Weaver
Fall 2019

CS 161
Computer Security

Project 1

Due: September 20th, 2019

Version 21.00.00.00

Preamble

You may work in teams of 1 or 2 students.

In this project, you will be exploiting a series of vulnerable programs on a virtual machine. In order to aid in immersion, this project has a story. It is not necessary to read the story in order to do the problems.

> We use a shaded box to denote story which is not necessary for completing the project.

NOTE: You are only allowed to perform attacks against targets in your own virtual machine. It is a violation of campus policy and the *law* when directing attacks against parties who do not provide their informed consent!

> It is a time of rebellion. The evil empire of Caltopia oppresses its people with relentless surveillance, and the emperor has recently unveiled his latest grim weapon: a supremely powerful botnet, called *Calnet*, that aims to pervasively observe the citizenry and squash their cherished Internet freedoms.
>
> Yet in the enlightened city of Berkeley, a flicker of hope remains. The brilliant alumnus Neo, famed for his hacking skills, has infiltrated the empire's byzantine networks and hacked his way to the very heart of the Calnet source code repository. As the emperor's dark lieutenant, Lord Dirks of Leland Junior University, attempts to hunt him down, Neo feverishly scours the Calnet source code hunting for weaknesses. He's in luck! He realizes that Lord Dirks enlisted ill-trained CS students from Leland Junior University in writing Calnet, and unbeknownst to the empire, the code is assuredly not memory-safe.
>
> Alas, just as Neo begins to code up some righteous exploits to pwn Calnet's components, a barista at the coffeeshop where Neo gets his free WiFi betrays him to Lord Dirks, who swoops in with a SWAT team to make an arrest. As the thugs smash through the coffeeshop's doors, Neo gets off one final tweet for help. Such are his hacking skillz that he crams a veritable boatload of key information into his final 280 characters, exhorting Berkeley University's virtuous computer security students to carry forth the flame of knowledge, seize control of Calnet, and let freedom ring throughout Caltopia ...

Getting Started

> Neo expects your team to develop exploits for vulnerabilities in Calnet's components. All you have to go by are your wits, your grit, and Neo's legacy: guidelines on how to proceed, and a virtual machine (VM) image that contains the vulnerable code samples.

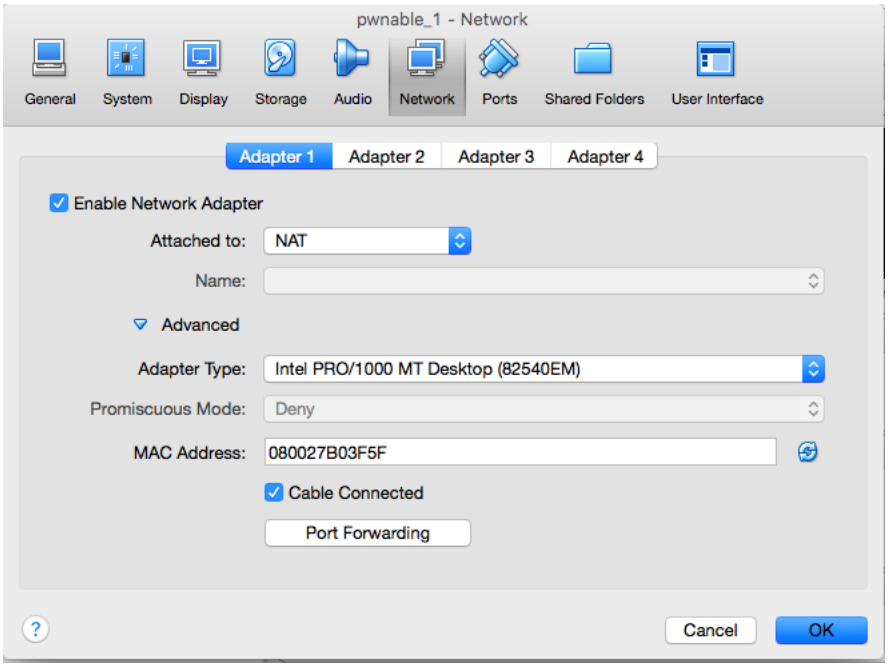
Recommended Setup: VirtualBox

To begin the project, you will need to set up a virtual machine. You will need the following programs installed on your computer:

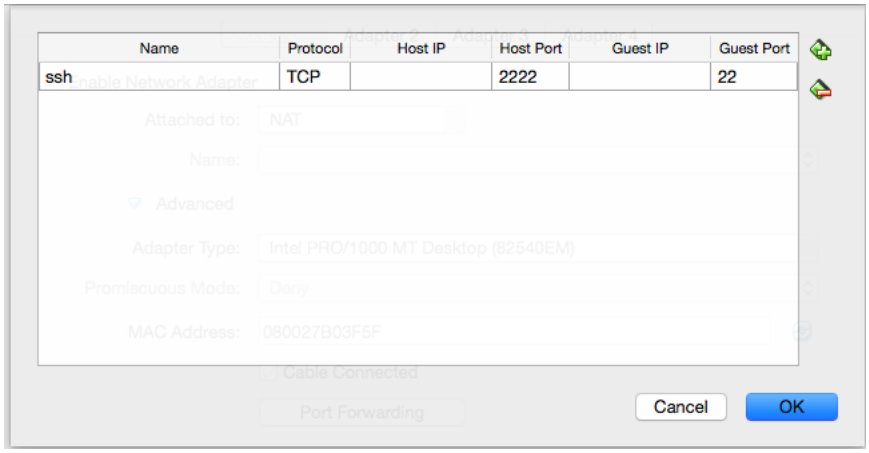
1. VirtualBox
2. A text editor
3. An SSH client (on Windows, use Putty or Git Bash)

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-fa19.ova) via `File -> Import Applicance`.

Make sure your network is configured correctly. Click your VM's settings: under `Network -> Adapter 1`, make sure the first NAT adapter is enabled and open the advanced settings.



Click the `Port Forwarding` button and ensure that you have a rule to forward from 16119 on your host to port 22 on the VM. The image below shows that port 2222 is being forwarded. Make sure that yours shows port 16119.



You can now start the VM, in which you will run the vulnerable programs and their exploits.

FAQ

Question: I configured the VM wrong! What do I do?

Answer: Just repeat the steps above. You can customize the VM without losing any of your files. However after the VM's customization changed, old exploits you created may no longer work.

Question: My partner and I configured the VM the same way, but they are not syncing?

Customizing just means that they are configured the same way, not that they will sync together. You and your partner are running on two identical, but distinct, VMs. You'll need to share your work via some out-of-band process.

Question: I get the error message: "INTERNAL ERROR! POST ON PIAZZA. MISSING IDENTITY"?

Answer: Try recustomizing the VM. If this fails, make a Piazza post.

Backup Setup: the Hive

Neo does not recommend this setup, but if you are having significant difficulty running VirtualBox on your local machine, this will work out-of-the-box on many configurations: most Linux systems, default macOS, Windows Linux Subsystem, Git Bash, and so on.

To start the VM, execute the following commands in your terminal:[1]

```
# Replace XXX with last three letters of your instructional account
$ u=XXX
$ ssh -t cs161-$u@hive$((36#${u:2}%26+1)).cs.berkeley.edu \~cs161/proj1/start
```

The `$((36#${u:2}%26+1))` portion load-balances you on a different hive machine, depending on the last letter of your login. Sometimes some of the hive machines are offline. In this case, you can use Hivemind to select a different hive machine which is online.[2] For best experience, try to choose one not used by too many other students.

If everything works successfully, a lot of output will scroll by (from the virtual machine booting up). Finally, you should see the following prompt:

```
Welcome to Alpine Linux 3.8
Kernel 4.14.89-0-virt on an i686 (/dev/ttyS0)

pwnable login:
```

Normally when you are done with the virtual machine, 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 commands on your local computer:

```
# Replace XXX with last three letters of your instructional account
# Repalce YY with the number of the hive machine which the VM is on
$ u=XXX
$ ssh cs161-$u@hiveYY.cs.berkeley.edu \~cs161/proj1/stop
```

[1] This command simply SSHes you into one of the hive machines (determined by the last letter of your login), and then starts the program `~cs161/proj1/start`.

