Name: Archit Benipal

Subject: Software Supply Chain Security (CY 653)

Assignment 2: File size increased after compilation

1. **Explain why the compiled program vuln1 has a much larger size than the size of the source program vuln1.c**

Source Code:

```
#include <stdio.h>
int main() {
char username[10];
printf("Enter your username: ");
scanf("%s", username); // <- Vulnerable scanf usage
printf("Welcome, %s!\n", username);
return 0;
}
```

Short Answer:

The compiled program vuln1 is linked with lots of binaries. When the compiler is finished with the linking process the file size increases. The libc is always linked against when building the C program.

Id command combines a number of object and archive files, relocates their data and ties up symbol; references. Usually the last step in compiling a program is to run Id.

The linkers put a lot into the binary file.

To understand this better I will break down the process into the 4 stages of C program has to go through before creating the binary file.

Detailed breakdown:

The four stages for a C program to produce an executable are the following:

1. Pre-processing
2. Compilation
3. Assembly
4. Linking

To explain the four stages I will use the source code provided.

Step 1: Pre-processing:

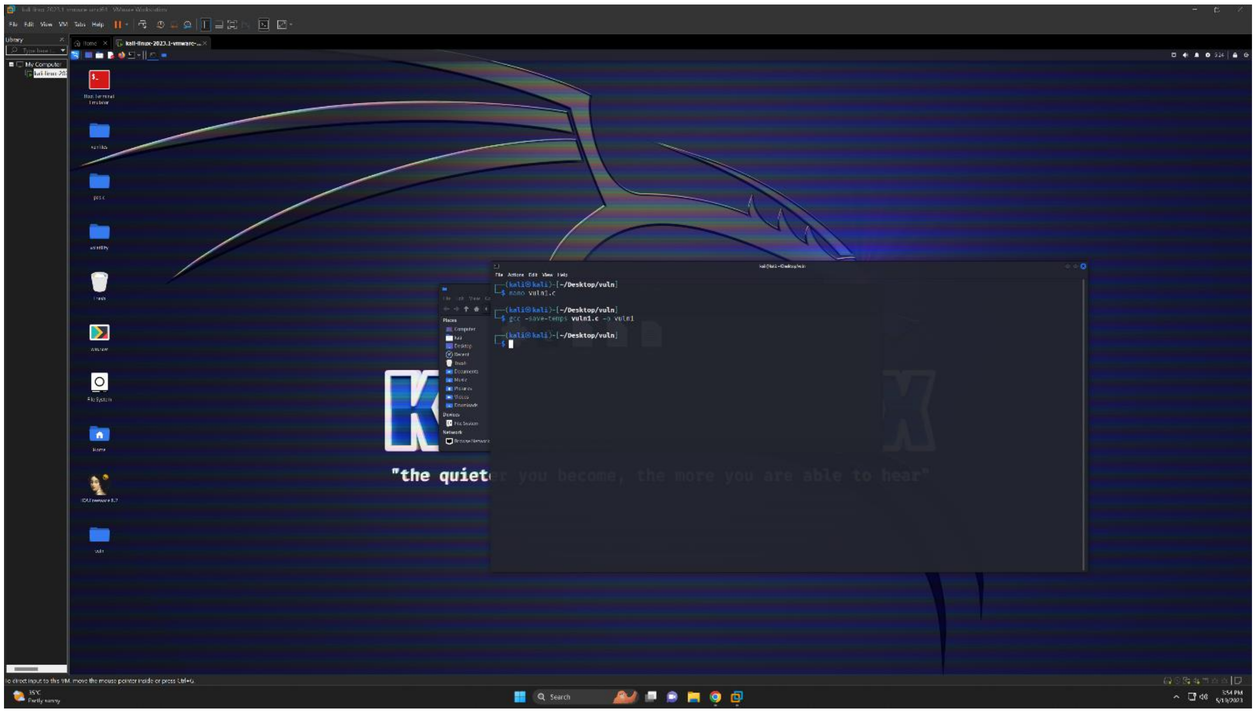
This is the very first stage through which a source code passes. In this stage the following tasks are done:

