refer to this list to make sure you’ve found distinct vulnerabilities—you cannot receive credit for the same category more than once. You may not receive credit for two exploits that rely on the same vulnerability categories.

The source code of cs666’s course infrastructure can be found at /course/cs1660/pub/handin on the department machines.

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:

<table>
<thead>
<tr>
<th>Severity Category</th>
<th>Description</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arbitrary Code Execution</td>
<td>Execute arbitrary code as the TA group.</td>
<td>10</td>
</tr>
<tr>
<td>Data Modification</td>
<td>Change existing data that you should not be allowed to modify.</td>
<td>7</td>
</tr>
<tr>
<td>Data Exfiltration</td>
<td>Get access to data that you should not have access to.</td>
<td>6</td>
</tr>
<tr>
<td>Data Theft</td>
<td>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).</td>
<td>4</td>
</tr>
<tr>
<td>Metadata Exfiltration</td>
<td>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.).</td>
<td>2</td>
</tr>
</tbody>
</table>

CS166 students are tasked with finding 35 points worth of exploits, with an extra credit cap of 40 points. CS162 students are given a more robust version of the system, and must find 30 points worth of exploits, with an extra credit cap of 34 points.

1.2 Deliverables

For each exploit you submit, you must submit an exploit script that automatically executes your attack (see Section 1.2.1) as well as a README analyzing your attack (see Section 1.2.2).

1.2.1 Exploit Scripts

For the purposes of grading, there are some rules on how your exploits should be scripted:
• We should be able to run your attack by executing a single `exploit.sh` script using `
bash(1)`, though your `exploit.sh` may then call other scripts (or refer to any other files or payloads) if necessary.

• Section 3 describes how your handin should be structured. Pay close attention to those requirements—especially if your exploits rely on files in particular locations. If your attack relies on certain files being in a different place than in your `/home/alice` directory, your `exploit.sh` script should handle moving those files to the required locations (but you should not need to move the `exploit.sh` script itself).

• For certain exploits, it’s okay if your exploit doesn’t work 100% of the time, though your script should be consistent enough to complete the attack after running `exploit.sh` a few times.

1.2.2 README

In addition to documenting the `vulnerability category` of the exploit (see Appendix B) and the `severity category` of the exploit (see Section 1.1), your README for each exploit should document the following:

• 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.).

• 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). Please also document any utilities or languages your attack program relies on outside of the `exploit.sh` script, if applicable.

• An analysis of how the vulnerability could be fixed (include specific references to the source code as to where fixes should be applied).

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 their man pages for usage information):

- `lsof` — lists open file descriptors
- `strace` — traces system calls
- `gdb` — binary debugger
- `objdump` — displays binary information
- `strings` — prints strings in binary
- `ps` — lists processes
- `htop` — live process viewer
- `watch` — execute a program periodically
- `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.\footnote{If you want to run the `sftp` command yourself or want to access The Hotel CIT’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`.}
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 Handing In

Your handin should consist of several directories named `exploit#`, where # is a number. Each directory should contain a README, a `exploit.sh` file that executes your attack when run, and an optional files sub-directory that contains any files that are necessary to run your exploit. When grading, we will upload your handin **directly** into `/home/alice` on a fresh VM. Make sure that your exploits work with this setup—especially if they rely on files in particular locations. For example, your handin might be structured as:

```
/some/containing/folder  # this is the directory from which you should create a zip
  |- /exploit1
  |    |- README
  |    |- exploit.sh  <-- when run, executes your attack
  |    |- /files
  |    |    |- ...  <-- any necessary files for running the exploit go here
  |- /exploit2
  |    |- README
  |    |- exploit.sh  # if no files needed for attack, ‘files’ directory not necessary
  |    |- ...
```

You should then upload a zip archive named `handin.zip` to Gradescope. To create a zip of the contents in your current directory, you can run:

```
zip -r handin.zip  
```

3.1 Interactive Grading

To give you the opportunity to exercise your security presentation skills, we will host 20-minute interactive grading sessions after the project deadline in which you will demonstrate all of your exploits to a TA.

- In your presentation, you’ll walk through each of your exploits in a way that convinces us that your exploit allows you to perform a normally unauthorized action in the system. In addition to running your exploit, you should cover each of the components of your README—the severity category, why the system is vulnerable, what your attack does, and how the vulnerability can be fixed.

- As stated in Section 3, we’ll load the contents of your handin directly into `/home/alice` on a fresh VM during your interactive grading. You may **not** load any files that weren’t handed in onto your VM during the grading, so make sure all the necessary files are in your handin. However, you may refer to your README or other notes as a memory aid during your presentation.

- The TA may ask you questions that your presentation raises, so make sure to account for this in your presentation time—we will only give full credit for exploits presented within the 20 minute window.

We will send out more details on the interactive gradings towards the end of the project.
Part II
Linking

CS162 students must complete the following additional problem.

In this problem, you will explore another security hole in The Hotel CIT’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 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 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 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.

---

2 https://en.wikipedia.org/wiki/Honeypot_(computing)
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/alice/student/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 will not be asked to demonstrate your Linking exploit during the interactive grading.

---

\(^4\)-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 Hotel CIT’s system).
Part III

Appendix

A The Hotel CIT’s Ivy Handout

Coincidentally, THE HOTEL CIT’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: Monday, March 16, 2020 @ 11:59 pm

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 OS 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 Hotel CIT’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.

- Exfiltrated Process Information
- Buffer Overflow / Memory Corruption
- Path Sanitation Bypass
- Symlink Traversal
- Unsanitized Environment Variables
- Outdated System Components with Known Vulnerabilities (i.e. `bash` or OS kernel vulnerabilities)
- Misconfigured Blocklists / Safelists
- TOCTOU (Race Condition)
- Misconfigured File / Directory Permissions
- Escaping `chroot` or Sandbox

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.