[2] The command for this: `ssh -t cs161-$u@hiveYY.cs.berkeley.edu ~cs161/proj1/start`, where YY is the number of whichever hive machine you've chosen.

FAQ

Question: When I try to SSH in, I get the message "WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!"

Answer: We will learn more about SSH's trust-on-first-use model later in the semester. For now, just run the command `ssh-keygen -R hive$((36#${u:2}%26+1)).cs.berkeley.edu`. This removes the old host identification, allowing you to use the new one.

Question: When I try to SSH in, I get the message "Connection refused."

Answer: Check your Internet connectivity. If you are on campus, ensure you are connected to Airbears2 and not CalVisitor. Also try using a different Hive machine.

Question: My hive machine crashed while my VM was running on it, and now I can't stop it.

Answer: You can actually run the `stop` command on any hive machine, but it's impolite. Running it on a different hive machine leaves the old VM running, which is harmless but doesn't free the resources. But if the hive machine actually crashed, this is a moot point, and you should feel free to run `stop` on any available hive machine.

Customizing

> A tweet from Neo assures you that given its hasty development by poorly educated programmers, Calnet's components contain a number of memory vulnerabilities.

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 in the first place.

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 stage.

The Task

> Neo's intelligence sources revealed that, once broken in the system, the required login credentials necessary for further access are located inside the system itself. Escalate your privileges in the machine by reading the credentials for each part, and then logging into the accounts with more and more authority to carry out your attack.

For each question, you are provided a vulnerable piece of code, and its compiled executable, in the home directory. Each binary has its `setuid` bit set, and is owned by the user of the next stage: the program will run with *their* effective privileges. Therefore, exploiting this program will allow you to assume the next user's permissions.

Develop an exploit at each step to access / read the restricted `README` file, where you will find the username and password for the next stage.

> You know from having watched his YouTube channel that Neo advocates a three-step approach for breaking into a system:
>
> **Reconnaissance.** Investigate what software/which services is/are running. Determine if there is anything you can access. What can you discover about the software? Using this information you can seek out potential vulnerabilities.
>
> **Development.** After you have found a vulnerability, you can create an exploit using the found bugs (generally, as an attacker, this means crafting a malicious input to the buggy program).
>
> **Profit.**

Use your knowledge of software security to first, identify the vulnerability, and then second, develop an exploitative user input. Neo has provided an `exploit` script in each step that will take your malicious input and automate your exploit. Look at this script to determine which executables you need to create (e.g. `egg` in question 0). If you have an concerns about using Unix to do this, read this useful Appendix.

For each step, you can confirm that your solution works by running `exploit`. For most of the problems, a correct exploit will launch a new shell waiting for input: typing commands like `whoami` and looking for the expected output (e.g., the username for the following problem) can help verify that your malicious input is working.

Once you have a working exploit, you can view the `README` file via a command like `cat README`. Sanity check: why does `cat README` not work *before* running the exploit? What happens when you try it?

For each question you will submit both a working exploit, and a brief writeup. For details, see the Submission Summary section.

Warnings

Exploit development is fussy business, which means you need to be careful.

Before Beginning We recommend you review the material from the lectures, and Discussions 1 and 2. For each question's vulnerable code, try to absorb the high-level concepts of exploiting stack overflows rather than every single line of assembly.

Executable Privileges Before invoking `exploit`, make sure that your executables have the execute permission set — this can be done using chmod +x filename.

Python 3 Neo does not recommend using Python 3, because of the distinction between unicode bytes and strings. Python 2 doesn't make this distinction, which makes it significantly easier. Alternatively, you can use a different scripting language such as Bash, Perl, or Ruby, or even write your exploits in C.

