Project
Capture The Flag!

Learning Goals of this Project:

Students will learn introductory level concepts about binary exploitation. This lab develops understanding of control flow hijacking through different tasks/challenges designed to show certain vulnerabilities or weaknesses in a C program. A python library pwntools will be used to show some exploitation techniques and automation to successfully hack a program

The final deliverables:

A single json formatted file will be submitted to Gradescope. This file should be named ***project_ctf.json.*** A template can be found in the Home directory.
See Submission Details for more information

Important Reference Material:

Assignment Project Exam Help

This Intro to **pwntools/pwndbg video** showing how to automate some exploits and use our exploit framework on the VM

- If you're an *absolute beginner* with no Linux experience, **This Website** may be able to help
- **pwntools Documentation**
  GDB command **cheat sheet**

https://powcoder.com

Submission:

Add WeChat powcoder

Gradescope (autograded) - see Submission Details

Virtual Machine:

(Note: downloads can be very slow when project first releases due to very high traffic in first few hours/day)

- Parallels vm for apple m1 based systems
  - You can install Parallels for mac here
  - VM Download Link
  - Username: parallels, Password: password
- Intel/AMD x64 version (suggest using 6.1.16 but can try any version if already installed)
  - VM Download
  - Windows Virtualbox 6.1.16 Download
  - Mac VirtualBox 6.1.16 Download
  - Username: cs6035, Password: cs6035
- Note, there is no root permissions on the VM

00_intro

Step 1: Open a terminal and cd into the project directory **`project_ctf/00_intro`**.

```
$ cd ~/project_ctf/00_intro
```

Inspect the contents of the readme file

```
$ cat readme
```

Follow the instructions in the readme to modify e.py with your GTID (9 digit numeric school ID number that looks like 901234567 and afterwards execute the script to get your first flag! Your output will look like this. Copy this submission hash and place in the json file in your home directory ~/project_ctf.json

```
cs6035@cs6035:~/project_ctf/00_intro$ ./e.py
[+] Starting local process './flag': pid 6481
/home/cs6035/project_ctf/00_intro/./e.py:33: BytesWarning: Text is not bytes; assuming ASCII, no guarantees. See https://docs.p
  p.sendline(payload)
[*] Switching to interactive mode
nateclaflin
binary hash = 5461f2bb95f6aee1adaad76fb7bb3c8b75d51443caae089a869c018b500033a2
WOO! Your Submission Hash is: f8e6[illegible]7daccc336a1dc35fd5fbd1f853a8b27e7102b2f1b

[*] Process './flag' stopped with exit code -17 (SIGCHLD) (pid 6481)
$
```

Assignment Project Exam Help
https://powcoder.com
Add WeChat powcoder

SUBMIT YOUR FIRST FLAG TO MAKE SURE IT WORKS BEFORE CONTINUING

Also, it is a very good idea to submit each flag you get to make sure it works before moving on, in case of any issues

(Applicable for all flags): If for whatever reason you don't get a flag and you're positive you should, try running the exploit once or twice. The flag generator can have some unexpected behaviors. When in doubt, make a post in Ed Discussion to 'All Instructors' and we will assist you if possible

01_buffer_overflow_1

(watch the intro video first please, or if you want to try the experimental instruction program BoxxY, see Appendix for details, for Intel/AMD chips)

This task is a very simple buffer overflow that, upon inspection, will check if a variable is non-zero. Using the information you have gathered from reading and the videos, it is your task to get this program to get to the call_me() function, and get the flag printed.

Note: you are free to use GDB if you need to for this project but you need to run the program on the command line (i.e. ./e.py) in order to get the real flag for submission and submit it!

Assignment Project Exam Help

https://powcoder.com

Add WeChat powcoder

01_buffer_overflow_2

In this task you will learn details about binaries compiled from C code, and how some basic things **can be exploited such as process redirection or control flow hijacking. The steps in this flag are discussed in-depth in the intro video.**

In this directory you have an executable binary named 'flag' which is vulnerable to a buffer overflow in one of its functions. We will be using an exploitation library called pwntools to automate some of the overflow techniques and get the binary to call a function it otherwise wouldn't have. This function called 'call_me' generates a key using your Gradescope User ID to get a valid flag that will pass the autograder.

Now we will run the binary just to see what the program is doing by running the executable

```
$ cd ~/project_ctf/01_buffer_overflow_2
$./flag
```

Assignment Project Exam Help

```
[*] Interrupted
cs6035@cs6035:~/project_ctf/01_basic_overflow_2$ ./flag
Feed Me A Stray String:
```

https://powcoder.com

Add WeChat powcoder

We see the binary is asking for a string, input any text you want or just press enter, and you'll see that the program does nothing and just exits. That's just to simplify the code so we can focus on the exploit.

The binary is statically linked to a shared object which has a lot of methods that construct the key and has a simple function called 'call_me()' which will print out your key.

This is where we will start learning about binary file formats. Without going into a deep dive about program structure, operating systems, compilers, assembly language, machine code, etc. you will still be able to understand that there are two aspects that are key in binary exploitation

- **Data**
  - is simple enough, it is just any collection of bits that represent some kind of data element (like an ASCII character, integer value, pointer, etc)
- **Addresses**
  - At this scope we can just think of addresses as fully unique identifiers of specific data elements. These are logical locations the computer understands.

A buffer overflow occurs when too much data is fed into an unprotected (or poorly protected) data buffer. The way that 64-bit C programs work is, a small amount of bytes past the beginning of the stack frame, data is stored at an address called the ***Instruction Pointer*** which is a register pointing to the currently executed instruction. If we override this with a valid address we can manipulate the control flow of the program and have it execute arbitrary (or otherwise unintended) code, with a well-formed attack. Starting off easy, we are going to modify `e.py` and learn a few basics of the pwntools library.

