CYBR 437: Secure Coding

C Review-Part III

Instructor: Abinash Borah

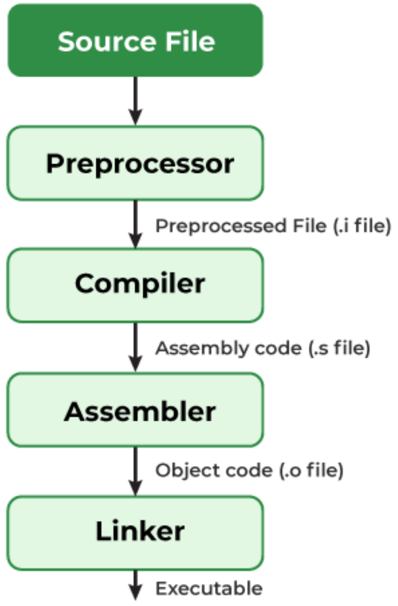
Outline

- C Compilation Process
- Storage Classes in C
- Type Qualifiers in C

Compilation Process

- Compilation is the process of converting source code into machine code
- There are four phases in the compilation process of a C program

    1. Preprocessing
    2. Compilation
    3. Assembling
    4. Linking



- We need to understand each process because, for Secure Coding, understanding where the potential threats can occur is important to determine the best mitigation techniques

Preprocessing

The preprocessor does the following:

- Removes Comments – strips out all comments throughout the code
- Handles Line Breaks
  - The \ allows a long line of code to be broken into multiple lines

```
printf("This is going to be a really long line"\
        " that needs to be broken into multiple lines\n");
```

- Includes the Header File code – copies the code from the header files
- Expands Macros (e.g., `#define MIN(a, b) ((a) < (b) ? (a) : (b))` – replaces all the macros with their values

Preprocessing (contd.)

- Conditional compilation – includes and excludes parts of the program based on various conditionals (e.g., `#ifdef`, `#ifndef`, `#if`, `#elif`, `#else`, `#endif`, etc.)
- Line Control – informs the C compiler of the location of each token in the source code
- The result of preprocessing is a .i file – known as a preprocessed file or a translation unit
- -E option with the gcc command is used to stop the compilation process after preprocessing: `gcc -E test.c -o test.i`
- The compiler does not run in this case

Compilation

- The compiler takes the preprocessor output file, checks the syntax, and converts the code to assembly language
- The output file is a .s file (can also be .asm)
- The -S option informs the compiler to stop after compiling

```
gcc -S test.i -o test.s
```

- The gcc compiler delegates the compilation steps (preprocessing, assembling, and linking) to cpp (preprocessor), as (assembler), and ld (linker)
- It only performs the compilation step

Assembling

- The assembler converts assembly code into machine code, a .o object file
- This step requires an input file of type .s (gcc specifically requires it)

```
gcc -c test.s -o test.o
```

- For other compilers, it can be .s or .asm file
- Each assembly instruction represents a single machine code instruction
- Each instruction contains the opcode, memory addresses of variables, numerical values, etc.
- The assembly machine code is highly platform-specific and typically not compatible with CPUs from different manufacturers

Linking

- The linker merges multiple C files (.o code) into a single .o file
- It links the code from the C standard libraries into the executable code
- There are two types of libraries:
- **Static libraries** are known as archives (.a files in Unix/Linux, .lib files in Windows)
    - Static libraries contain code that is linked in the final executable
    - Linking these libraries in the final executable creates a larger executable file
- **Dynamic libraries** are known as shared object libraries (.so in Unix/Linux, .dll in Windows)
    - Dynamic libraries contain code that is not linked into the final executable – code is only linked during execution
    - This allows for a smaller executable file and allows easy maintenance

Linking (contd.)

- For libraries outside the C standard library, we need to link them manually
- Example: to use the math library in C, we need to specify as follows during compilation:

```
gcc test.c -o test -lm
```

- The executable file is known as an Executable and Linkable Format (ELF) file

Storage Classes in C

- Storage classes are used to describe the features of a variable declaration and/or a function declaration
- Storage classes include information about the visibility, the scope of a variable or the scope of the function, and their lifetime
- There are four commonly used storage classes:
    - auto
    - extern
    - static
    - register

The Auto Storage Class

- The auto storage class is the default storage class
- When we declare a variable such as int x = 0; behind the scenes, the declaration is auto int x = 0;
- This declaration applies to all variables declared within a function or a block
- The auto storage class is rarely used explicitly
- When declared, auto variables are assigned a garbage value unless explicitly initialized
- Auto variables can only be accessed within the block/function in which they are declared and not outside of that block/function
- Auto variables within nested blocks are visible to the parent block where they are declared

The Extern Storage Class

- The extern storage class tells the compiler that the variable is not defined within the same block where it is used
- An extern variable is meant to be used elsewhere, typically outside the file in which it was declared in
- An extern variable can be accessed within any function/block
- Placing ‘extern’ before a declaration of a variable in a block/function signifies that the variable references the global variable only
- All extern variables can be accessed between many different files that are within the program

The Static Storage Class

- Static variables have the property of preserving their value even after they are out of the block/function scope
- Static variables are initialized only once and exist till the termination of the program
- If a static variable is defined within a function, its scope is local to the function where it is defined
- Global static variables can be accessed anywhere in the program
- If a static variable is not initialized, the compiler sets the value to zero and stores the variable in the bss section
- If a static variable is initialized, the compiler stores the variable in the initialized data section

The Register Storage Class

- The register storage class declares register variables that have the same functionality as auto variables
- If available, a register variable is stored in the processor register
- If no register is available, the variable is stored in the data segment
- During runtime, register variables are accessed much faster than variables stored in the data segment
- The address of register variables cannot be obtained using pointers

Summary of Storage Classes

Storage Specifier	Storage	Initial Value	Scope	Life
auto	stack	garbage	Within block	End of block
extern	data segment	zero	global	End of program
static (uninitialized)	bss	zero	Within block	End of program
static (initialized)	initialized data	initialized value	Within block	End of program
register	processor	zero	Within block	End of block

Type Qualifiers in C

- A type qualifier is used to refine the declaration of a variable, a function, and the parameters of the function
- The qualifier specifies if the value of the variable can be changed
- The value of a variable must always be read from memory, not a register
- There are many qualifiers – the ones primarily used are:
  - const
  - volatile
  - signed
  - unsigned

const Qualifier

- The const qualifier tells the compiler that the value of an object cannot be modified after its initialization by the program
- Any attempt to change a const-qualified object will result in a compile-time error
- Helps prevent accidental data alteration and allows the compiler to perform optimizations, such as placing the variable in read-only memory

volatile Qualifier

- The volatile qualifier tells the compiler that the value of the variable may change at any time outside the normal program flow (e.g., hardware interactions or interrupt service routines)
- The volatile qualifier notifies the processor to fetch the current value before use
- In practice, only three types of variables could change outside the normal program flow :
  - Memory-mapped peripheral registers
  - Global variables modified by an interrupt service routine
  - Global variables accessed by multiple tasks in a multi-threaded application

signed and unsigned Qualifiers

- The signed or unsigned qualifier tells the compiler how to process the sign bit of a variable
- The keyword signed is optional – any variable declared is signed by default
- The unsigned qualifier informs the compiler that the variable can only hold zero or positive values
- For variables declared as unsigned, all the bits are used to represent the magnitude of a variable