

Project 1 - Part 2 : Stack Buffer Exploit

Learning Goals of this Project:

- Execute a stack buffer overflow exploit
- How stack buffer overflows work
- How stack registers are manipulated during program execution

The final deliverables:

You will submit `<canvas username>_data.txt` in 2 places.

1. **Project 1 - Part 2 - Autograder** on Gradescope.
2. **Project 1 - Part 2 - Stack Overflow Exploit** on Canvas.

The submission name format is `<canvas username>_data.txt` for example:

`ctaylor308_data.txt` or `jlohse3_data.txt`, be sure to use your login **and not** your 9-digit Student ID.

***** If you do not submit to both (Canvas and Gradescope), you will receive a 0. This is non-negotiable and will be enforced heavily. *****

Tools Needed :

- Use the latest version of [VirtualBox](#) (any 6.x.x version!)
- If you are using an M1 based Mac, you will need to follow the document showing how to use [Azure Cloud VM Instructions](#)
- Download the VM OVA **cs6035_project1_vm.ova** from the following link:
 - [VM Download](#)
 - SHA256 Checksum:
20dbc08724bef87999891b0998f9934e65addc1b511866498155efddf8dee83b
- **VM Tutorial/Overview:**
 - If you don't know how to use virtualbox or want to see what a successful exploit looks like in a live video, watch this link below
 - [CS6035 Project 1 VM Walkthrough](#)
- GDB Tutorials
 - Here is an in-depth video showing some common usage for GDB in a live program
 - [CS6035 Project 1 GDB Tutorial](#)
 - Text based: [Intro GDB \(pwndbg\) Tutorial](#)
 - GDB command [cheat sheet](#)

NOTE: Using an old VM from earlier semesters is not permitted and WILL result in a 100% penalty. If you are retaking this course you will have to download the VM again from the link given above.

Submission:

Late Work Will Be Penalized or Not Accepted

Please refer to the course syllabus for more information about submitting, deadlines, and deductions.

Virtual Machine:

- VM Name: cs6035_project1_vm
- User Name: kali
- Password: kali

Project Idea:

You'll be performing a [return-to-libc](#) attack. This kind of attack usually starts with a [buffer overflow](#) in which a subroutine [return address](#) on a [call stack](#) is replaced by an address of another subroutine that is already present in the [process](#) executable memory. This other subroutine usually exists as part of the standard c libraries or "libc", hence the name. The original subroutine "returns to libc" by calling that new function instead of properly returning to the caller.

Project Tasks (80 points):

Run:

The VM will automatically download the files needed to run the exploit in

`~/Desktop/32_bit_overflow`.

These files include an executable `exploit` and its source code `exploit.c`. You will create an input file as described below.

NOTE* From time to time, and depending on your location (if your country does a lot of filtering for instance) If your VM has issues downloading these files you can run the following commands to download them yourself. Or you may have to manually download them and copy over using a shared folder

```
$ cd /home/kali/Desktop/32_bit_overflow
$ curl -L -o exploit
"https://drive.google.com/uc?export=download&id=1nWfNi7BnLVJEQ8oausstUmYOqxDuAe-B"
$ chmod 777 exploit
$ curl -L -o exploit.c
"https://drive.google.com/uc?export=download&id=11cVTpJAm3CJXLYdu3Na0fsL4B2_4Vf4F"
```

You will notice that `exploit.c` sorts the information included in `canvasusername_data.txt`

Completing the Exploit:

1. Open the terminal and navigate to the exploit folder with the command:
cd /home/kali/Desktop/exploit

2. Create a text file with the command:
touch canvasusername_data.txt

If your canvas username is ctaylor308 you would use: ctaylor308_data.txt

3. Open the text file with the command:
mousepad canvasusername_data.txt &

You'll want to keep the text file and the terminal windows open as you'll be editing in both.

4. Also open the source code for the program you'll be exploiting with the command:
mousepad exploit.c &

It will open it in the same window as your text file and you can switch between them with the tabs at the top.

5. Let's examine the exploit.c source file to learn more about the code and how to exploit it. The Sort function starting on line 17 reads the data from our text file, sorts it, and outputs it to the terminal. The sort on lines 34-49 simply sorts the data by ascending order. The output on lines 51-54 merely displays the data in that sorted order. The more important part of the program to understand is where the file is being read in on lines 13-32. You'll want to step through this code by hand and make sure that you understand everything it is doing. To put it simply, however, a line from the file gets read into a character array (named line) that can hold 8 characters. That line is then written to a larger array (named array) of uint32_t and printed to the terminal (prior to sorting). This array is the buffer we'll be exploiting as it is designed to hold 5 uint32_t values but keeps reading values until the end of the file due to the while loop on line 23, allowing us to overflow it.
6. Before we can overflow anything we need to disable the ASLR protection. You'll need to find the command to use in the linux terminal to temporarily disable it. You'll need to disable it every time you restart the operating system.
7. Let's try a brute force attack. Add a line of 8 characters to your text file such as:
AAAAAAAA

Save the file.