Open `e.py` with your favorite text editor and analyze the content and comments.

Once you understand what they do, proceed to fill in the cyclic size (this number is up to you, based on your understanding of the program and what would break it) to get a segmentation fault message by running

```
$ ./e.py dbg
```

This will open up a gdb terminal with a breakpoint set at main()

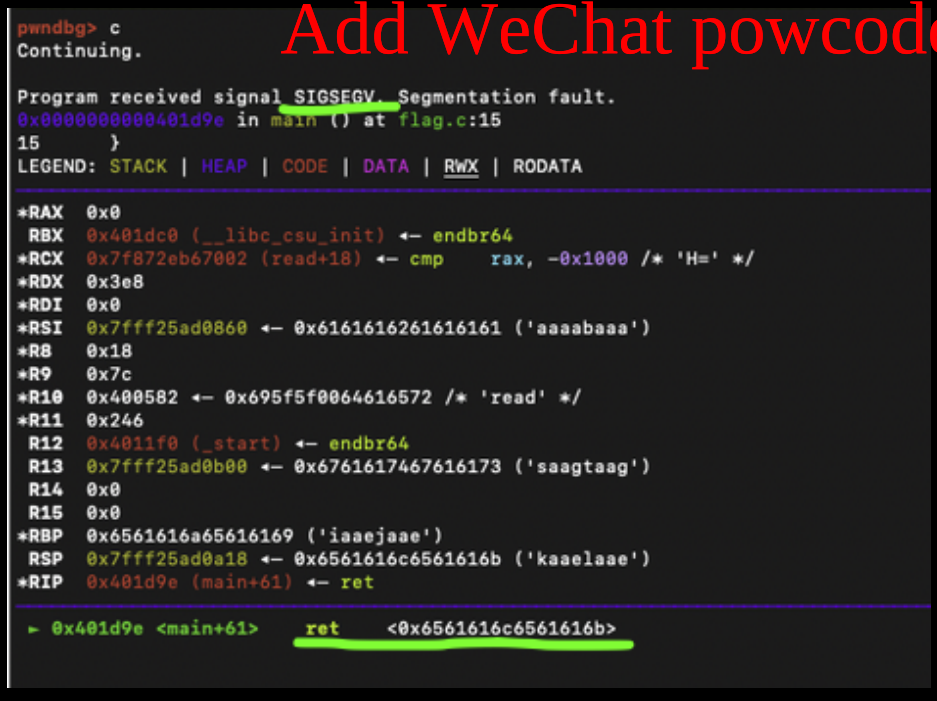
Assignment Project Exam Help

Type 'c' to continue from the breakpoint (**sometimes need to press 'c' twice if you don't see the error, this is an issue with how gdb attaches to processes**)

https://powcoder.com

Note: The screenshot below is taken from an Intel based OS. For ARM based OS (Apple M1), the registers will be different. For more info you can visit the ARM Documentation

Add WeChat powcoder



We see the program received an interrupt signal for a **SEGMENTATION FAULT** (SIGSEV, or an invalid access to memory) or on AARCH64 it will be a **SIGBUS** Error. This happens when the program tries to access memory at a certain location that it either isn't allowed to access, or doesn't exist. In this case the return address for the function was overwritten by cyclic()'s data in the form of long strings of characters. Pay attention to the bottom of the screenshot where the instruction pointer is currently trying to 'ret' (return) to 0x6561……616b which is just a string of ascii characters in hexadecimal form.
Now we know how to break the binary, let's figure out how to purposefully break it. Using a pwntools method called 'cyclic_find()' we enter in the bottom 32 bits (4 bytes) of the return string (in this example is 0x6561616b) which will give the number of characters before reaching that value. This is **important** because we are now going to reach our first step of **control flow hijacking** by overflowing enough data that we can place a value and change the course of the program's normal path.

In **e.py**, on the commented line below your cyclic command, we are now going to use **cyclic_find()** which will automate our buffer length calculation, and feed that number into cyclic(). Place in your 4 character bytes (preceded by a 0x, like 0x6561616b). Uncomment the two lines beneath our original cyclic() call, and fill in the hex value described above. This will fill the buffer with our calculated buffer length, appended by the ASCII byte equivalent of the variable by using another pwntools method **p64(<string>)**.

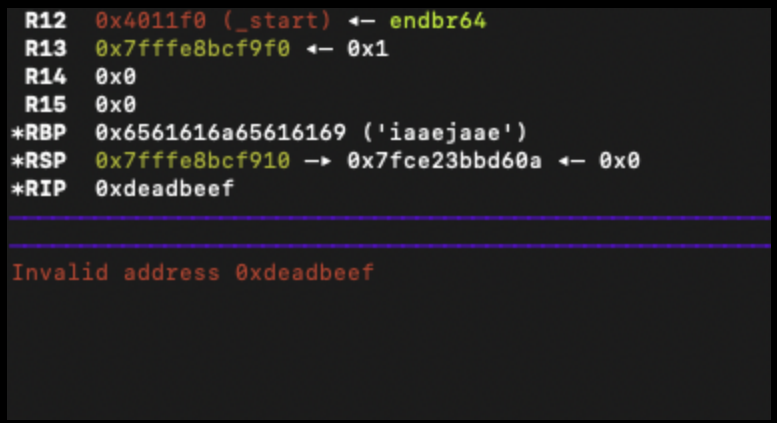
Assignment Project Exam Help
https://powcoder.com

After you have done that, rerun

```
./e.py dbg
```

Add WeChat powcoder

And hit 'c'



If done correctly, you should see something like this screenshot, where if you check the 'ret' instruction, we are now failing on an invalid access to our dummy address.

