HO# 3.1: Recap of C and Vulnerabilities

A Quick Recap of C Language and its Constructs

C is a general-purpose programming language that was developed in the early 1970s by Dennis Ritchie at Bell Labs. It was created as an evolution of the B language, which itself was derived from BCPL. Initially, C was designed to develop the UNIX operating system, and its success in this area contributed to its widespread adoption. The language was first implemented on the PDP-11 computer.

Today, in 2024, C language continues to be actively used and maintained, with minor updates and revisions being proposed. The language's simplicity, efficiency, and portability ensure its relevance in modern computing, including system programming, embedded systems, and high-performance applications.

- **History and Evolution of C Language**

  The C programming language has a rich history marked by its development, standardization, and evolution. Here's a chronological overview of its key milestones:

  - **Early Beginnings (1960s):** Before C, there were precursors like BCPL (Basic Combined Programming Language) and B, which influenced C's development. BCPL was created by Martin Richards in the 1960s and B by Ken Thompson, which was itself derived from BCPL.
  - **Development of C (1970s):** Dennis Ritchie at Bell Labs develops C as an evolution of the B language. It was initially used to rewrite the UNIX operating system, which helped in demonstrating C's capabilities and efficiency. In 1978, Dennis Ritchie and Brian Kernighan publish *The C Programming Language*, often referred to as K&R C. https://www.cs.sfu.ca/~ashriram/Courses/CS295/assets/books/C_Book_2nd.pdf
  - **Standardization:** The C programming language has several standard versions, that are aimed to unify various implementations of C and ensure code portability across different platforms. The most commonly used ones are C89/90, C99, C11, and C18.
    - **C89/C90 (ANSI C or ISO C)** was the first standardized version of the language, released in 1989 and 1990 respectively. This standard introduced many of the features that are still used in modern C programming, including data types, control structures, and the standard library.
    - **C99 (ISO/IEC 9899:1999):** In 1999, the International Organization for Standardization (ISO) and International & Electrotechnical Commission (IEC) publishes the C99 standard. This update introduces several new features, including, new data types, inline functions, variable-length arrays, improved support for floating-point arithmetic and new standard library functions and macros. In 2001, C99 is adopted as the standard by some organizations, though its widespread adoption is gradual due to compatibility issues with existing codebases.
    - **C11 (ISO/IEC 9899:2011):** In 2011, the **C11** standard is published by ISO, introducing several enhancements like support for multi-threading, improved Unicode support, static assertions and enhanced standard library functions.
    - **C18 (ISO/IEC 9899:2018):** This is the most recent standard and includes updates and clarification to the language specification and the library.

- **Features and Characteristics**
    - **Low-Level Access**: C provides low-level access to memory through the use of pointers, which allows for efficient manipulation of data.
    - **Portability**: C code can be compiled and executed on a wide variety of hardware platforms with minimal modification.
    - **Efficiency**: C is known for its performance and minimal runtime overhead, making it suitable for system-level programming.
    - **Structured Programming**: C supports structured programming constructs such as conditions, loops, and functions, which promote clear and maintainable code.
    - **Modularity**: C allows for the division of code into functions and files, aiding in modular design and code reuse.
    - **Standard Library**: C provides a rich set of standard libraries for tasks such as input/output, string manipulation, and mathematical computations.
    - **Pre-processor**: C includes a pre-processor that handles macros, file inclusion, and conditional compilation, which can be used to make the code more flexible and portable.
- **Uses of C**
    - **System Programming**: C is widely used in operating systems development, including UNIX, Linux, and Windows kernel components.
    - **Embedded Systems**: Its low-level capabilities and efficiency make C a popular choice for programming embedded systems and microcontrollers.
    - **Compilers and Interpreters**: Many modern compilers and interpreters for other programming languages are written in C.
    - **Networking**: C is used to develop networking protocols and software that require direct access to hardware and network interfaces.
    - **Database Systems**: The development of database systems and management tools often leverages C for its efficiency and control.
- **C Language Constructs:** Every programming language has its own set of constructs that influence how code is written and executed, shaping the overall design and idioms of the language. Some of the major C programming language constructs are given below:
    - **Data Types:** C provides a variety of data types to store different kinds of data:
        - **Basic Data Types**:
            - `int`: Integer (e.g., `int age = 30;`)
            - `char`: Character (e.g., `char letter = 'A';`)
            - `float`: Floating-point (e.g., `float price = 9.99;`)
            - `double`: Double-precision floating-point (e.g., `double pi = 3.14159;`)
        - **Derived Data Types**: Arrays store collections of elements of the same type:
            - **Arrays:**
                - **Array Declaration**: Defines an array and its size (e.g., `int arr[5];`)
                - **Array Access**: Accesses elements using indices (e.g., `arr[0] = 1;`)
            - **Pointers**: Pointers store memory addresses and provide powerful capabilities:
                - **Pointer Declaration**: Declares a pointer variable (e.g., `int *ptr;`)
                - **Pointer Usage**: Accesses and manipulates memory locations (e.g., `*ptr = 10;`)
                - **Pointer Arithmetic**: Performs arithmetic operations on pointers (e.g., `ptr++`)
                    - **Structures:** Group related variables of different types (e.g., `struct Employee { char name[30]; int id; };`)
                    - **Unions**: Store different data types in the same memory location (e.g., `union Data { int i; float f; char str[20]; };`)
                    - **Enumerations**: Define a set of named integer constants (e.g., `enum Color { RED, GREEN, BLUE };`)

- **Operators:**
  - **Arithmetic Operators**: `+, -, *, /, %` (e.g., `int sum = a + b;`)
  - **Relational Operators**: `==, !=, <, >, <=, >=` (e.g., `if (a > b)`)
  - **Logical Operators**: `&&, ||, !` (e.g., `if (a > 0 && b < 10)`)
  - **Bitwise Operators**: `&, |, ^, ~, <<, >>` (`int result = a & b;`)
  - **Assignment Operators**: `=, +=, -=, *=, /=, %=` (e.g., `x += 5;`)
  - **Increment/Decrement Operators**: `++, --` (e.g., `i++;`)
  - **Conditional (Ternary) Operator**: `?:` (e.g., `int max = (a > b) ? a : b;`)

- **Control Flow Constructs:** These constructs manage the flow of execution in a program:
  - **Conditional Statements**:
    - if and else (e.g., `if (condition) { /* code */ } else { /* code */ }`).
    - switch (e.g., `switch (variable) { case 1: /* code */ break; default: /* code */ }`).
  - **Loops**:
    - for (e.g., `for (int i = 0; i < 10; i++) { /* code */ }`).
    - while (e.g., `while (condition) { /* code */ }`).
    - do-while (e.g., `do { /* code */ } while (condition);`).
  - **Jump Statements**:
    - `break` (exits loops or switch statements).
    - `continue` (skips the current iteration of a loop).
    - `goto` (transfers control to a labeled statement, though its use is generally discouraged).

- **Functions:** Functions in C allow code to be modular and reusable:
  - **Function Declaration**: Specifies the function's name, return type, and parameters (e.g., `int add(int a, int b);`)
  - **Function Definition**: Provides the implementation of the function (e.g., `int add(int a, int b) { return a + b; }`)
  - **Function Call**: Invokes a function and can pass arguments (e.g., `int sum = add(5, 10);`)

- **Preprocessor Directives:** Preprocessor directives are instructions for the C preprocessor:
  - **Macros**: Define constants or code snippets (e.g., `#define PI 3.14`).
  - **File Inclusion**: Include header files (e.g., `#include <stdio.h>`).
  - **Conditional Compilation**: Include code conditionally (e.g., `#ifdef DEBUG`).

- **Error Handling:** C does not have built-in exception handling. Error handling is typically done using return codes and checking error conditions.

C Compilation Process and its Tool Chain

https://www.youtube.com/watch?v=a7GhFL0Gh6Y&list=PL7B2bn3G_wfC-mRpG7cxJMnGWdPAQTViW&index=2
https://www.youtube.com/watch?v=A67t7X2LUsA&list=PL7B2bn3G_wfC-mRpG7cxJMnGWdPAQTViW&index=3

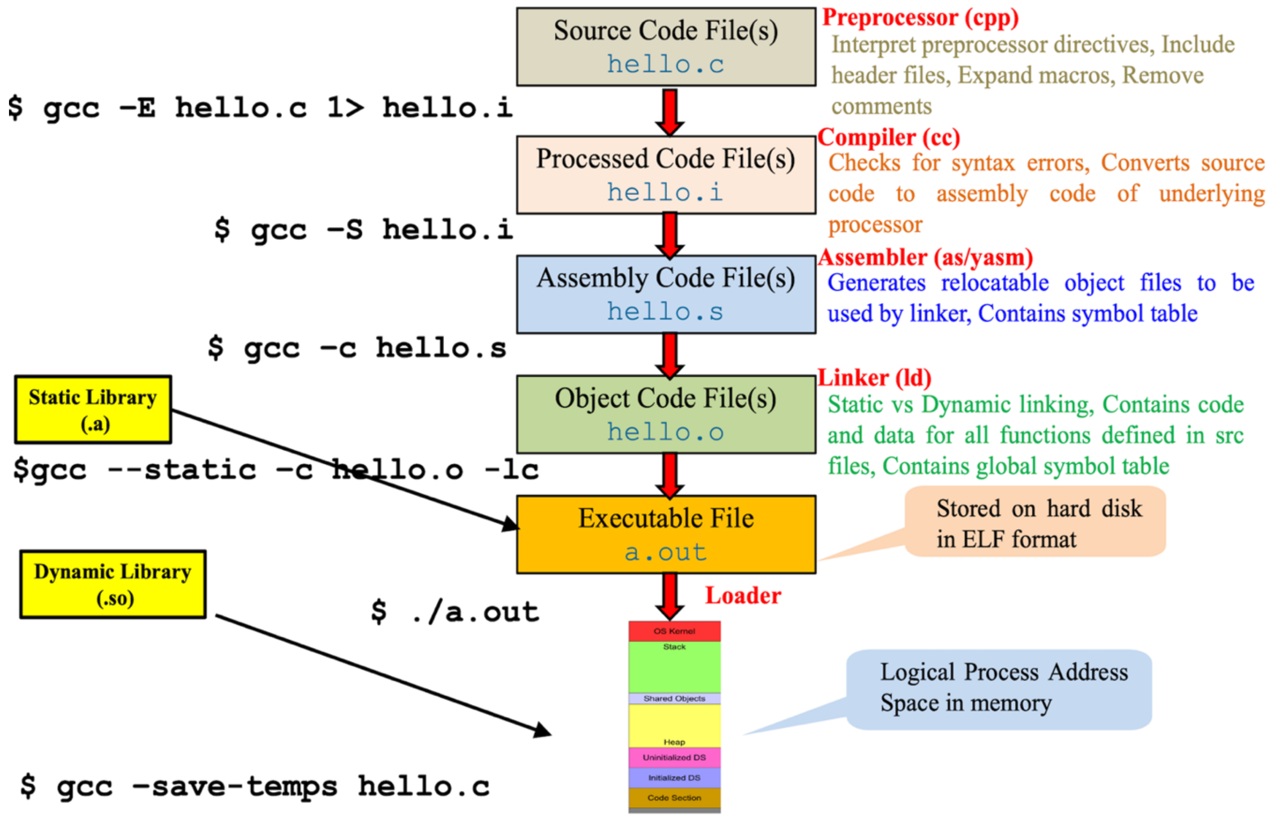
The set of programming tools used to create a program is referred to as the Tool Chain.

