C Compilation Under the Hood and its Vulnerabilities

# A Bit of Word about C

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.

The first formal definition of C was provided by Brian Kernighan and Dennis Ritchie in the book *The C Programming Language*, commonly known as K&R C, published in 1978. In 1989, the ANSI (American National Standards Institute) standardization of C was completed, leading to the ANSI C standard (also known as C89 or C90). This was followed by the C99 standard, which introduced several new features and improvements. The most recent major standard is C11, with updates for more recent revisions including C17 and C18.

* **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:

1. **Early Beginnings (1960s):** 
   1. **BCPL and B**: 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.
2. **Development of C (1970s)**
   1. **1972**: 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.
   2. **1978**: Dennis Ritchie and Brian Kernighan publish *The C Programming Language*, often referred to as K&R C. This book serves as the first formal definition of C, detailing its syntax and features.
3. **Standardization (1980s)**
   1. **1983**: The American National Standards Institute (ANSI) forms a committee to standardize C. The goal was to provide a consistent and portable version of the language.
   2. **1989**: ANSI publishes the first standard for C, known as ANSI C or C89/C90. This standardization aimed to unify various implementations of C and ensure code portability across different platforms.
4. **New Features and Updates (1990s)**
   1. **1999**: The International Organization for Standardization (ISO) publishes the C99 standard. This update introduces several new features, including:
      1. New data types like long long int and \_Bool.
      2. Inline functions and variable-length arrays.
      3. Improved support for floating-point arithmetic.
      4. New standard library functions and macros.
   2. **2001**: C99 is adopted as the standard by some organizations, though its widespread adoption is gradual due to compatibility issues with existing codebases.
5. **Modern Enhancements (2000s to Present)**
   1. **2011**: The C11 standard is published by ISO, introducing several enhancements:
      1. Support for multi-threading with the \_Atomic type and thread support.
      2. Improved Unicode support with the \_Generic keyword for type-generic macros.
      3. Static assertions and enhanced standard library functions.
   2. **2017**: The C17 standard (also known as C18) is released. It primarily includes bug fixes and clarifications rather than major new features.
   3. **2020**: The C18 standard is published, further refining the language with minor corrections and clarifications to the C17 standard.
6. **Current Status and Ongoing Developments**
   1. **Ongoing**: The 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.

* **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, thanks to its standardized syntax.
  + **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 loops, conditionals, 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.
  + **Preprocessor**: C includes a preprocessor that handles macros, file inclusion, and conditional compilation, which can be used to make the code more flexible and portable.
  + **Simplicity**: While C provides powerful features, it maintains a relatively simple syntax, which helps in learning and understanding the language.
* **Language Constructs**

C language constructs are the building blocks that define the structure and functionality of C programs. Here’s an overview of the fundamental constructs in C:

1. **Data Types:** C provides a variety of data types to store different kinds of data:
   * + **Basic Data Types**:
       - int: Integer data type (e.g., int age = 30;).
       - char: Character data type (e.g., char letter = 'A';).
       - float: Floating-point data type (e.g., float price = 9.99;).
       - double: Double-precision floating-point data type (e.g., double pi = 3.14159;).
     + **Derived Data Types**:
       - **Arrays**: Collection of elements of the same type (e.g., int numbers[10];).
       - **Pointers**: Variables that store memory addresses (e.g., int \*ptr;).
       - **Structures**: User-defined data types that group different types of variables (e.g., struct Person { char name[50]; int age; };).
       - **Unions**: User-defined data types that store different types of variables but only one at a time (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 };).
2. **Operators:** C includes a wide range of operators for various operations:
   * + **Arithmetic Operators**: +, -, \*, /, % (e.g., int sum = a + b;).
     + **Relational Operators**: ==, !=, <, >, <=, >= (e.g., if (a > b) { /\* code \*/ }).
     + **Logical Operators**: &&, ||, ! (e.g., if (a > 0 && b < 10) { /\* code \*/ }).
     + **Bitwise Operators**: &, |, ^, ~, <<, >> (e.g., 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;).
3. **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).
4. **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);).
5. **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++).
6. **Arrays:** Arrays store collections of elements of the same type:
   * + **Array Declaration**: Defines an array and its size (e.g., int arr[5];).
     + **Array Access**: Accesses elements using indices (e.g., arr[0] = 1;).
7. **Structures and Unions**
   * + **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]; };).
8. **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).
9. **Error Handling**
   * + **Error Handling**: C does not have built-in exception handling. Error handling is typically done using return codes and checking error conditions.

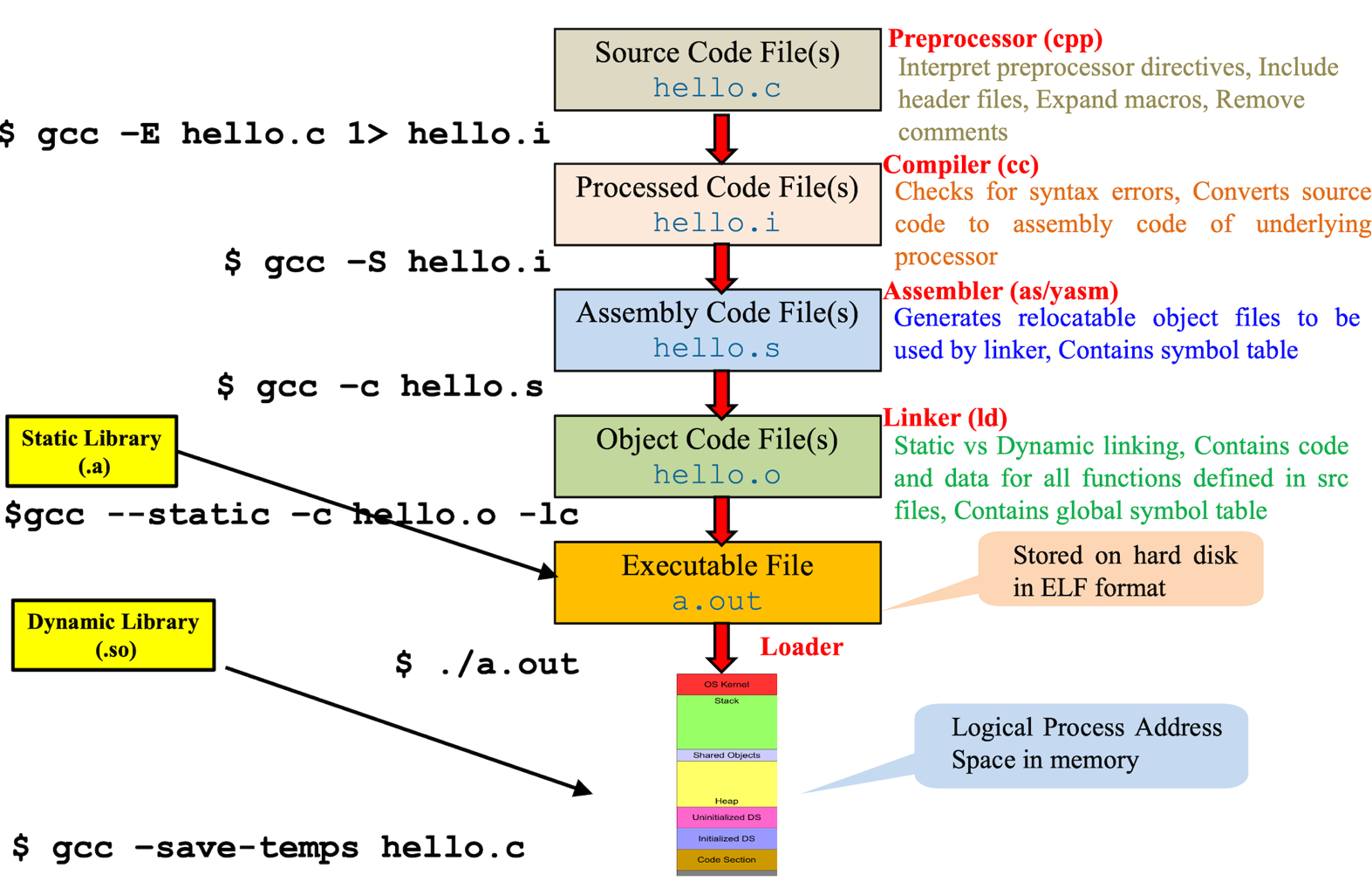
* **Uses of C**

1. **System Programming**: C is widely used in operating systems development, including UNIX, Linux, and Windows kernel components.
2. **Embedded Systems**: Its low-level capabilities and efficiency make C a popular choice for programming embedded systems and microcontrollers.
3. **Application Development**: C is used in various application domains, including game development, desktop applications, and performance-critical applications.
4. **Compilers and Interpreters**: Many modern compilers and interpreters for other programming languages are written in C.
5. **Networking**: C is used to develop network protocols and software that require direct access to hardware and network interfaces.
6. **Database Systems**: The development of database systems and management tools often leverages C for its efficiency and control.

