Name: Archit Benipal

Subject: Software Supply Chain Security (CY 653)

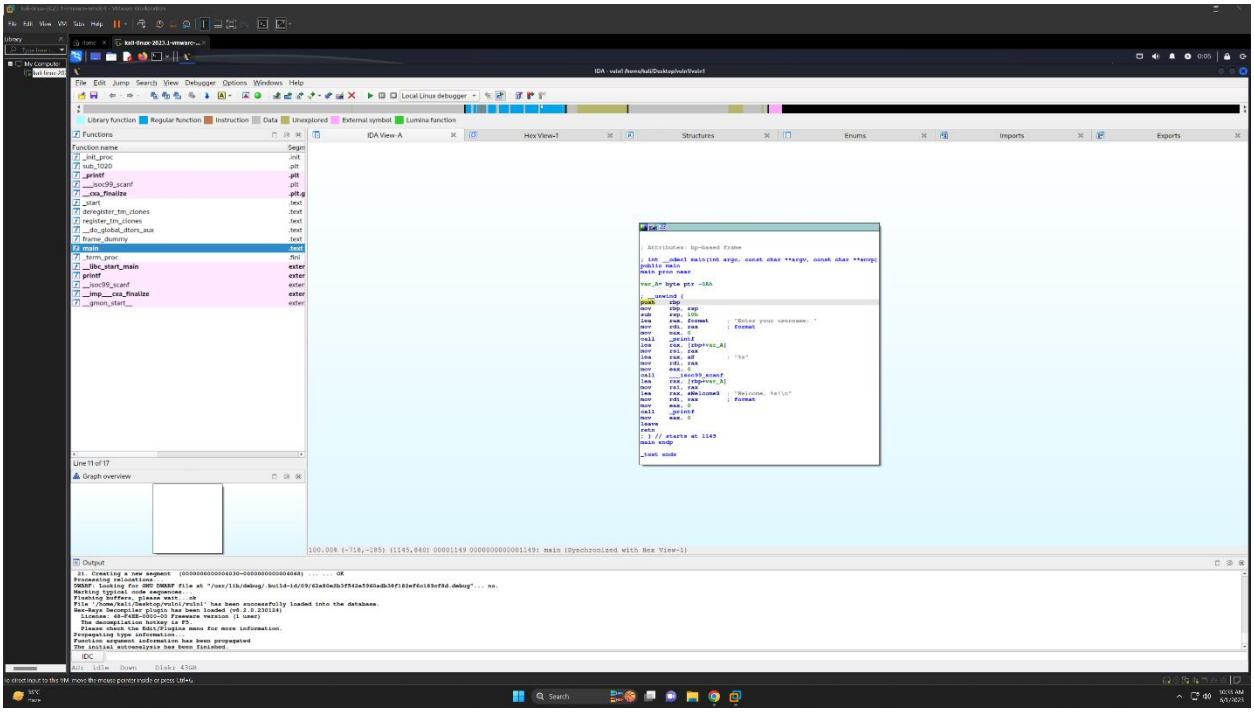
Assignment 5: Static Scanning

Static scanning: Static scanning, in the context of an ELF (Executable and Linkable Format) file or any other executable binary, refers to the analysis of the file without actually executing it. This form of analysis allows us to understand various aspects of the file, such as its structure, dependencies, functions, and potential vulnerabilities, without having to run the code.

Tools used for static analysis of ELF files include objdump, readelf, radare2, IDA Freeware/Pro, Ghidra, etc. It's important to note that static analysis won't catch all potential vulnerabilities or issues, and it's often used in combination with dynamic analysis (running the code and analyzing its behavior) for a more complete view of the program's behavior.

