Jawale & Dutra
Summer 2019

CS 161
Computer Security

Project 1

Due: July 11th, 2019, 11:59PM

Version 20.00.00.00

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!

> 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 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 the National Berkeley University's virtuous computer security students to carry forth the flame of knowledge, seize control of Calnet, and let freedom ring once more throughout Caltopia ...

Getting Started

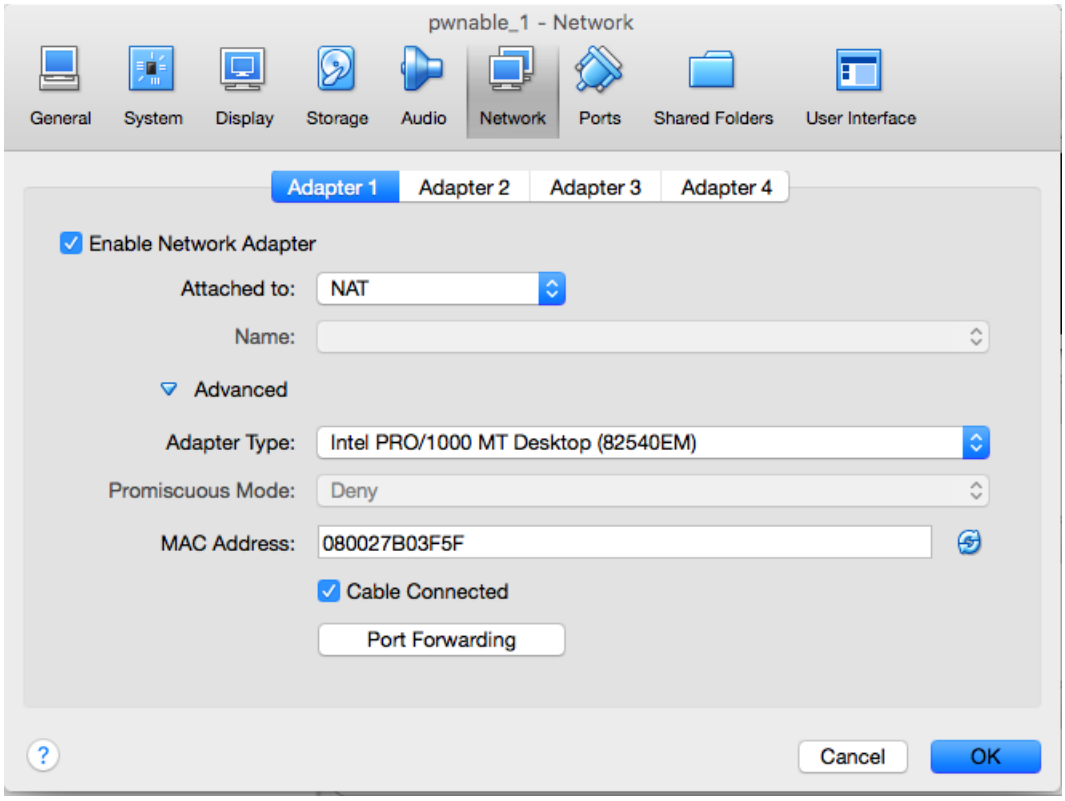
> Neo expects your team to develop exploits for 5 vulnerabilities in Calnet's components. As they topple you will move closer and closer towards pwning the nefarious botnet. All you have to go by are your wits, your grit, and Neo's legacy: guidelines on how to proceed, and, most precious, a virtual machine (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. You will need the following programs installed on your computer:

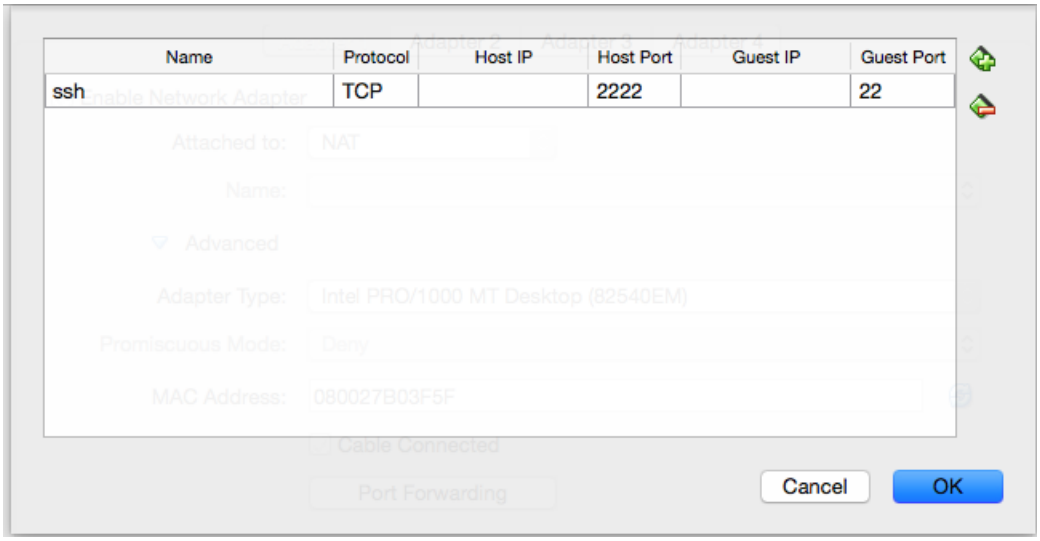
1. VirtualBox
2. A text editor
3. An SSH client (on Windows, use Putty or Git Bash)