i. Macro substitution

ii. Comments are stripped off

iii. Expansion of the included files

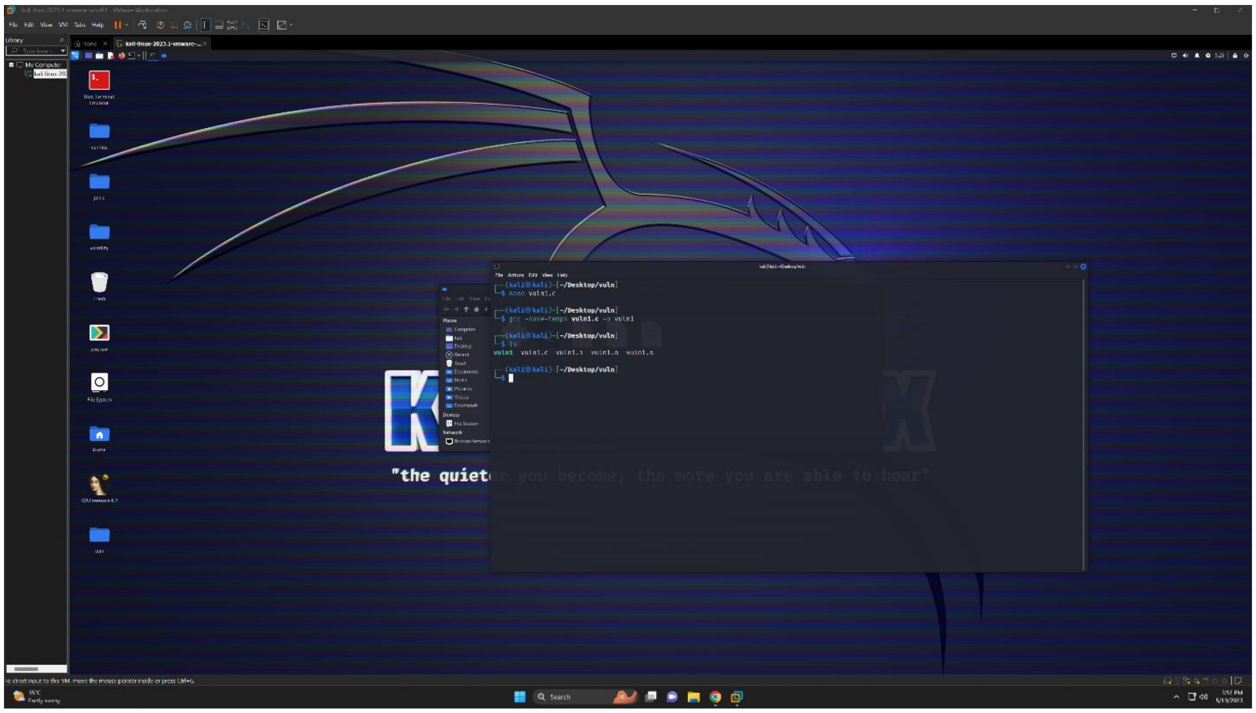
To understand the pre-processing better I have used a flag “-save-temps” this flag instructs compiler to store the temporary intermediate files used by gcc compiler in the current directory.



```
┌──(kali㉿kali)-[~/Desktop/vuln]
└─$ nano vuln1.c

┌──(kali㉿kali)-[~/Desktop/vuln]
└─$ gcc -save-temps vuln1.c -o vuln1

┌──(kali㉿kali)-[~/Desktop/vuln]
└─$ 
```

Here we have used the command “gcc -save-temps vuln1.c -o vuln1”