8. Now in the terminal, we'll run the program with our text file with the command:
./exploit canvasusername_data.txt

Now we see that the output contains "aaaaaaaa" in both the original and ascending order. But nothing else unexpected has happened though. Our goal is to break the normal operation of the program to exploit the system by overflowing the return address.

9. You know that you've overflown the frame pointer when the program segfaults. So to find the location of the frame pointer, add a new line of 8 characters to your text file (save the

file) and run the exploit with the command from step 8. Keep doing this, one line at a time, until you see a segfault.

10. Once you get the segfault you know you've overflowed the frame pointer and that the next address should be the return address, where you want to place your malicious data. This garbage data that you've entered is the padding required before your malicious data. But why was it as much data as it was? We can actually manually calculate the amount of data that was required to get to this point and confirm that our padding is correct.
 - a. In the terminal run GDB with the command:
gdb exploit
 - b. We need to set a breakpoint in order to examine the program. You can pick multiple lines but for now we'll choose the Sort function. Enter the command:
break Sort
 - c. Now run the binary with your data file, which will stop when we hit the breakpoint
run <your data file>.txt
 - d. Now we can find the address of the frame pointer, which is stored in the EBP register. Use the command:
print \$ebp
 - e. We'll also need the starting address of our buffer to determine the amount of padding we need. Enter the command:
print &array
 - f. Using a hexadecimal calculator, you can subtract the address of the buffer from the address of the frame pointer. You should get a difference of 28 bytes. Recall that the start of the return address is after the frame pointer (at EBP + 4). This means we add 4 bytes to the 28 bytes for a total of 32 bytes. So we need 32 bytes of padding to fill our buffer all the way up to the return address, so that the next byte will overflow the return address. Reexamining our brute force attack, take the number of lines of padding you had and multiply them by 8 bytes (1 byte per character). You may notice that this is too much data; double the data in fact, as you should have gotten 64 bytes. How can this be? Well, recall that exploit.c stores the data in an array of type uint32_t, not char. The translation from the 8 char line to the uint32_t array cuts the data in half (uint32_t values occupy only 4 bytes). So really you should multiply your lines of padding by 4, not 8. Now you should have the correct number of bytes and understand why your brute force attack worked.
11. In this exploit, we'll replace the return address with the address of the **system()** function. This will allow us to supersede our permissions and have the system run something for us. In this case, we want to open a shell as **/bin/sh** as opposed to the **/usr/bin/zsh** shell we are currently in. We will also want to **exit** cleanly so we don't get any errors. While still in GDB from the last step, you'll want to use GDB to find these addresses. Once you've found them, add them to your text file after your padding (saving the file again).
 - a. Find the **system** function address.
 - b. Find the **/bin/sh** address.
 - c. Find any **exit** function address that exits the shell cleanly (no errors displayed).

Important Note:

You are **NOT** allowed to use environment variables to store these addresses.

12. Quit out of GDB with the command:

quit

13. Now try running the exploit again with the command you used in step 8. After that, check to see if you're in the **/bin/sh** shell with the command:

echo \$\$ \$0

Compare your results with the screenshot in the **Verifying your Results** section below. If there is a white \$ waiting for your input and the echo statement returned **/bin/sh** then you've successfully entered the correct shell. If not and the program had a segfault, then one of your addresses is wrong. It is likely your exit address, try replacing it with a copy of either your **system** or **/bin/sh** address and see if that works.

14. If you have the white \$ awaiting input, then enter the command:

exit

If you get a segfault then your exit address is incorrect; you need to find one that works with the sorting of the program (google **man pages exit** and then investigate the **see also** section). If you do not get a segfault and your results match the screenshot below then you've successfully completed the project.

15. Upload your text file to gradescope and confirm your score as in the **Submitting to the Gradescope Autograder** section below. If you received full credit, then also upload to canvas.