# C Compilation Process and its Tool Chain

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), Scalable Processor ARChitecture (Sun SPARC)
* **Operating System:** Windows, UNIX, Linux, Mac OSX
* **Editor/IDEs:** gedit, vim, atom, sublime, Visual Studio, Eclipse, Xcode
* **Assembler:** NASM, YASM, GAS, MASM
* **Linker:** LD a GNU linker
* **Loader:** Default OS
* **Debugging/RE:** gdb, radare2, objdump and readelf



* **C Compilation Process**

The C compilation process is a series of steps that transforms C source code into an executable program. This process involves several phases and tools that work together to produce the final output. Here’s a detailed overview of the C compilation process:

* + **Preprocessing:** The first step in the compilation process involves preprocessing. The preprocessor handles directives in the source code that start with #. These directives are not part of the C language but are used to perform operations before the actual compilation begins.
    - **File Inclusion**: #include directives are replaced with the contents of the specified files. This is typically used to include header files.
    - **Macro Expansion**: #define macros are expanded to their defined values or code snippets.
    - **Conditional Compilation**: #ifdef, #ifndef, #else, #elif, and #endif are used to include or exclude parts of the code based on certain conditions.
    - **Output**: The result of preprocessing is a preprocessed source file, which is usually saved with .i or .ii extension.
  + **Compilation:** The compiler takes the preprocessed source code and translates it into assembly code. This step involves several key activities:
    - **Syntax Analysis**: The compiler checks the syntax of the code to ensure it conforms to the C language rules.
    - **Semantic Analysis**: It verifies the semantic correctness of the code, such as type checking and ensuring that variables are declared before use.
    - **Optimization**: The compiler may optimize the code to improve performance or reduce resource usage.
    - **Code Generation**: The compiler generates assembly code, which is a low-level representation of the source code.
    - **Output**: The result is an assembly file, usually with .s or .asm extension.
  + **Assembling:** The assembler converts the assembly code into machine code.
    - **Translation**: It translates assembly language instructions into machine code instructions that the CPU can execute.
    - **Symbol Resolution**: The assembler resolves symbolic names (e.g., variable names) to actual memory addresses.
    - **Output**: The result is an object file, typically with .o or .obj extension, containing machine code.
  + **Linking:** The linking phase is a critical part of the C compilation process where object files and libraries are combined into a single executable file. This phase involves several important tasks to ensure that all symbols (e.g., functions and variables) are correctly resolved and all code and data are properly placed in memory. Here’s a detailed description of the linking phase:

1. **Purpose**:
   1. To combine multiple object files into a single executable.
   2. To resolve references between object files and external libraries.
   3. To assign final memory addresses to code and data.
2. **Types of Linking**:
   1. **Static Linking**: Combines object files and static libraries into a single executable file. Static libraries are included in the executable at compile time.
   2. **Dynamic Linking**: Links against shared libraries (dynamic link libraries) at runtime. The executable contains references to these libraries, which are loaded into memory when the program runs.
3. **Linking Process Details**

* **Symbol Resolution**
  + 1. **Symbols**: Names representing functions, variables, and other entities in the code.
    2. **Definitions**: Locations where symbols are defined (e.g., function implementations).
    3. **References**: Places where symbols are used (e.g., function calls or variable accesses).
    4. **Resolving References**: The linker matches references in the object files to their corresponding definitions. For example, if main.o references a function foo() that is defined in foo.o, the linker connects these references to ensure the call to foo() is properly linked to its definition.
  1. **Address Binding**
     1. **Relocation**:
        1. Object files contain code and data at addresses that are placeholders or relative to their own modules. The linker adjusts these addresses to fit the final memory layout.
        2. **Relocation Entries**: Each object file includes relocation entries that specify where in the code or data the addresses need to be updated.
     2. **Memory Layout**: The linker determines the final memory layout of the executable, assigning absolute addresses to all functions and variables.
  2. **Combining Object Files**
     1. **Merging Sections**:
        1. **Code Sections**: .text sections from multiple object files are combined into a single code section in the executable.
        2. **Data Sections**: .data and .bss sections are merged, with .data holding initialized data and .bss holding uninitialized data.
     2. **Handling Duplicate Symbols**:

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.

* 1. **Library Linking**
     1. **Static Libraries**:
        1. **Library Archives**: Static libraries are archives of object files. The linker extracts and includes only the object files needed for the final executable.
        2. **Link-Time Optimization**: The linker may perform optimizations like eliminating unused functions or data.
     2. **Dynamic Libraries**:

1. **Shared Libraries**: Dynamic libraries (e.g., .dll on Windows, .so on Linux) are not included in the executable. Instead, the executable contains references to these libraries.
2. **Dynamic Linking at Runtime**: The dynamic linker/loader (part of the OS) loads the shared libraries into memory and resolves symbols when the program is executed.
   1. **Generating the Executable**
      1. **Executable File Format**: The final output is an executable file with a specific format (e.g., ELF on Linux, PE on Windows). This format includes headers and sections for code, data, and metadata.
      2. **Executable Headers**:
         1. **Program Header**: Contains information about how to load the executable into memory.
         2. **Section Headers**: Describe the sections in the executable (e.g., code, data).
      * **Output**: The result is an executable file, often with an extension like .out, .exe, or no extension, depending on the operating system.
   * **Execution:** The loader, part of the operating system, loads the executable into memory and prepares it for execution.
     + **Loading**: It reads the executable file from disk into memory.
     + **Dynamic Linking**: If the executable depends on shared libraries (dynamic linking), the loader resolves these dependencies at runtime.
     + **Output**: The program is now in memory and ready to be executed.

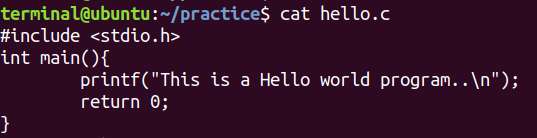
# Hands On Example of C Compilation Process

Let’s get practical overview of the above discussed C Compilation phases. First, you need to write your C source code using some text editor. For instance, Vim, nano, or gedit are the most commonly used editors in Linux.

* You can install vim editor using following command:

sudo apt-get install vim

* Here’s the snapshot of a hello world program written in C language that simply prints a hello world msg on screen.



* The next step is to run the discussed four phases one by one on this hello world program to generate an executable. For that purpose, we are going to use the GCC compiler. The GNU Compiler Collection (GCC) 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:

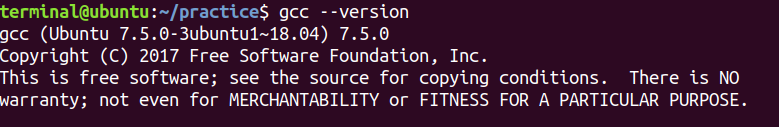
sudo apt update

sudo apt install gcc

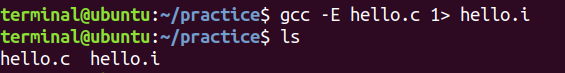
* If you also need the GNU C++ compiler and other development tools, you might want to install the build-essential package, which includes GCC, G++, and other necessary tools:

sudo apt install build-essential

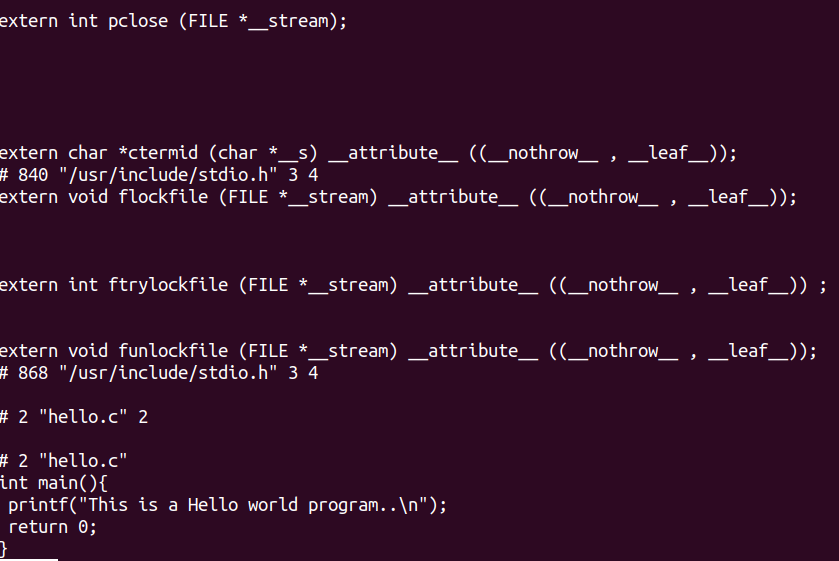
* Once it is installed, you can verify installation using following command:



