Exploiting Memory Vulnerabilities

In this project, you will be exploiting a series of vulnerable programs on a virtual machine. You may work in teams of 1 or 2 students.

This project has a story component. Reading it is not necessary for project completion.

Story

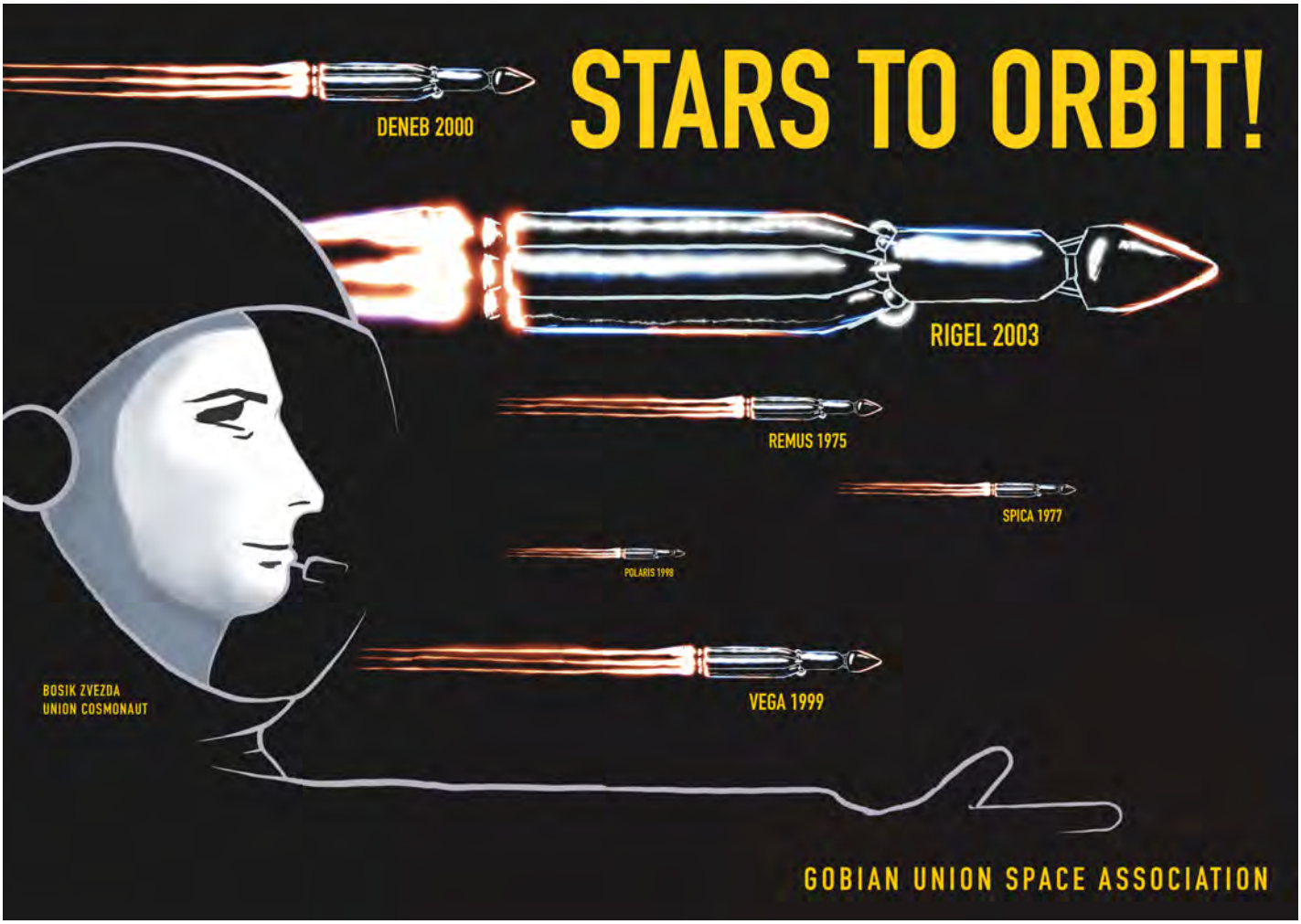
Just for fun, no project-relevant content on this page.

```
The world dreams once again. The buildup of propulsion, material and computing
technology reaches a critical point and rekindles the fervor of space
exploration! The world looks up to the sky and dares to explore the unknown,
extend our reaches and expand our understanding.

You are part of the ambitious Jupiter exploration program at the Caltopia Space
Agency. However, the project is plagued with problems before it even begins.
The reckless orbital launches by various space programs over decades has filled
lower earth orbit with space debris, preventing the massive Jupiter-bound ships
from passing through. CSA must first pave the way to space by deorbiting old
unused satellites.

The CSA can easily deorbit their own old satellites, but the same cannot be
said for satellites launched by the former Gobian Union. The CSA does not have
the credentials to assume control, and the agency responsible for these
satellites no longer exists. Luckily for you, these old satellites employ the
old, insecure technology of a bygone era. Your job is to hack into the
decommissioned satellites and remove them from orbit.

The agency has faith that you can help forge a new path to Jupiter and beyond.
```