Running Without `invoke` You should always run executables with `invoke`, for example:

```
$ ./dejavu           # bad
$ invoke ./dejavu    # good
$ gdb dejavu         # bad
$ invoke -d ./dejavu # good
```

Note that it is *not* necessary to run `exploit` or `debug-exploit` scripts with `invoke`, since `exploit` already uses `invoke`. For more details, see the appendix.

We **highly recommend** that you test each of your submissions against our autograder, in order to debug potential issues before the project deadline.

Question 1 ***Neo's Guide*** **(10 points)**

For this question, while you will still have to develop your own exploit, Neo has done much of the conceptual work for you. In the back of these instructions, you will find Neo's *write-up*: a detailed but brief description of how he developed this exploit on his copy of Calnet (**Note:** his copy will have been customized differently).

Begin the project by SSHing into the VM from your local machine: since we use a rule to forward to port 16119, run the command `ssh -p 16119 vsftpd@127.0.0.1` on your local machine, where `vsftpd` is the username you obtained in the customization step above. Use the associated password to log in.

Neo already provided the `exploit` script, a scaffold that takes a malicious input and feeds it to the vulnerable program (which you can find in the home directory).

You are to continue his work and use this to spawn a shell. More concretely, write a script called `egg` that outputs a malicious buffer to standard out (ie, `print` in Python). You buffer should, when `exploit` is called, inject the following *shellcode*[3]:

```
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: unless otherwise stated, you will be using the same shellcode for the subsequent parts to this project as well.

Recall that x86 has little-endian byte order, e.g., the first four bytes of the above shellcode will appear as `0xcd58326a` in the debugger.

Once you have a shell running with the privileges of the next user, run the command `cat README` to learn their password for the next problem.

Submission and Grading. You will submit a script `egg`, written in your favorite scripting language, that integrates with the above displayed script `exploit`. Your script should inject shellcode to spawn a shell. Make sure it works by running `./exploit`.

Our grading tool will log into a clean VM image as user `vsftpd` and put your submission into the directory `/home/vsftpd`. A script will then invoke `exploit` and check for the existence of a shell prompt with effective privileges of the next user `(10 points)`.

For the rest of the project, you will also submit your own write-up for each question, which includes a description of the vulnerability, how it could be exploited, and how you determined any relevant "magic numbers." For now, this is provided for you.

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

Example Write-Up

Main Idea

The vulnerability in this question is that `gets(door)` does not check the length of the input from the user, which allows for a buffer overflow attack. By overwriting the return address of the frame to point at the shellcode, which we inserted after the return address, we can execute the shellcode.

Magic Numbers

We first determined the addresses of the `door` buffer (`0xbffffc18`) and the value of the `eip` register (`0xbffffc2c`) (which is the return instruction pointer) when executing the `deja_vu` function. This was done by invoking GDB and setting a breakpoint at line 7.

```
(gdb) x/16x door
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 deja_vu (dejavu.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 (`0xbffffc2c - 0xbffffc18 = 20`).

Exploit Structure

We used this information to structure our final exploit. The exploit consisted of three sequential sections:

1. `0xbffffc18`: First, we include 20 dummy characters to pad the buffer until we reach the return address pointer.
2. `0xbfffc2c`: Next, we insert our new return address. Since we want the return address to be the address of the shellcode (which is inserted directly after), we inserted `0xbffffc30` (`0xbffffc2c + 4`) into the return address so it points to 4 bytes after the return address.
3. `0xbffffc30`: Finally, we inserted the rest of the shellcode immediately after.

When executed, the buffer is completely overwritten with exploit code by the `gets` function. When the `deja_vu` function is done executing, it pops the return address off the stack, reading it to be `0xbffffc30`. It starts executing at the point, running the shellcode that we placed there.

Exploit GDB Output

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

```
(gdb) x/16x door
0xbffffc18:  0x61616161  0x61616161  0x61616161  0x61616161
0xbffffc28:  0x61616161  0xbffffc30  0xcd58316a  0x89c38980
0xbffffc38:  0x58466ac1  0xc03180cd  0x2f2f6850  0x2f686873
0xbffffc48:  0x546e6962  0x8953505b  0xb0d231e1  0x0080cd0b
```

As predicted, the `gets()` function wrote past the buffer boundary, overwriting the return instruction pointer to point to the given shellcode afterwards.

