CS-501

Introduction to Malware, Threat Hunting
& Offensive Capabilities Development

Lecture 2:
C, C++ and Static Analysis

Agenda for Today

- C/C++ on Windows
- Ghidra
- WinApi
- GCC, Mingw, Zig/clang
- Assembly, x86/x64
- Calling Conventions

Overview

- C++ is a statically typed, compiled language that is very similar to C
- A large portion of malware you will encounter is written in C/C++
- C/C++ has no Garbage collector:
  - This Forces the developer to manage their own memory
- While it is general purpose, we will use it as a systems programing language
- C++ has many different versions, many different compilers and tools to help with debugging
- Reflection isn’t supported by default in C++
  - There is no such thing as type() like there is in python

Difference between C/C++

C++ is a (near) superclass of C. The standard library has extra features that can make using C++ easier and safer. This can come with a performance penalty.

- C++ has support for classes and structs. C only supports structs.
- C++ supports encapsulation
- C++ has namespaces
- While you can directly call malloc/HeapAlloc in C++, it has new/delete and “smart” pointers that simplify managing memory
  - You are welcome to use C as opposed to C++

Modern C++

- Supports functional programing in addition to OOP,auto/decltype,Variadic templates and more!
- We will only cover a small portion of these features but you are encouraged to use the standard library to its fullest.
- C++ is considerably more user friendly than it was when it was first released and with the release of C++20, programining

C/C++ and Memory Safety

Microsoft: 70 percent of all security bugs are memory safety issues

Percentage of memory safety issues has been hovering at 70 percent for the past 12 years.

By Catalin Cimpanu for Zero Day | February 11, 2019 -- 15:48 GMT (07:48 PST) | Topic: Security

C/C++

- It is hard to write safe C
- I don’t care what anyone says. Writing C is hard.

Shout out to @jgeigerm who I stole this joke from

C/C++

- Simple mistakes like using int* instead of size_t* can lead to serious vulns.
- Modern C++ has tools to make managing memory safer/easier
  - Spoiler! It is still hard

Compiler toolchain for this class

- For this class you may choose one of the following:
- MinGW (GNU gcc/g++ compiler)
  - Minimal GNU GCC on Windows
- clang/clang++
  - Compiler toolchain built on top of LLVM
- zig C/C++ compiler
  - Technically, zig is itself a programming language that is built on top of llvm. It provides a drop in replacement for most C/C++ compilers and provides cross compilation out of the box! It also tends to create smaller, more hackable builds as it supports directly emitting LLVM (not covered in this class, but you should definitely check it out!)
- In either case, we will use GNU Make to simplify the build process

Development tools

- We will use Visual Studio Code (blue, not purple program) as our IDE
  - Feel free to install c/c++ intellisense tools to get tab completion
- The officially supported compiler will be Mingw (gcc/g++)
- clang/Clang++, which is in most cases a drop in replacement, is also supported, and preferable when leveraging certain C++ Std libs.
  - Threading with mingw doesn’t always work/isn’t always supported :)
- For Debugging, we will use x64dbg
- For analyzing the executable statically, we will use a combination of PE-Bear and Ghidra

Why not Visual Studio?

- Windows has an official C compiler: cl.exe
- Visual Studio IDE + cl.exe does not easily support cross compilation (i.e., compiling our code on a Linux machine targeting a windows machine)
- Windows is a pretty “heavy” OS in that it is not as cheap to deploy as a linux box.
- For Tailored payloads, having a build server that can be run on any Linux machine is an incredibly attractive feature for our purposes
- ...It also makes grading easier/possible :D
- ...Also I don’t like .sln (Solution) files (build files used by Visual Studio)

...Also

It can run slowly in a VM and is very memory hungry.

Kidding aside, it is an incredible IDE, and the standard in industry for large scale software development.

This class doesn't cover it.

I also on principal hate all Microsoft products that have "Enterprise" in the name.

C, Microsoft Visual C, and C Runtimes

- C is a programming language
- Similar to LibC on Linux, Windows has a C runtime environment called the Microsoft Visual C Runtime
- This is different from the C programming language, and is usually compiler/implementation specific.
- This gives you access to standard functions
- Visual C != C, but they are very similar, and let you program basically the same way.
- Visual C has different compiler/linker directives
  - This means you might have to modify Microsoft example code for it to run with Mingw/clang

Hello World!

```c
#include <stdio.h>

int main(){
    printf("Hello %s!\n", "world");
    return 0;
}
```

Hello World: Win32

```
#include <windows.h>
int main(){
    // Message
    char message[] = "Hello world!\n";
    // Message size in bytes, not including the NULL-Byte
    DWORD messageSize = 13;
    DWORD dwBytesWritten = 0;

    // get Handle to STD_OUTPUT
    HANDLE stdOut = GetStdHandle(STD_OUTPUT_HANDLE);

    WriteFile(
        stdOut,
        message,
        messageSize,
        &dwBytesWritten,
        NULL
    );
    return 0;
}
```

WTH: that is so much more code!