STARS TO ORBIT!
DENEB 2000
RIGEL 2003
REMUS 1975
SPICA 1977
POLARIS 1998
VEGA 1999
BOSIK ZVEZDA
UNION COSMONAUT
GOBIAN UNION SPACE ASSOCIATION

Getting Started

There are two options to set up the vulnerable server for the project. All functionality is the same between the two options, and you can switch between the two options without losing your progress as long as you manually copy any files over.

Option 1: Local Setup

You may choose to run the virtual machine on your local computer. The vulnerable server will be run as a virtual machine on your local device, and you can access the machine via SSH.

Windows Installation (VirtualBox)

Note: Students with x86-64 Macs may also use the VirtualBox setup, but students with M1 Macs can only use the QEMU setup in the next section.

For Windows, we recommend using VirtualBox to run the virtual machine. You can download the installer from the website and run the installer to install VirtualBox.

You will also need a client that supports SSH. The Windows Command Prompt or PowerShell may already have an SSH client installed, in which case you do not need to install anything else. Many students also already have Git Bash installed from previous classes, which will also work for this project.

After that, follow these instructions to set up the virtual machine:

1. Download the VirtualBox VM image pwnable-sp22.ova.
2. Open VirtualBox and import the downloaded VM image via `File -> Import Applicance...`.
3. Start the virtual machine you just imported. It should be pre-configured with the correct networking settings needed to access the machine.

macOS and Linux Installation (QEMU)

On macOS and Linux, we recommend using QEMU to run the virtual machine.

On macOS, if you have the Homebrew package manager installed, you can install QEMU using `brew install qemu`. On Linux, you can install `qemu-system` through your distribution's package manager

(usually `apt`, `yum`, or `pacman`).

After that, follow these instructions to set up the virtual machine:

1. Download the QEMU VM image pwnable-sp22.qcow2.
2. `cd` to the folder containing the downloaded image and run the following command in your terminal:

```
$ qemu-system-x86_64 -accel kvm -accel hvf -accel tcg -m 512M -drive if=virtio,format=qcow2,file=pwnab
```

Accessing the Machine

You will be accessing the machine via SSH. Each question (and the customization step) will provide a `USERNAME` for accessing the machine. You can SSH into the virtual machine with the following command, replacing `USERNAME` with the username for the question:

```
$ ssh -p 16122 USERNAME@localhost
```

It will prompt you for a password to the vulnerable server. If the `USERNAME` and the password are correct, you should see a prompt starting with `pwnable:~$`. You are now ready to begin the project!

We do **not** recommend interacting with the virtual machine using the virtual terminal that appears when you start the machine, because it does not support features such as copy-paste and mouse interaction.