Question 2 ***Compromising Further*** **(15 points)**

SSH into the VM again, using the username `smith` and the password you learned in the previous question (the command to run is `ssh -p 16119 smith@127.0.0.1`).

> Calnet uses a sequence of stages to protect intruders from gaining root access. The inept Leland Junior University programmers actually attempted a half-hearted fix to address the overt buffer overflow vulnerability from the previous stage. In this problem you must bypass these mediocre security measures and, again, inject code that spawns a shell.

In the home directory of this stage, `/home/smith`, you will find a small helper script `generate-file-contents`. This script takes arbitrary input via *stdin* and prints the first 127 bytes to *stdout* in the format that the program `agent-smith` expects (which is an initial byte specifying the length of the input, followed by the input itself):

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

Neo realized that this helper script always generates safe files to be used with the buggy `agent-smith` program—but nothing prevents you from instead feeding `agent-smith` an arbitrary file of your choice. In particular, Neo started a script `exploit` representing an initial exploit attempt.

Warning: Note that (the length of) the input filename used affects your stack addresses! Make sure you take this into account while debugging, and ensure that your exploit works when running `./exploit`. We recommend using the filename `anderson.txt`.

Submission and Grading. As in the previous question, you will submit a script `egg`, written in your favorite scripting language, that integrates with the script `exploit`. Your script should inject shellcode to spawn a shell. Make sure it works by running `./exploit`.

Our grading tool will log into a clean VM image as user `smith` and put your submission into the directory `/home/smith`. A script will then invoke `exploit` and check for the existence of a shell prompt with effective privileges of the next user `(10 points)`.

You must also submit a write up for this question in `explanation.pdf`, that includes a description of the vulnerability and your exploit, as well as how you determined any relevant "magic numbers." Also provide GDB output demonstrating the before/after of your exploit working. Please try to limit your write (GDB output excluded) to no more than a page. `(5 points)`

Question 3 ***Secret Exfiltration*** **(25 points)**

Lord Dirks has learned from your previous exploits that buffer overflows are **bad news**. Rather than rewrite his code to fix this issue, Lord Dirks decides to enable stack canaries as a fool-proof solution.

The `agent-jz` program takes in any number of lines, and converts them so that their hexadecimal escapes are decoded into the corresponding ASCII characters. Any non-hexadecimal escapes are outputted as-is. For example:

```
$ ./agent-jz
\x41\x42   # outputs AB
XYZ        # outputs XYZ
# Control-D ends input
```

Neo has helped you by creating three files: `interact.scaffold`, `exploit` and `scaffold.py`. Neo intended for you to work from the `interact.scaffold` file, but alas! The VM will not let you write to these files. Instead, copy `interact.scaffold` over to a fresh `interact` script. The command `cp interact.scaffold interact` should do this. Then, make your new file writable by running `chmod +w interact`. All of your work will go in this new `interact` file.

The `exploit` script simply runs your `interact` script three times in a row (since your solution might have a small chance of failure.) Finally, the `scaffold.py` script contains functions which will help you to interact with the output of the program. In particular, you have access to the following:

1. `SHELLCODE`: the shellcode that you should execute. Rather than opening a shell, it prints the `README` file, which contains the password.
2. `p.send(s)`: sends a string `s` to the program. **Be sure to send a newline `\n` at the end of each line of your input.**
3. `p.recv(num_bytes)`: reads the given number of bytes from the program's output.

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

```
# Note the newlines!
p.send('\\x41\\x42' + '\n') # p.recv(3) == 'BC\n'
p.send('XYZ' + '\n')        # p.recv(4) == 'XYZ\n'
```

Note that this question is particularly difficult to debug. Neo suggests that you begin with exploring the problem using `gdb` and a pen-and-paper, rather than trying to start by writing the `interact` script.