Stepping away from the pwntools library for a moment, we now need to find something **usable** within the binary that will allow us to actually call a function or do something other than just crashing the program.

Now we will use a linux command 'objdump' which takes a binary file and will output a **dump** of the binary which will give some key information about the binary. The `-D` flag will output binary addresses, machine code, and assembly code of the binary into a file.

```
objdump -D flag > flag.asm
```

Then open **`flag.asm`**

You will see a bunch of (likely) confusing information that at a high level translates to the code that you can see in the 'flag.c' file. You aren't going to have to go through this file in any extreme expanse (unless you want to?) we are just going to focus on finding an address within the binary file that holds the machine code responsible for making a function call to 'call_me()'.

Assignment Project Exam Help

Search for the string '**`call_me`**' in **`flag.asm`** and keep looking until you find the assembly instruction:

https://powcoder.com

For Intel/AMD CPUS:

```
call <some address> <call_me>
```

For M1/aarch64 based systems

```
bl <some address> <call_me>
```

Add WeChat powcoder

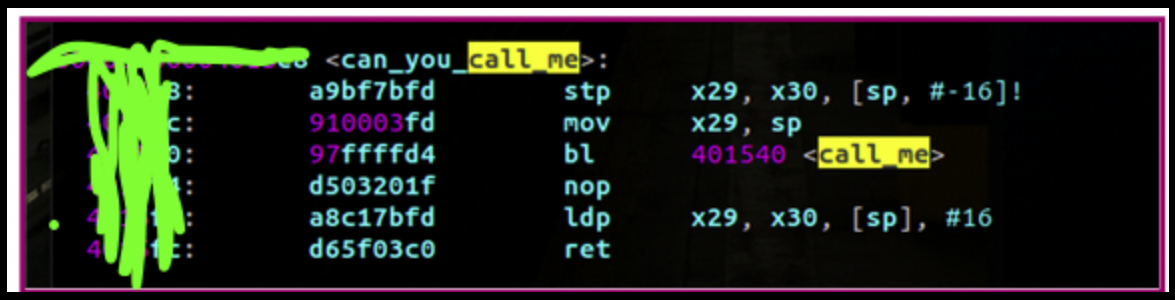
For Intel/AMD CPUS: Note down the highlighted address showing the call (it will be different in your binary):

```
  401bdd:       c9                      leave
  401bde:       c3                      ret

0000000000401bdf <can_you_call_me>:
  401bdf:       55                      push   %rbp
  401be0:       48 89 e5                mov    %rsp,%rbp
  401be3:       b8 00 00 00 00          mov    $0×0,%eax
  401be8:       e8 20 ff ff ff          call   401b0d <call_me>
  401bed:       90                      nop
  401bee:       5d                      pop    %rbp
  401bef:       c3                      ret

0000000000401bf0 <__libc_csu_init>:
```

On M1 / aarch64 based systems, the command will be a `bl` that looks like this.



Now open `e.py` and adjust the line (see the commented useful commands section)

```
payload += p64( 0xdeadbeefdeadbeef )
```

With the hexadecimal value of the address above (prepend 0x to the value highlighted)

Now run `./e.py` again from the command line (without dbg) and check the terminal output.

Assignment Project Exam Help

Did you get it? Awesome! Submit your first flag to gradescope (follow APPENDIX for more details)

- If not, retrace your steps in this task and also make sure you used the `call call_me` address in the earlier step and not the address of the actual function `call_me()`

https://powcoder.com

Add WeChat powcoder

02_assemble_the_assembly

This task will get you to determine which assembly instructions will properly construct a call using the address of the call_me() function (the actual address of the function, as opposed to task 1 which needed the call to a function). Analyze the different instructions and look up the usage/behavior of them to figure out which one will construct the address.

You can use objdump or gdb to find the address of call_me() and figure out how you calculate it.

For debugging, I highly recommend using gdb, setting a breakpoint on the gadget function, and stepping through the options once you think you know the correct path to get to the function call.

(FYI:: you don't have to use pwntools for this one)

Assignment Project Exam Help

https://powcoder.com

Add WeChat powcoder

02_bad_rando

This Program (very conveniently) leaks out part of the libc base address

- this address is randomized via ASLR so it will change a little bit every time the program is launched
- run the program a few times and notice what bytes are different and which ones aren't

Next step will be analyzing the C file and see what we are comparing against in order to get to 'call_me'

- system() is a libc function, use GDB to get the address of system using 'p system'
- **repeat printing the system address a few times (CLOSING GDB BETWEEN EACH RUN) and see if you notice a pattern for the address returned from system, libc is a group of c functions that always have a static offset from the base, even when the addresses are randomized**

Assignment Project Exam Help

Fortunately there's only one byte that is missing from our formula, so we can do some scripting in python to try out the remaining values.

https://powcoder.com

- pwntools has a function called recvlineuntil(): that will let us do some manipulation with the string returned (before we send the payload) and allow that to coerce the input we send in.

Add WeChat powcoder

- the recv functions will return a BYTES object, so you will need to do some clever manipulation of said strings that are returned, this will probably take a few iterations and permutations to get the value in the right format

- note that the C file is using scanf to read in a hexadecimal number, meaning you don't need to use p64(), you are sending in the STRING REPRESENTATION of a hex number, that means WITHOUT the '0x' in the beginning, and you send the string directly on the command line like 'ffaabbccdd' or 'f701234abcd' etc!

