Raluca Popa
Spring 2018

CS 161
Computer Security

Project 1

Due: Febuary 19, 2018, 11:59PM

Version 2.3: January 29, 2018

Background

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 Taipei, a flicker of hope remains. The brilliant University of Caltopia alumnus Nick Weaver, 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, Nick Weaver 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 Nick Weaver begins to code up some righteous exploits to pwn Calnet's components, a barista at the coffeeshop where Nick Weaver gets his free WiFi betrays him to Lord Dirks, who brutally deletes Nick Weaver's YouTube account and swoops in with a SWAT team to make an arrest. As the thugs smash through the coffeeshop's doors, Nick Weaver gets off one final tweet for help. Such are his hacking skillz that he crams a veritable boatload of key information into his final 140 characters, exhorting the National Taipei 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

Nick Weaver has determined that the correct mojo for this task is teams of 1 or 2 students. He 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 Nick Weaver'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 will be able to run and investigate the VM on your own computer. You will need the following on your computer:

1. VirtualBox
2. A text editor
3. An SSH client (on Windows, use Putty or the ssh that comes with Git)

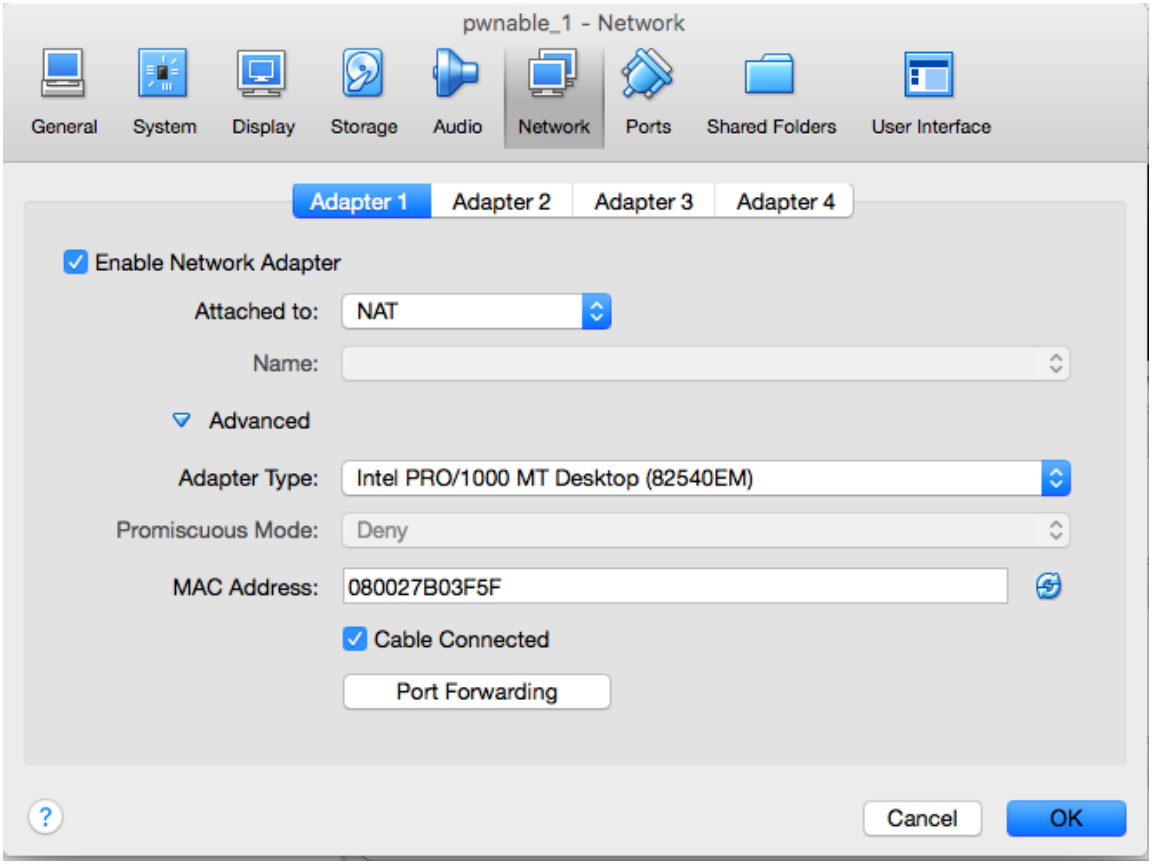
On Linux and Mac, you can install these programs from your package manager (e.g., `apt` or `brew`).