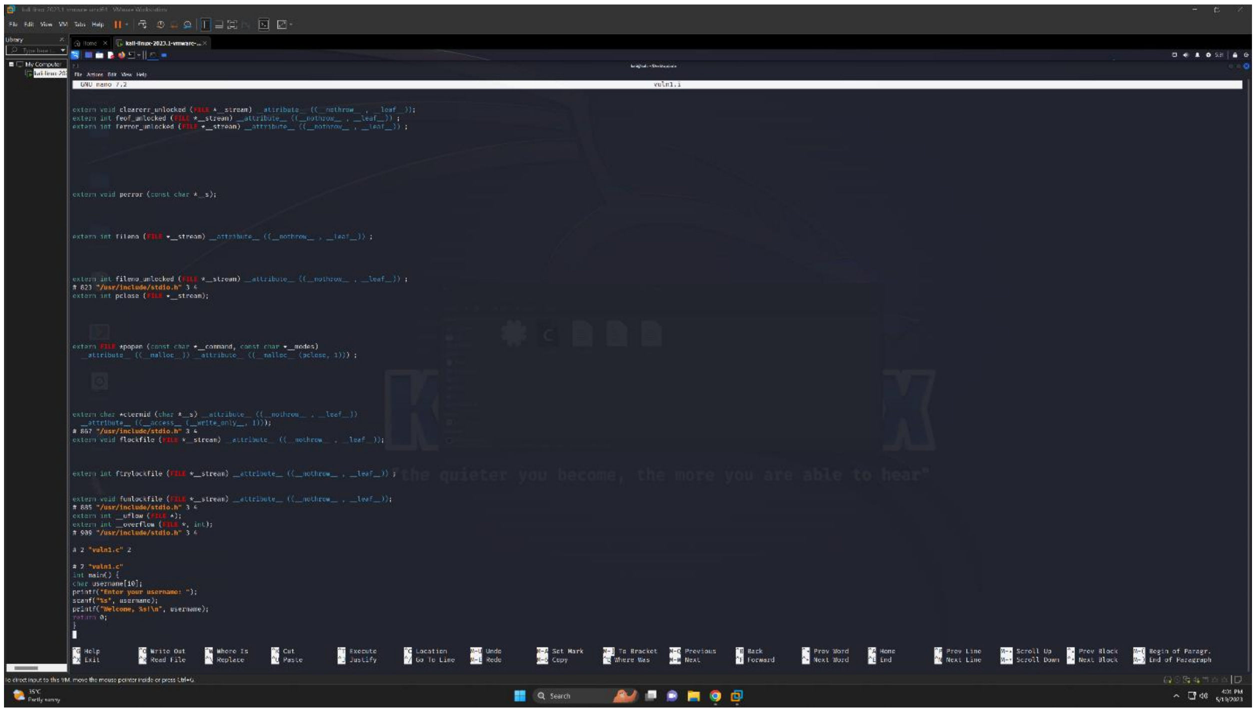
- gcc – Invokes the GNU C compiler
- vuln1.c – Input C program
- -o vuln1 – Instruct C compiler to create the C executable as vuln1. If you don’t specify -o, by default C compiler will create the executable with name a.out



```
┌──(kali㉿kali)-[~/Desktop/vuln]
└─$ nano vuln1.c

┌──(kali㉿kali)-[~/Desktop/vuln]
└─$ gcc -save-temps vuln1.c -o vuln1

┌──(kali㉿kali)-[~/Desktop/vuln]
└─$ ls
vuln1  vuln1.c  vuln1.i  vuln1.o  vuln1.s

┌──(kali㉿kali)-[~/Desktop/vuln]
└─$ 
```

Here we can see that we got the binary file “vuln1” with three other files vuln1.i, vuln1.s and vuln1.o.

The pre-processed output is stored in the temporary file that has the extension .i