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-su19.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.



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.

Customizing

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.

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.

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.
>
> 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.

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`.

Warnings

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

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.

Changing the VM. 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 using `make`! This way you ensure that your solutions will work with our grading tool and you do not run the risk of losing unnecessary points.

If you accidentally overwrote a file, you will need to redownload the VM. Alternatively, you can use the snapshot feature of Virtualbox, and make a snapshot every time you do something important. Then, you can revert to the snapshot if something breaks

We **highly recommend** that you test each of your submissions against our autograder, in order to debug potential issues before the project deadline. To do so, see the Submission Summary section.

Question 1 ***Behind the Scenes*** **(10 points)**

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.

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

In the VM that Neo provided, you will find the first code piece located in the home directory. This is a memory-vulnerable binary with 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`. Therefore exploiting this program will allow you to assume the permissions of `smith`.

Neo already provided an exploit scaffold that takes a malicious input and feeds it to the vulnerable program via a script called `exploit`. 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*:

```
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"
```

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

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.

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. 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 user `smith (10 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 clearly but succinctly demonstrates the effects of your exploit (before/after) `(5 points)`. Please keep your write-ups to no more than a page, excluding GDB outputs and diagrams.

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 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 your writeup for Question 1 on Gradescope. `(5 points)`

Question 3 ***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. 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 `oracle (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 ***Oracle 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.

> Lord Dirks, hiding behind the perceived safety of his canaries, beseeches his oracles for advice in stopping Neo's looking threat. Your new job is to steal their prophetic wisdom, and use it against them.

As usual, your goal is to exploit the poorly-written code to get higher credentials. Consider reading "Introduction to String Format Vulnerabilities" by Alex Reece. Neo also managed to provide some assistance, and left you a `hint` for this problem, in the `/home/oracle` directory itself. Be careful, your solution needs to be able to fit in the space provided by the buffer.

Warning: Alpine Linux (the OS of the VM) uses `muscl libc`. This does not behave like `glibc`, which you may be more familiar with from 61C. In particular, `muscl libc` treats string formats like `$` differently, so Neo highly recommends avoiding this string format.

`muscl libc` also treats `gets()` slightly differently: it writes **two** null terminators at the end of the string, instead of the standard one null byte. Be wary of this when debugging.

Submission and Grading. For this question, you will submit 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/oracle`. A script will then invoke `exploit` and check for the existence of a shell prompt with effective privileges of user `jones (15 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) `(10 points)`. Please keep your writeups to no more than a page, excluding GDB outputs and diagrams.

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. 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 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 ***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.

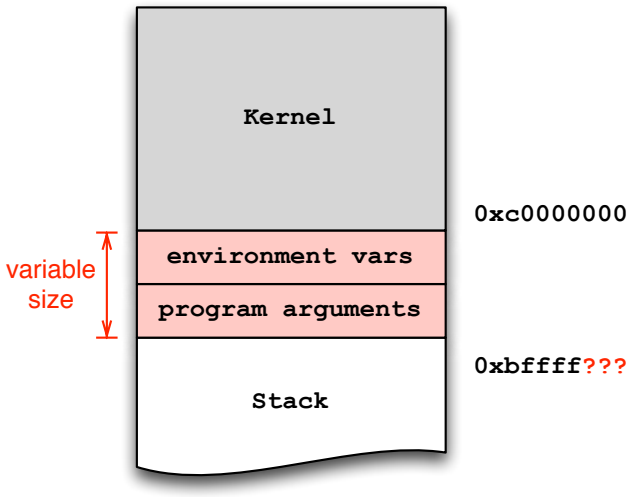
You **should not** copy and paste your exploits from the VM onto your computer, since this might insert weird characters which will cause you to fail our autograder. Using `scp`, create the following directory tree:

```
customizer/.customization
vsftpd/egg
smith/egg
brown/arg
brown/egg
oracle/egg
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 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.