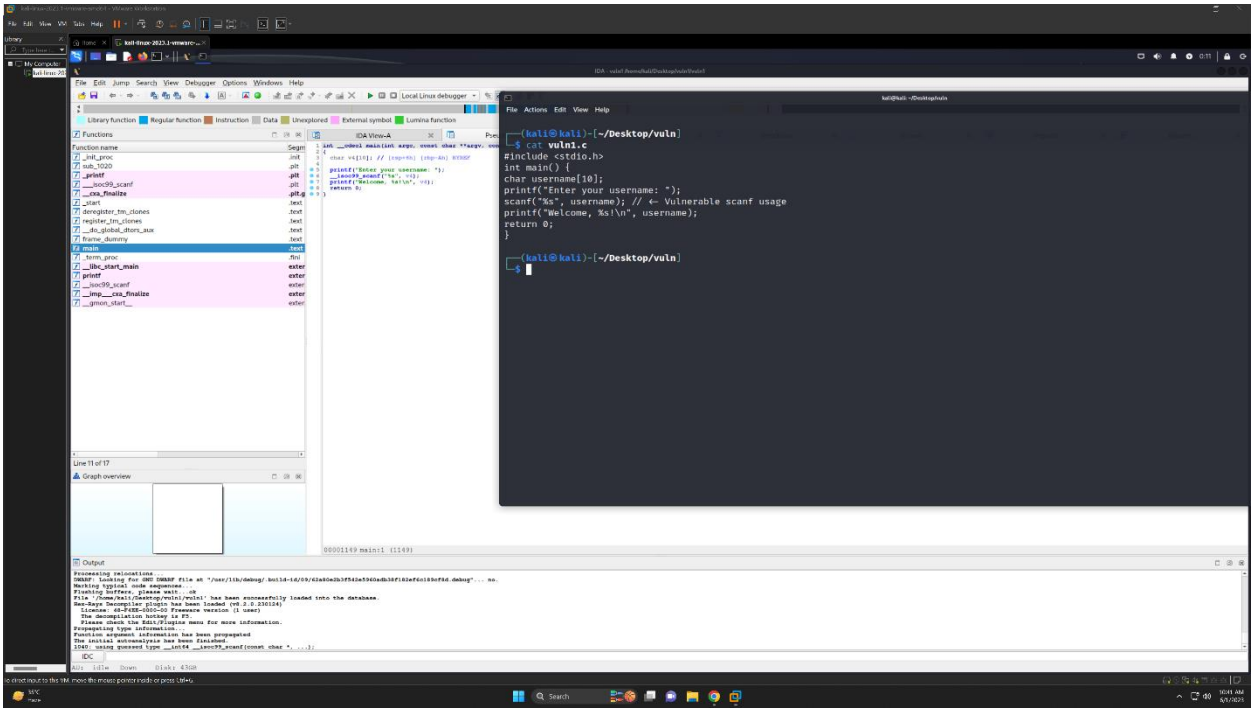
For this Lab I am going to stick with IDA freeware

Step 1: Opened the vuln1 binary with IDA freeware



Here under the main function we can see the assembly level code of the binary file, I can either recreate the c-code from this or view the decompiled code

Step 2: To check the c-level pseudo-code we need to press F5 which would show us the decompiled code



Here I have also compared it with the original vuln1.c file

Step 3: Breakdown of the version of code provided by IDA

At a first glance I can see that it is vulnerable to buffer-overflow attack as there is no proper length check for input in place

In the line " isoc99_scanf("%s", v4);"

The line reads the standard input and stores in the variable v4. The **v4** variable is a character array of size 10, which means it can hold 9 characters plus the null-terminating character (**'\0'**) that denotes the end of the string in C. If a user enters more than 9 characters, those additional characters will overflow the **v4** array and start overwriting other parts of the memory, which can lead to a variety of issues, including crashes and potential security vulnerabilities.