NOTE: Only use these tools against your own infrastructure. You violate campus policy when directing them against parties who do not provide their informed consent!

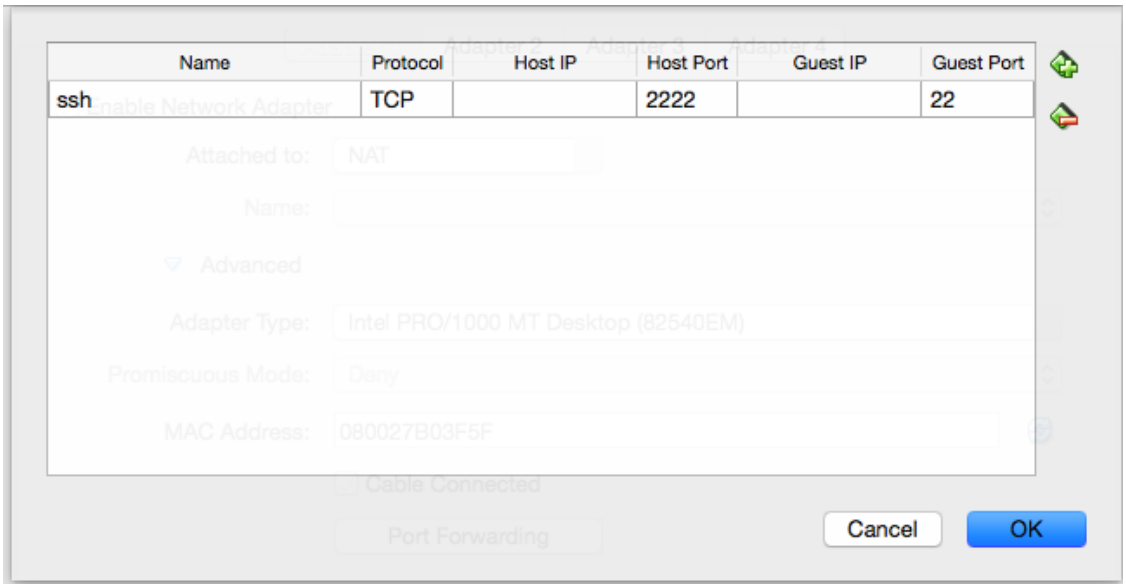
VM Setup

Open VirtualBox, and download and import the VM image (pwnable.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 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. The image is a bare-bones Ubuntu Linux server installation on a 32-bit Intel architecture.

If you don't have a class account for this class, go to http://inst.eecs.berkeley.edu/webacct to get one. The first time you boot the image, you have to enter your class accounts in the format cs161-$x_1x_2x_3$,cs161-$x_4x_5x_6$, where $x_1, \ldots, x_6$ are the letters of your class accounts. You need to list the accounts in alphabetical order. For example, if a student with class account `cs161-wed` teams with a student with class account `cs161-vvz`, then you would enter the string `cs161-vvz,cs161-wed`.[1]

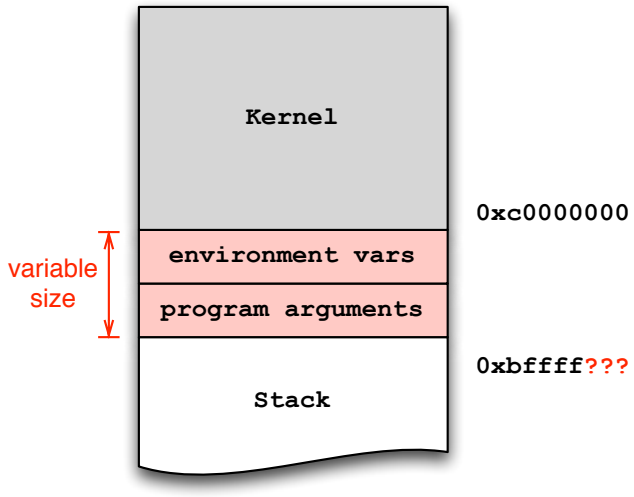
The VM configures its addresses using this login so if it is not entered correctly, you and your partner will fail the autograder tests. Make sure that you include your EXACT login. If you receieve an error verb—Verifying... syntax error.—, make sure other VMs are stopped (such as a VM for 162). Before submitting you may want to check the login you used for the VM by reading the file `/etc/default/pwnable` from the VM.