- C-Runtimes abstract away details of utilizing OS provided APIs for interacting with system resources and objects
- The text you observe when running a terminal application is actually displaying content from STDOUT, STDERR and takes inputs from STDIN
- We will discuss more about windows internals later, but everything is accessed by *handles*
- For those familiar with Linux internals, a handle fulfils more or less the same role as a Binary File Descriptor
- Under the hood, printf will also get a handle to Standard output and push data to it!

Interop: Libc and MSVC

Most functions implemented in Libc’s stdlib are also implemented in MSVCR (Microsoft Visual C Runtime)

Using the stdlib adds a (tiny) amount of overhead

In particular, it requires

Entry Point:

Process creation is a very complex topic, and has many moving parts

For now, all you need to know is the OS performs a sequence of operations, and if there are no errors at the end, kernel will pass execution of the process to the *entry point*

This is the first bit of user defined code that is executed when the process starts

This will look different depending on what *type* of executable we create! Windows supports executables targeting different *subsystems*

We will focus on *console* applications for this lecture.

Code Sample 0:

- The C-Runtime is used by most executables, and as a result most compiler toolchains will call the C runtime startup code by default!
- In this example, we will compile a simple hello world program with and without it!
- Just like in Linux, the executables produced by our compilers have support for debugging information: in particular, we can view the symbols associated with the generated assembly code
- Let’s take a look at this!

Fundamental Concepts For this class

- C/C++ Runtime
- Windows API
  - Win32 API
  - Native API
  - More undocumented APIs
- The Windows Runtime
- If time permits:
  - Component Object Model
  - Common Language Runtime

Ghidra

- Interactive Disassembler
- Has a decompiler
- Other tools useful for reverse engineering that we will cover another time
- Developed by the NSA.
- It is free, and fully open source!
- The alternative is IDA (interactive Disassembler) that costs ~$10k
  - Update… now it is only 5k a year but you only get it so long as you pay for it.Shout out to Hex-Rays for putting a 5MB maximum on binaries for students, and going the Adobe cloud route for SAAS. -_-...
- Let’s use Ghidra to take a look at what our compiler produced!

Demo 0: Hello world

What exactly just happened?

What does gcc/g++/zig cc/clang/cl.exe actually do to our source code files?

1) Compiling
2) Linking

Converting Text Programs into Executables

What is a C/C++ compiler toolchain responsible for? Converting text (code) into an application that a computer can run!

This is accomplished with two operations.

Compiling:

Linking:

Compiling: Seriously oversimplified

- Convert text file into an intermediate file called an *object file*
  - Object files contain assembled code!
  - If you are using LLVM, this is a little more complicated
- The compiler evaluates “preprocesses” steps
- Constructs an abstract syntax tree from our code
- From there it can actually generate machine code that the CPU will execute using its backend assembler
- It also creates a location to store constant data
- The end result of this stage are “Object files”

Linking: Seriously oversimplified

- Once we compile source files, we need to “link” the binary
- The linker finds where each symbol and function is, and links them together!
  - Ie: the linker replaces the references to undefined symbols with the correct addresses
- This is an attractive feature, as if there is lots of code reuse across multiple binaries, we can store the referenced code in a *library*.
- There are multiple ways to link against external code

Linking

Static Linking: The external code is directly embedded in your final executable. This can be useful if you are unsure if the user running your code will have the dependencies!

Dynamic Linking: The reference to the external library is set in your binary, and at runtime, it loads the external library! This gives you smaller binaries, and allows for lots of code reuse!

Dynamic Link Libraries (DLLS)

DLLs are analogous to Linux Shared Objects (SOs)

They are portable executables that contain exported, callable functions that can be dynamically loaded at run time.

Some notable ones being

- NTDLL.dll: core functionality for talking with the kernel
- kernel32.dll (one of the API subsystem libraries)
- Advapi.dll (another one of the API subsystem libraries)l
- Msvcrt.dll (c runtime libraries)

Dynamic Linking

Implicit Linking: The program declares in the binary that it wants the OS’s PE Loader to resolve it’s dependencies, and declares them. If the PE loader cannot find the dependency, the program exits.

