Preamble

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!

Background Story

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 University of Caltopia alumnus Neo, famed for not only his hacking skills but also the excellent YouTube videos he produces illustrating his techniques, 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 brutally deletes Neo’s YouTube account and swoops in with a SWAT team to make an arrest. As the thugs smash through the coffeeshop’s doors, Neo gets off one final
Getting Started

Neo expects your team to develop exploits for 5 vulnerabilities in Calnet’s components. As they topple you will move closer and closer towards pwning the nefarious botnet. All you have to go by are your wits, your grit, and Neo’s legacy: guidelines on how to proceed, and, most precious, a virtual machine (VM) image that contains code samples from the main Calnet components.

You can work in teams of 1 or 2 students. To begin the project, you will need to set up a virtual machine. There are two methods to do so—both methods require an instructional account.

Recommended Setup: “No Fuss”

This setup requires very little work on your part. It works 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:\[1\]

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

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

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

That’s it! If this works for you, skip the “fussy” version below and go directly to “Configuring your VM”.

---

\[1\] 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. If it does not work, feel free to SSH into any of the hive machines and run the command manually.
Alternate Setup: “Fussy”

There is also the old style setup method, for people who prefer to run things locally. You are also able to run and investigate the VM on your own computer. You will need the following 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-sp19.ova) via File -> Import Applicance.

Make sure your network is configured correctly by clicking your VM’s settings. Under Network -> Adapter 1, make sure the first NAT adapter is enabled and open the advanced settings.

![VirtualBox Adapter Settings](image)

Click the Port Forwarding button and ensure that you have a rule to forward port 22, for SSHing to the machine, to port 16161 on your host. (The image below shows that port 2222 is being forwarded. Make sure that yours shows port 16161.)
You can now start the VM, in which you will run the vulnerable programs and their exploits.
Customizing

Regardless of which setup you use, you will need to customize the virtual machine. If you don’t have an instructional account for this class, you will need to get one first. To do so, follow the instructions given here.

Log in as the user customizer with the password customizer (same username and password), and follow the prompts. Note that customization requires your partner’s login. Both you and your partner should customize the VM using the same logins (the order of the logins 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 class account 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 (redetermination of the addresses to use in them). This should go quickly once you understand the exploit in the first place.

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

Once you have finished customizing your virtual machine, you will receive the username and password for the first stage.
An Important 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:

![Stack Diagram]

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. More broadly, since our grading tool uses the exact same VM that you downloaded, do not perform any system modifications, only add/upload new content. (For example, do not attempt to recompile the given executables.) This way you ensure that your solutions will work with our grading tool and you do not run the risk of losing unnecessary points.
The Task

Neo’s intelligence sources revealed that, once broken into 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.

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 Neo’s three-step plan to solve the following problems.

**No Fuss Setup.** Enter the username (*vsftpd*) and password you received in the customization step above.

**Fussy Setup.** Begin the project by SSHing into localhost, using the username and password you received in the customization step above. Since we use a rule to forward to port 16161, use the command `ssh -p 16161 vsftpd@127.0.0.1`, where *vsftpd* is the username you obtained in the customization step above.

For each step, look at the *exploit* script to determine which executables you need to create (e.g. *egg* in question 1). Before invoking *exploit*, make sure that your executables have the execute permission set — this can be done using `chmod +x filename`. For each step, you can confirm that your solution works by running *exploit*, which should launch a shell waiting for input, and then typing commands like `whoami` and looking for the expected output, the username for the following problem, in this case. Once you have a working exploit, the *README* file will let you see the username and password for the next stage. You can view it via a command like `cat README`.
**Question 1  Behind the Scenes** (10 points)

A tweet from Neo assures you that given its hasty development by poorly educated programmers, Calnet’s components contain a number of memory-safety vulnerabilities. In the VM that Neo provided, you will find the first code piece located in the directory `/home/vsftpd`.²

You are to continue his work and write an exploit that spawns a shell, for which you can use the following `shellcode`:

```shellcode
shellcode = 
"\x6a\x31\x58\xcd\x80\x89\xc3\x89\xc1\x6a" + 
"\x46\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"
```

Shellcode is x86 machine code which performs some action—typically spawning a shell for further attacker interaction. Recall that x86 has little-endian byte order, e.g., the first four bytes of the above shellcode will appear as `0xcd58316a` in the debugger.