- **Processor:** Intel IA-32, Intel IA-64, AMD x86-64, Microprocessor without Interlocked Pipeline Stages (MIPS), Advanced RISC Machine (ARM), Sun Scalable Processor ARChitecture (Sun SPARC)
- **Operating System:** Windows, UNIX, Linux, MacOS
- **Editor/IDE:**
  - **Text Editors:** `gedit, vim, notepad`
  - **Code Editors:** `Atom, Sublime, VS Code, Notepad++, Brackets`
  - **IDEs:** `Visual Studio, Code::Blocks, PyCharm, Spider, Eclipse, Xcode`
- **Assembler:** `nasm, yasm, gas, masm`
- **Linker:** `ld` a GNU linker
- **Loader:** Default OS
- **Debugging/RE:** `readelf, objdump, nm, strings, file, hexedit, objcopy, strip, addr2line, gdb (PEDA/GEF), valgrind, strace, ltrace, ftrace, bftrace, perf, IDA Pro, ghidra, radare2, cutter, OllyDbg, binaryninja`

C Compilation Process

The C compilation process is a series of steps that transforms C source code into an executable program. This process can be divided into four phases, with each phase using tools that work together to produce the final executable. The image below gives an overview of the entire C-compilation process.

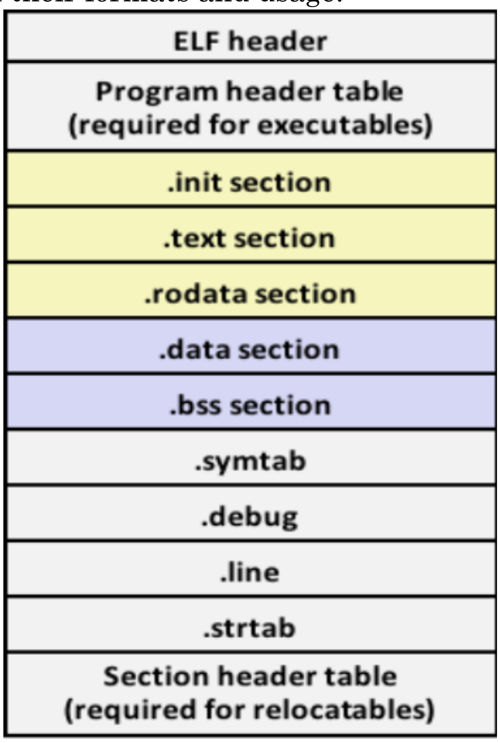
```
#include <stdio.h>
int main() {
    printf("This is a Hello World program..\n");
    return 0;
}
```



I. **Preprocessing:** The preprocessing step handles pre-processor directives, and remove comments. The pre-processor directives are not part of the C language but are used to perform operations before the actual compilation begins. The result of preprocessing is a preprocessed source file, which is usually saved with `.i` or `.ii` extension. The pre-processor directives perform following tasks:
   a. **File Inclusion**: `#include` directives are replaced with the contents of the specified files. This is typically used to include header files.
   b. **Macro Expansion**: `#define` macros are expanded to their defined values or code snippets.
   c. **Conditional Compilation**: `#ifdef, #ifndef, #else, #elif`, and `#endif` are used to include or exclude parts of the code based on certain conditions.

II. **Compilation:** The compiler takes the preprocessed source code and translates it into assembly code and the result is an assembly file, usually with `.s` or `.asm` extension. This step involves several key activities:
   a. **Syntax Analysis**: The compiler checks the syntax of the code to ensure it conforms to the C language rules.
   b. **Semantic Analysis**: It verifies the semantic correctness of the code, such as type checking and ensuring that variables are declared before use.
   c. **Optimization**: The compiler may optimize the code to improve performance or reduce resource usage.
   d. **Code Generation**: The compiler generates assembly code, which is a low-level representation of the source code.

III. **Assembling:** The assembler converts the assembly code into machine code, which is an object file, typically with `.o` or `.obj` extension. The assembling phase translates assembly language instructions into machine code instructions that the CPU can execute. Moreover, the assembler resolves symbolic names (e.g., variable names) to actual memory addresses and generates a symbol table. In Linux, object files can be classified based on their formats and usage:



   - **Relocatable object file** (***.o file***) is a file generated by a compiler or assembler that contains machine code, data, and metadata, but is not yet a complete executable or library. Object files are intermediate files that are linked together to produce final executables or shared libraries. They are crucial in the software build process, allowing modular development and incremental compilation. Each `.o` file is produced from exactly one `.c` file.
   - **Executable object file** (***a.out file***) Contains binary code and data in a form that can be copied directly into memory and executed. Linkers generate executable object files.
   - **Shared object file** (***.so file***) A special type of relocatable object file that can be loaded into memory and linked dynamically, at either load time or run time. Called dynamic link libraries (dlls) in Windows. Compilers and assemblers generate shared object files.
   - **Core file:** A disk file that contains the memory image of the process at the time of its termination. This is generated by system in case of abnormal process termination.

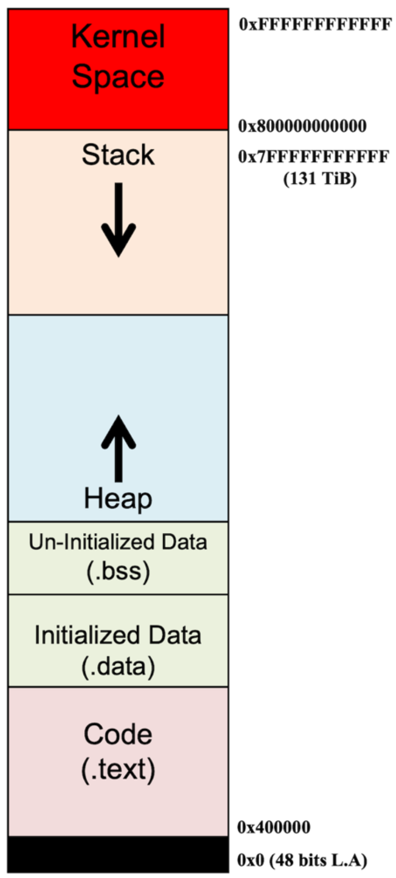
   In Linux, Executable and Linking Format (ELF) is a binary format used for storing programs or fragments of programs on disk, created as a result of compiling and linking. ELF not only simplifies the task of making shared libraries, but also enhances dynamic loading of modules at run time. An executable file using the ELF format consist of ELF Header, Program Header Table and Section Header Table.

IV. **Linking:** The linking phase is a critical part of the C compilation, where object files and libraries are combined into a single executable file. The `.text` sections from multiple object files are combined into a single `.text` section in the executable. Similarly, the `.data` and `.bss` sections from multiple object files are merged, with `.data` holding initialized data and `.bss` holding uninitialized data. Moreover, since every `.o` file has its own symbol table, so if the same symbol (e.g., a function or variable) is defined in multiple object files, the linker must resolve which definition to use. In most cases, the linker uses the first definition it encounters or provides a mechanism to handle multiple definitions. The output of linking phase is an executable file with a specific format (e.g., ELF on Linux, PE on Windows). This format includes headers (containing information about how to load the executable into memory) and sections for code, data, and metadata. The two types of linking are briefly described below:

   a. **Static Linking**: Combines object files and static libraries (`libc.a`) into a single executable file. Static libraries are included in the executable at link time. The linker may perform optimizations like elimination unused functions or data. This way the final executable is self-contained and do not require libraries on the machine on which it will be executed later. However, the size of a statically linked file is quite large.
   b. **Dynamic Linking**: Links against shared libraries (`libc.so`) at runtime. The dynamically linked executable file contains references to these libraries, so the executable size is small and need the linked libraries on the machine on which it will be executed later. The dynamic linker/loader (part of the OS) loads the shared libraries into memory and resolves symbols when the program is executed.

Note:

- Finally, the loader is part of the OS, which loads the executable into memory and make it a process. If the executable depends on shared libraries (dynamic linking), the loader resolves these dependencies at runtime.
- The 64-bit x86 virtual memory map splits the address space into two: the lower section (with the top bit set to 0) is user-space, the upper section (with the top bit set to 1) is kernel-space.
- The x86-64 CPU chips that you can buy today, implement 48-bit logical address for virtual memory (as shown), and 40 bits for physical memory.