- your task is going to be:
    - get the value leaked from the program
    - modify it with the offset of the system() function
    - fill in the remaining byte with a random value
    - send to the process
    - (repeat until you get a flag)
        - note: i recommend using recvall() after you send in each payload, and write your loop logic around the output (see other flags for what kind of string output you can expect) to see if you got the right value!

02_p4s5w0rd

STRINGS!

Now it's time to learn a really useful technique to find all the available strings in a program.

And by strings, we mean any collection of printable characters that exist in the binary. So things like variable names, hardcoded paths, debug messages, or eeeeevenn.... passwords? Hopefully not in a real program but you would be surprised.

Assignment Project Exam Help

This binary has zero debugging information and you do not have the source code available, but guess what? The program is written terribly and is very unsafe, with passwords stored in plain text that can easily be dumped/searched in the binary!

https://powcoder.com

I would recommend running the program once or twice to see what it's doing (checking a series of responses to questions) and if you get every question right, then you will get the flag!

Add WeChat powcoder

To get the strings for the program, run the command:

```
$    strings flag
```

This will output it all to the terminal which isn't super helpful, so would suggest redirecting output to a file like:

```
$    strings flag > flag_str
```

Now you will be able to grep/search/navigate the file in a new terminal and will (hopefully) be able to figure out what the correct responses would be for the given questions.

(hint, strings are stored in the binary in the order that they're written in the C code, might be a good idea to search for the questions they're asking and it should be pretty easy to determine the answer from there!)

Good luck!

02_the_server_client_one

This flag shows a communication between a server and a client. The client binary (flag) will send data to the server, and the server appends some (very conveniently structured) data to that message and sends it back to the client. Your goal for this task is to have the server return the ideal data to overwrite the instruction pointer with the data that is returned from the server.

Follow the same steps in previous tasks (buffer_overflow_2, more specifically) to break the program in gdb, and then figure out your buffer size, and try to fill in the response to correctly hit this function call!

If you use the pwntools e.py file, it will start the server for you so there is no need to explicitly start the server.

Assignment Project Exam Help

If you are running the program on the command line to experiment, then you must start the server each time you run the binary. You can either open a new terminal, and run

https://powcoder.com

```
./server
```

Or in the same terminal, each time you run the binary, run

Add WeChat powcoder

```
./server &
```

Your task is to figure out the breaking point, and heavily inspect the last bytes that are returned from the server in order to get the right return and get the flag!

03_XORbius

Time to rev up those Reverse Engineering motors, because you need to unravel the logic that this program is checking against in order to get to the call_me() function!

No buffer overflow this time, you just ‘simply’ need to input the right values that will correctly decode the logic and pass the checks.

If you’re unfamiliar with C operators, this TUTORIAL has all the necessary operations detailed.

Suggest pen and paper for this one to work through the logic by hand, or do a ton of experimentation to get the right value!

Assignment Project Exam Help

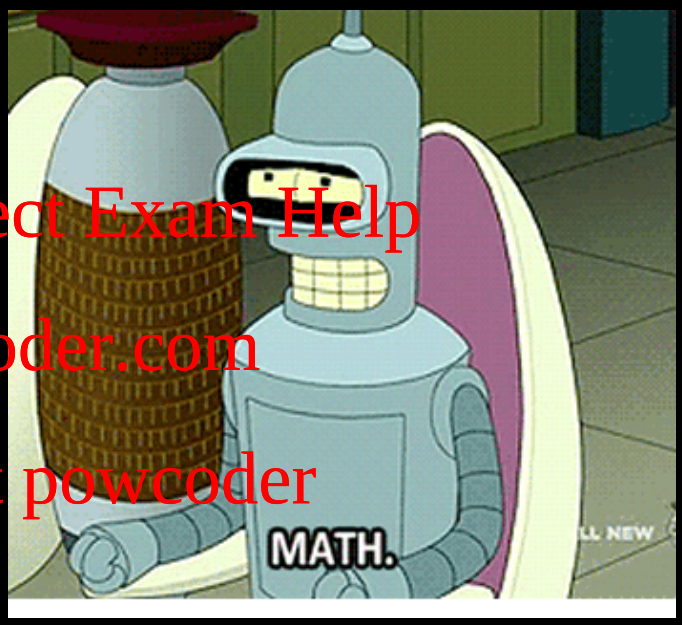
https://powcoder.com

Add WeChat powcoder

03_pointy_pointy_point

We see there is an unsafe() function which has some checks for different local variables. The positioning of these variables is important because they are declared _before_ the input buffer which means that a buffer overflow will cause data to be overwritten.

You will find additional details on this flag in the readme file of the folder. This program is a Buffer Overflow, however you will not be changing the control flow to a specific binary address, rather you will need to enter in the right values to trick the pointer arithmetic logic and get to the call_me() function.



Assignment Project Exam Help
https://powcoder.com
Add WeChat powcoder

03_hunt_then_rop (INTEL_AMD_x64 VERSION)

(if you have an Apple M1-based Mac please go to next section)

You've made it! You are now on your final task. In this directory is the entire contents of /usr/bin, a collection of binary files that make up a lot of common linux uses. One of these files has been overwritten by a vulnerable program. It is your task to figure out which one. You are given a list of checksum values that are known good, so your first task will be determining the sha256 hash of all of the files in this directory, and then finding the one that does not match. You are free to do this however you would like. **NOTE: in your scripting method, ignore the files 'checksums' and 'user.txt'.** They will likely report a mismatch but you can be certain neither are not the file in question

Once you find the file it is time to begin our exploit of that file. This is a bit more complex than the other flags and will require a full ROP (return oriented programming) exploit to chain calls together, and we will also need a new tool called Ropper to find a 'gadget' in order to supply a function argument and pass a specific check.