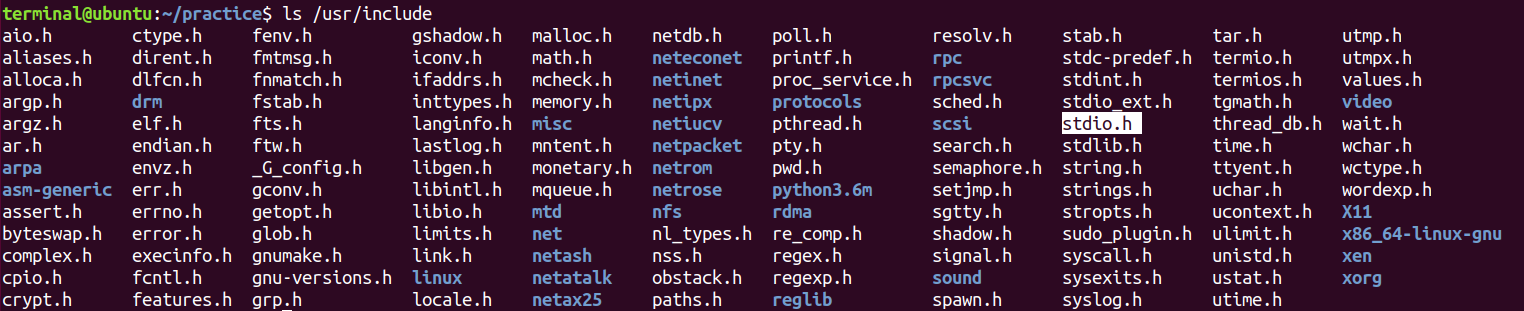
* **Preprocessing**: Let’s start with the preprocessing phase which interprets the preprocessor directives as it includes the header files, remove the comments and expand the macros.
  + Using –E option we can instruct the gcc to just perform preprocessing and then redirect the output to hello.i file otherwise it will print the preprocessed code to the standard output or screen.



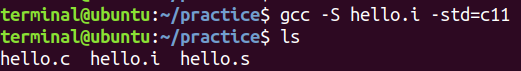
* We can use the less or cat command to view the contents of the file. You can see lots of information from different header files that have been included in it. And at the very end you can see the actual lines of code you've written.



* For information purpose, you can check the C header files in /usr/include directory as the C preprocessor searches for the header file in this directory. This is the standard location for C header files.

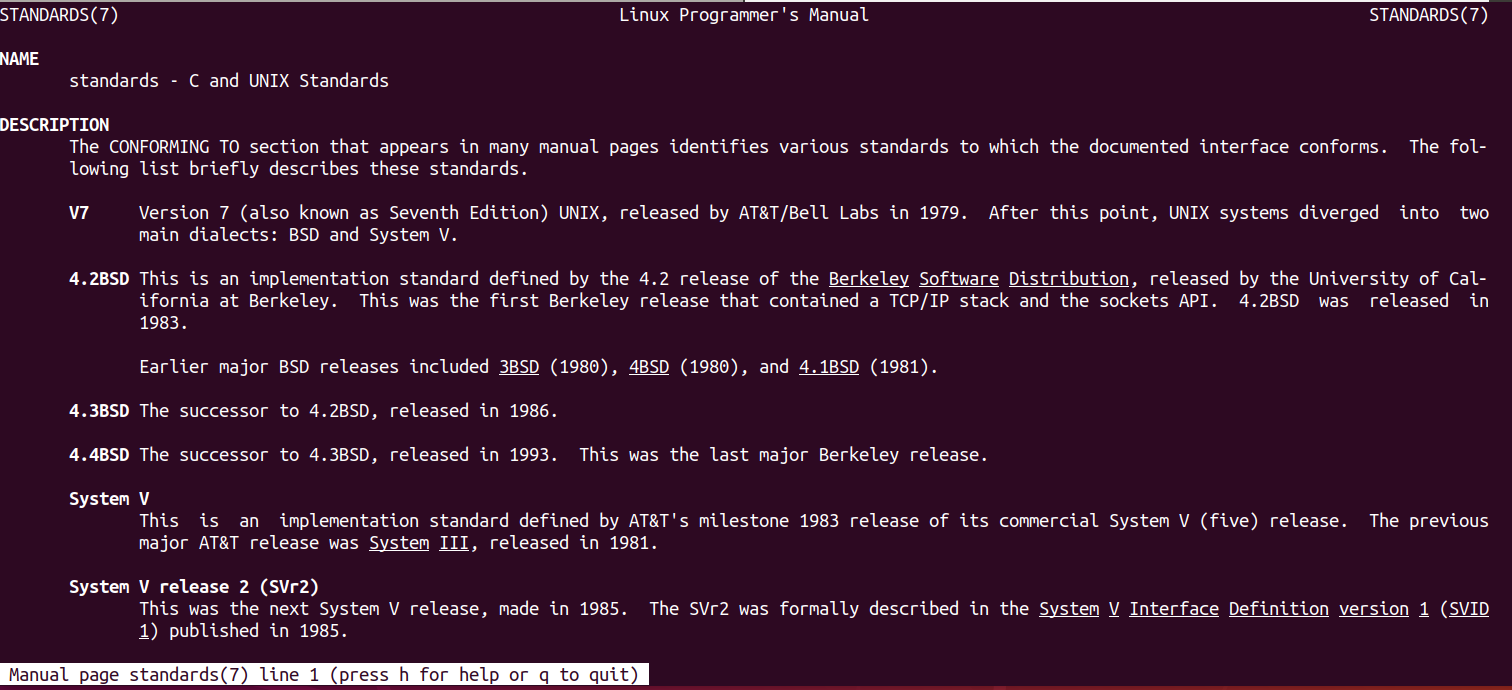


* **Compilation**: The next step is compilation that checks for syntax errors and generates an ascii file containing assembly code. This phase takes the preprocessed file as input which is hello.i in our case and generates the assembly code for underlying architecture and in our case, it is x64 and the generated output file is hello.s.
  + Using –S option you can instruct gcc to perform just compilation rather than performing all the steps at once. You can also instruct gcc to follow specific C laguage standard while compiling the code for example using –std option and passing the C standard of your choice such as –std=c11.

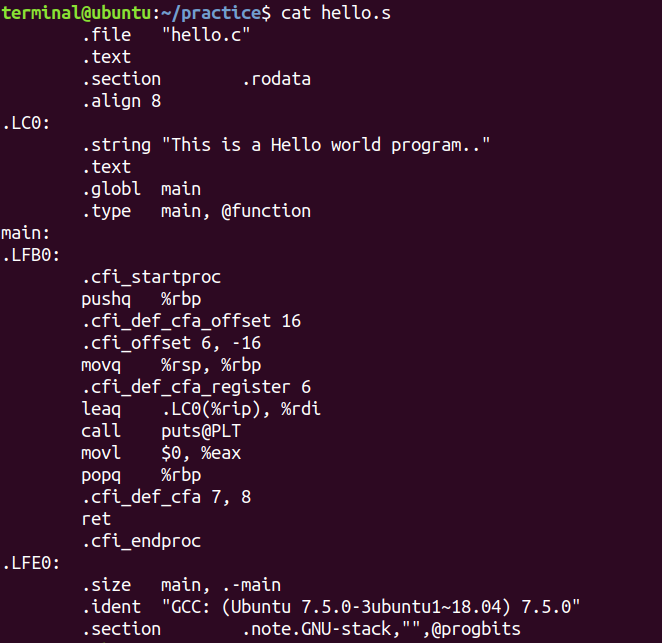


* + You can view the man page to check for standards using following command:

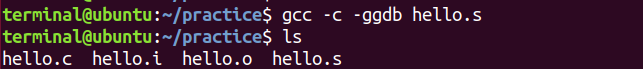
man 7 standards



* You can also view the assembly code from hello.s file:



* **Assembling**: The third one is the assembling phase where assembly code is converted into the machine dependent opcode.
  + Using the –c option you can instruct gcc to perform assembling only. It takes hello.s file as input and generates hello.o file as output. You can also include debugging symbols using –ggdb option.



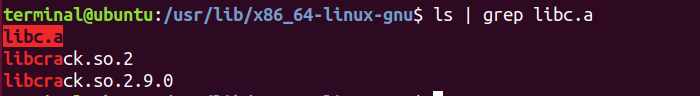
* The hello.o file is a relocatable object file because the code section and data section of this file will be merged to the code section and data section of other object files in the linking phase. This object file with many other things also contains:
  + Symbol table that contains name, type and relative addresses of global and static variables in the source file if any.
  + Name and relative addresses of functions defined in the source file.
  + Name of the library functions used in source file, whose addresses will be updated in the linking phase. For instance, in our case the symbol table of hello.c file contains only the name of the printf function and not the address of the function, which we will get during linking phase.
* **Linking**: The last step is the linking phase. In simple words, a linker actually links or merges the multiple .o files or relocatable object files and creats a single executable.
  + For example, in our case we have a single .o file which needs to be merge with another .o file and that is the printf.o file. The code of printf function resides in the standard C library. You can get the information of standard C library from the man page as:

man 7 libc

* + Standard C library and other libraries resides in a standard location which is /usr/lib/x86\_64-linux-gnu/ location. We can get lots of libraries at this location. We can cd to this directory and look for standard C library using following command:



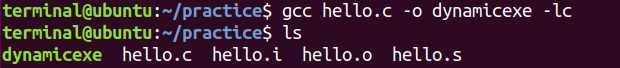
* + These libraries are pre-compiled set of functions and are ready to use by our programs. These libraries come into two flavors such as static library and dynamic library. The above shown libc.so is dynamic library and .so represents shared object, while libc.a, where. a represents the archive is the static version.