Don't worry if the VM screen shows nothing but "Ready for pwning" or eventually the screen turns off. Since it's your job to gain access to it from your host machine over the network, you won't need the VM's GUI.

[1] 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 configuration step. Once you have your team in place, you'll need to start again with a clean VM image configured 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've learned how to figure out the addresses in the first place.

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:



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
```

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

```
% invoke -e X=Y foo arg1   # sets environment variable X=Y in foo
```

You must always use `invoke` to launch (or debug via `-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

Unfortunately Nick Weaver did not have enough time to figure all out the necessary login credentials. It is up to you to break into the VM and continue his mission, with the ultimate goal to gain root privileges on the machine to have full control over Calnet. Nick Weaver'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 Nick Weaver advocates a three-step approach for breaking into a system:

Step 1: Reconnaissance. Investigate what software/which services is/are running. Determine if there is anything you can access. What can you discover about the software (e.g., in terms of version; do you have the source code)? Using this information you can seek out potential vulnerabilities.

Step 2: 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).

Step 3: Profit.

Use Nick Weaver's three-step plan to solve the following problems. Begin the project by SSHing into localhost, using the username `vsftpd` and password `r4e8kWpeFC`. Since we use a rule to forward to port 16161, use the command `ssh -p 16161 vsftpd@127.0.0.1`.

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. When exploiting a program that reads input up to a newline, make sure your `exploit` script outputs a newline at the end; i.e., don't require the user to type a newline manually after running `exploit`.

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

A tweet from Nick Weaver 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 Nick Weaver 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:

[2]The 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`.

```
shellcode =
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07" +
  "\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d" +
  "\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80" +
  "\xe8\xdc\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68"
```

NOTE: Recall that x86 has little-endian byte order, e.g., the first four bytes of the above shellcode will appear as `0x895e1feb` in the debugger.

Nick Weaver already provided an exploit scaffold that takes your malicious buffer and feeds it to the vulnerable program via a script called `exploit`:

```
#!/bin/sh
( ./egg ; cat ) | invoke dejavu
```

(As one of Nick Weaver's tweets explains in a concise but strikingly lucid fashion, the expression before the shell pipe is necessary so that if the attack input generated by `egg` succeeds, then you will be able to interact with the shell that the exploit spawns by typing via *stdin*. Be aware that when the shell spawns there will not be any immediate visual feedback, such as a prompt. 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, read "Smashing The Stack For Fun And Profit" by AlephOne [1]. Nick Weaver recommended that you try to absorb the high-level concepts of exploiting stack overflows rather than every single line of assembly. He also warned you that some of the example codes are outdated and may not work as-is.

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. For this problem you will submit the missing script `egg`, which can be written in your favorite scripting language (e.g., Python, Ruby, Perl, Bash). Your code should print the buffer used by the `exploit` script to spawn a shell. Make sure it works by invoking `./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 the script `exploit` *exactly as given above* and check for the existence of a shell prompt with effective privileges of user `smith (5 points)`.

You must also submit a file, `explanation.pdf`, that includes a brief description of the vulnerability, how it could be exploited, how you determined which address to jump to, and a sketch of your solution on Gradescope. This includes gdb output that very clearly demonstrates the effects of your exploit (before/after). Be sure to label it with as question 1. Keep it to no more than one page `(5 points)`.

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

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.

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`). 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):

```
% ./generate-file-contents < anderson.txt
```

Nick Weaver 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, Nick Weaver started a script `exploit` representing an initial exploit attempt:

```
#!/bin/sh
./egg > pwnzerized
invoke agent-smith pwnzerized
```

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 invoking `./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 for the previous Question `(5 points)` on Gradescope.

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.

Nick Weaver, again on top of it, started a scaffold called `exploit` that you can use:

```
#!/bin/sh
invoke -e egg=$(./egg) agent-brown $(./arg)
```

(Note that a shell expression like "`$(foo)`" means "run the command `foo` and substitute its *stdout* output here." So "`$(./egg)`" means "run the command `./egg` and assign the output it generates to the variable `$egg`.")

To solve this problem, you are pretty sure that a cryptic reference in Nick Weaver's tweets indicates you'd benefit from reading Section 10 of "ASLR Smack & Laugh Reference" by Tilo Müller [2]. (Although the title suggests that you have to deal with ASLR, you can ignore any ASLR-related content in the paper for this question.)

Hint: The VM will output a line saying "Check out the hint" while running the program if you happen to have set your stack up so that it's difficult to accomplish the exploit with the addresses as they are. In this case, you may want to add bogus environment variables to move the stack around and give yourself enough room to operate.

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 invoking `./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)`.

As for the previous question, you must also submit a write up for this question in `explanation.pdf` that includes a brief description of the vulnerability, how it could be exploited, how you determined which address to jump to, and a sketch of your solution. This includes gdb output that very clearly demonstrates the effects of your exploit (before/after) `(15 points)`.

Question 4 *Secret Exfiltration* (20 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.[3]

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
```

Nick Weaver 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:

[3] As an implementation detail, you should note that stack canaries on Debian systems always have a NUL byte as their least-significant byte. Because of little-endian, this means that the NUL byte comes first in memory.

1. `SHELLCODE`: the shellcode that you should execute. 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. **If no string is available, your program will timeout while waiting.**

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

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

Note that this question is particularly difficult to debug. Nick Weaver 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. Once you think that you've identified the vulnerability and playing with in `gdb`, it's time to start thinking about interactivity.

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. (10 points)

As in the previous questions, you must also submit a write up for this question in `explanation.pdf` that includes a brief description of any vulnerabilities, how they can be exploited, how you determined what addresses to jump to, how you determined what characters to input, and a sketch of your solution. (10 points)

Question 5 ***The Last Bastion*** **(20 points)**

To protect the Calnet source from advanced hackers, Lord Dirks's minions persuaded him that he must enable address layout randomization (ASLR) as a final layer of defense for the VM. They assured him that it was inconceivable that anyone even of super-human intelligence would possess the uber-h4x0r skillz required to overcome this. **Once you have started this part of the project ASLR will be enabled on your VM so you'll need to restart your VM if you'd like to go back to the previous parts.**

Yo, Taipei! 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! Nick Weaver didn't dare hope you might hack your way this far and this deeply ... but he could never abandon his dream of freedom, and to that end provided an exceedingly cryptic clue in his final tweet that after a caffeine-fueled all-nighter you eventually realize suggests you should consider reading Section 8 of "ASLR Smack

& Laugh Reference" by Tilo Müller [2]. Nick Weaver 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 Nick Weaver *could* figure out for you is that the service to exploit listens locally on TCP port 42000. 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 6666.
bind_shell =
  "\x31\xdb\xf7\xe3\x53\x43\x53\x6a\x02\x89\xe1\xb0\x66\xcd" +
  "\x80\x5b\x5e\x52\x68\x02\x00\x1a\x0a\x6a\x10\x51\x50\x89" +
  "\xe1\x6a\x66\x58\xcd\x80\x89\x41\x04\xb3\x04\xb0\x66\xcd" +
  "\x80\x43\xb0\x66\xcd\x80\x93\x59\x6a\x3f\x58\xcd\x80\x49" +
  "\x79\xf8\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3" +
  "\x50\x53\x89\xe1\xb0\x0b\xcd\x80"
```

This should finally suffice to pull off the Final Stage! Somehow you must code up the program `egg` so that Nick Weaver's `exploit` script can launch the final, fatal strike:

```
#!/bin/sh
echo "sending exploit"
./egg | nc 127.0.0.1 42000 &
sleep 1
nc 127.0.0.1 6666
```

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 and pipes it to `nc`. Make sure your scripts work by invoking `./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 (10 points)`.

You must also submit a write up for this question in `explanation.pdf` in the same fashion as for the previous question `(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

In summary, you must submit the following directory tree on `glookup`:

```
q1/egg
q2/egg
q3/arg
q3/egg
q4/interact
q5/egg
```

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. Instead, you can use the `bundle` command. This will aggregate all of your submissions into a single directory, which can then be copied safely onto your computer. Once you have done so, please submit your answers to these questions on `glookup`.

You must also submit your writeup `explanation.pdf` with your solution to each question on Gradescope.

References

[1] Aleph One. Smashing The Stack For Fun And Profit. *Phrack*, 7(49), November 1996. http://www-inst.eecs.berkeley.edu/~cs161/fa08/papers/stack_smashing.pdf.

[2] Tilo Müller. ASLR Smack & Laugh Reference. http://www.icir.org/matthias/cs161-sp13/aslr-bypass.pdf, February 2008.