Assignment Project Exam Help
https://powcoder.com
Add WeChat powcoder

In 64-bit programs, the function gets arguments through registers, in the case of intel architecture the RDI register supplies the first function argument.

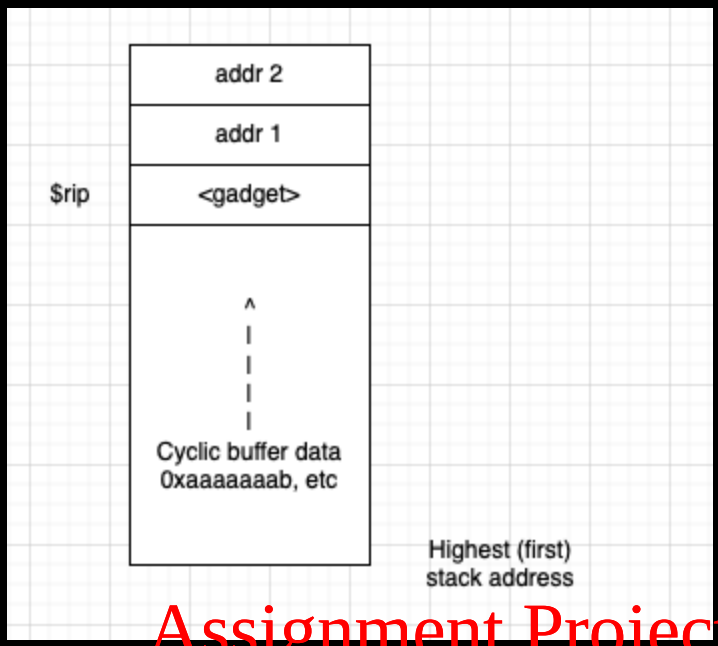
So we need to find a gadget (a piece of code that we can override the instruction pointer with, that will perform a certain action and then continue with the control flow hijack) that will pop a value from the stack into the RDI register.
-
Let's use ropper like this

```
$ ropper --file flag | grep "pop"
```

This will give you all gadgets within the binary that have a keyword 'pop' (spoiler, there's a LOT of them). An objective for this task is to figure out what gadget will likely work best to get the required argument passed into the function you are trying to call. **This Writeup** is a helpful reference to understand how calling convention works for x86_64 cpu's
Kali

Note the addresses that are output for each gadget. Once you find a gadget you think will work, we will need that as our first override value in pwntools

Pictorially, this is what our crafted exploit needs to look like (remember stack grows down)



Assignment Project Exam Help

Now we will need to supply the argument, which will be on the stack immediately after our pop gadget, figure out what that value needs to be, and add it as p64(<value>) after the pop gadget

https://powcoder.com

Then we need to put the address of the function as the next call, use objdump or gdb to find the addresses (you should probably get the second function address while you're at it). The call to our pop gadget will 'ret' and then hit this second function call to enter one of the unsafe functions

Add WeChat powcoder

Finally, we need to finish our execution chain by calling the second function which will allow for exploitation. Append that address to your chain and see if you get a flag!

03_hunt_then_rop (APPLE_M1_AARCH64)

You've made it! You are now on your final task. In this directory is the entire contents of /usr/bin/, a collection of binary files that make up a lot of common linux uses. One of these files has been overwritten by a vulnerable program. It is your task to figure out which one. You are given a list of checksum values that are known good, so your first task will be determining the sha256 hash of all of the files in this directory, and then finding the one that does not match. You are free to do this however you would like. **NOTE: in your scripting method, ignore the files 'checksums' and 'user.txt'.** They will likely report a mismatch but you can be certain they are not the files in question

```
$ cat checksums
```

Assignment Project Exam Help

Once you find the file it is time to begin our exploit of that file. This is a bit more complex than the other flags and will require a full **ROP (return oriented programming)** exploit to chain calls together, and we will also need a new tool called Ropper to find a 'gadget' in order to supply a function argument and pass a specific check.

https://powcoder.com

In 64-bit programs, the function gets arguments through registers, in the case of aarch64 based ARM CPUs, the x0 register supplies the first function argument.

Add WeChat powcoder

So we need to find a gadget (a piece of code that we can override the instruction pointer with, that will perform a certain action and then continue with the control flow hijack) that will pop a value from the stack into the appropriate register. A **helpful reference here** should show you which registers might be useful to you. We want a gadget that will use the LOAD REGISTER (ldr) operation to access something from the stack pointer [sp].