Neo already provided an exploit scaffold that takes your malicious buffer and feeds it to the vulnerable program via a script called `exploit`. To test whether the exploit worked, try running a command such as `ls` or `whoami`. To exit the shell, type Ctrl-D.

To get started, review the material from the lectures and Discussion 1. Neo recommended that you try to absorb the high-level concepts of exploiting stack overflows rather than every single line of assembly.

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

**Submission and Grading.** We highly recommend that you test your submission against our autograder, in order to debug potential issues before the project deadline. To do so, see the section “Submission Summary” near the end.

You must also submit a write up for this question in `explanation.pdf` that includes a description of the vulnerability, how it could be exploited, how you determined which address to jump to, and a detailed explanation of your solution. This includes GDB output that very clearly demonstrates the effects of your exploit (before/after) (5 points). Please keep your writeups to no more than a page, excluding GDB outputs and diagrams.

---

²The vulnerable binary has the `setuid` bit set and is owned by the user of the next stage, meaning it will run with the effective privileges of user `smith`. 
Question 2  Compromising Further  (15 points)

No Fuss Setup. Press Ctrl-D (or use the command logout) until you are back at the login prompt. Use the username and password you learned in the previous question.

Fussy Setup. 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 16161 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.

Submission and Grading. As in the previous question, 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 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 user brown (10 points).

You must also submit a write up for this question in explanation.pdf, that includes the same type of information as the sample writeup for Question 1 on Gradescope. (5 points)
Question 3  Deep Infiltration  (35 points)

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.

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.

To solve this problem, you might benefit from reading Section 10 of “ASLR Smack & Laugh Reference” by Tilo Müller [1]. (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. Your code should integrate with the script exploit as shown above. Make sure your scripts work 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 user jz (20 points).

You must also submit a write up for this question in explanation.pdf that includes a description of the vulnerability, how it could be exploited, how you determined which address to jump to, and a detailed explanation of your solution. This includes GDB output that very clearly demonstrates the effects of your exploit (before/after) (15 points). Please keep your writeups to no more than a page, excluding GDB outputs and diagrams.
Question 4  Secret Exfiltration  

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:

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

Neo has helped you by creating three files: interact, exploit and scaffold.py. Your work will go in the interact script – do not modify the other files as they will not be graded. The exploit script simply runs your interact script three times in a row. (This is helpful, 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.

```python
# 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. Do not edit any of the other files as they will not be graded. It is OK if your exploit does not work 100% of the time, although reasonable solutions should work at least 90% of the time. Our grading tool will log into a clean VM image as user jz and then put your submission into the directory /home/jz. A script will then run exploit and check that your submission correctly reads the README file. (15 points)

As in the previous questions, you must also submit a write up for this question in explanation.pdf. Please include a brief description of all vulnerabilities, how they can be exploited, how you determined what addresses to jump to, how you determined what characters to input, and a detailed explanation of your solution. (10 points)
Question 5  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 [1]. 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 xc5d xc8d xc4a xc31 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 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 xe04 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 question, you will submit a script egg, written in your favorite scripting language, that prints the exploit buffer to standard output. Make sure your scripts work 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 user root (15 points).

You must also submit a write up for this question in explanation.pdf in the same fashion as for Questions 1–3 (10 points).
Question 6  \textit{Feedback (optional) (0 points)}

If you wish, you may submit feedback at the end of \texttt{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

You will need to move your files off the VM and submit them to the “Project 1 Autograder” assignment on Gradescope. The method to do this depends on how you set up your VM:

\textbf{No Fuss Setup.}  Run the following commands in your local computer’s terminal:

\begin{verbatim}
$ u=XXX # replace XXX with last three letters of instructional account
$ ssh cs161-$u@hive$((36#${u:2}\%26+1)).cs.berkeley.edu \ 
  ~/cs161/proj1/make-submission > proj1-subm.zip
\end{verbatim}

This creates a file \texttt{proj1-subm.zip} in the current directory, which you can submit to Gradescope as-is.

\textbf{Fussy Setup.}  Using \texttt{scp}, create the following directory tree:

\begin{itemize}
  \item \texttt{customizer/.customization}
  \item \texttt{vsftpd/egg}
  \item \texttt{smith/egg}
  \item \texttt{brown/arg}
  \item \texttt{brown/egg}
  \item \texttt{jz/interact}
  \item \texttt{jones/egg}
\end{itemize}

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.

Submit your writeup \texttt{explanation.pdf} to the assignment “Project 1 Writeup”.

References