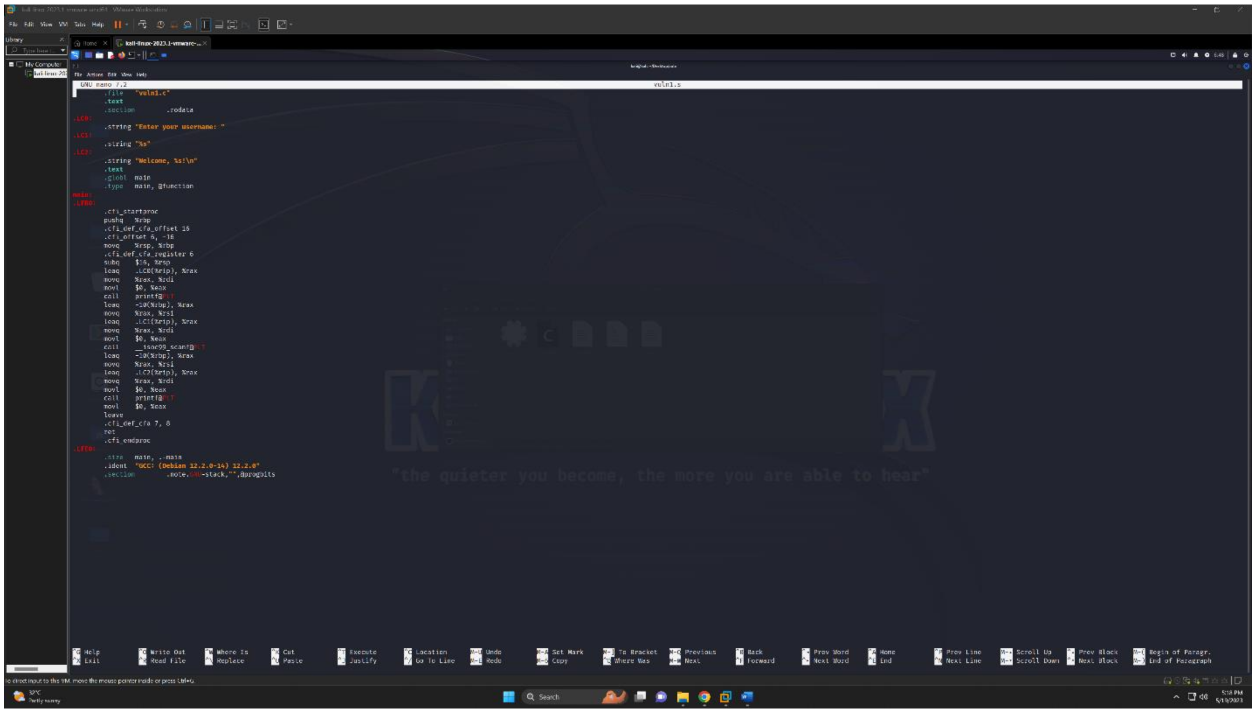
```
GNU nano 7.2                                    vuln1.i

extern void clearerr_unlocked (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__));
extern int feof_unlocked (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__)) ;
extern int ferror_unlocked (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__)) ;




extern void perror (const char *__s);



extern int fileno (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__)) ;



extern int fileno_unlocked (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__)) ;
# 823 "/usr/include/stdio.h" 3 4
extern int pclose (FILE *__stream);




extern FILE *popen (const char *__command, const char *__modes)
  __attribute__ ((__malloc__)) __attribute__ ((__malloc__ (pclose, 1))) ;





extern char *ctermid (char *__s) __attribute__ ((__nothrow__ , __leaf__))
  __attribute__ ((__access__ (__write_only__, 1)));
# 867 "/usr/include/stdio.h" 3 4
extern void flockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__));



extern int ftrylockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__)) ;


extern void funlockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__));
# 885 "/usr/include/stdio.h" 3 4
extern int __uflow (FILE *);
extern int __overflow (FILE *, int);
# 909 "/usr/include/stdio.h" 3 4

# 2 "vuln1.c" 2

# 2 "vuln1.c"
int main() {
char username[10];
printf("Enter your username: ");
scanf("%s", username);
printf("Welcome, %s!\n", username);
return 0;
}
```

Upon viewing the vuln1.i file I spotted two differences the first being the comment was stripped and expansion of the included files took place.

I can see that the #include is missing and instead of that I see whole lot of code in its place. So its safe to conclude that stdio.h has been expanded and literally included in source file. Hence the compiler is able to see the declarations of printf() function.

Step 2: Compiling:

After the compiler is done with the pre-processor stage. The next step is to take vuln1.i as input, compile it and produce an intermediate compiled output. The output file for this stage is ‘vuln1.s’. The output present in vuln1.s is assembly level instructions.



```
GNU nano 7.2                                   vuln1.s
        .file   "vuln1.c"
        .text
        .section        .rodata
.LC0:
        .string "Enter your username: "
.LC1:
        .string "%s"
.LC2:
        .string "Welcome, %s!\n"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $16, %rsp
        leaq    .LC0(%rip), %rax
        movq    %rax, %rdi
        movl    $0, %eax
        call    printf@PLT
        leaq    -10(%rbp), %rax
        movq    %rax, %rsi
        leaq    .LC1(%rip), %rax
        movq    %rax, %rdi
        movl    $0, %eax
        call    __isoc99_scanf@PLT
        leaq    -10(%rbp), %rax
        movq    %rax, %rsi
        leaq    .LC2(%rip), %rax
        movq    %rax, %rdi
        movl    $0, %eax
        call    printf@PLT
        movl    $0, %eax
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Debian 12.2.0-14) 12.2.0"
        .section        .note.GNU-stack,"",@progbits
```