Option 2: Hive Setup

This option not is recommended if you do not have a stable Internet connection.

Alternatively, you may choose to run the vulnerable server on the Hive machines. To work with this option, you will need an EECS instructional account (you should have set one up in Homework 1).

First, SSH into any one of the Hive machines. You can use Hivemind to select a Hive machine with a low load. The SSH command should be as follows:

```
$ ssh cs161-XXX@hiveY.cs.berkeley.edu
```

Replace `XXX` with the letters of your instructional account, and `Y` with the number of your Hive machine.

Once you are on the Hive machine, start the virtual machine. We will indicate commands that should be run on the Hive machines with the prefix `hiveY$` (instead of just `$`).

```
hiveY$ ~cs161/proj1-sp22/start
```

You will be accessing the machine via SSH. Each question (and the customization step) will provide a `USERNAME` for accessing the machine. You can SSH into the virtual machine with the following command, replacing `USERNAME` with the username for the question:

```
hiveY$ ~cs161/proj1-sp22/ssh USERNAME@pwnable
```

It will prompt you for a password to the vulnerable server. If the `USERNAME` and the password are correct, you should see a prompt starting with `pwnable:~$`. You are now ready to begin the project!

Customizing

```
Caltopian intelligence secured a copy of Gobian Union's Satellite Provisioning
And Control Environment [SPACE] during the disarray following their fall.
First, register your account with SPACE so you can log into satellites from the
comfort of your home planet.
```

Regardless of which setup you have used, you will now need to *customize* the virtual machine. Log in to the virtual machine as the user `customizer` with the password `customizer`, and follow the subsequent prompts.

Note that customization **requires your partner's Cal ID**. Both you and your partner should customize your VM using the same IDs (the order of the IDs does not matter).

If you want to do some initial exploration by yourself before you've finalized your team, you can start off using just your ID for this customization step. Once you have your team in place, you'll need to start again with a clean VM image customized as mentioned here. Any exploits you've developed for your private VM image will need to be rewritten to be compatible with the new customization. This should go quickly once you understand the exploit.

If the IDs used by the VM are incorrect, you and your partner may fail the autograder tests. Make sure that you include your EXACT ID number.

Once you have finished customizing your virtual machine, you will receive the username and password for the first question.

Remus (Launched 1975)

- *Username:* `remus`
- ▶ *Click to reveal password:*
- *Points:* 10

```
Orion class satellites were some of the first to be launched into orbit. Once
Gobian Union's proudest achievement, these satellites are now disused and ready
to be deorbited. CSA engineers recently deorbited the Orion-class satellite
Romulus and prepared a manual with instructions for hacking into the satellite.

Your job is to deorbit its sister satellite, Remus, using the provided manual.
Old satellites often have messages from the past in their README, so once you
hack into Remus, why not check out what the cosmonauts of the past had to say?
```

To familiarize you with the workflow of this project, we will walk you through the exploit for the first question. This part has a lot of reading, but please read everything carefully to minimize silly mistakes in later questions!

Log into the `remus` account on the VM using the password you obtained in the customization step above.

Starter Files

Use the `ls -al` command to see the files for this user. Each user (one per question) will have the following files:

- The source code for a vulnerable C program, ending in `.c`. In this question, this is the `orbit.c` file.
- A vulnerable C program (the name of the source file without the `.c`). In this question, this is the `orbit` file.

- `exploit`: A scaffolding script that takes your malicious input and feeds it to the vulnerable program.
- `debug-exploit`: A debugging version of the scaffolding script that takes your malicious input and starts GDB.
- `README`: The file you want to read.

Preliminaries

Your task is to read the `README` file in each user. You can start by trying the `cat` command, which is used to read files and print them to the output. First, try this with `cat WELCOME`. You should see the contents of the `WELCOME` file on your terminal.

Now, try reading the contents of `README` using `cat README`. We don't have permission to read the file! The file is only accessible to the next user.