Hands On Example of C Compilation Process

Dear students, it is time to make our hands dirty to practically perform all of the above-mentioned steps of C-compilation process. For compilation, we will use the GNU Compiler Collection (GCC), which is a powerful and widely used open-source compiler system developed by the GNU Project. GCC is known for its versatility and is capable of compiling code written in various programming languages. You can install GCC using following command if it is not already installed on your system:

```
$ sudo apt update
$ sudo apt install gcc
$ sudo apt install build-essential
$ cat /proc/os-release
$ uname -a
$ gcc --version
gcc (Debian 13.3.0-5) 13.3.0
$ ldd --version
ldd (Debian GLIBC 2.38-13) 2.38
```

Let us write down a multi-file C source code on our Kali Linux machine using some text editor like `vim`, `nano`, or `gedit` and go step by step through all the C compilations steps/phases:

```c
//module3/3.1/compilation/driver.c
#include <stdio.h>
#include <stdlib.h>
#include <mymath.h>

int main(int argc, char* argv[]) {
   int num1 = atoi(argv[1]);
   int num2 = atoi(argv[2]);
   printf("%d + %d = %d \n", num1, num2, myadd(num1,num2));
   printf("%d - %d = %d \n", num1, num2, mysub(num1,num2));
   return 0;
}
```

```c
//mymath.h
int myadd(int, int);
int mysub(int, int);
```

```c
//mysub.c
int mysub(int a, int b) {
    int ans = a - b;
    return ans;
}
```

```c
//myadd.c
int myadd(int a, int b) {
    int ans = a + b;
    return ans;
}
```

I. **Preprocessing**: Using `-E` option we can instruct the `gcc` to just perform preprocessing, which will display the output on screen. In order to save the output in a file, we need to redirect the output in a file named `<somename.i>`. Moreover, the default location where the C preprocessor searches for the header files is the `/usr/include/` directory. Since in our case, one header file `myheader.h` is in the current working directory, so we need to use the `-I` flag followed by the directory path to instruct `gcc` to look for header files in that specific directory.

```
$ gcc -E  myadd.c   1>  myadd.i
$ gcc -E  mysub.c   1>  mysub.i
$ gcc -E  driver.c  -I ./   1>  driver.i
```

We can use the `less` or `cat` command to view the contents of the preprocessed files as their file type is C source, ASCII text, which can be checked using the `file` command. Once you do that, you will see lot of information from different header files that have been included in it and at the very end you can see the actual lines of code you've written. Moreover, please do check out different C header files that are there in the `/usr/include/` directory.

```
┌──(kali㉿kali)-[~/IS/module3/3.1]
└─$ ls /usr/include/
aarch64-linux-gnu    error.h             iconv.h               mtd                   regulator           ttyent.h
aio.h                et                  ifaddrs.h             ncurses_dll.h         resolv.h            uchar.h
aircrack-ng          eti.h               inttypes.h            ncurses.h             riscv64-linux-gnu   ucontext.h
aliases.h            etip.h              iproute2              ncursesw              rpc                 ulimit.h
alloca.h             execinfo.h          kadm5                 net                   rpcsvc              unctrl.h
alpha-linux-gnu      expat_external.h    kdb.h                 netash                ruby-3.1.0          unicode
argp.h               expat.h             KHR                   netatalk              s390x-linux-gnu     unistd.h
argz.h               fcntl.h             krb5                  netax25               sched.h             utime.h
ar.h                 features.h          krb5.h                netdb.h               scsi                utmp.h
arm-linux-gnueabi    features-time64.h   langinfo.h            neteconet             search.h            utmpx.h
arm-linux-gnueabihf  fenv.h              lastlog.h             netinet               selinux             uuid
arpa                 file                libgen.h              netipx                semaphore.h         uv
arping.h             finclude            libintl.h             netiucv               sepol               uv.h
asm                  FlexLexer.h         libmount              netpacket             setjmp.h            values.h
asm-generic          fmtmsg.h            libr                  netrom                sgtty.h             video
assert.h             fnmatch.h           libxml2               netrose               sh4-linux-gnu       wait.h
bits                 form.h              limits.h              nfs                   shadow.h            wchar.h
blkid                fpu_control.h       link.h                nl_types.h            signal.h            wctype.h
byteswap.h           fstab.h             linux                 nss.h                 sound               wordexp.h
c++                  fts.h               llvm-16               obstack.h             sparc64-linux-gnu   X11
capstone             ftw.h               llvm-17               openssl               spawn.h             x86_64-linux-gnu
cifsidmap.h          gawkapi.h           llvm-c-16             openvpn               stab.h              x86_64-linux-gnux32
clang                gconv.h             llvm-c-17             panel.h               stdc-predef.h       xcb
clif.h               gdb                 locale.h              paths.h               stdint.h            xen
com_err.h            getopt.h            loongarch64-linux-gnu pcre2.h               stdio_ext.h         xorg
complex.h            gio-unix-2.0        lz4frame.h            pcre2posix.h          stdio.h             z3_algebraic.h
cpio.h               GL                  lz4.h                 poll.h                stdlib.h            z3_api.h
crypt.h              GLES                lz4hc.h               powerpc64le-linux-gnu string.h            z3_ast_containers.h
ctype.h              GLES2               m68k-linux-gnu        powerpc64-linux-gnu   strings.h           z3_fixedpoint.h
```

II. **Compilation**: Using `-S` option we can instruct the `gcc` to just perform compilation, which takes the `.c` or `.i` file(s) as input and generates the assembly code for underlying architecture and in our case, it is x64 and the generates output file(s) with `.s` extension.

- ***C-Standard to use:*** The C programming language has several standard versions, that are aimed to unify various implementations of C and ensure code portability across different platforms. The most commonly used ones are C89/90, C99, C11, and C18. During compilation step, you can also instruct `gcc` to follow specific C language standard while compiling the code for example using `-std` option and passing the C standard of your choice such as `-std=c11`. To check out the details about C standards, read the man page of `standards` from section 7.
- ***Optimization Level:*** During the compilation step, we can also instruct the `gcc` to generate optimized code using `-O0, -O1, -O2, -O3, -Ofast, -Os, -Og` flags, which instruct the compiler to optimize the generated machine code for *performance*, *size*, or a balance of both. To check out the details about C optimization flags, read the `man` page of `gcc`.
- ***Generate Code for Specific Architecture:*** During the compilation step, we can also instruct `gcc` to generate code for specific architecture using the `-m32` or `-m64` flags, which affects the size of pointers, integer types, and function calling conventions used in the generated binary. By default, `gcc` generates 64-bit binaries on a 64-bit Linux system, so you usually don't need to specify any additional flags if you're targeting a 64-bit architecture. Moreover, if you want to generate 32-bit binary, you need to have the 32-bit libraries and development tools installed on your system. On Debian-based systems like Ubuntu, you can install the necessary packages by installing `gcc-multilib` package (`sudo apt install gcc-multilib g++-multilib`), and then need to mention the `-m32` option during compilation, assembling and linking phases.

```
$ gcc -S *.i -std=c18 -m64        //will generate three .s files
```

Above command will generate three assembler source, ASCII text files namely `driver.s`, `myadd.s` and `mysub.s`. Once again you can view their contents using the `cat` or `less` commands. Students are advised to view and understand contents of intermediate output files.

III. **Assembling**: Using `-c` option we can instruct the `gcc` to perform the steps till assembling phase, which takes the `.c` or `.i` or `.s` file(s) as input and generates the object code files with `.o` extension. If you want to include debugging symbols, so as to load this file inside a debugger, use `-ggdb` option. In our case three object files will be generated namely `driver.o`, `myadd.o` and `mysub.o`. However, this time, you CANNOT view the contents of these object files using the `cat` or `less` commands, since these are ELF 64-bit LSB relocatable files as can be seen using the `file` command.

```
$ gcc -c -ggdb *.s          //will generate three .o files
$ file driver.o
driver.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), with debug_info, not stripped
```

IV. **Linking**: In our example the object file `driver.o` needs to be merged with the `myadd.o` and `mysub.o` file, other than the `printf.o` file whose code resides in the standard C library `libc.a` or `libc.so`. Please make time and read more information about standard C library from the man libc man page in section 7.
For x86_64 architecture, the standard C library along with other libraries resides in a standard location which is `/usr/lib/x86_64-linux-gnu/` directory as well as in the `/lib/x86_64-linux-gnu/` directory. The second location is used during the booting process even before `/usr/` is mounted. These libraries are pre-compiled set of functions and are ready to use by our programs. These libraries come into two flavors static library and dynamic library.

```
┌──(kali㉿kali)-[~/IS/module3/3.1]
└─$ ls -lh /usr/lib/x86_64-linux-gnu/ | grep libc.a
-rw-r--r--  1 root root  5.3M Jun 11 01:25 libc.a
lrwxrwxrwx  1 root root    21 Jun 25 03:44 libclang-16.so.1 → libclang-16.so.16.0.6
-rw-r--r--  1 root root   33M Jun 25 03:44 libclang-16.so.16.0.6
lrwxrwxrwx  1 root root    17 Jul 31 11:40 libclang-17.so.1 → libclang-17.so.17
-rw-r--r--  1 root root   32M Jul 31 11:40 libclang-17.so.17
lrwxrwxrwx  1 root root    33 Jun 25 03:44 libclang-cpp.so.16 → ../llvm-16/lib/libclang-cpp.so.16
lrwxrwxrwx  1 root root    33 Jul 31 11:40 libclang-cpp.so.17 → ../llvm-17/lib/libclang-cpp.so.17
lrwxrwxrwx  1 root root    17 Jan  6  2024 libcrack.so.2 → libcrack.so.2.9.0
-rw-r--r--  1 root root   42K Jan  6  2024 libcrack.so.2.9.0

┌──(kali㉿kali)-[~/IS/module3/3.1]
└─$ ls -lh /usr/lib/x86_64-linux-gnu/ | grep libc.so
lrwxrwxrwx  1 root root    18 Jun 23 15:12 libcjson.so.1 → libcjson.so.1.7.18
-rw-r--r--  1 root root   34K Jun 23 15:12 libcjson.so.1.7.18
lrwxrwxrwx  1 root root    24 Jun 23 15:12 libcjson_utils.so.1 → libcjson_utils.so.1.7.18
-rw-r--r--  1 root root   18K Jun 23 15:12 libcjson_utils.so.1.7.18
-rw-r--r--  1 root root   283 Jun 11 01:25 libc.so
-rwxr-xr-x  1 root root  1.9M Jun 11 01:25 libc.so.6
```

For the linking process, you can instruct the `gcc` to perform linking using `-lc` option which instructs `gcc` to link the source file with standard C library and generate an executable. Since every C program need to be linked with standard C library, so this is the default, i.e., you need not to mention `-lc` option explicitly while linking. One more point is that `gcc` will look for the dynamic version of the library by default (i.e., `libc.so`), if it exists it will link the source code with `libc.so`, otherwise it will look for the static version, i.e., `libc.a`. To force static linking, you can use the `--static` flag of `gcc`. Finally, you can also choose the name of executable of your own choice using `-o` option otherwise, it will by default generate the exe file with the name `a.out`. You can use any of the following two commands to generate the final executable, either using dynamic linking or static linking:

```
$ gcc *.o -o dynamicexe -lc
$ gcc *.o --static -o staticexe -lc
```