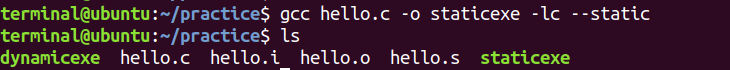
* + We can also check whether printf.o resides in this library using follwoing command and can check that it actually exists there:



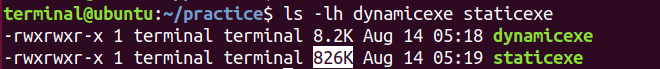
* + Now we need to link printf.o from libc library to with our hello.o file. We can perform both static linking as well as dynamic linking and can check the difference between them.
  + You can instruct the gcc to perform linking using –lc option which tells gcc to link the source file with libc library and generate an executable. 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. One more point is that gcc will look for the libc.so by default, if it exists it will link the source code with libc.so otherwise it will look for libc.a. Thus, here’s the command that performs dynamic linking with libc and outputs an exe file with name dynamicexe.



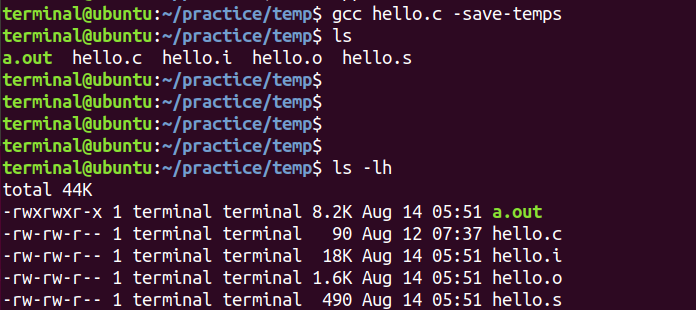
* + You can use --static option to perform static linking with libc.



* + To see the difference between static and dynamic linking, first check 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 static exe is much larger. This executable can even run on those machines where libc doesn’t exist. However, in case of dynamic exe the final executable doesn’t contains the code of external function (only contains a stub), that is why it is small in size, and the executable requires the libc to run on that system.



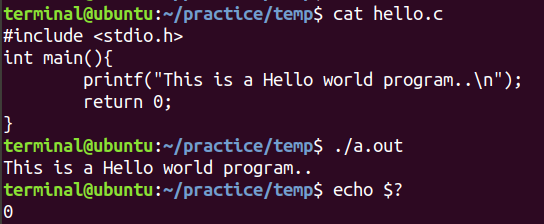
* The hello.i, hello.s and hello.o files that we have been generated one by one in separated steps can be created at once using –save-temps option. Using this option, we can generate all the temporary files as well as the final executable a.out already linked with libc library.



* **Return Value**: As we have created the final executable let’s execute and check its output.



* + When we execute a program, it returns some value to its parent which is shell in our case. 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 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. The specific non-zero value can often provide information about the type of error.
  + After running a program, you can check its return value using the special shell environment variable $? in Linux.



* + 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.
* **Optimization Flag:** Optimization flags instruct the compiler on how to optimize the code it generates. The goal of optimization is typically to make the code run faster, consume less memory, or use fewer resources. Different optimization levels and specific flags enable different types of optimizations.
  + Optimization flags in the context of compiling code with GCC (GNU Compiler Collection) are options you pass to the compiler to control how it optimizes the generated code. These flags can significantly impact the performance, size, and debugging capabilities of the resulting executable. GCC provides several standard optimization levels that can be set using the -O flag:
    - **No optimization**: Using –O0 this level is used for debugging and ensures that the code is compiled as closely as possible to the source code. It may result in larger and slower code

gcc hello.c -o myprog -O0

* + - **Basic optimization:** This level enables a reasonable set of optimizations that do not significantly increase compilation time.

gcc hello.c -o myprog -O1

* + - **Moderate optimization:** This level enables more optimizations than -O1 and aims to improve performance without significantly increasing compilation time or code size.

gcc hello.c -o myprog -O2

* + - **Aggressive optimization:** This level enables all the optimizations from -O2 and includes additional optimizations that may improve performance further but could also increase compilation time and code size.

gcc hello.c -o myprog -O3

* + - **-Ofast**: Optimizations that are not strictly standard-compliant but aim to provide the fastest possible code. This may include disabling some standard compliance features.

gcc hello.c -o myprog -Ofast

* + - **Optimize for size:** This level optimizes the code to reduce its size rather than its execution speed.

gcc hello.c -o myprog -Os

* + - **Optimize for debugging:** This level provides a balance between optimization and debugging, making debugging easier than higher optimization levels but with some optimization applied.

gcc hello.c -o myprog –Og

* **Creating Binaries for Different Architectures:** To create binaries for different architectures on Linux, such as 32-bit and 64-bit, you need to use appropriate flags with the compiler (gcc for GNU Compiler Collection) and sometimes specify the target architecture explicitly.
  + **Creating 32-bit Binaries:** To compile a 32-bit binary on a 64-bit Linux system, you need to use the -m32 flag with gcc. This flag tells the compiler to generate code for a 32-bit target.

gcc hello.c -o myprog –m32

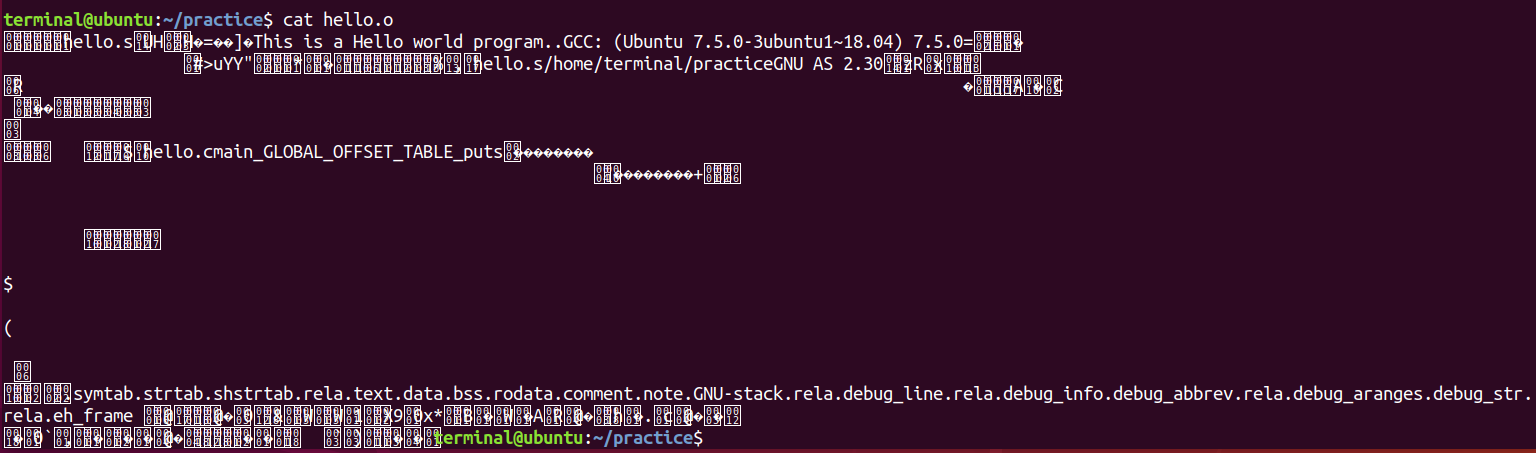
* + You may need to have the 32-bit libraries and development tools installed on your system. On Debian-based systems like Ubuntu, you can install the necessary packages using:

sudo apt-get install gcc-multilib

* + 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.

# Reading/Viewing Contents of Object Files

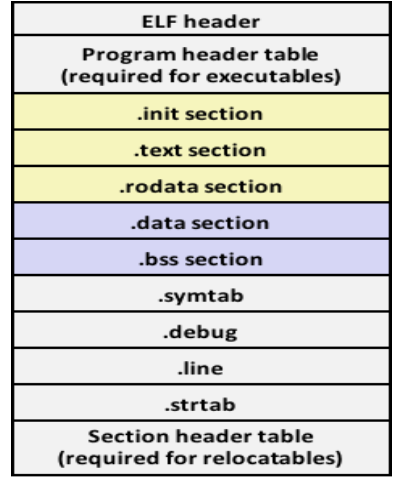
We have covered all four phases of the C Compilation process, and generated the preprocessed file, assembly file, object file and final executable. As we have already read the contents of preprocessed file with .i extension and the assembly file with .s extension. But we haven’t checked the contents of relocatable object file and final executable. So, let’s try to view the content of these files using cat or less commands:



You can see that we can’t view the contents using cat utility. Now let’s try to check the content of final exe.