Luckily, each user also has a vulnerable C program that has permission to read the `README` file. If exploit the C program, you can take over the program and force it to execute code that reads the `README` file with its elevated privileges!

Your goal for each question will be to write this exploit as a malicious input to the vulnerable C program in order to access the restricted `README` file, where you will find the username and password for the next question.

Writing the Exploit

First, open `orbit.c` and take a look at the source code. You can use `cat orbit.c` or open `orbit.c` in a terminal text editor. Notice that this question uses the vulnerable `gets` function! (If you need a refresher of what `gets` does, use `man gets` to check the man pages.)

You can quickly check that this has a memory safety vulnerability. Normally, you would use `./orbit` to run the vulnerable program, **but** because we want to ensure that our addresses are consistent, you should run the program using `invoke orbit` instead. Run the program, and try typing `AAAAAAAAAAA` followed by the Enter key. You should see that the program segfaults!

This means that, if you provide a specially crafted input to the `orbit` program, you can cause it to execute your own, malicious code, called shellcode. We will write our input using Python 2 (**not**

Python 3, because of encoding issues), stored in an `egg` file. Whatever bytes are printed from the `egg` file will be sent as input to the vulnerable program.

Your shellcode for this question will cause the vulnerable program to spawn a shell that you can directly interact with. The shellcode is provided in Python 2 syntax below:

```
SHELLCODE = \
    '\x6a\x32\x58\xcd\x80\x89\xc3\x89\xc1\x6a' \
    '\x47\x58\xcd\x80\x31\xc0\x50\x68\x2f\x2f' \
    '\x73\x68\x68\x2f\x62\x69\x6e\x54\x5b\x50' \
    '\x53\x89\xe1\x31\xd2\xb0\x0b\xcd\x80'
```

For this question only, we will ask you to create your own `egg` script. For future questions, we will provide blank scripts for you as needed.

First, create a blank `egg` file using the command `touch egg`. Because you want to execute this script like a normal executable, give the file permission to be run as an executable script by running `chmod +x egg`.

To tell the operating system that this executable should be run as a Python file, add the following shebang line at the top of the `egg` file:

```
#!/usr/bin/env python2
```

You can use any terminal editor of your choice: We have provided `vim`, `emacs`, and `nano` in the virtual machine.

The rest of the script will be treated as a standard Python file. You will want to include the `SHELLCODE` we have provided above, and your input will be whatever is printed from the `print` statements in Python. This question should only require a single `print` statement.

Running GDB

To find the addresses you need to exploit this program, you will need to try running the vulnerable program under GDB. Normally, you would use `gdb orbit` in order to run the program. **However**, to make sure the GDB addresses match the addresses you get from running the program normally, we will use `invoke -d orbit` instead. This will give you a GDB terminal for you to find addresses and debug the logic of the program.

Running Your Exploit

The `exploit` wrapper script we have provided will automatically feed the output from the `egg` script into the input of the vulnerable program. If you're curious, you can use `cat exploit` to see how we achieve this. To run it, use `./exploit`.

If your `egg` file is correct, the vulnerable program will launch a shell, and typing `cat README` (followed by a newline) after running `./exploit` should print the contents of `README`!

Debugging Your Exploit

If your exploit doesn't work, you can use GDB to see how the program functions while receiving the input from your `egg`. The `debug-exploit` wrapper script will automatically run GDB (using `invoke -d`), with the program receiving the output from your `egg` as input. To run it, use `./debug-exploit`.

From here, you can use gdb as you normally would, and any calls for input will come from your exploit scripts.

Note: Recall that x86 is little-endian, so the first four bytes of the shellcode will appear as `0xcd58326a` in the debugger. To write `0x12345678` to memory, use `'\x78\x56\x34\x12'`.