You can see the difference between static and dynamic linking by checking the size of both executables, which clearly shows that size of `dynamicexe` is much smaller than `staticexe`. Because in case of static linking, linker resolves all the external references and final executable contains the complete code, that is why the size of `staticexe` is much larger. This executable can even run on those machines where standard C library doesn't exist. However, in case of dynamic linking the final executable doesn't contain the code of external function (only contains a stub), that is why the file `dynamicexe` is smaller in size, and the executable requires the standard C library to exist on that system.

```
┌──(kali㉿kali)-[~/IS/module3/3.1]
└─$ gcc *.o -o dynamicexe -lc

┌──(kali㉿kali)-[~/IS/module3/3.1]
└─$ gcc *.o --static -o staticexe -lc

┌──(kali㉿kali)-[~/IS/module3/3.1]
└─$ ls
driver.c  driver.o  dynamicexe  myadd.i  myadd.s   mysub.c  mysub.o  staticexe
driver.i  driver.s  myadd.c     myadd.o  mymath.h  mysub.i  mysub.s

┌──(kali㉿kali)-[~/IS/module3/3.1]
└─$ ls -lh
total 840K
-rw-rw-r-- 1 kali kali  287 Oct 27 21:01 driver.c
-rw-rw-r-- 1 kali kali  46K Oct 27 21:01 driver.i
-rw-rw-r-- 1 kali kali 3.2K Oct 27 21:15 driver.o
-rw-rw-r-- 1 kali kali 1.1K Oct 27 21:06 driver.s
-rwxrwxr-x 1 kali kali  18K Oct 27 21:28 dynamicexe
-rw-rw-r-- 1 kali kali   69 Oct 27 20:56 myadd.c
-rw-rw-r-- 1 kali kali  187 Oct 27 20:59 myadd.i
-rw-rw-r-- 1 kali kali 2.5K Oct 27 21:15 myadd.o
-rw-rw-r-- 1 kali kali  255 Oct 27 21:06 myadd.s
-rw-rw-r-- 1 kali kali   53 Oct 27 20:55 mymath.h
-rw-rw-r-- 1 kali kali   68 Oct 27 20:56 mysub.c
-rw-rw-r-- 1 kali kali  187 Oct 27 21:00 mysub.i
-rw-rw-r-- 1 kali kali 2.5K Oct 27 21:15 mysub.o
-rw-rw-r-- 1 kali kali  265 Oct 27 21:06 mysub.s
-rwxrwxr-x 1 kali kali 721K Oct 27 21:28 staticexe
```

V. **Execute the Program**:

Now it is time to execute the program. The `./` before the program name actually specifies that the loader should look for the program file in the current working directory. Otherwise, the loader will look for the program inside the directories mentioned inside the PATH variable, which is an environment variable that contains colon separated absolute path names of the directories where the shell will look for the executables.

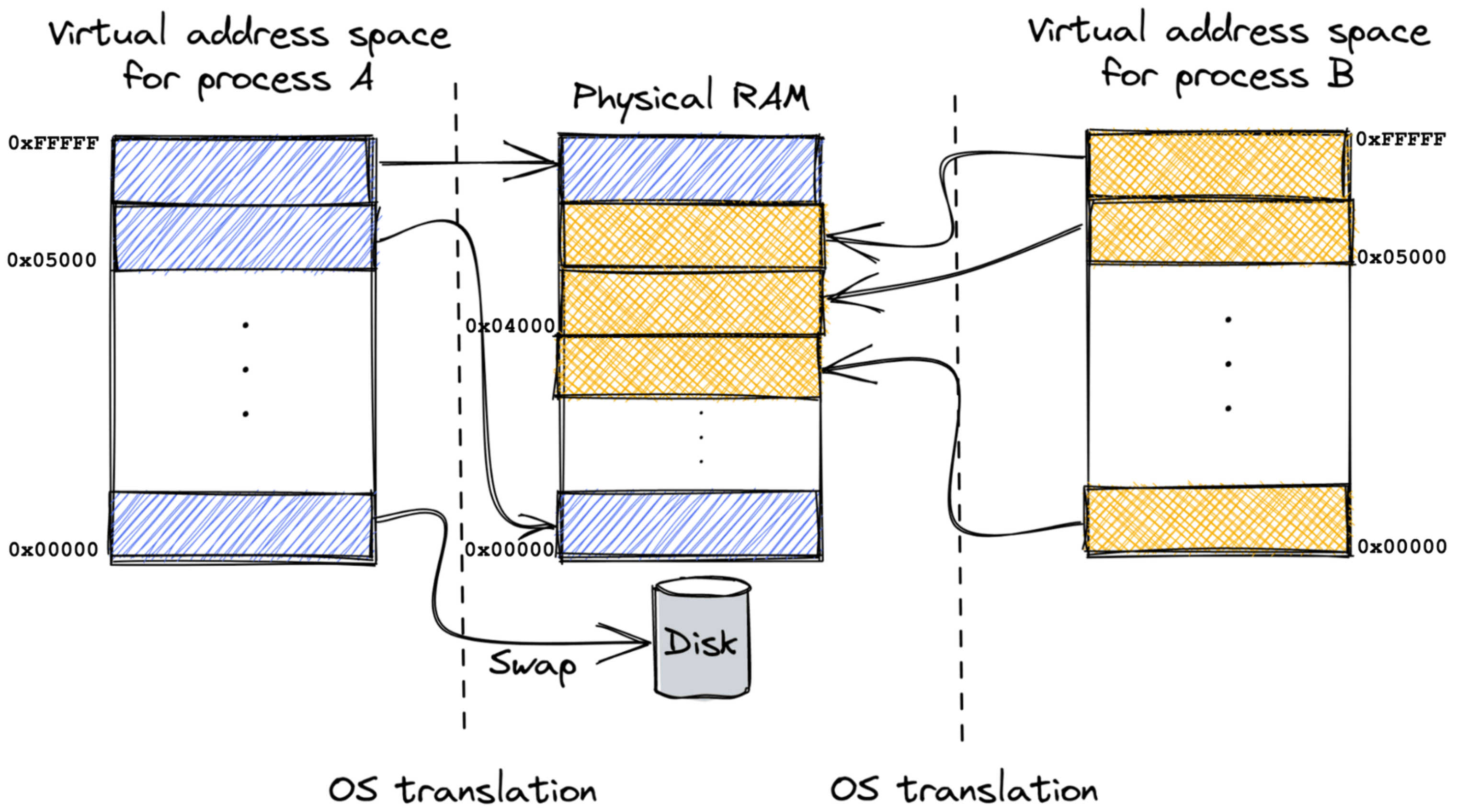
```
┌──(kali㉿kali)-[~/IS/module3/3.1]
└─$ ./dynamicexe 5 3
5 + 3 = 8
5 + 3 = 2

┌──(kali㉿kali)-[~/IS/module3/3.1]
└─$ echo $?
0
```

- A process is created using `fork()` / `clone()` system call that create a nearly exact copy of the parent process.
- The program binary (`a.out`) is loaded inside the child process usually using the `execve()` system call.
- The binary is initialized, using constructors/functions that are there in every ELF, e.g., libc initialize memory regions for dynamic allocations when the program is initialized.
- A normal ELF automatically calls `__libc_start_main()` in `libc`, which in turn calls the program's `main()` function and your code starts running.
- The binary reads its input from the outside world, from command-line arguments and environment variables.
- The binary code executes and does what the developer has coded it for.
- The binary terminates by either receiving an unhandled signal or by calling the `exit()` system call. After termination, the process will remain in a zombie state until they are `wait()`ed on by their parent. When this happens, their exit code will be returned to the parent, and the process will be freed. If their parent dies without `wait()`ing on them, they are re-parented to the `init/systemd` process and will stay there until they're cleaned up.

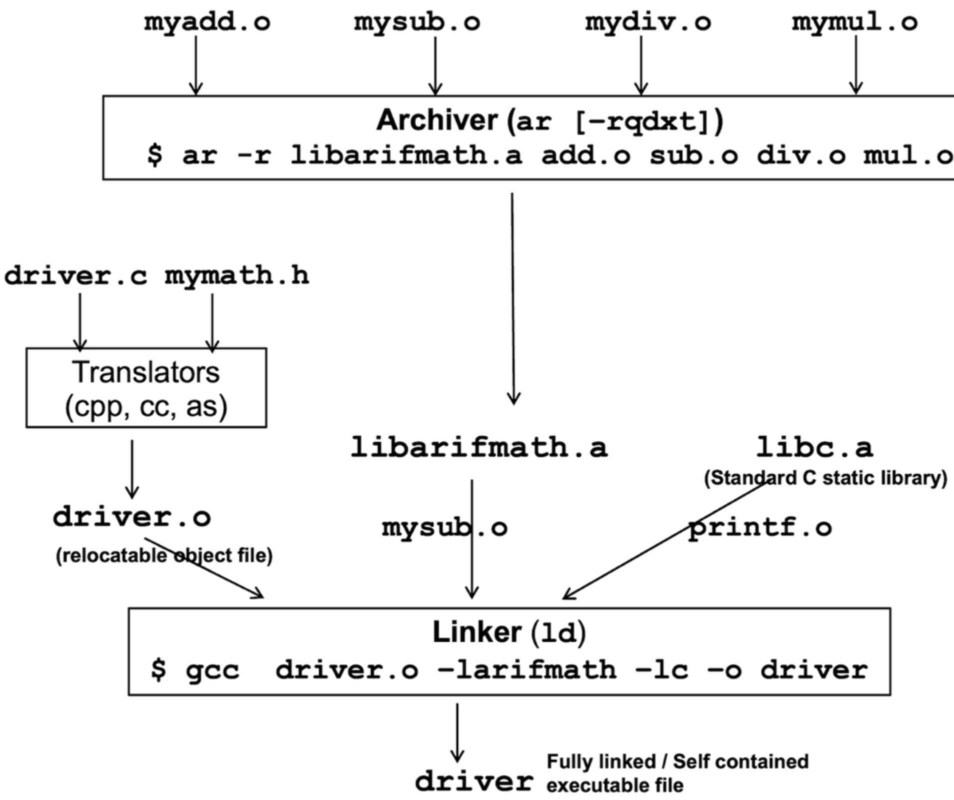
Note: In programming, a program's return value (often called the exit status or exit code) indicates whether the program completed successfully or if an error occurred. This value is returned to the operating system/parent process (shell in our case) when the program finishes execution. By convention, a return value of 0 typically indicates that the program was executed successfully. Any non-zero return value usually indicates that an error occurred, specifying the type of error. In Linux Bash shell, you can check the return value of the last executed program inside the environment variable **`$?`**. In Linux and other Unix-like operating systems, the exit status of a process is represented as an 8-bit integer. This means that the exit status is effectively limited to values between 0 and 255. However, the convention is that exit statuses above 127 are reserved for special purposes related to process termination via signals ☺

Physical vs Virtual Address Space of a Process



Creating & using your own Static Library

Creating a static library is the process of concatenating related relocatable object files into a single file called an archive. In Linux the archiver tool **ar** is used to create a static library, and the process is shown in the image below:



Linker's algorithm for resolving external references:

- Scan all the `.o` and `.a` files in the command line order.
- During the scan, keep a list of the current unresolved references.
- Try to resolve each unresolved reference in the list against the symbols defined in obj files.
- If any entries in the unresolved list at end of scan, then error.