use ropper like this (See: screenshot for example output)

```
ropper --file flag | grep "ldr"
```

```
parallels@ubuntu-linux-20-04-desktop:~/project1_CTF/apple_m1_aarch64/flag1$ ropper --file flag | grep x30
[INFO] Load gadgets for section: LOAD
[LOAD] loading... 100%
[INFO] Load gadgets for section: GNU_STACK
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
0x00000000004014cc: add x0, x0, #0x828; bl #0x820; movz w0, #0; ldp x29, x30, [sp]; add sp, sp, #0x4b0; ret;
0x0000000000400798: add x16, x16, #0x30; br x17;
0x0000000000400790: adrp x16, #0x12000; ldr x17, [x16, #0x30]; add x16, x16, #0x30; br x17;
0x00000000004000ec0: b.hi #0xe34; nop; nop; ldp x29, x30, [sp], #0x40; ret;
0x0000000000401270: b.ls #0x10a4; nop; nop; ldp x29, x30, [sp], #0x30; ret;
0x0000000000401664: b.ne #0x1648; ldp x19, x20, [sp, #0x10]; ldp x21, x22, [sp, #0x20]; ldp x23, x24, [sp, #0x30]; ldp x29, x30, [sp], #0x40; re
t;
0x0000000000401504: bl #0x1284; nop; ldp x29, x30, [sp], #0x30; ret;
0x0000000000401574: bl #0x14e4; nop; ldp x29, x30, [sp], #0x60; ret;
0x00000000004015d8: bl #0x1540; movz w0, #0; ldp x29, x30, [sp], #0x10; ret;
0x00000000004015f0: bl #0x1540; nop; ldp x29, x30, [sp], #0x10; ret;
0x00000000004015d4: bl #0x1584; bl #0x1540; movz w0, #0; ldp x29, x30, [sp], #0x10; ret;
0x00000000004015a8: bl #0x810; str w0, [sp, #0x1c]; nop; ldp x29, x30, [sp], #0x20; ret;
0x00000000004015d0: bl #0x820; bl #0x1584; bl #0x1540; movz w0, #0; ldp x29, x30, [sp], #0x10; ret;
0x00000000004014d0: bl #0x820; movz w0, #0; ldp x29, x30, [sp]; add sp, sp, #0x4b0; ret;
0x0000000000400700: bl #0x87c; ldp x29, x30, [sp], #0x10; ret;
0x0000000000400918: bl #0x890; movz w0, #0x1; strb w0, [x19, #0x98]; ldr x19, [sp, #0x10]; ldp x29, x30, [sp], #0x20; ret;
```

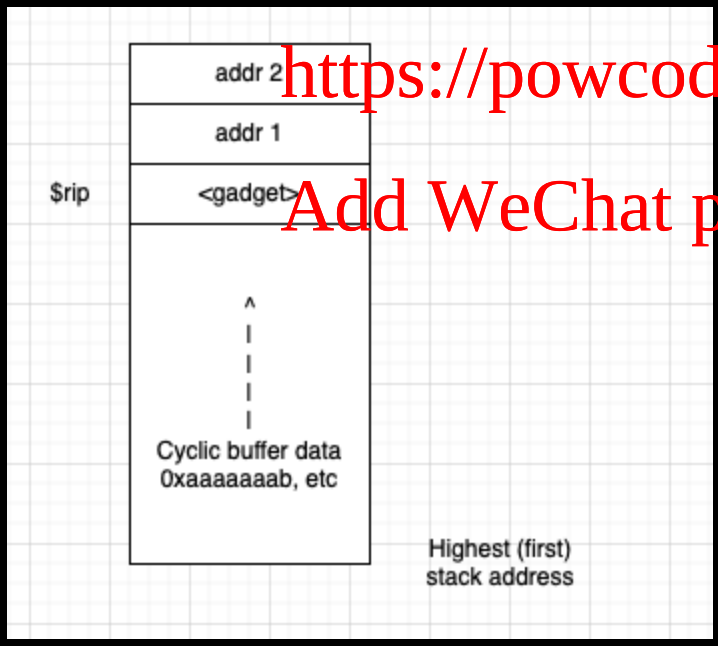
Your job is to pick/find a gadget that will perform the necessary task of loading the register with the specific argument value, and THEN we also need to perform a load operation to the LINK REGISTER containing the RETURN ADDRESS so that we can then call the function

Assignment Project Exam Help

Pictorially, this is what our crafted exploit needs to look like (remember stack grows down)

https://powcoder.com

Add WeChat powcoder



Now we will need to supply the argument, which will be on the stack immediately after our pop gadget, figure out what that value needs to be, and add it as p64(<value>) after the pop gadget

Then we need to put the address of the function as the next call, use objdump or gdb to find the addresses (you should probably get the second function address while you're at it). The call to

our pop gadget will ‘ret’ and then hit this second function call to enter one of the unsafe functions

Finally, we need to finish our execution chain by calling the second function which will allow for exploitation. Append that address to your chain and see if you get a flag!

Assignment Project Exam Help

https://powcoder.com

Add WeChat powcoder

Rubric

This project is worth 15% of your grade.

There are a total of 110 points for this project, if you complete all flags and get all 110 points, you get an extra 10% of the project applied to your grade

If you complete all flags you will get an effective extra credit of 1.5% final course grade applied

Flag	%grade
00_intro	0
01_basic_overflow_1	10
01_basic_overflow_2	10
02_assemble_the_assembly	15
02_bad_ranDc	15
02_p4s5w0rd	15
02_the_server_client_one	15
03_hunt_then_rop	10
03_pointy_pointy_point	10
03_XORbius	10
Total % Possible	110

Assignment Project Exam Help
https://powcoder.com
Add WeChat powcoder

Submission Details

File submission instructions:

The contents of the submission file should be the following. There is a ~/project_ctf.json file in your vm with a template set up, or you can copy-paste this to your newly created ***project_ctf.json*** file elsewhere and replace the placeholders with the flags you retrieve from each relevant task. (the name of the file doesn't matter, it is just named that for clarity)

Note: You can use TextEdit or Vim to create and edit this file. Do not use LibreOffice or any Word Document editor. It must be in proper JSON format with no special characters in order to pass the autograder and these Word Document editors are likely to introduce special characters.