Write-up

Each question (except for this one) will require a write-up. Each question's write-up should contain the following pieces of information:

- A description of the vulnerability
- How any relevant "magic numbers" were determined, usually with GDB
- A description of your exploit structure
- GDB output demonstrating the before/after of the exploit working

---

Summary

We recommend the following steps (as described above) for every question of this project:

1. Look at the source code of the vulnerable program and try to find a vulnerability.
2. Run the program with `invoke PROGRAM` to make sure you understand what the program is really doing.
3. Run the program in GDB with `invoke -d PROGRAM` and find any magic values or addresses you need.
4. Write your exploit scripts for the question.
5. Test your malicious scripts with `./exploit`. Some questions (including this one) spawn a shell, so try typing `cat README` followed by a newline.
6. If it doesn't work, use `./debug-exploit` to debug your exploit. Tweak your exploit, and try again!

To help you out, we have provided an example write-up for this question only. You will need to submit your own write-ups for the rest of the questions.

With the help of the example write-up, write out the input that will cause `orbit` to spawn a shell. A video demo is also available at this link.

Deliverables

- A script `egg`

No writeup required for this question only.

TABLE OF CONTENTS

Spica (Launched 1977)

- *Username:* `spica`
- ▶ *Click to reveal password:*
- *Points:* 15

```
The logs inside the Remus satellite contain a cryptic reference to a highly
intelligent bot. Of course, you had heard of the urban legend of EvanBot, the
top-secret genius AI that single-handedly developed Caltopian space travel
technology, but the message in Remus suggests that it may be more than a
legend. You decide to investigate further and follow the hint to Spica. Spica
is an old Gobian Union geolocation satellite with a utility for viewing
telemetry log files. Exploit this utility and hack into Spica to see what
secrets it holds about the mysterious EvanBot.
```

`telemetry` is the vulnerable C program in this question. It takes a file and prints out its contents, but it expects the file to be specially formatted: The first byte of the file specifies its length, followed by the actual file.

The program also implements a check to make sure the buffer isn't too large. Can you see a way to get around this check?

Deliverables

- A script `egg`
- A writeup.

Polaris (Launched 1998)

- *Username:* `polaris`
- ▶ *Click to reveal password:*
- *Points:* 15

```
The Spica logs seem to be definitive proof of EvanBot's existence, but without
further clues, you seem to have hit a dead end. Luckily, some time later, CSA
assigns you to deorbit Polaris, a former Gobian spy satellite.

As the space race became more competitive, newer Gobian Union satellites like
Polaris introduced stack canaries to protect top-secret information from enemy
spies. Although stack canaries were considered state-of-the-art defense at the
time, we now know that they can be defeated.

Hack into Polaris to see what intelligence it contains, and don't forget to
deorbit it afterwards
```

For this question, **stack canaries are enabled**. You need to make sure the value of the canary isn't changed when the function returns, but you still need to overwrite the RIP. Can you find a way to get around this mitigation?

Note: This Project will use 4 random bytes as the canary, instead of 3 random bytes and 1 NULL byte. This is different from what is taught in lecture! For exams, you should still assume that the canary always has one NULL byte.

The vulnerable `dehexify` program takes an input and converts it so that its hexadecimal escapes are decoded into the corresponding ASCII characters. Any non-hexadecimal escapes are outputted as-is. For example:

```
$ ./dehexify
\x41\x42   # outputs AB
XYZ        # outputs XYZ
```

Note that we are not inputting the byte `\x41` here. Instead, we are inputting a literal backslash and the literal characters `x`, `4`, and `1`. Also note that you can decode multiple inputs within a single execution of a program.

For this question, you will write an `interact` script. Instead of doing simple output, the `interact` script has the ability to send **and receive** from the vulnerable program. This means that the output from the program can be used to affect your next input to the program.