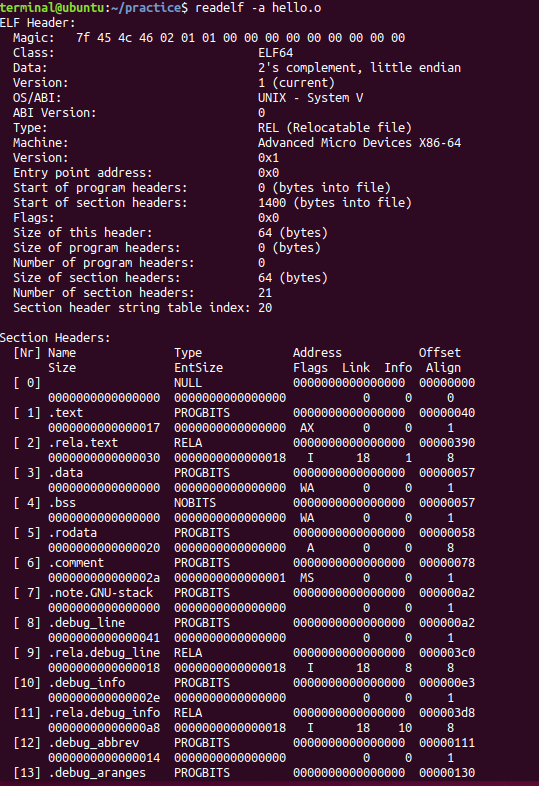


* **Types of Object Files:** In Linux, an object file is a file generated by a compiler or assembler that contains machine code, data, and metadata, but is not yet a complete executable or library. Object files are intermediate files that are linked together to produce final executables or shared libraries. They are crucial in the software build process, allowing modular development and incremental compilation. Object files can be classified based on their formats and usage:
  + **Relocatable object file**: (***.o file***) Contains binary code and data in a form that can be combined with other relocatable object files at compile time to create an executable object file. Each .o file is produced from exactly one .c file. Compilers and assemblers generate relocatable object files.
  + **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.
* **Formats of Object Files:** Object file formats vary from system to system. Some famous formats are mentioned below:
  + a.out: Original file format for UNIX. It consists of three sections: text, data, and bss, which are for program code, initialized data and uninitialized data respectively.
  + COFF: Common Object File Format was introduced with SVR3 Unix. COFF files may have multiple sections, each prefixed by a header. The number of sections is limited. The COFF specification includes support for debugging but the debugging info was limited. Later ECOFF was introduced by MIPS and XCOFF by IBM.
  + ELF: Executable and Linking Format came with SVR4 UNIX. ELF is similar to COFF in being organized into a number of sections, but it removes many of COFF's limitations. ELF is used on most modern UNIX systems, including GNU/Linux, Solaris and Irix. Also used on many embedded systems.
  + PE: Portable Executable format is used by Windows for their executables. PE is basically COFF with additional headers. The extension normally is .exe.
* **ELF Format:** Executable and Linking Format is binary format, which is used in SVR4 Unix and Linux systems.
  + It is a format 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
  + The files that are represented in this formats are:
    - Relocatable file objects (.o)
    - Normal executable files (a.out)
    - Shared object files (.so)
    - Core files



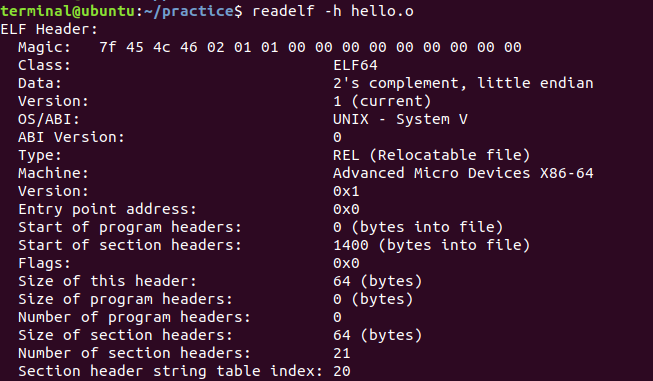
* **Inspecting Object Files using *readelf***: readelf is a command-line utility used to display detailed information about ELF (Executable and Linkable Format) files on Linux and other Unix-like operating systems. ELF is the standard file format for executables, object code, shared libraries, and core dumps in Linux. readelf allows you to inspect various sections of ELF files, providing valuable insights for debugging, analysis, and development. You can view the man page of readelf utility to get more information.
  + **Display All Information:** To display all available information about an ELF file, including headers, sections, and symbols:

readelf –a hello.o



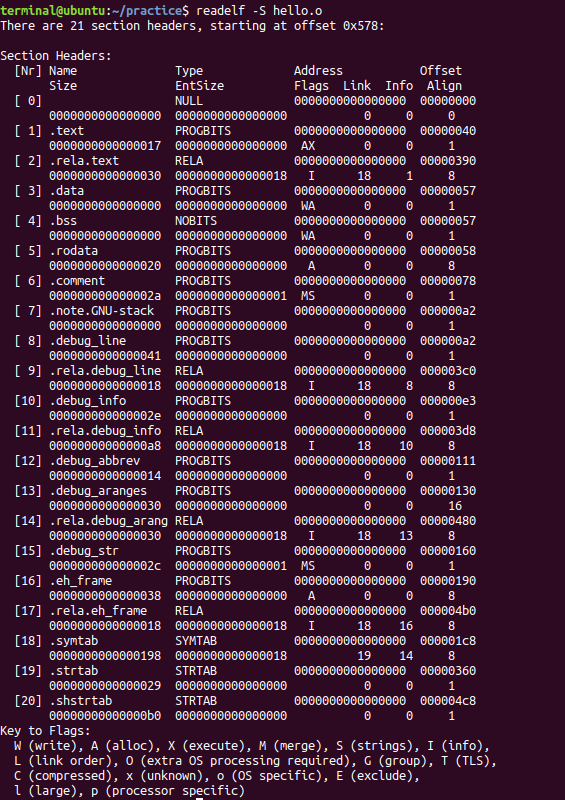
* + **Display ELF Header:** To show the ELF header, which includes information about the file type, architecture, entry point, and more:

readelf –h hello.o



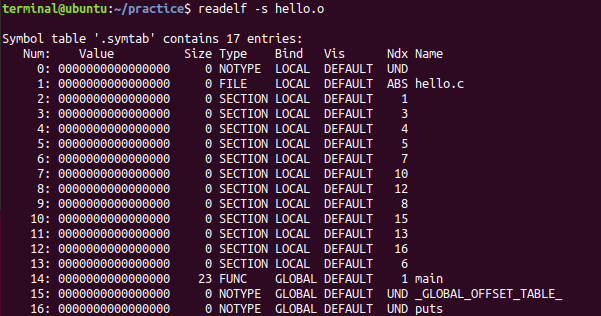
* + **Display Section Headers:** To list the section headers of the ELF file, which describe different sections like .text, .data, and .bss:

readelf –S hello.o



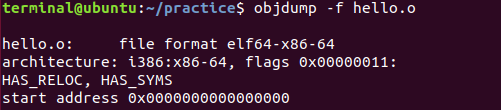
* + **Display Symbol Table:** To display the symbol table, which includes information about functions, variables, and other symbols:

readelf –s hello.o



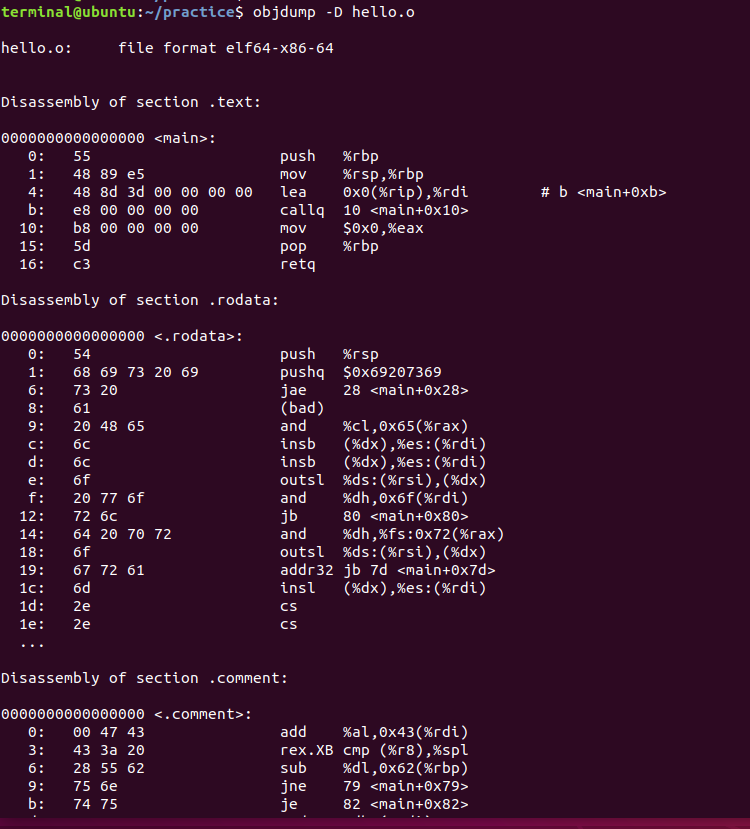
* **Inspecting Object Files using *objdump***: objdump is a powerful tool in Linux used to display information about object files, executable files, and libraries. It is part of the GNU Binutils package and provides detailed insights into the content and structure of binary files. This can be particularly useful for debugging, analyzing binaries, and understanding how code is translated into machine instructions.
  + **Display File Headers:** To display the header information of an object file or executable, including architecture, entry point, and section headers:

