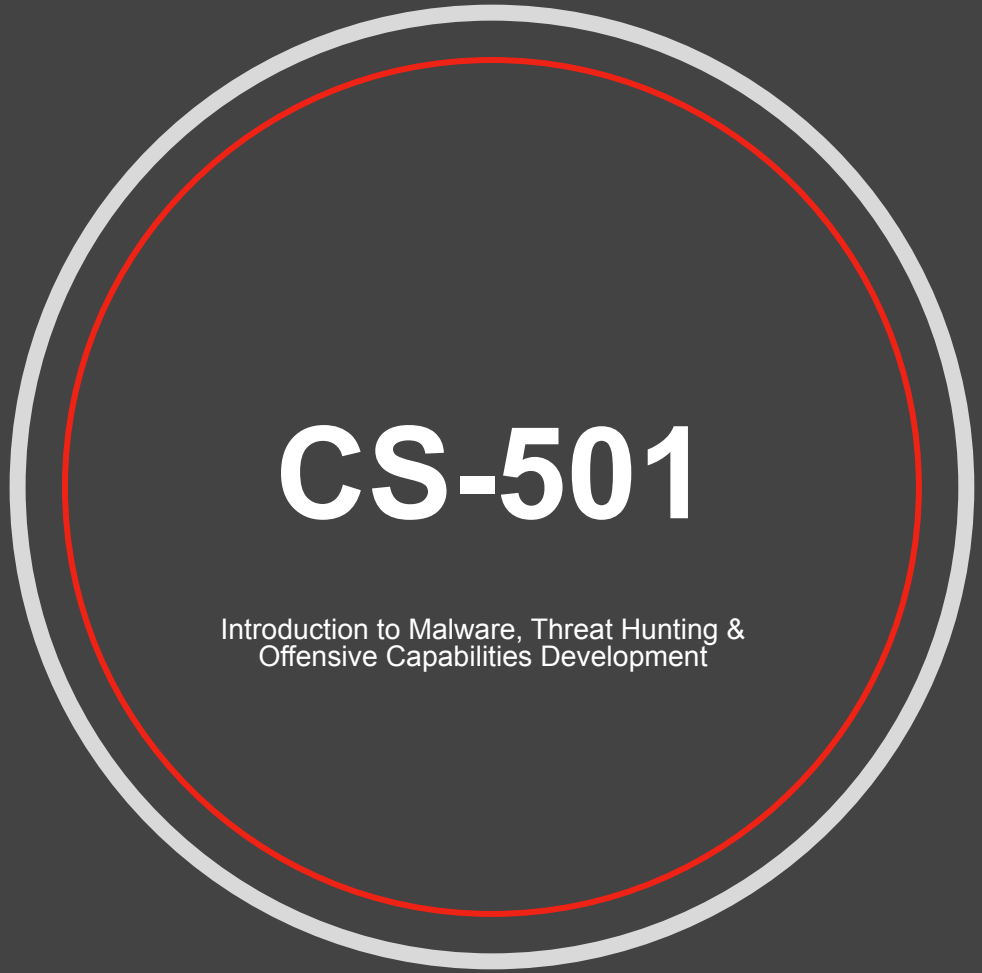


CS-501

Introduction to Malware, Threat Hunting &
Offensive Capabilities Development

Lecture 2: Crash course in C++

C++ In a nutshell

Overview C++

- C++ is a statically types, 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, safer and faster. 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
- ...etc
- 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

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

Example Program

```
#include <iostream>
#include <string>

using namespace std;

int main(){
    cout << "Greetings! What should I call you?" << endl;
    string name;
    getline(cin, name);
    cout << "Hello there "<< name << endl;
    return 0;
}
```

Compilers for this class

For this class you may choose either one of the following:

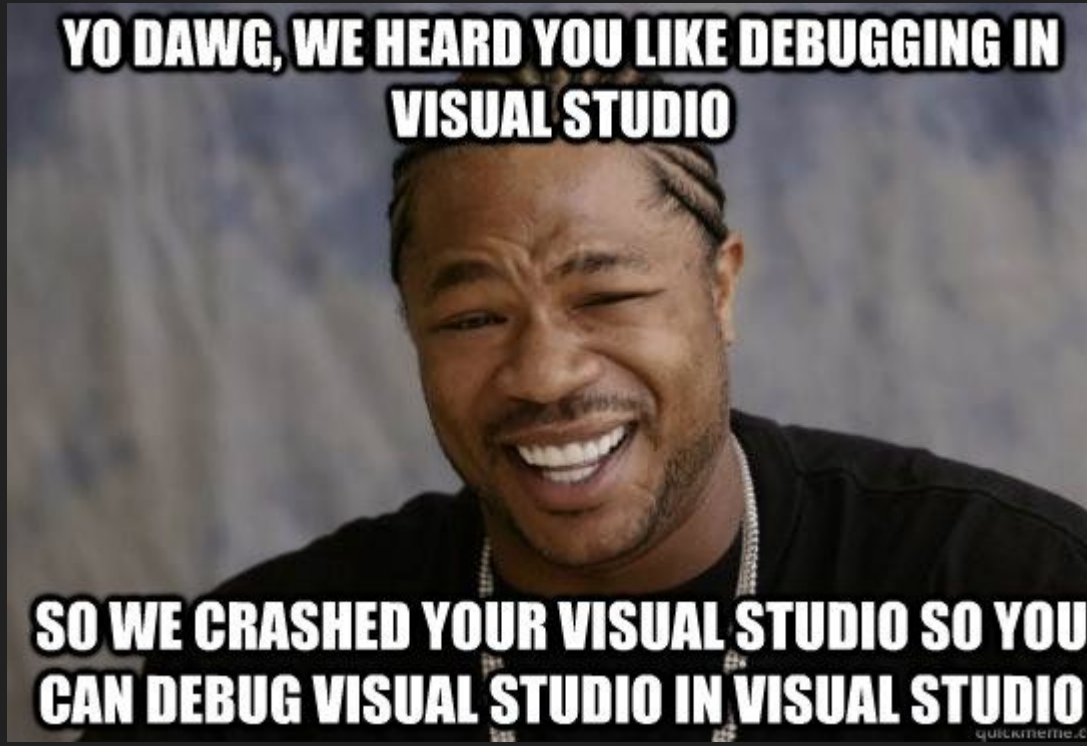
- MinGW (GNU gcc/g++ compiler)
- zig C/C++ compiler
  - Technically, zig is itself a programming language that is built on top of llvm. It provides, however, a drop in replacement for most C/C++ compilers and provides cross compilation out of the box! It also tends to create smaller, more hackable builds 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

Why not Visual Studio?

- Windows has an official C/C++ compiler
- This toolchain relies on the Visual Studio IDE and 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
- ...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.



Converting Text Programs into Executables

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

This is accomplished with two operations.