Verifying your Results:

Your goal is to craft your **canvasusername_data.txt** in such a way that when you run **./exploit canvasusername_data.txt** a shell spawns (note the '\$' symbol which means you are at a shell in the screenshot) where you will then type the commands:

```
$ echo $$ $0
```

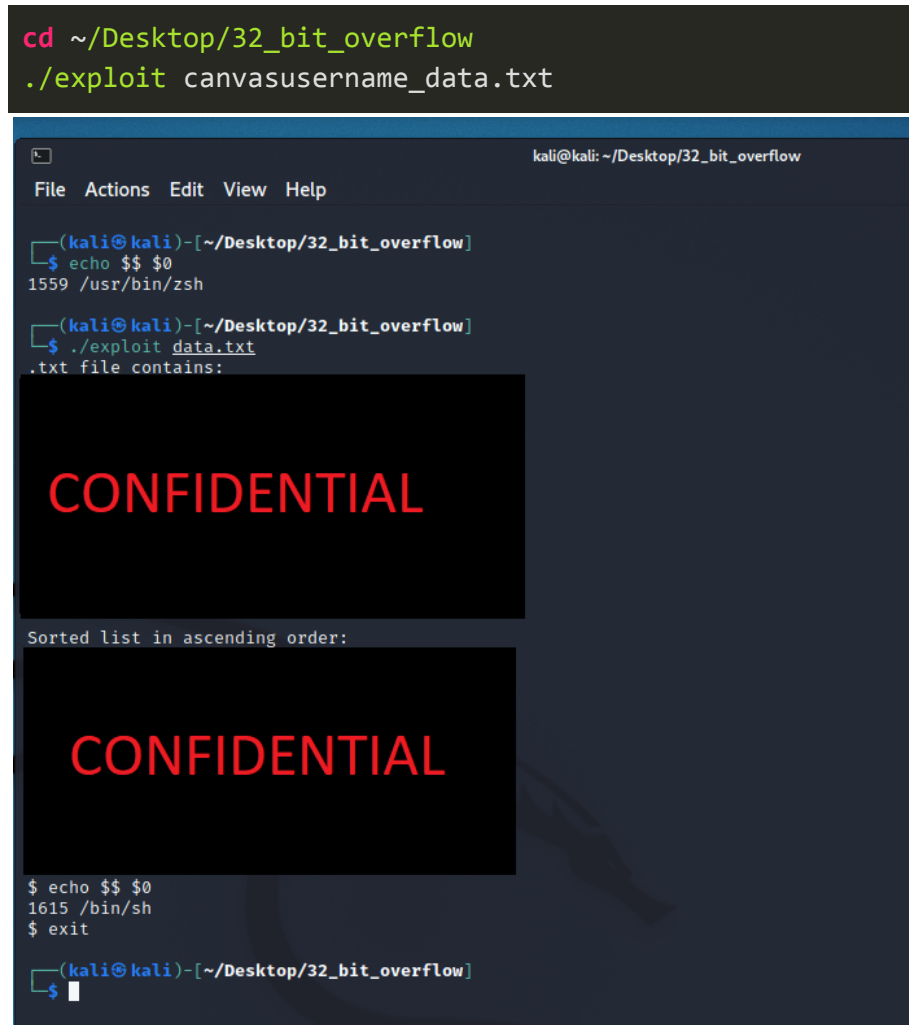
and

```
$ exit
```

The program MUST exit cleanly, which means ZERO error messages are displayed and you return to your normal shell, if you have any follow-up questions please check the video tutorial in the beginning of the instructions

Here is an example of a successful execution NOTE: example shows data.txt, not canvasusername_data.txt:

```
cd ~/Desktop/32_bit_overflow
./exploit canvasusername_data.txt
```



```
(kali@kali)-[~/Desktop/32_bit_overflow]
$ echo $$ $0
1559 /usr/bin/zsh

(kali@kali)-[~/Desktop/32_bit_overflow]
$ ./exploit data.txt
.txt file contains:

CONFIDENTIAL

Sorted list in ascending order:

CONFIDENTIAL

$ echo $$ $0
1615 /bin/sh
$ exit

(kali@kali)-[~/Desktop/32_bit_overflow]
$
```

Important Notes:

- Your terminal should show the values both before and after sorting
- No segfaults are allowed. This must execute a clean shell, then exit cleanly.
- Be sure you check the ID and name of the shell you are in before and after with the echo commands to verify that they've changed and the exploit was successful.