objdump –f hello.o



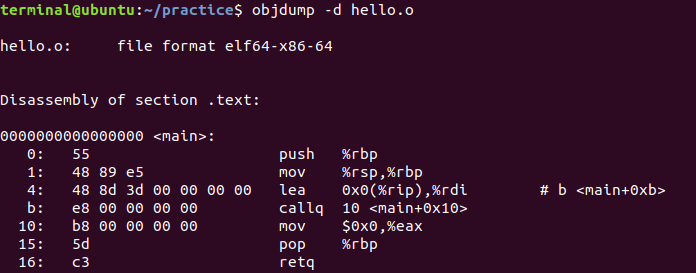
* + **Display Disassembly:** To disassemble the entire program and show the assembly instructions:

objdump –D hello.o



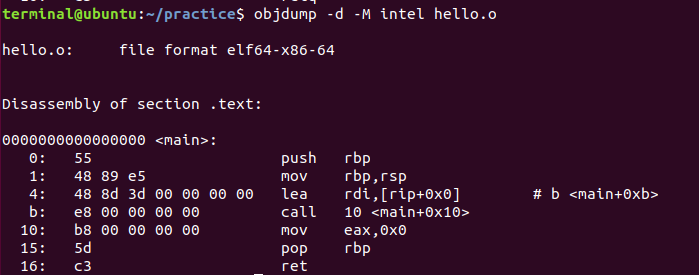
* + **Display Disassembly of Main only:**

objdump –d hello.o



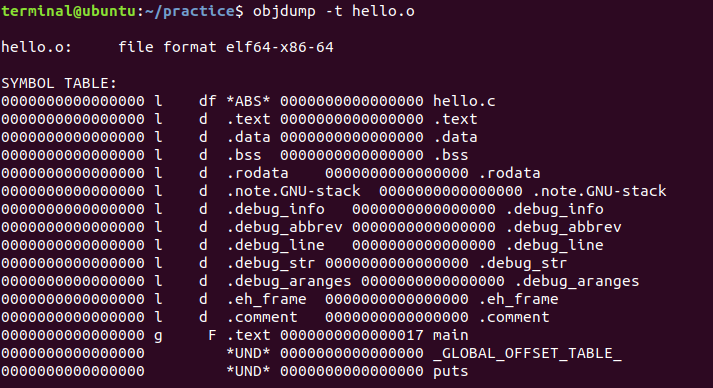
* + **Show Disassembly in Intel Format:** By default, objdump shows assembly in AT&T format. We can also view the assembly in intel format:

objdump –d –M intel hello.o



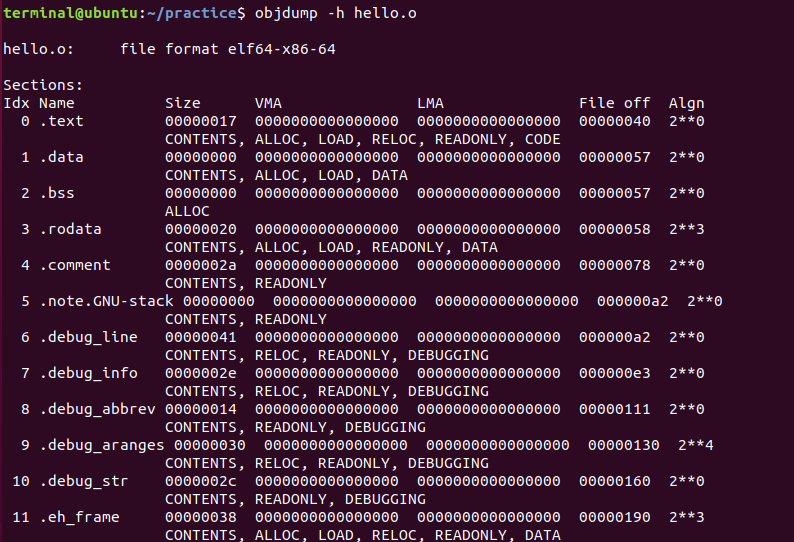
* + **Display Symbol Table:** To list the symbols in the object file, including functions, global variables, and more:

objdump –t hello.o



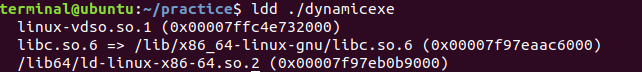
* + **Display Section Headers:** To display detailed information about the sections in the object file, such as their sizes and offsets:

objdump –h hello.o



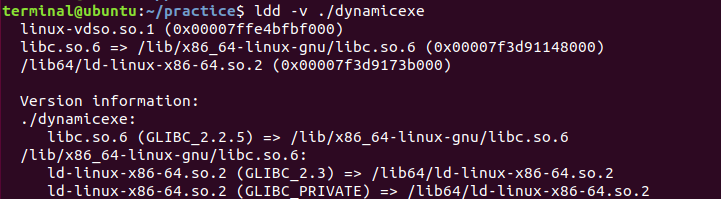
* **Inspecting Object Files using *ldd***: ldd is a command-line utility used on Linux and other Unix-like operating systems to display the shared library dependencies of an executable or shared library. It shows which dynamic libraries an executable or shared library relies on, helping you understand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

ldd ./dynamicexe



* + - 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.
  + **Display Library Dependencies with Verbose Output:** To get more detailed information about the libraries and their paths:

ldd –v ./dynamicexe



# C Code Vulnerabilities and Mitigations

A **vulnerability** is a weakness or flaw in a system that can be exploited by an attacker to compromise the system's integrity, availability, or confidentiality. Vulnerabilities can exist in various types of systems and can lead to unintended behaviors or security breaches. Understanding vulnerabilities is crucial for maintaining robust security across different domains.

1. **Abstract Concept of Vulnerability:** At an abstract level, a vulnerability can be described as:
   * **A Deficiency**: A gap or shortcoming in a system's defenses or design that can be exploited.
   * **An Opportunity for Exploitation**: A potential point of attack that, if discovered and exploited, can lead to unauthorized access, damage, or disruption.
   * **A Risk Factor**: An element that introduces potential risks or threats to the security and functionality of a system.
2. **Categories of Vulnerabilities**
   * **Hardware Vulnerabilities**
     + **Definition**: Weaknesses or flaws in physical components of a computer system or network devices.
     + **Examples**:
       - **Spectre and Meltdown**: Vulnerabilities in modern CPUs that can allow unauthorized access to sensitive data.
       - **Hardware Faults**: Physical defects or design issues that might lead to failures or unauthorized access.
     + **Location**: Microprocessors, memory modules, hardware devices.
   * **Operating System Vulnerabilities**
     + **Definition**: Weaknesses or flaws in the OS software that can be exploited to gain unauthorized access or control.
     + **Examples**:
       - **Privilege Escalation**: Flaws that allow users to gain higher privileges than intended.
       - **Buffer Overflow**: Errors that allow attackers to execute arbitrary code.
     + **Location**: Kernel, system libraries, user interfaces.
   * **Application Program Vulnerabilities**
     + **Definition**: Flaws or weaknesses in software applications that can be exploited to perform unauthorized actions.
     + **Examples**:
       - **SQL Injection**: Exploiting vulnerabilities in database queries.
       - **Cross-Site Scripting (XSS)**: Injecting malicious scripts into web pages.
       - **Code Injection**: Injecting malicious code into applications.
     + **Location**: Software code, libraries, web applications.
   * **Network Vulnerabilities**
     + **Definition**: Weaknesses in network protocols, configurations, or devices that can be exploited to disrupt or intercept communications.
     + **Examples**:
       - **Man-in-the-Middle (MitM) Attacks**: Intercepting and potentially altering communication between two parties.
       - **Denial of Service (DoS)**: Overloading a network service to make it unavailable.
       - **Unsecured Wireless Networks**: Exploiting weaknesses in wireless encryption or authentication.
     + **Location**: Network protocols, routers, switches, wireless access points.
   * **Human Factors**
     + **Definition**: Vulnerabilities arising from human behavior or decision-making that can compromise security.
     + **Examples**:
       - **Social Engineering**: Manipulating individuals into divulging confidential information.
       - **Weak Passwords**: Using easily guessable or reused passwords.
     + **Location**: User practices, security awareness training.
3. **Finding Vulnerabilities:** Vulnerabilities can be discovered through various methods:
   * **Code Analysis**: Examining source code for security flaws.
   * **Penetration Testing**: Simulating attacks to find exploitable weaknesses.
   * **Security Audits**: Comprehensive reviews of systems and practices.
   * **Vulnerability Scanning**: Using automated tools to detect known vulnerabilities.
   * **Configuration Reviews**: Checking system and network configurations for weaknesses.
   * **Research and Reporting**: Identifying and reporting vulnerabilities through research or bug bounty programs.
4. **Vulnerable C Language**: 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. Here’s a summary of common vulnerabilities associated with C programming and the reasons behind them:
   * **Buffer Overflow**
     + **Description**: Occurs when a program writes more data to a buffer than it can hold, causing adjacent memory locations to be overwritten.
     + **Why It Happens**: C does not automatically check the bounds of arrays or buffers. Functions like strcpy, sprintf, and gets can cause buffer overflows if the input data exceeds the allocated size.