Problem:

- Command line order matters. So best practice is to put libraries at the end of the command line.

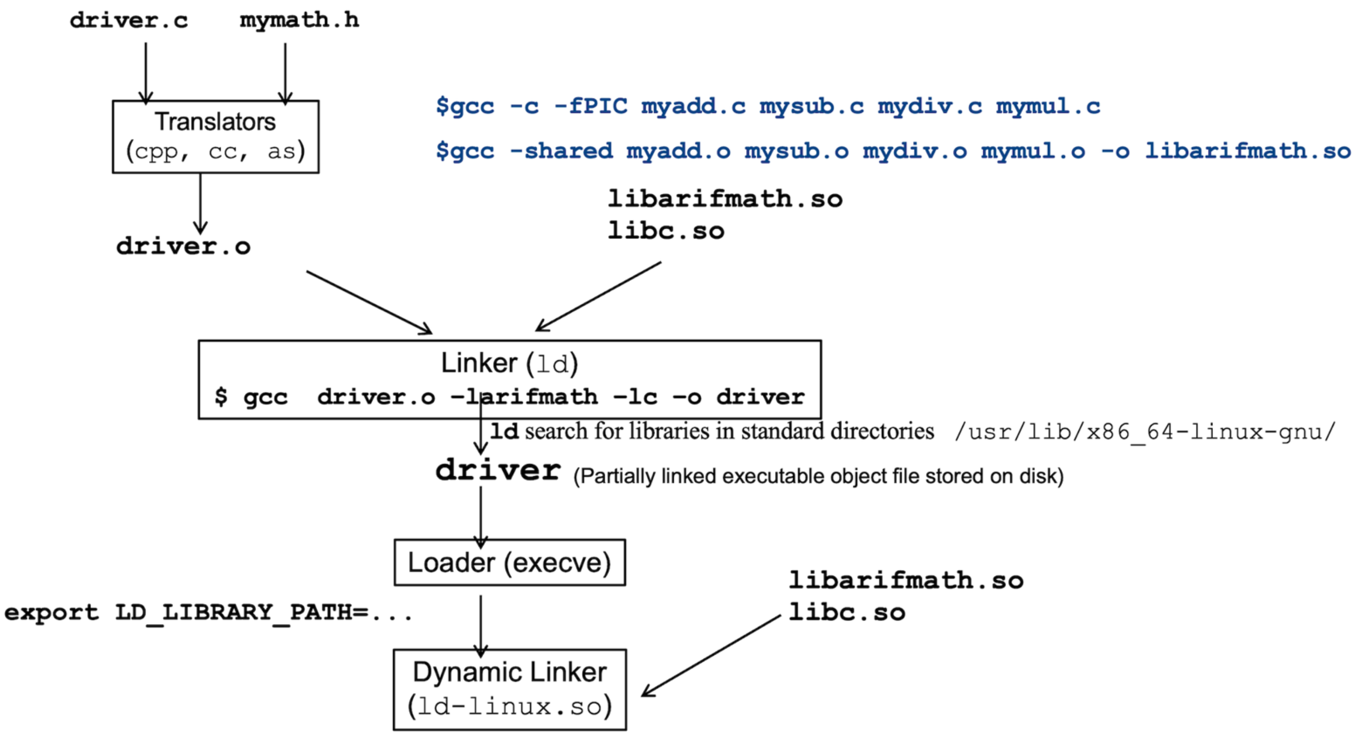
To check out what all object files are there inside the standard static C library:

```
$ ar -t /usr/lib/x86_64-linux-gnu/libc.a
```

Example Code: Visit `module3/3.1/staticlibrary/` directory for the related files to create a static library `libarifmath.a`

Creating & using your own Dynamic/Shared Library

A dynamic library or shared object is similar to static library because it is also a group of object files. However, a dynamic library differs from a static library as the linker and loader both behave differently to a dynamic library. To create a dynamic library in Linux, you simply generate the object files from source files using `-fPIC` flag, that instruct `gcc` to generate position-independent code. A code that can be loaded and executed at any address without being modified by the linker is known as position-independent code. Then you use the `-shared` flag to `gcc` to link all the object files to a shared object file. The process is shown in the image below:



In dynamic linking the goal is to keep the library code outside your final executable. Therefore, the library is there on the system, but it is not actually copied into your executable. If you need it during the run time, it's going to be linked to your program. The advantage of dynamic linking is that you do not need to have duplicate copies of this library in each of the executable. Now if you have 100 different programs, they all use some of the library functions, like `printf`, in static linking, all these hundred programs need to have a copy of `printf` function and also all the other functions that `printf` depends on. That adds a lot of extra space to the program. Also, another problem with statically linked executable is, if in the future, the library gets updated, for example `printf` has a security problem, and it gets fixed, then all these binaries need to be re-linked.

Last but not the least, remember, when you will run the final executable, the loader will search for the shared object files in the standard locations (in Linux it is `/usr/lib/x86_64-linux-gnu/` directory). In Linux, to temporary override the default shared library paths, we can use the `LD_LIBRARY_PATH` environment variable. This variable is especially useful when running applications that require specific or custom versions of libraries that are not installed in the default locations.

Example Code: Visit `module3/3.1/dynamiclibrary/` directory for the related files to create a static library `libarifmath.so`

Reading/Viewing Contents of Object Files

We have covered all four phases of the C Compilation process, and generated all the intermediate files (preprocessed file, assembly file, object file) and the final executable. We have already read the contents of preprocessed file with `.i` extension and the assembly file with `.s` extension, however, we haven't checked the contents of relocatable object files and final executable. You can't view the contents of these files using normal programs like `cat` and `less`. To deal with the object and executable files in Linux, you can use the following utilities:

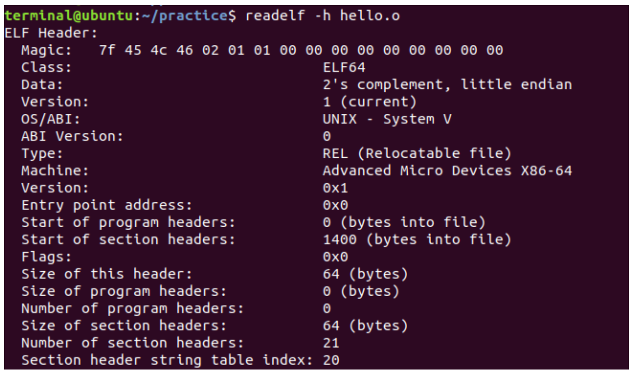
- **`readelf`** utility is used to display information about ELF files.
- **`objdump`** utility is used to disassemble and inspect object files, executables and libraries.
- **`nm`** utility is used to display the symbol table of object files, executables and libraries.
- **`strings`** utility is used to extract/display the ASCII/Unicode text embedded in binary files.
- **`file`** utility is used to determine the type of a file by inspecting the file's header.
- **`ldd`** utility is used to display the shared libraries with which the final executable is linked.
- **`strip`** utility is used to discard/remove symbols and debugging info from binaries.
- **`objcopy`** utility allows you to modify object files by copying them with alterations, such as stripping sections, changing formats, and extracting specific parts of the file.
- **`checksec`** utility is used to analyze security features of binaries and shows which exploit mitigation features are enabled/disabled in a binary (NX, PIE, Canary, RELRO, FORTIFY)
- **`addr2line`** utility is used to translate memory addresses into human-readable file names and line numbers in the original source code (if the binary contains debugging symbols).

Inspecting Object Files using `readelf`

The `readelf` is a command-line utility used to display detailed information of Executable and Linkable Format (ELF) files on Linux and other Unix-like operating systems. ELF is the standard file format for executables, object code, shared libraries, and core dumps in Linux. You can view the man page of `readelf` to get more information.

`$ readelf -[option] hello.o`

- The **`-a`** option displays all available info about the elf file, including headers, sections, segments and symbols
- The **`-h`** option displays information about ELF header, that contains metadata about the ELF file, such as:
  - Magic Number starts with `0x7F 45 4C 46` specifying that it is an ELF file. The `02` after that specifies the class of binary (64-bit). The `01` after that specifies data encoding (little-endian). The last `01` specifies the ELF version.
  - File type (Executable, Shared Library, or Object File)
  - Target architecture (e.g., x86-64, ARM)
  - Entry point address (where execution starts)
  - Offset locations for different sections



- The **`-S`** option displays different section headers (`.text, .plt, .got, .data, .rodata, .bss` etc) of the ELF file.
- The **`-s`** option displays the symbol table (function/variable names with addresses)

Inspecting Object Files using *objdump*

`objdump` is more general-purpose tool as compared to `readelf`, that is used for disassembling and inspecting binary files and works on multiple object formats like ELF, PE, COFF etc. It is particularly useful for debugging, analyzing binaries, and understanding how code is translated into machine instructions. By default, it displays the disassembly in AT&T format.

`$ objdump -[option] -M intel hello.o`

- The **`-f`** option displays the file header information (architecture, format, entry point, etc)
- The **`-h`** option displays information about the section headers, such as their sizes and offsets.
- The **`-t`** option displays the symbol table (function/variable names with addresses).
- The **`-d`** option disassembles only the executable sections (e.g., .text)
- The **`-D`** option disassembles all sections, including the non-executable ones.