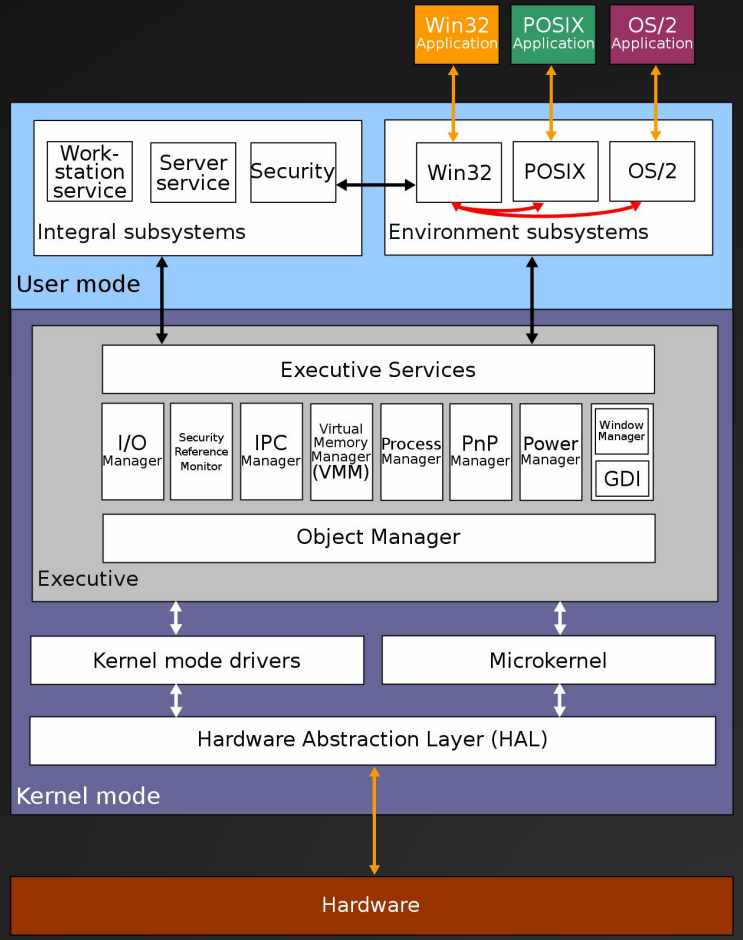
Explicit Linking: The programer explicitly loads the dependency at run time, and if it is unable to find the library, it can choose how to respond

Compiling and Linking

- Compiler front ends like gcc/g++ perform compilation and linking in 1 step
- You can pass options to the linker with `-lDllName`
- Because most binaries use Kernel32.dll, they generally automatically link against it

Windows System Architecture

- User Processes
- Subsystem DLLs
- NTDLL.dll
- Service Processes
- Executive
- Kernel
- Device Drivers
- Win32k.sys
- (sometimes) Hyper-V



Interacting with the Windows OS

- Windows API functions (win32): Documented, callable functions in the Windows API. For example, MessageBox, CreateFile, GetMessage
- Native system services (sys calls): Undocumented (officially) underlying serivices in the OS that are callable from user mdode. For example
  - NtAllocateVirtualMemory is the internal service used for VirtualAlloc
  - NtCreateUserProcess is the internal service used by CreateProcess
- Kernel support functions: functions inside the Windows OS that can only be called from in kernel mode

NtDLL.dll

Implements the Windows Native API. This is the lowest layer of code that is still Userland code.

It is used to communicate with the kernel for system call invocation.

NtDLL also implements the Heap Manager, the (executable) Image loader and some of userland thread pools. Every process loads this DLL in the same location in memory!

Kernel32.dll

Contains (more or less) the same functionality as NtDLL!

It exposes basic operations such as memory management, input/output (I/O) operations, process and thread creation, and synchronization functions

It can be thought of as a compatibility layer, as it almost always calls directly into NTDLL.dll

This is to maintain backwards compatibility– where the Win32 API rarely changes, but the Native API changes from release to release.

Win32 API

We will mostly leverage documented functions from the Windows API

The function definitions are well documented

Reading that documentation however, is a skill that must be learned

Sometimes, we need more control over what we are trying to accomplish, and will leverage undocumented functions stored in NTDLL.dll

32bit vs 64bit

This class will focus on 64bit executables (Intel x86 64)

When developing code that needs to run on either a 32bit or 64bit systems, you need to take care when assuming the size of various types. For example, type sizes vary across 32bit and 64bit architectures and you need to take care when defining them.

Highlight of Win32 Data Types

- WORD: 16-bit unsigned integer
- DWORD (Double word): 32-bit unsigned integer
- QWORD (Quad word): 64-bit unsigned integer
- LPCSTR: Pointer to a c string (null terminated)
  - Each character is a char
- LPCWSTR: Pointer to wide character c string (double null terminated)
  - Each character is a wchar
- BYTE: unsigned char
- LPVOID: Pointer to any type
- HANDLE: Handle object

RTFM as needed:
https://docs.microsoft.com/en-us/windows/win32/winprog/windows-data-types

Example: Popping a message box

```c
#include <windows.h>

int main(){
    MessageBoxA(NULL, "Hello there", "General Kenobi", MB_OK);
    return 0;
}
```

Demo:
Compiling the example and linking against User32.dll