* + **Use After Free**
    - **Description**: Happens when a program continues to use a pointer after the memory it points to has been freed.
    - **Why It Happens**: 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.
  + **Dangling Pointer**
    - **Description**: A pointer that continues to reference a memory location after the memory it points to has been deallocated.
    - **Why It Happens**: 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.
  + **Integer Overflow and Underflow**
    - **Description**: Occurs when arithmetic operations produce results that exceed the representable range of the data type.
    - **Why It Happens**: C does not check for integer overflows or underflows. If the result of an arithmetic operation is too large or too small, it wraps around.
  + **Format String Vulnerabilities**
    - **Description**: Arise when untrusted data is used as a format string in functions like printf.
    - **Why It Happens**: C's printf and related functions do not validate format strings. An attacker can use format specifiers to read or write arbitrary memory locations.
  + **Stack Smashing**
    - **Description**: A specific type of buffer overflow that overwrites the stack, potentially altering the return address.
    - **Why It Happens**: Similar to buffer overflow, but specifically targets the stack area of memory. This can lead to control flow hijacking.

1. **Secure and Unsecure Function in C:** 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.
   * **Buffer Operations:** 
     + - **Unsecure Function** strcpy(): 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.
       - **Secure Function** strncpy(): Copies a specified number of characters from the source to the destination buffer, ensuring that the buffer is not overflowed. It also null-terminates the string if the length is less than the specified number.
   * **String Formatting**
     + **Unsecure Function** printf()/sprintf(): A versatile function for outputting formatted data, but if used with untrusted format strings, it can lead to format string vulnerabilities.
     + **Secure Function** snprintf(): Formats and stores a string in a buffer, ensuring that the buffer size is respected to prevent overflow. It also ensures null termination.
   * **Input Reading**
     + **Unsecure Function** gets(): Reads a line from standard input into a buffer. It does not check the size of the buffer, leading to buffer overflows if the input is larger than the buffer.
     + **Secure Function** fgets(): Reads a specified number of characters from a file stream, including standard input, and ensures that the buffer is not overflowed. It also handles newline characters properly.
   * **Memory Management**
     + **Unsecure Function** malloc() and free(): Functions 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.
     + **Secure Functions**: Use of 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.
2. **Vulnerable C Program:**
   * Vulnerable C Program Using gets(): The gets() function is unsafe because it does not perform bounds checking, which can lead to buffer overflows.
     + gets(buffer) reads a line of input from standard input into buffer.
     + If the input exceeds 50 characters, it will overflow buffer, potentially overwriting adjacent memory.
   * **Safe Program Using** fgets()**:** The fgets() function is safer because it allows you to specify the maximum number of characters to read, which helps prevent buffer overflows.
     + 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.
   * **Vulnerable Program Using strcpy():** In this example, we use the strcpy() function with a hardcoded string that exceeds the size of the destination buffer, leading to a buffer overflow.
     + **buffer[10]**: The buffer is only 10 bytes in size.
     + **strcpy(buffer, long\_string)**: The long\_string is much longer than the buffer, leading to overflow.
   * **Safer Program Using strncpy():** In this version, we use strncpy() to safely copy the string into the buffer, ensuring we do not exceed the buffer's size.
     + **strncpy(buffer, long\_string, sizeof(buffer) - 1)**: Copies up to sizeof(buffer) - 1 characters from long\_string into buffer, preventing overflow.
     + **buffer[sizeof(buffer) - 1] = '\0'**: Ensures that the buffer is null-terminated even if the source string is longer than the buffer.
   * **Vulnerable Program Using strcat():** The strcat() function is used to concatenate (append) one string to another. It does not check the size of the destination buffer, which can lead to buffer overflows if the buffer is not large enough to hold the concatenated result.
     + **buffer[20]**: The buffer is only 20 bytes in size.
     + **strcat(buffer, to\_append)**: The to\_append string is too long to fit into the buffer after the initial "Hello, ", leading to buffer overflow.
   * **Safer Example Using strncat():** The strncat() function allows you to specify the maximum number of characters to append, which helps prevent buffer overflows by ensuring you don't exceed the size of the destination buffer.
     + **strncat(buffer, to\_append, sizeof(buffer) - strlen(buffer) - 1)**: 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.
     + **sizeof(buffer) - strlen(buffer) - 1**: Ensures that the buffer has enough space for the null terminator and does not overflow.

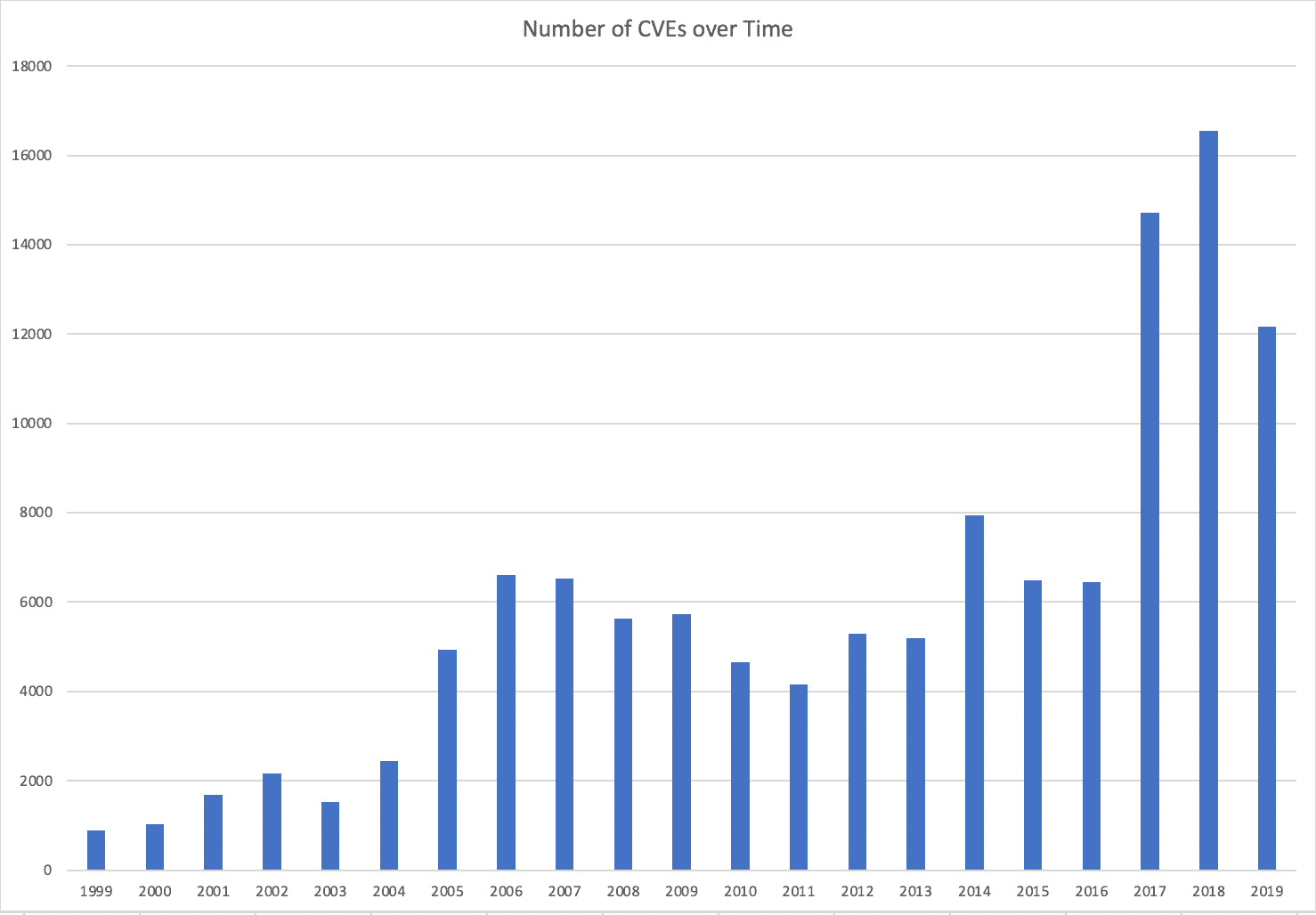
# Common Vulnerabilities and Exposures

**Common Vulnerabilities and Exposures (CVE)** is a database of publicly disclosed information security vulnerabilities and exposures. CVE provides a standardized method for identifying and discussing vulnerabilities in software and hardware. A CVE number uniquely identifies one vulnerability from the list. CVE provides a convenient, reliable way for vendors, enterprises, academics, and all other interested parties to exchange information about cyber security issues. Enterprises typically use CVE, and corresponding CVSS scores, for planning and prioritization in their vulnerability management programs.