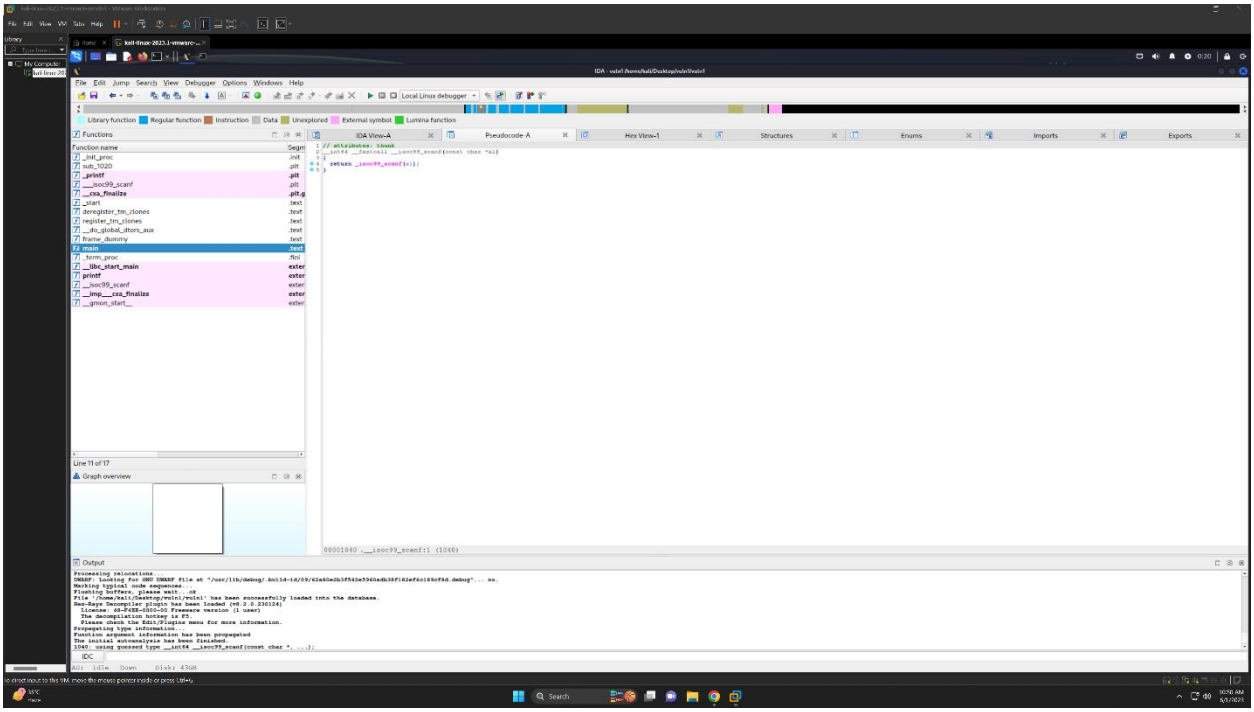
Fix for the issue:

This is a classic buffer overflow vulnerability. The correct way to avoid this issue is to specify a limit on the number of characters read by **scanf**,

"__isoc99_scanf("%9s", v4);"

This limits the input read into **v4** to 9 characters, leaving room for the null-terminating character and preventing a buffer overflow.

Step 4: Taking look at the scanf function using IDA and breakdown



The provided code snippet is a simple function that serves as a wrapper or thunk around the **_isoc99_scanf** function. It takes a single parameter (a character pointer **a1**) and passes it directly to **_isoc99_scanf**, returning the result.