Inspecting Object Files using `ldd`

`ldd` is a command-line utility used on Linux and other Unix-like operating systems to display the shared library dependencies of an executable or shared library. It shows which dynamic libraries an executable or shared library relies on, helping you understand the runtime requirements and ensuring that all necessary libraries are available on the system. To show the shared libraries required by an executable or shared library:

`$ ldd ./dynamicexe`

```
terminal@ubuntu:~/practice$ ldd ./dynamicexe
  linux-vdso.so.1 (0x00007ffc4e732000)
  libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f97eaac6000)
  /lib64/ld-linux-x86-64.so.2 (0x00007f97eb0b9000)
```

- Description of Output:
  - `linux-vdso.so.1` is a virtual dynamic shared object.
  - `libc.so.6` is the C standard library.
  - `/lib64/ld-linux-x86-64.so.2` is the dynamic linker/loader.

To get more detailed information about the libraries and their paths use `-v` option for verbose output:

`$ ldd -v ./dynamicexe`

```
terminal@ubuntu:~/practice$ ldd -v ./dynamicexe
  linux-vdso.so.1 (0x00007ffe4bfbf000)
  libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f3d91148000)
  /lib64/ld-linux-x86-64.so.2 (0x00007f3d9173b000)

  Version information:
  ./dynamicexe:
     libc.so.6 (GLIBC_2.2.5) => /lib/x86_64-linux-gnu/libc.so.6
  /lib/x86_64-linux-gnu/libc.so.6:
     ld-linux-x86-64.so.2 (GLIBC_2.3) => /lib64/ld-linux-x86-64.so.2
     ld-linux-x86-64.so.2 (GLIBC_PRIVATE) => /lib64/ld-linux-x86-64.so.2
```

C Code Vulnerabilities and Mitigations

- We all know that a **vulnerability** is a weakness or flaw in a system that can be exploited by an attacker to compromise the system's *integrity*, *availability*, and/or *confidentiality*. Vulnerabilities can exist in various types of systems and can lead to unintended behaviours or security breaches.
- The C programming language, while powerful and widely used, has several inherent features and common pitfalls that can lead to vulnerabilities. These vulnerabilities often arise due to the low-level nature of C, its handling of memory, and lack of built-in safety mechanisms.
- In C programming, **secure** and **unsecure** functions are terms that refer to how well a function handles potentially dangerous operations, particularly regarding memory safety and data integrity. Secure functions are designed to mitigate risks such as buffer overflows, format string vulnerabilities, and other common issues. In contrast, unsecure functions often lack built-in protections and can expose programs to these risks.

Example 1: Secure vs Unsecure C Program

The **`gets()`** function is unsafe because it does not perform bounds checking, thus can lead to buffer overflow. In the left code snippet, the `gets(buffer)` reads a line of input from standard input into `buffer`. If the input exceeds 20 characters, it will overflow `buffer`, potentially overwriting adjacent memory, thus generating a segmentation fault. In the code snippet shown on the right, the **`fgets()`** function is used, which is safer because it allows you to specify the maximum number of characters to read, which helps prevent buffer overflow. The `fgets(buffer, sizeof(buffer), stdin)` reads up to `sizeof(buffer) - 1` characters from `stdin` into `buffer`, ensuring it does not exceed the buffer size. It also appends a null terminator to `buffer` and handles newline characters properly.

```c
//vulnerable.c
#include <stdio.h>
void vulnerable_function() {
    char buffer[20];
    printf("Enter some text: ");
    gets(buffer);
    printf("You entered: %s\n", buffer);
}




int main() {
    vulnerable_function();
    return 0;
}
```

```c
//safe.c
#include <stdio.h>
#include <string.h>
void safe_function() {
    char buffer[20];
    printf("Enter some text: ");
    if (fgets(buffer, sizeof(buffer), stdin) != NULL) {
        int len = strlen(buffer);
        if (len > 0 && buffer[len - 1] == '\n') {
            buffer[len - 1] = '\0';
        }
        printf("You entered: %s\n", buffer);
    }
}
int main() {
    safe_function();
    return 0;
}
```

Example 2: Secure vs Unsecure C Program

The C **`strcpy()`** function is unsafe because it does not perform bounds checking, and copies a string from the source to the destination buffer *without* checking the size of the buffer. This can lead to buffer overflows if the source string is larger than the destination buffer. In the left code snippet shown below, the `strcpy(buffer, long_string)` tries to copy the `long_string` inside the `buffer`, and thus causing a buffer overflow, and generating a segmentation fault. In the code snippet shown in the right, the **`strncpy()`** function is used, which is safer because it allows you to copy up to `sizeof(buffer) - 1` characters from `long_string` into `buffer`, ensuring it does not exceed the buffer size. The `buffer[sizeof(buffer) - 1] = '\0'` ensures that the buffer is null-terminated even if the source string is longer than the buffer.

```c
//vulnerable.c
#include <stdio.h>
#include <string.h>
void vulnerable_function() {
    char buffer[5];
    char long_string[] = "Hello World…!";
    strcpy(buffer, long_string);  //
    printf("Buffer content: %s\n", buffer);
}


int main() {
    vulnerable_function();
    return 0;
}
```

```c
//safe.c
#include <stdio.h>
#include <string.h>
void safe_function() {
    char buffer[5];
    char long_string[] = "Hello World…!";
    strncpy(buffer, long_string, sizeof(buffer) - 1);
    buffer[sizeof(buffer) - 1] = '\0';
    printf("Buffer content: %s\n", buffer);
}
int main() {
    safe_function();
    return 0;
}
```

Example 3: Secure vs Unsecure C Program

The **`strcat()`** function is unsafe because it does not check the size of the destination buffer, which can lead to buffer overflow, if the buffer is not large enough to hold the concatenated result. In the code snippet shown in the left, the `to_append` string is too long to fit into the `buffer` after the initial "Hello, ". The line `strcat(buffer, to_append)`, will therefore cause a buffer overflow generating a segmentation fault. In the code snippet shown in the right, the **`strncat()`** function is used, which is safer because it allows you to copy up to specify the maximum number of characters to append, which helps prevent buffer overflow by ensuring you don't exceed the size of the destination buffer. The `strncat()` in the code on the right appends up to `sizeof(buffer) – strlen(buffer)-1` characters from `to_append` to `buffer`. This calculation ensures that the total length of `buffer` does not exceed its size and leaves room for the null terminator.

```c
//vulnerable.c
#include <stdio.h>
#include <string.h>
void vulnerable_function() {
    char buffer[10] = "Hello, ";
    char to_append[] = "Muhammad Arif Butt.";
    strcat(buffer, to_append);
    printf("Buffer content: %s\n", buffer);
}


int main() {
    vulnerable_function();
    return 0;
}
```

```c
//safe.c
#include <stdio.h>
#include <string.h>
void safe_function() {
    char buffer[10] = "Hello, ";
    char to_append[] = "Muhammad Arif Butt. ";
    strncat(buffer, to_append, sizeof(buffer) -
strlen(buffer)-1);
    printf("Buffer content: %s\n", buffer);
}

int main() {
    safe_function();
    return 0;
}
```

Example 4: Format String Vulnerability

The `printf()` and `sprintf()` functions are used to output formatted data, but if used with untrusted format strings, it can lead to format string vulnerabilities. It exist in programs where untrusted data is used as a format string in functions like `printf`. C's `printf` and related functions do not validate format strings. An attacker can use format specifiers to read or write arbitrary memory locations. On the contrary, the `snprintf()` function formats and stores a string in a buffer, ensuring that the buffer size is respected to prevent overflow. It also ensures null termination.

```
char user_input[100];
scanf("%s", user_input);
printf(user_input);  // Vulnerable if user_input contains format specifiers
```

Example 5: Use After Free Vulnerability

The `malloc()` and `free()` functions are used for dynamic memory allocation and deallocation. While `malloc` and `free` themselves are not inherently insecure, improper use, such as double-freeing or accessing freed memory, can lead to vulnerabilities. To avoid this, one should use memory management techniques that avoid common pitfalls, such as ensuring pointers are set to `NULL` after being freed and checking for `NULL` returns from `malloc`. The use-after-free vulnerability happens when a program continues to use a pointer after the memory it points to has been freed. C provides manual memory management using `malloc` and `free`. If a pointer is used after `free` without setting it to `NULL`, it can lead to undefined behavior or security issues.

```
char *ptr = malloc(20);
free(ptr);
strcpy(ptr, "vulnerable");
```

Example 6: Dangling Pointer

A pointer that continues to reference a memory location after the memory it points to has been deallocated. Similar to use after free, but can also occur if a pointer is left pointing to a local variable whose stack frame has been popped.

```
char* func() {
    char local_buffer[10];
    return local_buffer;  // Returning address of local variable
}
```

Example 7: Integer Overflow

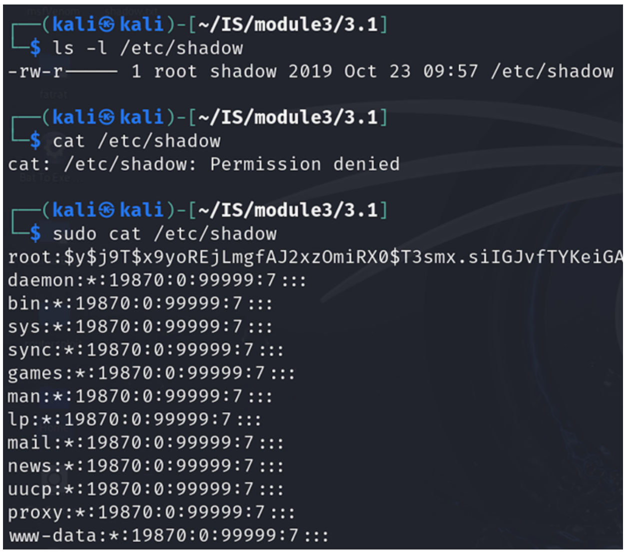
Occurs when arithmetic operations produce results that exceed the representable range of the data type. C does not check for integer overflows. If the result of an arithmetic operation is too large or too small, it wraps around.

```
unsigned int x = 4294967295; // Max value for 32-bit unsigned int
x += 1;  // Causes overflow and wraps around to 0
```

Exploiting Set-UID Privileged Programs

- **Understanding SUID Privileged Programs:**

We all know that the /etc/shadow file is a critical file in Unix-like operating systems that stores user account information, specifically the hashed passwords and related security attributes for user accounts. The /etc/shadow file is usually only accessible by the root user and certain privileged processes. This is to protect sensitive information from unauthorized access. From the screenshot, one can see that the permissions on this file are set to 640, meaning that the owner (root) has read and write permissions, the group (shadow) has only read permissions, and others have no access. So, a regular user cannot even view the content of this file. However, this can be done by a user who is in the sudo group, and since the user kali is a member of this group, so he/she can view its contents using the sudo command as shown in the screenshot.