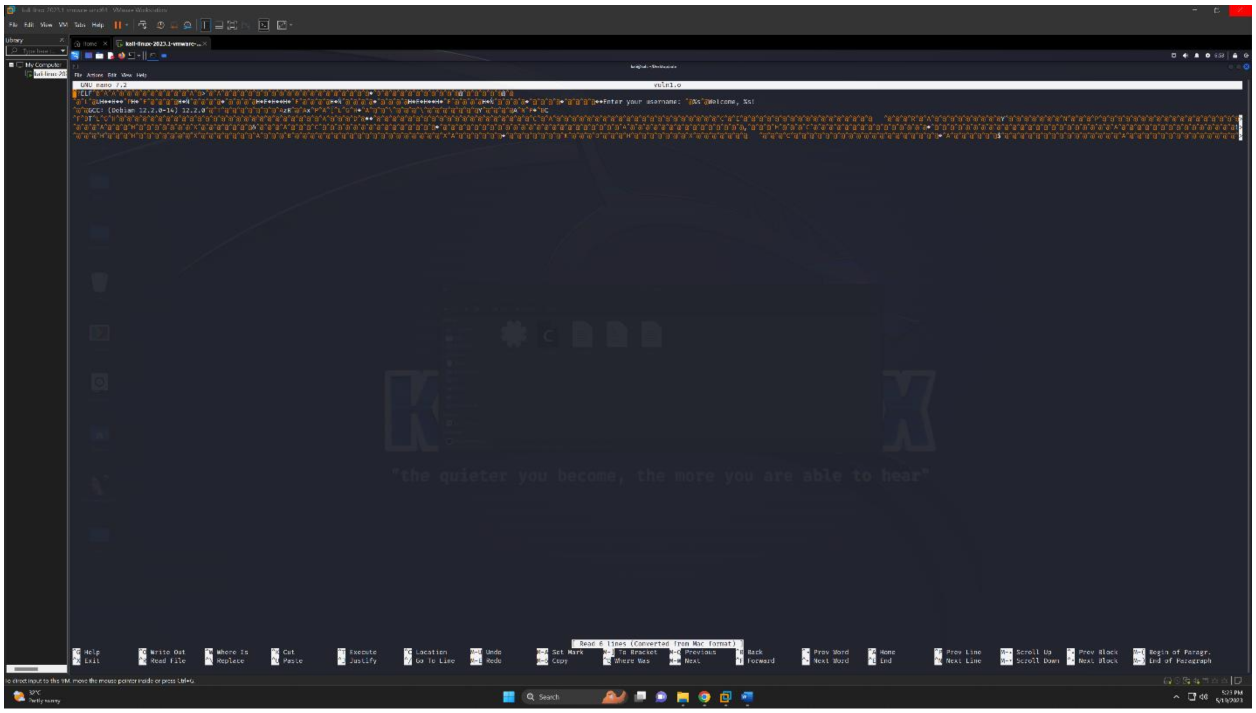
Here I can see that the source is converted into Assembly language.

Step 3: Assembly:

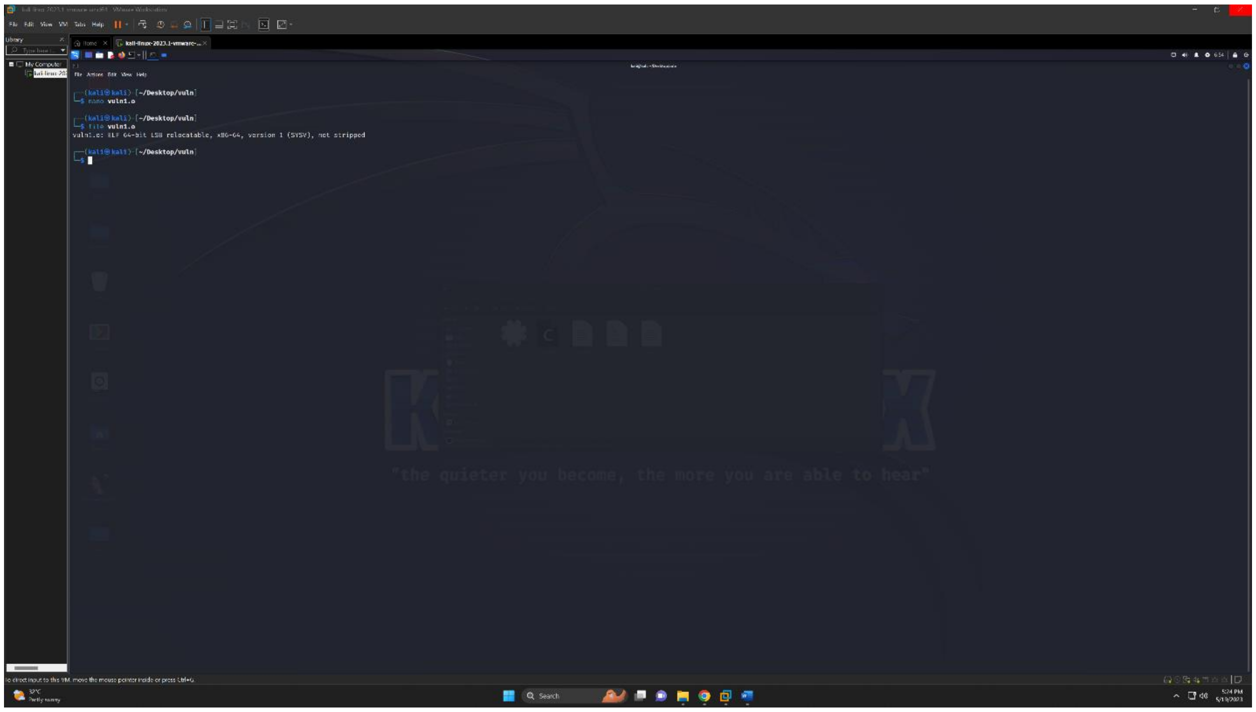
At this stage the vuln1.s file is taken as an input and an intermediate file vuln1.o is produced. This file is also known as the object file.

This file is produced by the assembler that understands and converts a ‘.s’ file with assembly instructions into a ‘.o’ object file which contains machine level instructions. At this stage only the existing code is converted into machine language, the function calls like printf() are not resolved.

Since the output of this stage is a machine level file (vuln1.o). So we cannot view the content of it.



```
  GNU nano 7.2                                    vuln1.o
^?ELF^B^A^A^@^@^@^@^@^@^@^@^@^A^@>^@^A^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@@^@^@^@^@^@@^@^@
[illegible]Enter your username: ^@%s^@Welcome, %s!
^@^@GCC: (Debian 12.2.0-14) 12.2.0^@[illegible]
[illegible]

                              [ Read 6 lines (Converted from Mac format) ]
```



```
┌──(kali㉿kali)-[~/Desktop/vuln]
└─$ nano vuln1.o

┌──(kali㉿kali)-[~/Desktop/vuln]
└─$ file vuln1.o
vuln1.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped

┌──(kali㉿kali)-[~/Desktop/vuln]
└─$ 
```

Here while checking file I came across the ELF 64bit. ELF stands for executable and linkable format.