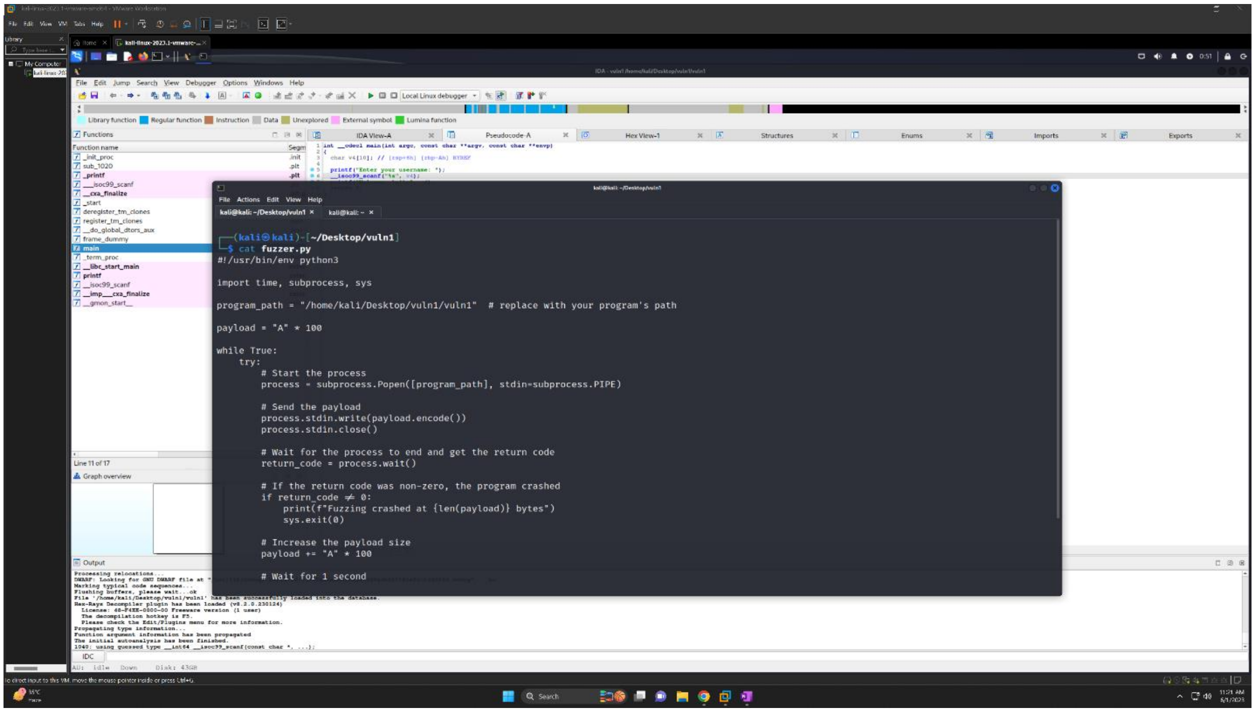
The term "thunk" refers to a function that performs a specific task on behalf of another function or piece of code. In this case, the thunk function **__isoc99_scanf** is merely calling the **_isoc99_scanf** function with the same arguments it received and returning the result.

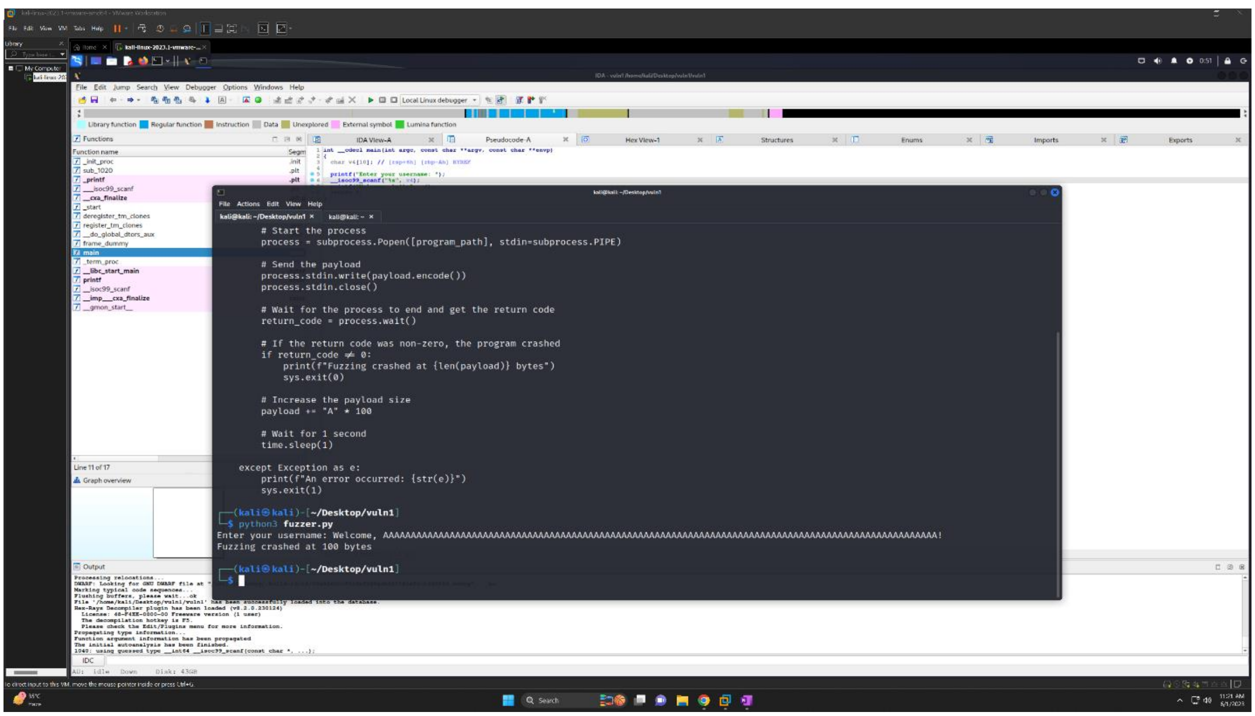
__fastcall is a keyword that specifies a calling convention where arguments are passed in registers when possible, which can result in faster function calls in some situations.