Submission and Grading. For this question, you will submit the Python script `interact`. Your script should successfully read the `README` file. Make sure it works by running `./exploit`.

Our grading tool will log into a clean VM image as user `jz` and put your submission into the directory `/home/jz`. A script will then invoke `exploit` and check for the output of the `README` file `(15 points)`.

You must also submit a write up for this question in `explanation.pdf`, that includes a description of the vulnerability and your exploit, as well as how you determined any relevant "magic numbers." Also provide GDB output demonstrating the before/after of your exploit working. Please try to limit your write (GDB output excluded) to no more than a page. `(10 points)`

Question 4 ***Deep Infiltration*** **(35 points)**

Find the subtle vulnerability in this code, and inject code that spawns a shell. Neo, again on top of it, started a scaffold called `exploit` that you should use. It might also help to review the explanation of `invoke` given above.

> Calnet is a pernicious and invasive piece of malcode. But Lord Dirks undertook all of his own studies at Leland Junior University, and as such he never really learned *how to count* without occasionally screwing it up.

To solve this problem, you might benefit from reading Section 10 of "ASLR Smack & Laugh Reference" by Tilo Müller. (Although the title suggests that you have to deal with ASLR, you can ignore any ASLR-related content in the paper for this question.)

Submission and Grading. For this question, you will submit a script `arg` and a script `egg` written in your favorite scripting language, that integrates with the script `exploit`. Your script should inject shellcode to spawn a shell. Make sure it works by running `./exploit`.

Our grading tool will log into a clean VM image as user `brown` and put your submission into the directory `/home/brown`. A script will then invoke `exploit` and check for the existence of a shell prompt with effective privileges of the next user `(20 points)`.

You must also submit a write up for this question in `explanation.pdf`, that includes a description of the vulnerability and your exploit, as well as how you determined any relevant "magic numbers." Also provide GDB output demonstrating the before/after of your exploit working. Please try to limit your write (GDB output excluded) to no more than a page. `(15 points)`

Question 5 ***Against the Clock*** **(35 points)**

Notice that this program includes two different types of user input.

Consider the varied differences in user-input between this code and the previous examples: find where users can interact with the program, and how this interaction is limited, compared to the other questions. Neo recommends first developing a high-level understanding of what the program does, and suggests that you pay particular attention to how it handles error-checking.

Hint: What if we consider "preventing invalid inputs" equivalent to "authorizing valid inputs"? What security vulnerabilities occur during authorization protocols?

> Despite your previous success, Lord Dirks thinks he's now safe: you now find yourself facing Calnet's antiquated file IO scheme, where user input comes to die. He proudly boasts to Leland Junior University that no one could ever develop an exploit without the convenience of standard input! Neo is confident you can prove him wrong.

Find, and exploit, the vulnerability in `dejavu.c` (wait, that name rings a bell). As usual, your goal is to exploit the poorly-written code to get higher credentials. Neo, once more, started a scaffold called `exploit` that you should use, as well as an `interact.scaffold` script. In particular, you have access to the following:

1. `SHELLCODE`: the shellcode that you should execute. Rather than opening a shell, it prints the `README` file, which contains the password.
2. `p.send(s)`: sends a string `s` to the program. **Be sure to send a newline `\n` at the end of each line of your input.**
3. `p.recv(num_bytes)`: reads the given number of bytes from the program's output.

To use these (and subsequently develop your exploit), copy `interact.scaffold` over to a fresh `interact` script. The command `cp interact.scaffold interact` should do this. Then, make your new file writable by running `chmod +w interact`.

Submission and Grading. For this question, you will submit the Python script `interact`. Your script should successfully read the `README` file. Make sure it works by running `./exploit`.

Our grading tool will log into a clean VM image as user `oracle` and put your submission into the directory `/home/oracle`. A script will then invoke `exploit` and check for the output of the `README` file `(20 points)`.