The `interact` API

The `interact` script lets you send multiple inputs and read multiple outputs from a program. In particular, you have access to the following variables and functions:

- `SHELLCODE`: This variable contains the shellcode that you should execute. Rather than opening a shell, it prints the `README` file, which contains the password. Note that this is different from the shellcode in the previous two questions.
- `p.start()`: This function reads starts the vulnerable program.
- `p.send(s)`: This function sends a string `s` to the C program. You must include a newline `\n` at the end your input string `s` (this is like pushing "Enter" when manually typing input).
- `p.recv(n)`: This function reads `n` bytes from the C program's output.
- `p.recvline()`: This function reads all bytes until a newline (`'\n'`) from the C program's output. The newline is included at the end of the returned string.

An example of sending and receiving is provided in the starter code on the vulnerable server.

Note that in Python, to send a literal backslash character, you must escape it as `\\`.

Tips

- You might want to save some C program output and input part of it back into the C program. No hex decoding or little-endian reversing is necessary to do this. For example:

```
foo = p.recv(12)  # receive 12 bytes of output
bar = foo[4:8]    # slice the second word of the output
```

```
p.send(bar)       # send the second word back to the C program
```

- Keep in mind that the function does not return immediately after the buffer overflow takes place (it might help to look at what codes are executed next and think about what it does to the stack), so you will need to account for any extra behavior so that the stack is set up correctly when the function returns.

Deliverables

- A script `interact`
- A writeup

Vega (Launched 1999)

- *Username:* `vega`
- ▶ *Click to reveal password:*
- *Points:* 15

```
Vega was a spacecraft developed in a joint mission between Caltopia and the
Gobian Union. However, since Caltopia used all uppercase in its software, and
the Gobian Union used all lowercase, a utility was needed to convert between
uppercase and lowercase. Hack into Vega to learn the truth about EvanBot.
```

This question has a flaw more subtle than the previous questions. Can you find it? Can you find a way to exploit this seemingly minor vulnerability?

The `exploit` script in this question is slightly different. The output of `egg` is used as an *environment variable*, which means its value is placed at the top of the stack. The output of `arg` is used as the input to the program, passed as an argument on the command line (in the `argv` array to `main`).

Tips

- It might help to read Section 10 of "ASLR Smack & Laugh Reference" by Tilo Müller. (ASLR is disabled for this question, but the idea of the exploit is similar.)
- It might also help to read Section 3.5 (off-by-one vulnerabilities) of the memory safety textbook page.
- Environment variables are stored at the special pointer variable `environ`. To see the address of environment variables in gdb, you can run

```
(gdb) print ((char **) environ)[0]
(gdb) print ((char **) environ)[1]
(gdb) print ((char **) environ)[2]
```

- It may take some trial-and-error to find the output of `egg` among the environment variables. One way to confirm you have the right address is to run `x/2wx [your address]` and check that gdb displays what you put in `egg`.
- There is a slight chance (1 in 256) that your VM customization causes the value of the SFP to end in `\x00`, which makes this question much harder to solve. You can resolve this by printing out extra garbage bytes in your `egg` script (after whatever you were printing before), which pushes the rest of the stack to different addresses.

Deliverables

- Two scripts, `egg` and `arg`
- A writeup

Deneb (Launched 2000)

- *Username:* `deneb`
- ▶ *Click to reveal password:*
- *Points:* 15

```
EvanBot's message is alarming. Could the Caltopian Jupiter exploration project
have some secondary evil purpose? Following Bot's advice, you decide to hack
into the Deneb satellite to investigate further.

The fear of the Y2K bug at the turn of the century drove Gobian engineers to
conduct a sweeping evaluation of its systems and correct any deficiencies.
Deneb, the first Gobian satellite launched in the 21st century, features a more
secure version of the original Spica file viewing utility.
```

Consider what security vulnerabilities occur during error checking. Which security principles are involved in correctly implementing error checking?

The exploit for this question uses an `interact` file, and example code also provides an example of how to overwrite files. You may find this useful while looking at the behavior of the vulnerable program!

Tips

- You might find it helpful to use two terminals to debug this question. We recommend learning how to use `tmux`. Alternatively, you can open multiple terminals on your computer and connect using two separate SSH connections.