However, this wrapper function does not resolve the issue of a potential buffer overflow if the string format specifier **%s** is used without a length limit in **scanf**, as I mentioned in my previous response.

Step 5: Wrote a simple python script for fuzzing the binary file

```
#!/usr/bin/env python3

import time, subprocess, sys

program_path = "/home/kali/Desktop/vuln1/vuln1"  # replace with your program's path

payload = "A" * 100

while True:
    try:

        process = subprocess.Popen([program_path], stdin=subprocess.PIPE)


        process.stdin.write(payload.encode())
        process.stdin.close()


        return_code = process.wait()


        if return_code != 0:
            print(f"Fuzzing crashed at {len(payload)} bytes")
            sys.exit(0)


        payload += "A" * 100


        time.sleep(1)

    except Exception as e:
        print(f"An error occurred: {str(e)}")
        sys.exit(1)
```

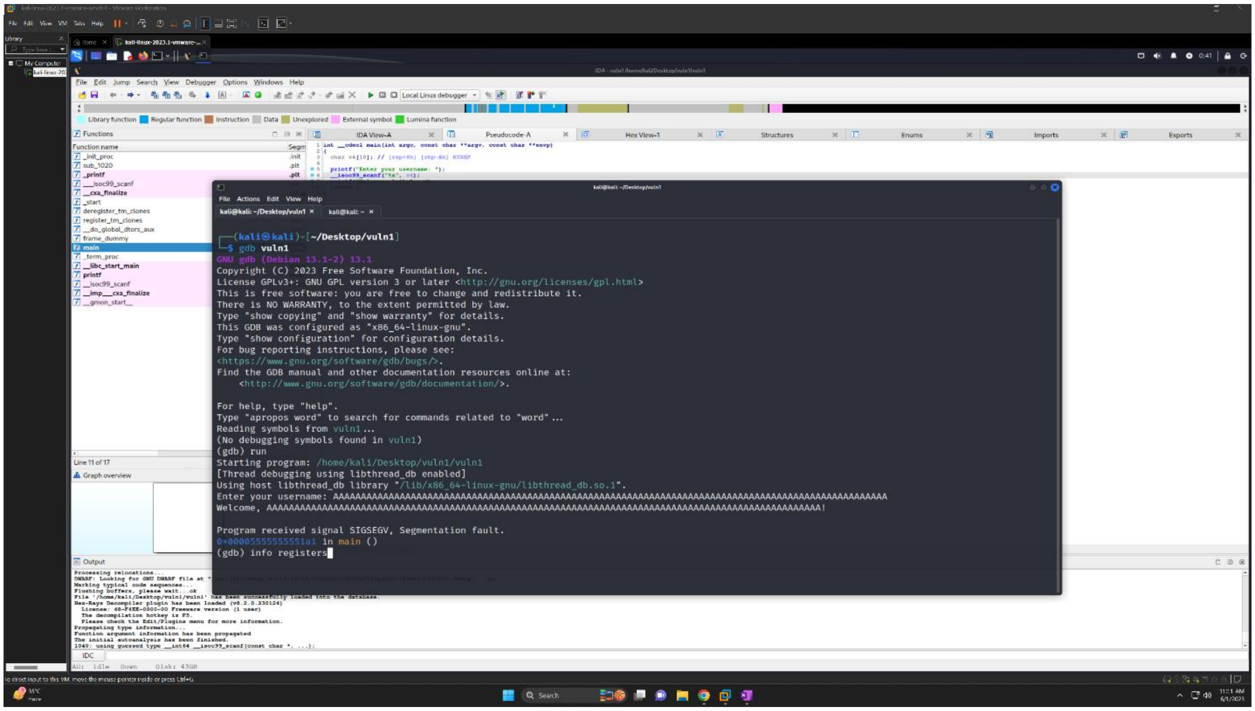




Here we can see the fuzzing crashed at 100 bytes

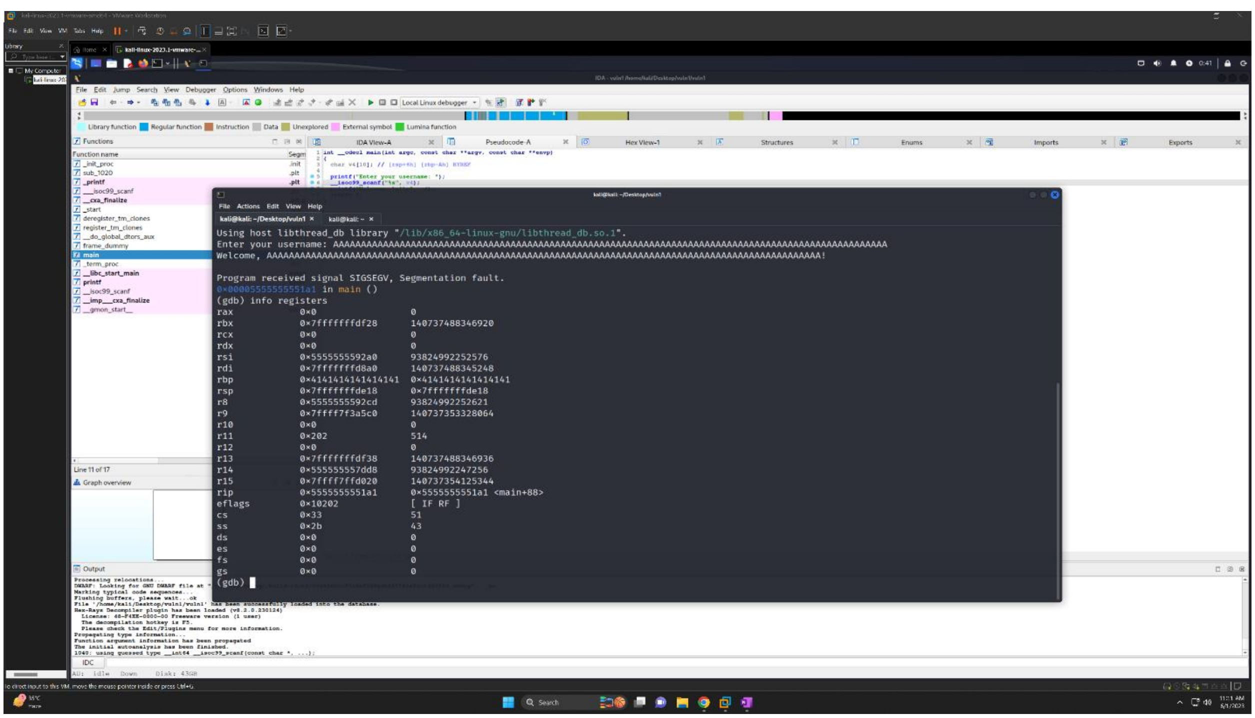
Buffer-overflow demonstration: (Exploitation)