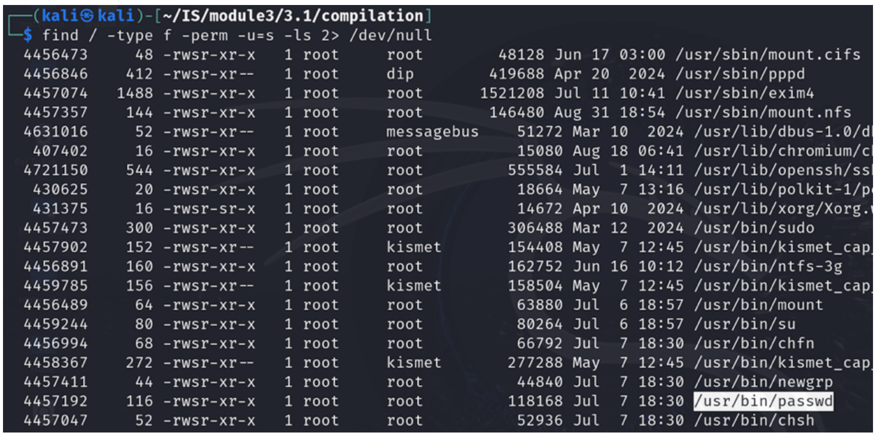
```
┌──(kali㉿kali)-[~/IS/module3/3.1]
└─$ ls -l /etc/shadow
-rw-r----- 1 root shadow 2019 Oct 23 09:57 /etc/shadow

┌──(kali㉿kali)-[~/IS/module3/3.1]
└─$ cat /etc/shadow
cat: /etc/shadow: Permission denied

┌──(kali㉿kali)-[~/IS/module3/3.1]
└─$ sudo cat /etc/shadow
root:$y$j9T$x9yoREjLmgfAJ2xzOmiRX0$T3smx.siIGJvfTYKeiGA
daemon:*:19870:0:99999:7:::
bin:*:19870:0:99999:7:::
sys:*:19870:0:99999:7:::
sync:*:19870:0:99999:7:::
games:*:19870:0:99999:7:::
man:*:19870:0:99999:7:::
lp:*:19870:0:99999:7:::
mail:*:19870:0:99999:7:::
news:*:19870:0:99999:7:::
uucp:*:19870:0:99999:7:::
proxy:*:19870:0:99999:7:::
www-data:*:19870:0:99999:7:::
```

Now a 100$ question is how come every user can use the passwd command to change his/her password, that resides in this /etc/shadow file. One way is to run a privileged daemon running at all times that can do this job for regular users, but this is of course expensive. So, the solution used by all UNIX systems for such tasks is the SUID bit. The SUID (Set User ID) bit is a special file permission in Unix-like operating systems that allows users to execute a file with the permissions of the file owner rather than the permissions of the user executing the file. The advantage of SUID bit is that it allows users to perform tasks that require higher privileges without giving them full access to the system. It is denoted by an 's' in the owner's execute permission or a capital 'S' if the owner's execute permission is off. An example of a **root-owned SUID program** is the /usr/bin/passwd program, that is used by regular users to change their own passwords by modifying the contents of /etc/shadow file owned by root. To identify executable files with the SUID bit set:

$ find / -type f -perm -u=s -ls 2> /dev/null



```
┌──(kali㉿kali)-[~/IS/module3/3.1/compilation]
└─$ find / -type f -perm -u=s -ls 2> /dev/null
 4456473     48 -rwsr-xr-x   1 root     root          48128 Jun 17 03:00 /usr/sbin/mount.cifs
 4456846    412 -rwsr-xr--   1 root     dip          419688 Apr 20  2024 /usr/sbin/pppd
 4457074   1488 -rwsr-xr-x   1 root     root        1521208 Jul 11 10:41 /usr/sbin/exim4
 4457357    144 -rwsr-xr-x   1 root     root         146480 Aug 31 18:54 /usr/sbin/mount.nfs
 4631016     52 -rwsr-xr--   1 root     messagebus    51272 Mar 10  2024 /usr/lib/dbus-1.0/d
  407402     16 -rwsr-xr-x   1 root     root          15080 Aug 18 06:41 /usr/lib/chromium/c
 4721150    544 -rwsr-xr-x   1 root     root         555584 Jul  1 14:11 /usr/lib/openssh/ss
  430625     20 -rwsr-xr-x   1 root     root          18664 May  7 13:16 /usr/lib/polkit-1/p
  431375     16 -rwsr-sr-x   1 root     root          14672 Apr 10  2024 /usr/lib/xorg/Xorg.
 4457473    300 -rwsr-xr-x   1 root     root         306488 Mar 12  2024 /usr/bin/sudo
 4457902    152 -rwsr-xr--   1 root     kismet       154408 May  7 12:45 /usr/bin/kismet_cap_
 4456891    160 -rwsr-xr-x   1 root     root         162752 Jun 16 10:12 /usr/bin/ntfs-3g
 4459785    156 -rwsr-xr--   1 root     kismet       158504 May  7 12:45 /usr/bin/kismet_cap_
 4456489     64 -rwsr-xr-x   1 root     root          63880 Jul  6 18:57 /usr/bin/mount
 4459244     80 -rwsr-xr-x   1 root     root          80264 Jul  6 18:57 /usr/bin/su
 4456994     68 -rwsr-xr-x   1 root     root          66792 Jul  7 18:30 /usr/bin/chfn
 4458367    272 -rwsr-xr--   1 root     kismet       277288 May  7 12:45 /usr/bin/kismet_cap_
 4457411     44 -rwsr-xr-x   1 root     root          44840 Jul  7 18:30 /usr/bin/newgrp
 4457192    116 -rwsr-xr-x   1 root     root         118168 Jul  7 18:30 /usr/bin/passwd
 4457047     52 -rwsr-xr-x   1 root     root          52936 Jul  7 18:30 /usr/bin/chsh
```

- ### Exploiting SUID Privileged Programs:

This is a security risk as well, because if an executable with the SUID bit set is compromised, it can be exploited by attackers to gain unauthorized access or privileges. This is especially risky if the executable has vulnerabilities. Now let's make use of SUID bit. We are going to use SUID bit to view the content of `/etc/shadow` file without using `SUDO`. We'll do this in two simple steps.

Step 1: Create a copy the `/bin/cat` program inside the `/tmp/` directory, change its owner to root, and see if you can view the contents of `/etc/shadow` file:

```
$ cp /bin/cat mycat
$ sudo chown root mycat
$ ./mycat /etc/shadow
```

```
┌──(kali㉿kali)-[~/IS/module3/3.1]
└─$ cp /bin/cat mycat

┌──(kali㉿kali)-[~/IS/module3/3.1]
└─$ sudo chown root mycat

┌──(kali㉿kali)-[~/IS/module3/3.1]
└─$ ls -l mycat
-rwxr-xr-x 1 root kali 48144 Oct 28 09:45 mycat

┌──(kali㉿kali)-[~/IS/module3/3.1]
└─$ ./mycat /etc/shadow
./mycat: /etc/shadow: Permission denied
```

Step 2: Now enable `SUID` bit for `mycat` program, and you can view the content of file `/etc/shadow` without using `sudo`:

```
$ sudo chmod u+s mycat
$ ./mycat /etc/shadow
```

```
┌──(kali㉿kali)-[~/IS/module3/3.1]
└─$ sudo chmod u+s mycat

┌──(kali㉿kali)-[~/IS/module3/3.1]
└─$ ls -l mycat
-rwsr-xr-x 1 root kali 48144 Oct 28 09:45 mycat

┌──(kali㉿kali)-[~/IS/module3/3.1]
└─$ ./mycat /etc/shadow
root:$y$j9T$x9yoREjLmgfAJ2xzOmiRX0$T3smx.siIGJvf
daemon:*:19870:0:99999:7:::
bin:*:19870:0:99999:7:::
sys:*:19870:0:99999:7:::
sync:*:19870:0:99999:7:::
games:*:19870:0:99999:7:::
man:*:19870:0:99999:7:::
lp:*:19870:0:99999:7:::
mail:*:19870:0:99999:7:::
news:*:19870:0:99999:7:::
uucp:*:19870:0:99999:7:::
proxy:*:19870:0:99999:7:::
www-data:*:19870:0:99999:7:::
```

Thus, if a SUID program has vulnerabilities (like buffer overflow), an attacker can exploit those to gain elevated privileges, potentially compromising the entire system. An attacker can manipulate input to a SUID program to execute arbitrary commands with elevated privileges.

To Do:

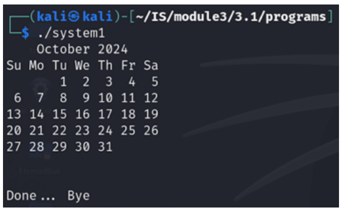
Suppose your friend *Kakamanna* gives you a chance to use his Linux account, and you have your own account on the same system. Can you take over *Kakamanna* account in 10 seconds? Hint: (Try `/bin/bash` and then `/bin/zsh`. Describe what is the difference?)