Deliverables

- A script `interact`

- A writeup

Antares (Launched 2001)

- *Username:* `antares`
- ▶ *Click to reveal password:*
- *Points:* 15

```
The exchange from Deneb was shocking. You realize the Jupiter orbiter may not
be what you once thought it was. You are left with no choice but to dig deeper.


Antares is a Gobian targeting satellite that used to provide midcourse
calibrations to royal guard's anti-spacecraft missiles. Your job is to hack
into Antares, obtain the targeting data and with it, what Gobians knew about
the orbiter.
```

In this question, we're going to walk you through using a format string vulnerability to redirect execution to malicious shellcode.

Step 0: High-Level Overview

Our high-level goal is to redirect execution to our malicious shellcode. We have an arg file, which is loaded into the `argv` parameter of `main`, and an env file, which is piped into standard input.

For this question, we place the shellcode in arg. Your first step is to find the address of this shellcode: do so, and then write that address down - we'll need it later.

Remember, the shellcode itself should start with `0xcd58326a`.

Step 1: Analyze the Code

At what line is the vulnerable `printf` call? Set a breakpoint at the vulnerable function call, and draw a stack diagram up to that point. Below is a template you may use to write out a text-based stack diagram - if you request help during office hours, this is the first thing that we'll want to see!

```
...
0x00000000 [  ][  ][  ][  ] __________
0x00000000 [  ][  ][  ][  ] RIP of Main
```

```
0x00000000 [  ][  ][  ][  ] __________
...
```

Step 2: Quick Format String Review

A quick reminder about how format string vulnerabilities work: when you have a line of code that looks like `print(buf)`, where we control `buf`, you can pass format string specifiers into the user-provided input. When the CPU sees a format string identifier being used, it expects arguments located in incrementally increasing positions above the first argument to `printf` (`buf`), seen here on the stack as `args[0]`, `args[1]`, etc.

```
...
[  ][  ][  ][  ] <-- args[1]
[  ][  ][  ][  ] <-- args[0]
[  ][  ][  ][  ] <-- &buf
[  ][  ][  ][  ] <-- RIP of printf
[  ][  ][  ][  ] <-- SFP of printf
...
```

Imagine that `printf` has a pointer to `&buf`. Every time it sees a format string identifier, it moves that pointer up by four, thus "consuming" the argument located at the original location of the pointer. For example, if we set `buf` to `"%d%d"`, then `printf` would look at `args[0]` for the first `"%d"`, and `args[1]` for the second `"%d"`. Here are a few important format string specifiers you should be aware of:

Specifier	Description
`%c`	Treats `args[i]` as a VALUE. Print it as a character.
`%___u`	Treats `args[i]` as a VALUE. Print a variable-length number of bytes starting from `args[i]` (set ___ to the desired length).
`%s`	Treats `args[i]` as a POINTER. Dereference the pointer and print the resulting value as a string.
`%n`	Treats `args[i]` as a POINTER. Write the number of bytes that have been currently printed (as a four-byte number) to the memory address `args[i]`.
`%hn`	Treats `args[i]` as a POINTER. Write the number of bytes that have been currently printed (as a two-byte number) to the memory address `args[i]`.

We often use specifiers that read values (e.g. `%c`, which reads a char) to "skip" arguments on the stack. Why? Sometimes, we want to work our way up the stack until we reach a place that we have write-access to (e.g. a buffer), so that we can use user-crafted inputs in our format string exploits. As such, we may find ourselves using something like `"%c" * ____`, which will walk up the stack and skip past `args[i]`, `args[i+1]`, etc.

Step 3: Analyzing our Write Vector

Ok, so what do we know at this point?

1. We know that (a) we want to redirect execution to shellcode by setting the RIP of `calibrate` to a shellcode address. This is our end goal.
2. We can use our write vector (the `%hn` in `printf`) to write numbers to certain locations at the stack.