Submitting to the Gradescope Autograder

- Your **canvasusername_data.txt** must be submitted to the Module: **Project 1 - Part 2 - Exploit Autograder** on Gradescope.
- **NOTE:** For the autograder to work, you **MUST** create **AND** submit your **canvasusername_data.txt** from **WITHIN** the VM. It is possible to create and submit from outside the VM; the autograder supports multiple encodings, but you run the risk of receiving a **0** if it doesn't support your system's. You will **NOT** be refunded any attempts or given any consideration for this.
- **The total number of submissions allotted to each student is limited to 5.** This means that you can submit your **canvasusername_data.txt** a maximum of 5 times without penalty.
- **IF YOU SUBMIT MORE THAN 5 TIMES, THE AUTOGRADER WILL NOT EVALUATE YOUR SUBMISSION AND YOU WILL GET A 0**
- Along with the test results, the autograder will display a Submission Count Disclaimer with information about your submission count.

Submission Count Disclaimer (0.0/0.0)

Submission #2/5. You can resubmit 3 more times.

- **NOTE:** You will also see the **Submission Count Disclaimer** under **Passed Tests** section on the results page even after you exceed the submission limit. This is related to how Gradescope works and does not hold any meaning for the students.
- To check your past submissions:
 - a. Click on **Project 1 - Part 2 - Autograder**
 - b. Click on **Submission History** at the bottom of the page

[🕒 Submission History](#)[📄 Download Submission](#)[Resubmit 📤](#)

This should show your past submissions results.

Submission History

#	SUBMITTED AT (EDT)	SUBMITTERS	SCORE	ACTIVE
7	May 10 at 8:52PM	JT	0.0	<button>Activate</button>
6	May 10 at 8:52PM	JT	0.0	<button>Activate</button>
5	May 10 at 8:51PM	JT	50.0	<button>Activate</button>
4	May 10 at 8:49PM	JT	25.0	<button>Activate</button>
3	May 10 at 8:29PM	JT	80.0	✓
2	May 10 at 8:28PM	JT	80.0	<button>Activate</button>
1	May 10 at 8:22PM	JT	0.0	<button>Activate</button>

✕ Close

- Please note - **Submission #6 and Submission #7** above are not graded by the autograder and have been assigned 0 for exceeding the allotted submission count.
- If you submit more than once, you are free to **Activate** any of your past submission scores before the submission deadline. The score with the check mark ✓ (see image above) is your current activated score. By default, the **Activated** score will be the latest submission.
- **IMPORTANT NOTE:** We will only consider the **Activated** score as your final score for the project.
- **By default, the score of your latest submission is Activated. IT IS YOUR RESPONSIBILITY TO ACTIVATE THE SCORE YOU WANT IF IT IS NOT THAT ONE.**

Debugging Tips

1. Make sure you disabled ASLR on every boot (it is possible to permanently disable this, but it is a good idea to check even if you disable “permanently”)
2. Make sure you understand exploit.c and what the code is doing
3. Verify your address are correct (typos are common)
4. Verify the addresses are in the correct order *after the sort*
5. Verify your padding is correct – Review the walkthrough above (brute force is also an acceptable method of figuring out the padding)
6. Run locally and verify a clean exit with no segfaults and correct echos
7. Run on the autograder and verify the same result

Grading Rubric:

Please remember only 5 submissions to Gradescope are allowed. Any submissions after that will not be evaluated by the autograder. It is your responsibility to activate the submission you want graded (by default, it is the latest submission). [See above](#).

You will only get credit for a **SUCCESSFUL EXPLOIT** (i.e, entering the shell successfully). Partial credit will be awarded if the exit is not clean.

Note: Remember that these milestones ***must be met outside GDB***.

Milestone 1: Successful Exploit (entered the shell)	50 Points
Milestone 2: Clean Exit (exited the shell with no errors or seg faults)	30 Points
TOTAL	80 Points

*** Remember to submit to both Canvas and Gradescope.***

Appendix:

Using an M1-based Mac

- Technically, you can run QEMU to emulate the x86_64 architecture, however depending on your hardware your mileage may vary significantly and this should only be used as a last resort (It will be quite slow)

Install Homebrew if you haven't already:

- <https://brew.sh/>

Install Qemu with command:

```
brew install qemu
```

Now download the cs6035_project1_vm.ova file from setup instructions

Open a new terminal, and cd into wherever the OVA was downloaded

Run the commands:

```
mv cs6035_project1_vm.ova cs6035_project1_vm.tgz
tar xvf cs6035_project1_vm.tgz
qemu-img convert -f vmdk -O qcow2 cs6035_p1p2_v4-disk001.vmdk
cs6035_p1p2_v4-disk001.qcow2
qemu-system-x86_64 -hda cs6035_p1p2_v4-disk001.qcow2 -m 4096 -cpu
qemu64 -smp 8
```