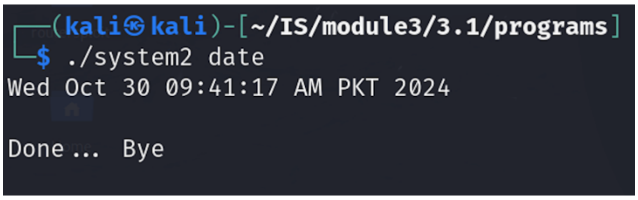
Exploiting the `system()` Function (using Command Injection)

The **`system(char* cmd)`** is a standard C library function, which is used to execute shell commands from within a program. It is passed a string and it spawns a shell program `/bin/sh` and use it to execute the argument passed as cmd (`/bin/sh -c "cal"`). The following two examples describes its behaviour:

```
//system1.c
#include <stdio.h>
#include <stdlib.h>
int main() {
    system("cal");
    printf("Done…Bye\n");
    return 0;
}
```

```
//system2.c
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char* argv[]) {
    system(argv[1]);
    printf("Done…Bye\n");
    return 0;
}
```

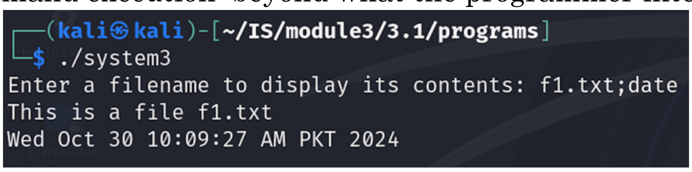




Now study the code of the following program, that prompts the user to enter a filename as input and constructs a command like `system("cat " + filename)` and display the contents of the file (`snprintf()` formats and stores a series of characters and values in the array as per the format string)

```
//system3.c
#include <stdio.h>
#include <stdlib.h>
int main() {
    char filename[100];
    printf("Enter a filename to display its content: ");
    fgets(filename, sizeof(filename), stdin);
    char command[150];
    snprintf(command, sizeof(command), "cat %s", filename);
    system(command);  // Potentially dangerous!
    return 0;
}
```

The `system()` function is really great however, it can introduce significant security vulnerabilities if not used carefully. If user input is passed to `system()`, w/o any validation, an attacker can manipulate that input to execute arbitrary commands. If the command string contains special characters like **;**, **&**, or **|**, it can lead to command execution beyond what the programmer intended.



To Do:

Now let us suppose that the attacker gives the input **`f1.txt;/bin/sh`**. What will happen, do you get a shell? Is this a shell with root privileges? If not, can we get a shell with root privileges?

Exploiting the `system()` Function (using Environment Variables)

Environment variables are name-value pairs. Each running process has a block of memory that contains a set of the name-value pairs which usually come from its parent process. You can view the environment variables of a shell program using the `env` command. So, when you run a command inside a Linux shell, the shell program (parent) send its own environment variables along with some new environment variables to the child process. These environment variables sit in the memory of the child process, and if the child process does not use these variables at all, then these variables will have no impact on its execution. But if the child process uses these variables, then these variables will of course have an impact on its behavior.

Let us consider the `system1.c` source file again:

```c
//system1.c
#include <stdio.h>
#include <stdlib.h>
int main() {
    system("cal");
    printf("Done…Bye\n");
    return 0;
}
```

- A natural question that might come to your mind is that how does `system()` function in the given code file, find the command `cal` as we have not passed the absolute path of this command which is `/usr/bin/cal`. The answer is using the PATH variable, which is an environment variable that contains colon separated list of absolute path names of all the directories where the shell is going to search for the binaries in the specified sequence. To display the current value of any environment variable, say PATH, you can use the following command:

```
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

- Now being a hacker, you're not interested in running the calendar program, rather you are interested in running a shell program with root privileges. Copy the `/bin/sh` program in the present working directory and rename it as `cal` using the following command:
  ```
  $ cp /bin/sh cal
  ```

- Now add the path of the pwd, i.e., a period in the very beginning of the PATH variable using the following command:
  ```
  $ PATH=.:$PATH
  $ echo $PATH
  .:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
  ```

- After modifying the PATH variable of the shell, when you will run the above program, the shell program will pass this modified PATH variable to the child process. So the `system()` function will now first search for `cal` program inside the `pwd`, instead of `/usr/bin/` directory. Hence you will get a shell program instead of the calendar program. Does the resulting shell have root privileges? If not why not?

To Do: Now let us suppose that in the above program the `system("cal");` is changed with `system("/usr/bin/cal");` Can you still hack this program (Hint: Use IFS environment variable)

Exploiting the Shared Libraries (using Environment Variables)

Consider compiling the following program, and linking it statically with `libc.a` and dynamically with `libc.so`.

```
//prog1.c
#include <stdio.h>
int main() {
    printf("Hello World!\n");
    return 0;
}
```

```
┌──(kali㉿kali)-[~/IS/module3/3.1/libraries]
└─$ ls -lh
total 740K
-rwxrwxr-x 1 kali kali  16K Oct 30 22:11 dynamicprog1
-rw-rw-r-- 1 kali kali   82 Oct 30 22:11 prog1.c
-rwxrwxr-x 1 kali kali 720K Oct 30 22:12 staticprog1
```

We can use the `ldd` program to check the dynamic dependency of the above executable files. The output shows that the `dynamicprog1` program rely on three shared object libraries. The first one `linux-vdso.so.1` is the Virtual Dynamic Shared Object library that is used for system calls. The second one `libc.so.6` is the standard C library. The third one `ld-linux-x86-64.so` is the linker itself.

```
┌──(kali㉿kali)-[~/IS/module3/3.1/libraries]
└─$ ldd dynamicprog1
        linux-vdso.so.1 (0x00007ffdf5b54000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f26e4d0d000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f26e4f18000)

┌──(kali㉿kali)-[~/IS/module3/3.1/libraries]
└─$ ldd staticprog1
        not a dynamic executable
```

A natural question that might come to your mind is that how does the loader (`ld`) knows, where to find these libraries. In `x86_64-linux`, the default paths for dynamic libraries are typically configured inside the `x86_64-linux-gnu.conf` file, which is shown below:

`$ cat /etc/ld.so.conf.d/x86_64-linux-gnu.conf`

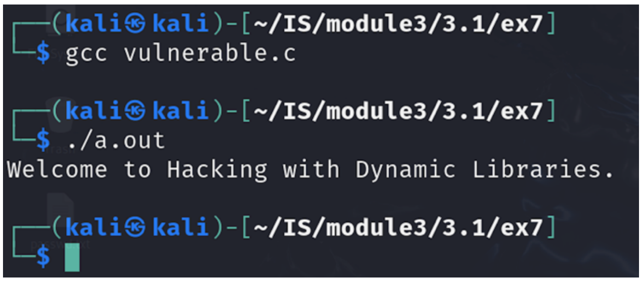
```
/usr/local/lib/x86_64-linux-gnu
/lib/x86_64-linux-gnu
/usr/lib/x86_64-linux-gnu
```

`LD_LIBRARY_PATH`: In Linux, to temporary override the default shared library paths, we can use the `LD_LIBRARY_PATH` environment variable. This variable is especially useful when running applications that require specific or custom versions of libraries that are not installed in the default locations. Remember, using relative paths in `LD_LIBRARY_PATH` can be dangerous. If an attacker can create files in the specified directories or influence the current working directory, he/she may load his/her own libraries.

`LD_PRELOAD`: Similarly, the `LD_PRELOAD` environment variable allows you to specify one or more shared libraries that should be loaded before any other libraries when running a program. This is particularly useful for overriding functions in other shared libraries, which can be helpful in debugging, testing, and custom function implementations. Since, `LD_PRELOAD` allows injecting arbitrary code into a process, it can be a security risk if set globally or used with untrusted programs. For security, `LD_PRELOAD` is ignored for `setuid/setgid` binaries to prevent privilege escalation.

Consider the following program, that is dynamically linked with the standard C library function and therefore, the loads the code of `printf()` and `sleep()` functions at runtime from the standard C library `libc.so` located in `/lib/x86_64-linux-gnu/` directory. Let us compile and execute this program:

```
//vulnerable.c
#include <stdio.h>
#include <unistd.h>
int main() {
    printf("Welcome to Hacking with Dynamic Libraries.\n");
    sleep(2);
    return 0;
}
```

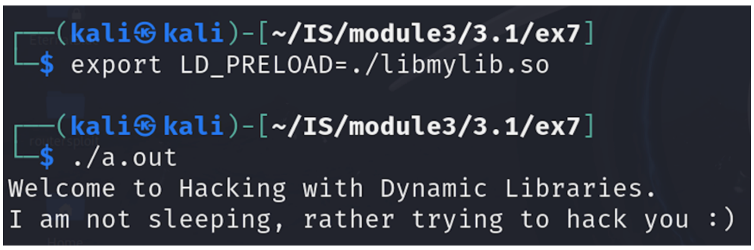


So, you see the above program is working perfectly fine as expected, i.e., it prints the message, sleeps for 2 seconds and then terminates. Let us suppose an attacker creates a malicious shared library having a `sleep()` function as shown below:

```
//mysleep.c
#include <stdio.h>
int sleep() {
    printf("I am not sleeping, rather trying to hack you ☺\n");
    return 0;
}
```

`$ gcc -fPIC mysleep.c mysleep.c`
`$ gcc -shared mysleep.o libmylib.so`

After creating the library `libmylib.so` having only one function in it with the name of sleep. The attacker places the path of this malicious library in the `LD_PRELOAD` environment variable. Now, when you execute the above executable again, instead of calling the `sleep()` function of the standard C library, linker will call the `sleep()` function of this malicious library.
`$ export LD_PRELOAD=./libmylib.so`
`$ ./a.out`



Once you are done, DONOT forget to unset this environment variable `$ unset LD_PRELOAD`

To Do:

Study the following CVEs. Write a detailed and comprehensive report for each CVE, that should cover at least the following points:

- What is this vulnerability?
- How to exploit this vulnerability?
- Practical step by step Proof of Concept (PoC).
- How to mitigate this vulnerability?

Task 1: Exploiting the `sudo` Vulnerability (CVE-2019-14287)

We all know that the `sudo` command is used to execute commands as a superuser. There exists a privilege escalation vulnerability in the `sudo` versions prior to 1.8.28, which allows a user to execute commands as root by bypassing restrictions in the `/etc/sudoers` file. The issue is triggered when the `ALL` keyword is specified in the `Runas` specification of the `sudoers` configuration file.

Task 2: Exploiting the Shellshock Vulnerability (CVE-2014-6271)

Shellshock, also known as the Bash bug, is a critical vulnerability in the Bash shell from versions 1.14 through 4.3. It allows an attacker to execute arbitrary commands on a vulnerable Linux system by sending specially crafted environment variable, a function definition that will be executed by Bash.

Disclaimer

The series of handouts distributed with this course are only for educational purposes. Any actions and or activities related to the material contained within this handout is solely your responsibility. The misuse of the information in this handout can result in criminal charges brought against the persons in question. The authors will not be held responsible in the event any criminal charges be brought against any individuals misusing the information in this handout to break the law.