You must also submit a write up for this question in `explanation.pdf`, that includes a description of the vulnerability and your exploit, as well as how you determined any relevant "magic numbers." Also provide GDB output demonstrating the before/after of your exploit working. Please try to limit your write (GDB output excluded) to no more than a page. `(15 points)`

Question 6 ***The Last Bastion*** **(25 points)**

This part of the project enables ASLR. **Once you have started this part of the project 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.**

> Yo, Berkeley! Your mission, should you choose to accept it, is to bypass the ASLR protection and spawn a shell with root privileges. Full control of the box ... *and thus Calnet itself* awaits you! Neo didn't dare hope you might hack your way this far and this deeply ... but he could never abandon his dream of freedom.

You should consider reading Section 8 of "ASLR Smack & Laugh Reference" by Tilo Müller. Neo has also noted that even though ASLR is enabled, position-independent executables were **not** enabled. Therefore, the `.text` segment of the binary is always at the same spot.

One detail Neo *could* figure out for you is that the service to exploit listens locally on TCP port 942. It turns out that the operating system watches the service and restarts it shortly when it crashes. You have to send the malicious shellcode to that service to successfully complete this task. To perform the exploit, run `exploit`. If you succeed in the exploit, you should see the output root on shell command whoami.

```
# Linux (x86) TCP shell binding to port 11111.
bind_shell =
    "\xe8\xff\xff\xff\xff\xc3\x5d\x8d\x6d\x4a\x31\xc0\x99\x6a" +
    "\x01\x5b\x52\x53\x6a\x02\xff\xd5\x96\x5b\x52\x66\x68\x2b\x67" +
    "\x66\x53\x89\xe1\x6a\x10\x51\x56\xff\xd5\x43\x43\x52\x56\xff" +
    "\xd5\x43\x52\x52\x56\xff\xd5\x93\x59\xb0\x3f\xcd\x80\x49\x79" +
    "\xf9\xb0\x0b\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89" +
    "\xe3\x52\x53\xeb\x04\x5f\x6a\x66\x58\x89\xe1\xcd\x80\x57\xc3"
```

> This should finally suffice to pull off the Final Stage!
>
> *The freedom of cybercitizens throughout Caltopia rests in your hands ...*

Submission and Grading. For this question, you will submit a script `egg` written in your favorite scripting language, that integrates with the script `exploit`. Your script should inject shellcode to spawn a shell. Make sure it works by running `./exploit`.

Our grading tool will log into a clean VM image as user `jones` and put your submission into the directory `/home/jones`. A script will then invoke `exploit` and check for the existence of a shell prompt with effective privileges of `root (15 points)`.

You must also submit a write up for this question in `explanation.pdf`, that includes a description of the vulnerability and your exploit, as well as how you determined any relevant "magic numbers." Also provide GDB output demonstrating the before/after of your exploit working. Please try to limit your write (GDB output excluded) to no more than a page. `(10 points)`

Question 7 ***Feedback (optional)*** **(0 points)**

If you wish, you may submit feedback at the end of `explanation.pdf`, 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.

Submission Summary

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

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

You **should not** copy and paste your exploits from the VM onto your computer, since this might insert weird characters which will cause you to fail our autograder. Using `scp`, create the following directory tree:

```
customizer/.customization
vsftpd/egg
smith/egg
jz/interact
brown/arg
brown/egg
oracle/interact
jones/egg
```

Drag and drop all of these folders onto Gradescope. (Drag and drop all of the folders directly–do **not** create a leading folder, such as `proj1/`.)

Appendix: Note on Unix

We fully understand not everyone has the same level of Unix experience. Below should encompass much of the Unix you should know for this project.

"Scripts" / Shebang Line We ask you many times to write an `egg` "script," in some language you choose. Typically, you would use file extensions to indicate code-type (ie, you could create a file named `egg.py`, if you wanted to use Python 2), However, Neo's `exploit` is looking for a script named *exactly* "`egg`", so this will unfortunately not work.