First launched in 1999, CVE is managed and maintained by the National Cybersecurity FFRDC (Federally Funded Research and Development Center), operated by the MITRE Corporation. CVE is sponsored by the US Federal Government, with both the US Department of Homeland Security (DHS) and the Cybersecurity and Infrastructure Security Agency (CISA) contributing operating funds. CVE is publicly available and free for anyone to use. [CVE Database](https://nvd.nist.gov/)

Here’s a detailed overview of CVE and its role in information security:

* **CVE Background:** Before CVE was started in 1999, it was very difficult to share data on vulnerabilities across different databases and tools. Each vendor maintained their own database, with their own identification system and different sets of attributes for each vulnerability. CVE ensures that every tool can exchange data with other tools, while also providing a mechanism by which different tools, such as vulnerability scanners, can be compared.
  + While some may question whether publicly disclosing vulnerabilities makes it easier for hackers to exploit those vulnerabilities, it is generally accepted that the benefits outweigh the risks. CVE includes only publicly known security exposures and vulnerabilities. This means that hackers could get their hands on data related to the CVE whether it is in the CVE list or not. Additionally, details of a CVE are often withheld from the vulnerability list until the corresponding vendor can issue a patch or other fix, ensuring that enterprises can protect themselves once the information is made public. Additionally, information sharing across the cybersecurity industry can help speed mitigations, as well as ensure that all organizations are protected more quickly than if left to identify and find resolutions to CVEs on their own.
* **Purpose of CVE:** The primary purpose of CVE is to provide a uniform identifier for publicly known vulnerabilities and exposures. By standardizing the identification of these issues, CVE helps:
  + **Facilitate Communication**: CVE identifiers (CVE IDs) allow security professionals, vendors, and researchers to discuss and reference vulnerabilities consistently.
  + **Support Integration**: Security tools and services can use CVE IDs to integrate and share information about vulnerabilities.
  + **Enhance Search and Reporting**: CVE IDs make it easier to search for, report on, and track vulnerabilities across different systems and databases.
* **Structure of CVE Entries**
  + **CVE ID**: A unique identifier for the vulnerability, formatted as CVE-YYYY-NNNNN. For example, CVE-2024-12345.
  + **Description**: A brief description of the vulnerability or exposure, including its impact and affected systems or software.
  + **References**: Links to additional information, such as advisories, patches, and reports. These may include vendor-specific details, security bulletins, and technical documentation.
* **Types of CVE Entries:** CVE entries fall into two main categories:
  + **Vulnerabilities**: Security flaws in software or hardware that can be exploited to compromise systems. Examples include buffer overflows, SQL injection flaws, and cross-site scripting (XSS) vulnerabilities.
  + **Exposures**: Weaknesses that could potentially be exploited but are not necessarily a direct security threat. Exposures are usually less critical but still relevant for security assessments.
* **CVE Process:** The process for a CVE entry typically involves several steps:
  + **Discovery**: A vulnerability or exposure is discovered by a security researcher, vendor, or another entity.
  + **Reporting**: The issue is reported to a CVE Numbering Authority (CNA) or directly to MITRE. CNAs are organizations authorized to assign CVE IDs and manage CVE entries for their respective domains.
  + **Review**: The reported issue is reviewed for accuracy and completeness. This may involve verifying the details and coordinating with the original reporter or affected vendor.
  + **Publication**: Once validated, the CVE entry is published in the CVE database and made publicly available.
* **CVE Database Access**
  + **CVE Website**: The official CVE website (cve.mitre.org) provides access to the CVE database, search functionality, and information on new and past CVE entries.
  + **Data Feeds and APIs**: CVE data can be accessed through feeds and APIs for integration into security tools, vulnerability management systems, and threat intelligence platforms.
  + **CVE Search**: Users can search for specific CVE IDs or keywords to find details about vulnerabilities and exposures.
* **Role in Security Ecosystem**
  + **Vulnerability Management**: CVE helps organizations track and manage vulnerabilities in their systems, apply patches, and implement mitigations.
  + **Incident Response**: Security teams use CVE entries to understand and respond to security incidents, determining which vulnerabilities may have been exploited.
  + **Threat Intelligence**: CVE data is used in threat intelligence to analyze emerging threats and assess risk.
  + **Compliance**: CVE helps organizations meet compliance requirements by providing a standardized way to reference and address known vulnerabilities.
* **Integration with Other Databases:** CVE is often integrated with other vulnerability databases and security resources, such as:
  + **National Vulnerability Database (NVD)**: Maintained by NIST, the NVD provides detailed information and metrics (e.g., CVSS scores) for CVE entries.
  + **Security Advisories**: Many vendors and security organizations reference CVE IDs in their advisories and security bulletins.
  + **Threat Intelligence Platforms**: CVE data is used in various threat intelligence platforms to enrich security data and improve threat detection.
* **CVE v/s CWE:** **CVE (Common Vulnerabilities and Exposures)** and **CWE (Common Weakness Enumeration)** are two distinct, but complementary frameworks used in the field of cybersecurity. They address different aspects of vulnerabilities and weaknesses in software systems. Here's a detailed comparison:
  + **CVE (Common Vulnerabilities and Exposures)**
    - **Purpose**: CVE provides a reference method for publicly known information-security vulnerabilities and exposures. It is a catalog of known vulnerabilities in software and hardware.
    - **Focus**: CVE focuses on specific, identifiable vulnerabilities or exposures in software and systems that have been publicly disclosed or are known.
    - **Structure**: Each CVE entry is a unique identifier assigned to a specific vulnerability or exposure. CVE entries include details such as a brief description of the vulnerability, affected products, and sometimes links to additional information or fixes.
    - **Example**: CVE-2024-12345
      * **Description**: A buffer overflow in a specific version of a software product that allows attackers to execute arbitrary code.
    - **Usage**: Security professionals, researchers, and organizations use CVE identifiers to track, manage, and remediate known vulnerabilities. CVEs help in communicating about vulnerabilities in a standardized way and in coordinating responses to vulnerabilities.
  + **CWE (Common Weakness Enumeration)**
    - **Purpose**: CWE is a community-developed list of software and hardware weaknesses. It provides a comprehensive list of common coding and design mistakes that can lead to vulnerabilities.
    - **Focus**: CWE focuses on the underlying weaknesses or flaws in software development processes and design that can lead to security vulnerabilities. It is more about the root causes rather than specific instances.
    - **Structure**: CWE entries describe types of weaknesses, providing detailed descriptions, examples, and suggestions for prevention or mitigation. Each weakness has a unique identifier and is categorized into various types and families.
    - **Example**: CWE-119
      * **Description**: Improper Restriction of Operations within the Bounds of a Memory Buffer, which can lead to buffer overflow vulnerabilities.
    - **Usage**: CWE helps developers, security professionals, and organizations understand and address the root causes of vulnerabilities. It provides guidance on best practices and helps in improving software development processes and security practices.
  + **Key Differences**
    - **Scope and Focus**:
      * **CVE**: Focuses on specific vulnerabilities that have been discovered and reported in software and hardware.
      * **CWE**: Focuses on general weaknesses in software development practices that could lead to vulnerabilities.
    - **Application**:
      * **CVE**: Used for tracking and managing specific vulnerabilities that have been identified and disclosed.
      * **CWE**: Used for understanding and mitigating the underlying weaknesses in code and design practices.
    - **Identification**:
      * **CVE**: Provides unique identifiers for individual vulnerabilities.
      * **CWE**: Provides identifiers for general types of weaknesses that can lead to vulnerabilities.
  + **Relationship Between CVE and CWE**
    - **Complementary Use**: CVE and CWE often work together in the context of vulnerability management. For instance, a specific CVE entry might be linked to one or more CWE entries to help understand the root cause of the vulnerability.
    - **Example**: A CVE entry for a buffer overflow vulnerability (CVE-2024-12345) might reference CWE-119 (Improper Restriction of Operations within the Bounds of a Memory Buffer) to explain the underlying weakness that led to the vulnerability.
* **How Many CVEs Are There?**
  + There are thousands of new CVEs every year. Since the CVE program was started in 1999, over 130,000 CVE Identifiers have been issued. Over the last few years, there have been 12,000-15,000 new CVEs annually.
  + Large software vendors with many products represent a large portion of the reported CVEs. Microsoft and Oracle, for example, each have over 6000 reported CVEs across their many product lines. In fact, the top 50 software vendors represent more than half of all CVEs issued since the inception of the CVE program.



<https://www.balbix.com/insights/what-is-a-cve/>