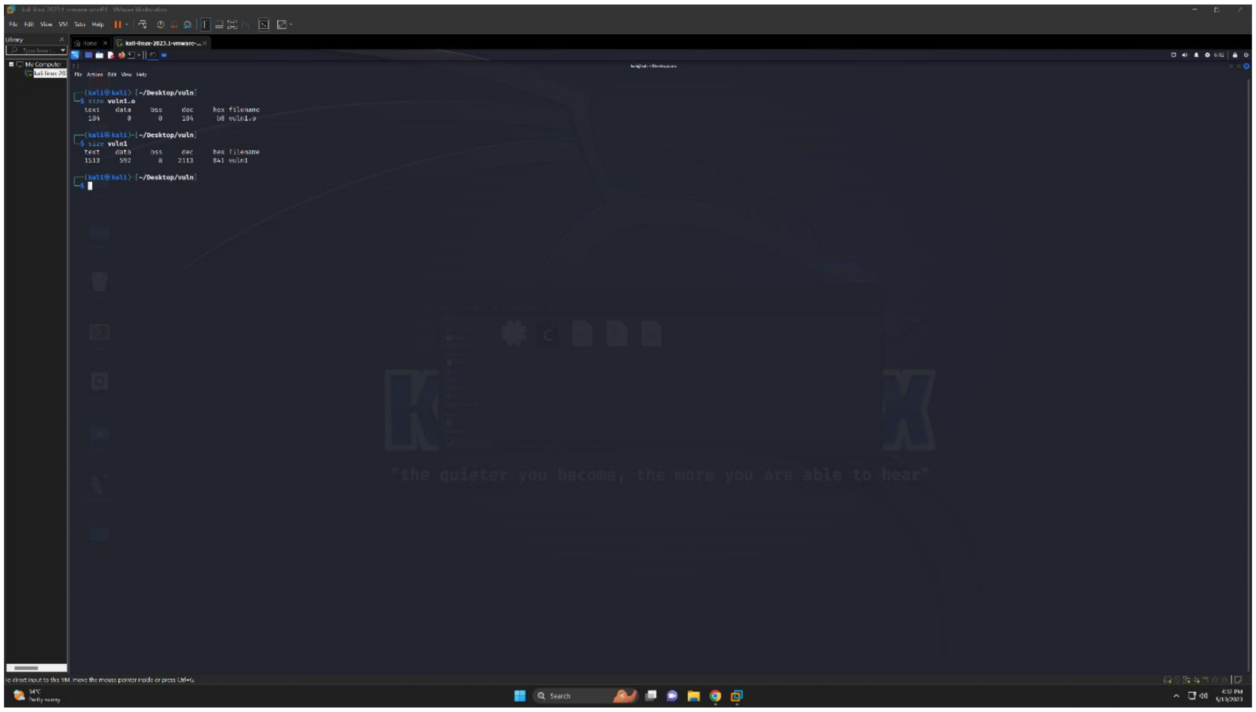
This is a relatively new format for machine level object files and executable that are produced by gcc. Prior to this, a format known as a.out was used. ELF is said to be more sophisticated format than a.out

Step 4: Linking:

This is the final stage at which all the linking of function calls with their definitions are done. As discussed earlier, till this stage gcc doesn't know about the definition of functions like printf(). Until the compiler knows exactly where all of these functions are implemented, it simply uses a place-holder for the function call. It is at this stage, the definition of printf() is resolved and the actual address of the function printf() is plugged in.

The linker comes into action at this stage and does this task.

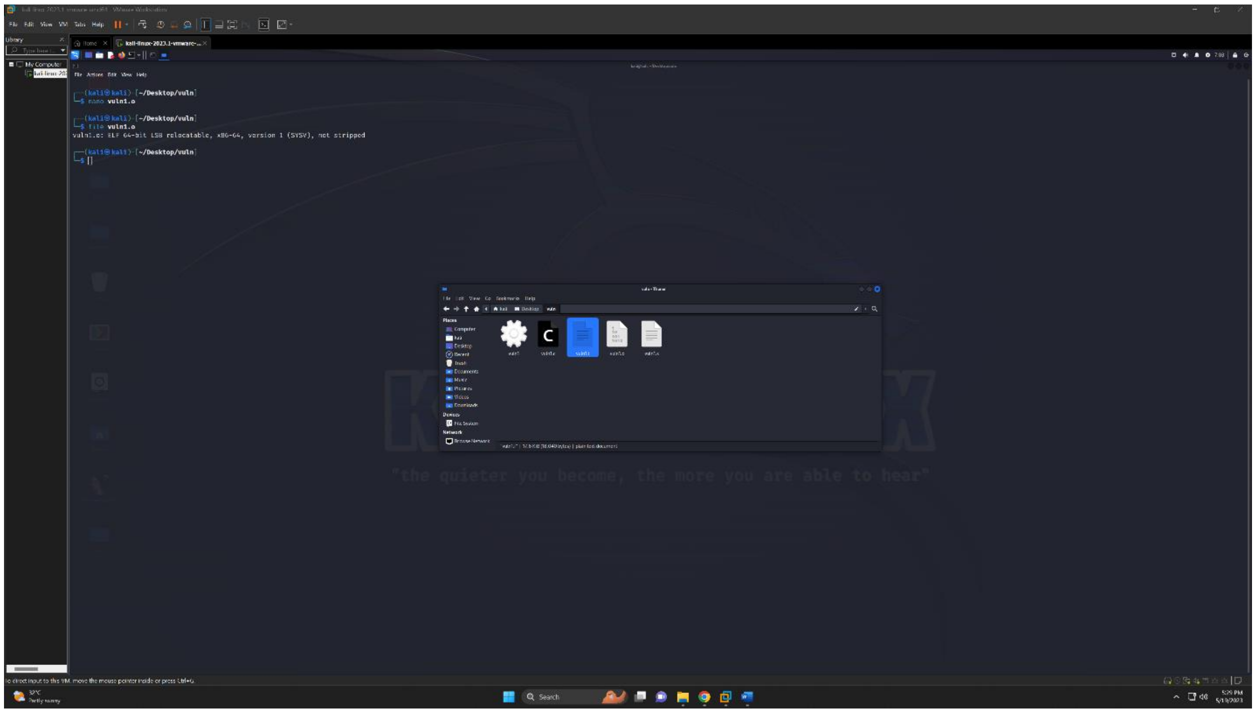
The linker also does some extra work; it combines some extra code to our program that is required when the program starts and when the program ends.



```
┌──(kali㉿kali)-[~/Desktop/vuln]
└─$ size vuln1.o
   text    data     bss     dec     hex filename
    184       0       0     184      b8 vuln1.o

┌──(kali㉿kali)-[~/Desktop/vuln]
└─$ size vuln1
   text    data     bss     dec     hex filename
   1513     592       8    2113     841 vuln1

┌──(kali㉿kali)-[~/Desktop/vuln]
└─$ 
```