You can include a shebang line at the top of your `egg` file, instead of using the "`.py`" extension, to indicate that your code is in Python (or any language). An example for Python shebang line might look like `#!/usr/bin/env python`

The `cat` and `echo` Commands Both of these are Unix commands that will cause the terminal to print some output. `cat` will print out the contents of a file, while `echo` will simply repeat back whatever arguments were passed in.

For example, `cat foo.txt` will print out the contents of `foo.txt`, whereas `echo foo.txt` will print out the exact phrase "`foo.txt`". Try them both out to see for yourself!

Redirection You may be familiar with redirection from 61C: as a reminder, these IO operators (`<` and `>`) redirect input, and output, respectively. Specifically, `<` is used for *input* redirection, and uses the righthand-side as the lefthand-side's input. `>` is used for *output* redirection, and send the lefthand-side's output to the righthand-side.

For example, `script > foo.txt` saves the output of `script` to `foo.txt`, whereas conversely, `script < foo.txt` uses the contents of `foo.txt` as input for `script`.

Piping You may notice that many of Neo's `exploit` scripts heavily use the symbol `|` and an obvious question might be: what does this do? This operator *pipes* the output of the process on its lefthand side, to the input of the process on its righthand side.

Or, more concretely, for `A | B`: the output of `A` is used as the input to `B`. Or, (using the commands from above) `cat foo.txt | script` use the contents of `foo.txt` (the output of `cat foo.txt`) as the input to `script`.

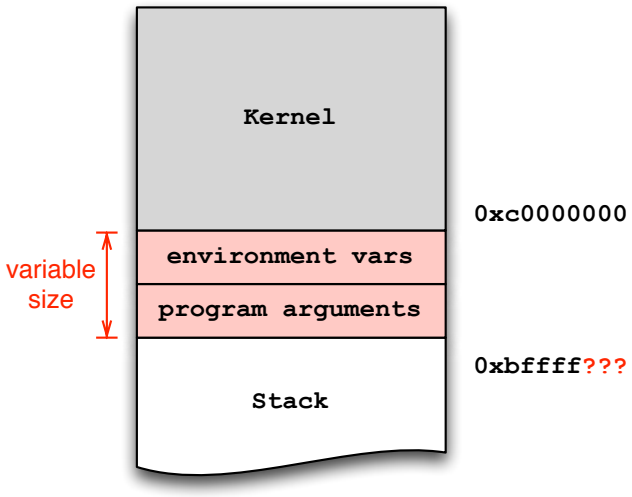
You can also almost think of piping (take `A | B`) as the following two redirection expressions, put together: `A > temp.txt` and `B < temp.txt`

Evaluating Expressions / Variables Occasionally, you might also see `$`'s show up across the project. In Unix, these are used to evaluate variables or expressions: `$u` is equivalent to whatever *value* the variable `u` is storing.

Consider the sequential commands `u=abc` and `echo $u`: these will print out to standard output the string "abc." For those who did the Backup Hive Setup, this should be familiar.

Appendix: Note on Execution Environments

Exploit development can lead to serious headaches if you don't adequately account for factors that introduce *non-determinism* into the debugging process. In particular, the stack addresses in the debugger may not match the addresses during normal execution. This artifact occurs because the operating system loader places both environment variables and program arguments *before* the beginning of the stack:



Already installed in the VM you'll find a small helper utility, `invoke`, that makes sure environment and arguments remain at the same location, regardless of whether using the debugger or not. For example, instead of invoking the program `foo` directly via `./foo`, you should instead use `invoke foo`:

```
$ ./foo arg1 arg2          # invocation dependent on environment state :-(
$ invoke foo arg1 arg2     # deterministic invocation
$ invoke -d foo arg1 arg2  # deterministic invocation in gdb
$ ./exploit                # deterministic invocation, handled by exploit
```

You may find it useful to pass an extra environment variable to the program. The `-e` switch serves that purpose:

```
$ invoke -e Y foo arg1  # sets environment variable ENV=Y in foo
```

You must always use `invoke` or `exploit` to launch (or debug via `invoke -d`) the provided executables because `invoke` additionally parameterizes the execution environment based on the ID you entered during the first boot.