If you can't find the file in the VM just copy this format below:

```
{
	"00_intro": "<copy flag here>",
	"01_basic_overflow_1": "<copy flag here>",
	"01_basic_overflow_2": "<copy flag here>",
	"02_assemble_the_assembly": "<copy flag here>",
	"02_bad_rando": "<copy flag here>",
	"02_p4s5w0rd": "<copy flag here>",
	"02_the_server_client_one": "<copy flag here>",
	"03_hunt_then_rop": "<copy flag here>",
	"03_pointy_pointy_point": "<copy flag here>",
	"03_XORbius": "<copy flag here>",
}
```

Assignment Project Exam Help
https://powcoder.com
Add WeChat powcoder

An example of what the submitted file content should look like:

```
{
 "00_intro":
"4ec60c3e084d8387f0f33916e9b08b99d5264a486c29130dd4a5a530b958c5c0f1faeaca2ce30b478281ec546a
4729f629b531a86cb27d86c089f0c542",
 "01_buffer_overflow_1":
"f496d9514c01e8019cd2bc21edfeb8e33f4a29af14a8bf92f7b3c14b5e06c5c0f1faeaca2ce30b478281ec546a
4729f629b531a86cb27d86c089f0c442",
 "01_buffer_overflow_2":
"b621bba0bb535f2f7a222bd32994d3875bcfcad651160c543de0a01dbe2e0c5c0f1faeaca2ce30b478281ec546
a4729f629b531a86cb27d86cf0c49542",

(etc)

}
```

Assignment Project Exam Help

https://powcoder.com

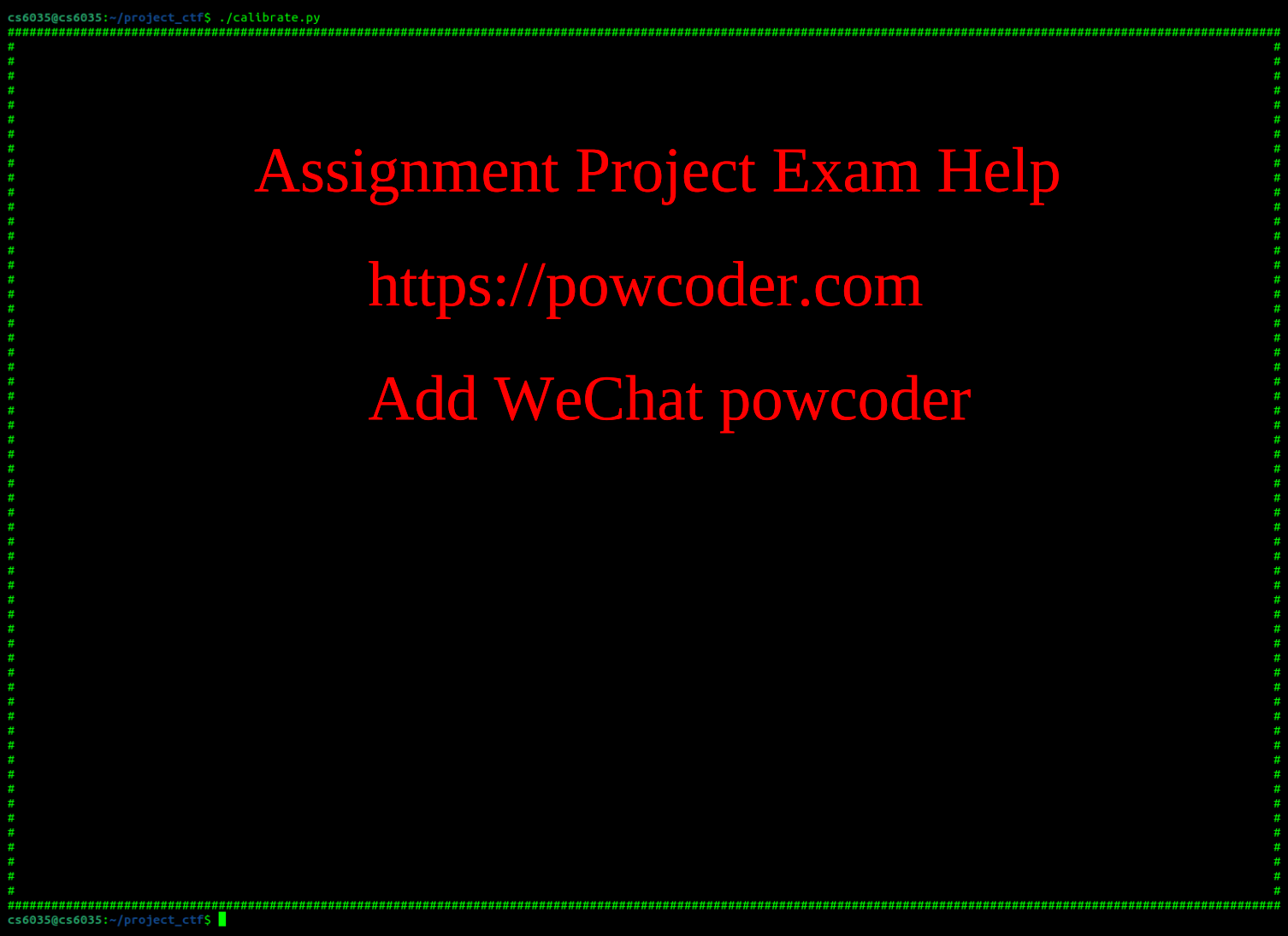
Add WeChat powcoder

Appendix

(Note: this is only currently released in the Intel/AMD version of the VM)
There is an experimental instructional tool I've created to guide you through the flow of some of these flags and give you an introduction to GDB and pwntools in a way that allows you to learn by doing!

You can start the program by
cd ~/project_ctf
./calibrate.py

Make sure your window at a _minimum_ will show the # symbols on top/bottom/left/right. If you have the resolution then feel free to make it bigger as it should enhance the readability.