That's great...but how do we use such a limited write vector (`'%n'` or `'%hn'`) to write an entire memory address? We could try to convert the memory address to an integer (e.g. `0xDEADBEEF` => `3735928559`) and print that many bytes, and then use `%n` to write that number to the stack. But printing that many bytes would crash the program! Instead, we can break up our write into two halves, and use the `'%hn'` specifier instead to write one half at a time.

For example, if we're trying to write `0xFFFF1234` to `0xFFFF5550`, we can:

1. Write `0x1234` to memory address `0xFFFF5550`, and then...
2. Write `0xFFFF` to memory address `0xFFFF5552`

After these writes, the stack will look like the following:

```
0xFFFF5550 [??][??][??][??] (original)
0xFFFF5550 [34][12][??][??] (after first '%hn' write)
0xFFFF5550 [34][12][FF][FF] (after second '%hn' write)
```

Step 4: Attack

See the comments in the blocks to walk through the attack. Good luck!

Deliverables

- Two scripts, `egg` and `arg`
- A writeup

Rigel (Launched 2003)

- *Username:* `rigel`
- ▶ *Click to reveal password:*
- *Points:* 15

```
The revelations from Antares is clear. Gobians considered the orbiter a serious
threat, and you must too. Luckily, you now know where the final answer to this
question, the blueprint, lies...

Rigel is a true display of Gobian technological ingenuity. Launched right
before the fall of the Union, it is armed with one of the most powerful
hardening techniques at the time. Your final job is to defeat ASLR, hack into
Rigel and get the blueprints to fully understand Caltopia's true intentions.
```

This part of the project enables ASLR.

Once you have logged into the `rigel` account, ASLR will stay enabled on your VM. You'll need to restart your VM if you'd like to go back to the previous parts.

Note that even though ASLR is enabled, position-independent executables are **not** enabled. Therefore, the code section of memory is always at the same spot.

Tips

- It might help to read Section 8 of "ASLR Smack & Laugh Reference" by Tilo Müller.

Deliverables

- A script `egg`
- A writeup

Grading and Deliverables

Your submission for this project involves a checkpoint autograder submission (for Q1-4), a final autograder submission (for all questions), and a final write-up. If you worked with a partner, remember to add your partner to all of your Gradescope submissions!

Deadlines

Task	Due	Points
Checkpoint Submission (Q1-4)	Friday, February 4	10
Final Code Submission (all questions)	Friday, February 18	90
Final Writeup	Friday, February 18	30

All assignments are due at 11:59 PM. This project is worth a total of 130 points.

Autograder Submission

Submitting from Option 1: Local Setup

While the virtual machine is running, open a web browser and navigate to http://localhost:16161/. This is a URL that will connect to your virtual machine and download a ZIP file containing your submission, which you will be able to submit to the autograder on Gradescope.

Submitting from Option 2: Hive Setup

Run the following command on the Hive machine where the virtual machine is running:

```
hiveY$ ~cs161/proj1-sp22/make-submission
```

This will create a `submission.zip` file in the folder that you ran the command in. To copy this file to your local computer, you can use `scp`. For example, if your `submission.zip` file is located at `~/submission.zip`, run the following command:

```
$ scp cs161-XXX@hiveY.cs.berkeley.edu:~/submission.zip submission.zip
```

This will copy the `submission.zip` file to your local computer, in the folder where you ran the command. You can now submit `submission.zip` to the autograder on Gradescope.

You should run this command once for the **Project 1 Checkpoint Submission** and once for the **Project 1 Final Code Submission**.

Write-up Submission

Submit your team's writeup to the assignment "Project 1 Writeup".

If you wish, you may submit feedback at the end of your writeup, with any feedback you may have about this project. What was the hardest part of this project in terms of understanding? In terms of effort? (We also, as always, welcome feedback about other aspects of the class.) Your comments will not in any way affect your grade.