Here I have compared the size of the two files vuln1.o and the binary file vuln1

Through the size command we get a rough idea about how the size of the output file increases from an object file to an executable file. This is all because of that extra standard code that linker combines with our program.



```
┌──(kali㉿kali)-[~/Desktop/vuln]
└─$ nano vuln1.o

┌──(kali㉿kali)-[~/Desktop/vuln]
└─$ file vuln1.o
vuln1.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped

┌──(kali㉿kali)-[~/Desktop/vuln]
└─$ []
```

vuln1
vuln1.c
vuln1.i
vuln1.o
vuln1.s
"vuln1.i" | 17.6 KiB (18,040 bytes) | plain text document
"the quieter you become, the more you are able to hear"

Here we can see the size of the vuln1.i file which the pre-processed file whose size is approximately 17.6 kb.

There are other factors for the size of the binary to be greater.

The factors contributing to the size of a compiled program can indeed vary widely, depending on a multitude of parameters including the language used, the specific compiler, the libraries linked, the target architecture, the optimization settings, and more. Below are the main factors that can influence the size of a compiled binary:

- **Source Code:** The size and complexity of the original source code. More code generally means a larger binary.
- **Compilation and Linking:** Compilation turns human-readable source code into machine code. Linking combines multiple object files and libraries into the final executable. Both processes add size to the binary.
- **Libraries:** Static libraries are included directly into the binary, increasing its size. Dynamic libraries aren't included in the binary, but they must be present on the system where the binary runs.
- **Debug Information:** Compiler options can include debug information in the binary, increasing its size.
- **Optimizations:** Some compiler optimizations aim for speed and can actually increase the binary size. Examples include loop unrolling and function inlining.
- **Binary Format Overhead:** Executable file formats include metadata, section headers, relocation and symbol tables, and more. All these add to the binary size.

- **Runtime Environment:** Some languages include a runtime environment in the binary.
- **Resource Embedding:** Any resources (e.g., images, icons, text files) included in the binary increase its size.
- **Code Generation Strategy:** Different compilers and settings can produce different machine code for the same source code.
- **Compiler Version and Type:** Different versions of a compiler, or different compilers altogether, can generate binaries of different sizes.
- **Target Architecture:** Different CPU architectures have different instruction sets, which can affect binary size.
- **Use of Templates/Generics:** In languages like C++ and Rust, heavy use of these features can lead to code bloat, increasing the binary size.
- **Exception Handling:** In languages that support exceptions, extra code and data structures may be needed to implement this feature.
- **Memory Alignment:** To optimize access speed, some architectures require certain kinds of data to be aligned at memory addresses that are multiples of some number, which can increase size due to padding.
- **Profiling Information:** If a binary is built with profiling information, this will increase the binary size.

The factors influencing the size of a binary can vary a lot depending on the language, compiler, libraries, and system used.