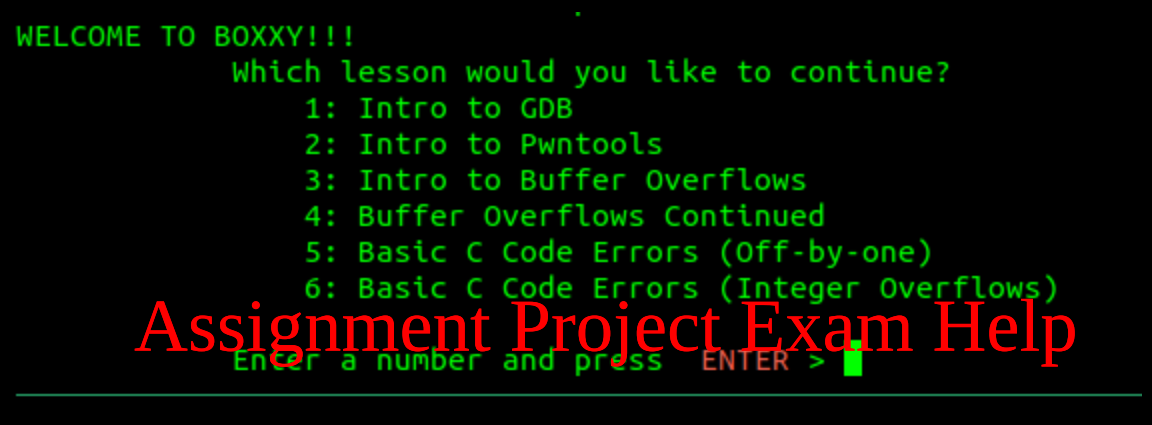


Assignment Project Exam Help
https://powcoder.com
Add WeChat powcoder

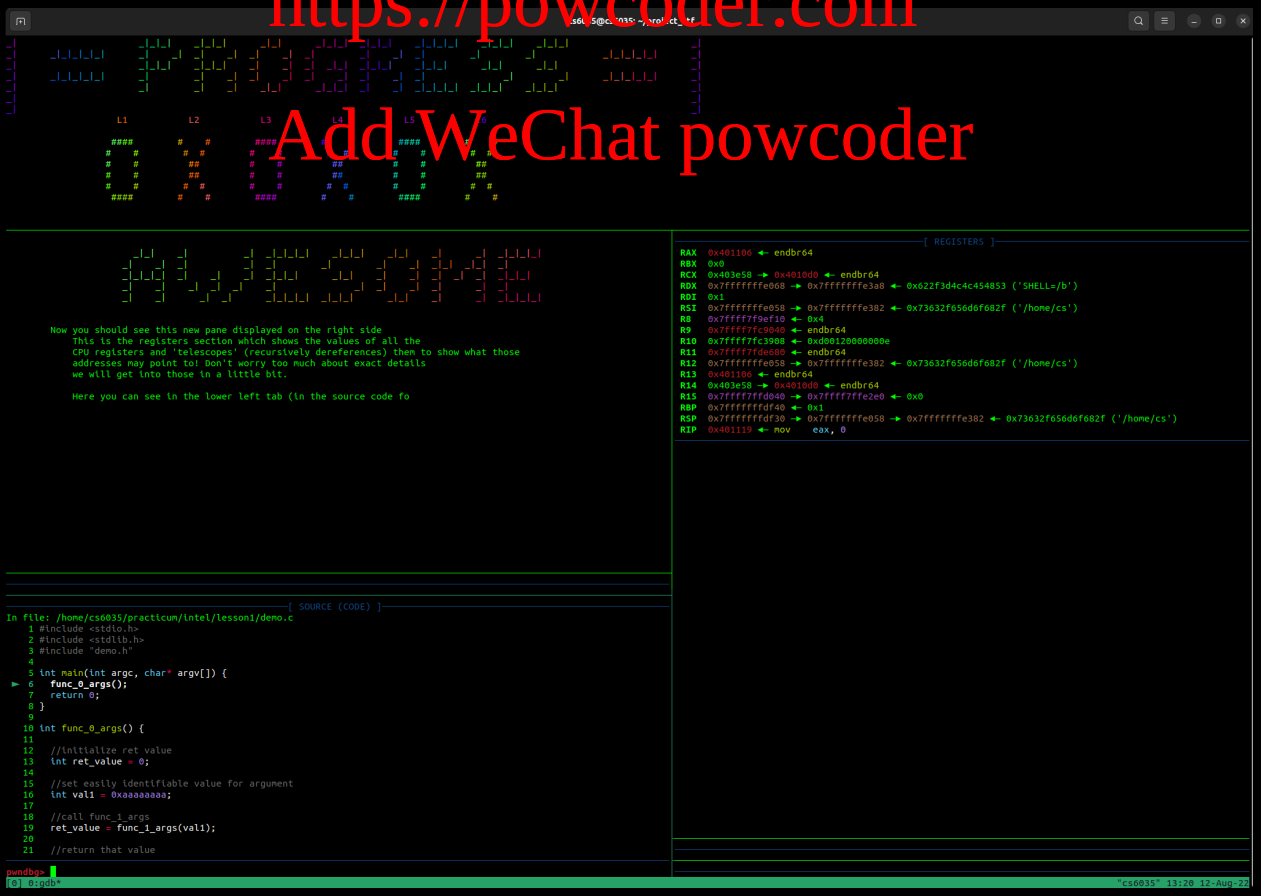
Once your terminal is set up, run

```
$ tmux
$ ./boxxy.py
```

You will be asked which lesson you want to start, I would suggest going through all of them but you are free to start wherever you want. (Note, once you start a section there isn't currently an easy way to navigate through the other sections, that is still being developed, thanks for your patience and I appreciate feedback on the lesson material!)



Assignment Project Exam Help
https://powcoder.com
Add WeChat powcoder



So long for now

Assignment Project Exam Help
https://powcoder.com
Add WeChat powcoder