Step 1: Here I have entered an random input of A*100

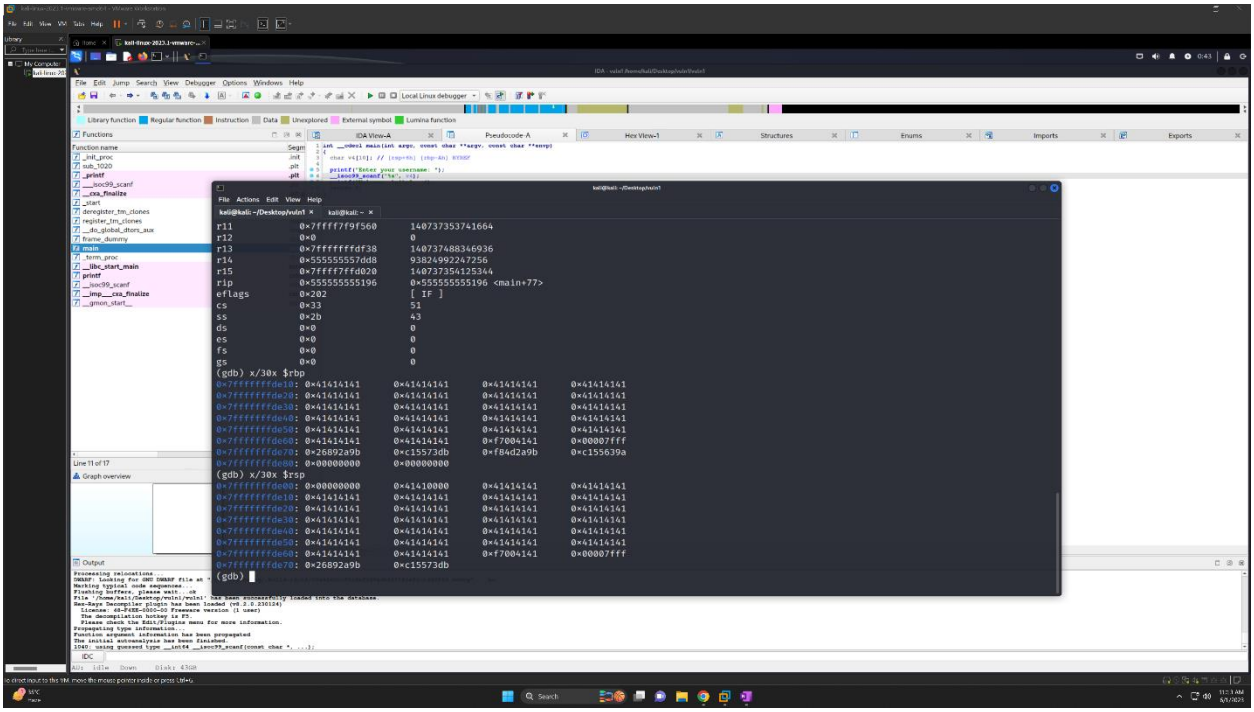


Here we can see a segmentation fault

Step 2: Viewed the registers



Step 3: Viewed the base pointer and stack pointer



Here I can see that the base and stack pointer have been overflown and see the hex value for “A” as 0x41.

Tools used:

IDA freeware: IDA Freeware is a free version of IDA (Interactive DisAssembler) offered by Hex-Rays. IDA is a widely used disassembler and debugger intended for software reverse engineering. It can be used to dissect binary files, analyze their structure, the flow of execution, and more.

GDB: The GNU Debugger, is a powerful, open-source debugger that is part of the GNU Project. It is widely used for debugging programs written in various languages such as C, C++, and Fortran. It supports a variety of Unix-like systems and works for many programming languages, including Ada, C, C++, Objective-C, Free Pascal, and more.