Compiling

Linking

Compiling: Seriously oversimplified

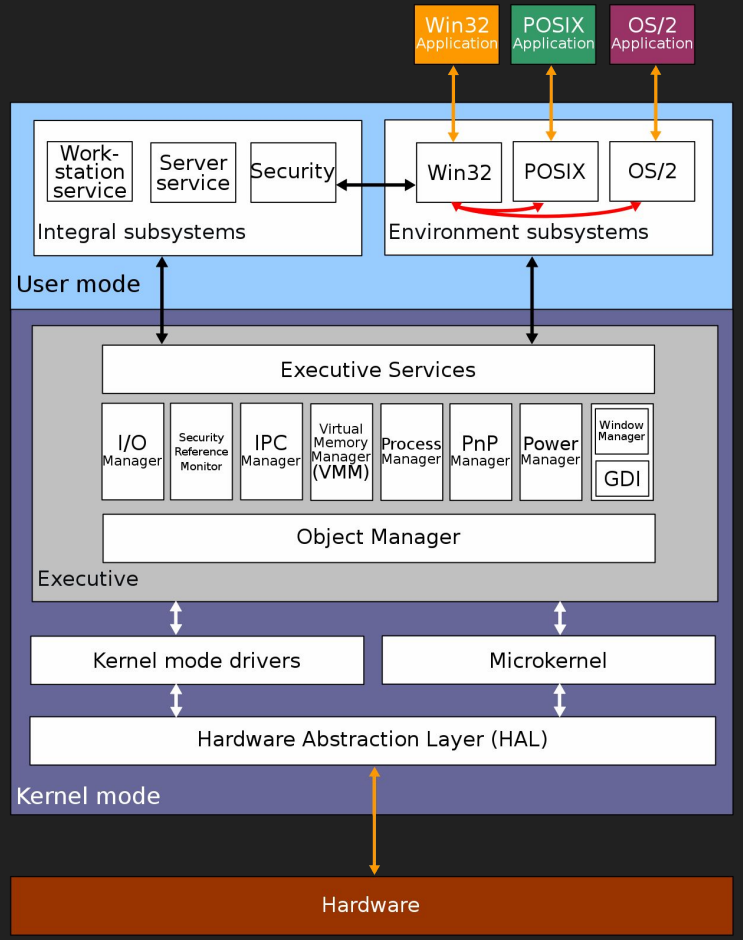
- Convert text file into an intermediate file called an object file
  - If you are using LLVM, this is a little more complicated
- The compiler evaluated “preprocesses” steps
- Constructs an abstract syntax tree from our code
- From there it can actually generate machine code that the CPU will execute 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

Windows System Architecture

- User Processes
- Subsystem DLLs
- NTDLL.dll
- Service Processes
- Executive
- Kernel
- Device Drivers
- Win32k.sys
- HyperV



Fundamental Concepts For this class

- Windows API
  - Win32 API
  - Native API
  - More undocumented APIs
- The Windows Runtime

If time permits:

- Component Object Model
- Common Language Runtime

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 allied in kernel mode

Dynamic Link Libraries (DLLS)

DLLs are analogous to Linux Shared Objects (SOs)

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

Some notable ones being

- kernel32.dll (one of the API subsystem libraries)
- Advapi.dl (another one of the API subsystem libraries)l
- Msvcrt.dll (c runtime libraries)

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.

Example: Popping a message box via Import

```
#include <windows.h>

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

Example Continued

name (38)	blacklist (1)	group (7)	ordinal (0)	library (3)
TlsGetValue	-	execution	-	kernel32.dll
SetUnhandledExceptionFilter	-	exception	-	kernel32.dll
GetLastError	-	diagnostic	-	kernel32.dll
__C_specific_handler	-	-	-	kernel32.dll
__getmainargs	-	-	-	msvcrt.dll
__initenv	-	-	-	msvcrt.dll
__iob_func	-	-	-	msvcrt.dll
__lconv_init	-	-	-	msvcrt.dll
__set_app_type	-	-	-	msvcrt.dll
__setusermatherr	-	-	-	msvcrt.dll
_acmdln	-	-	-	msvcrt.dll
_amsg_exit	-	-	-	msvcrt.dll
_cexit	-	-	-	msvcrt.dll
_commode	-	-	-	msvcrt.dll
_fmode	-	-	-	msvcrt.dll
_initterm	-	-	-	msvcrt.dll
_onexit	-	-	-	msvcrt.dll
abort	-	-	-	msvcrt.dll
calloc	-	-	-	msvcrt.dll
exit	-	-	-	msvcrt.dll
fprintf	-	-	-	msvcrt.dll
free	-	-	-	msvcrt.dll
signal	-	-	-	msvcrt.dll
strlen	-	-	-	msvcrt.dll
strncmp	-	-	-	msvcrt.dll
vfprintf	-	-	-	msvcrt.dll
MessageBoxA	-	-	-	user32.dll

Dynamically Loading via LoadLibrary

```
#include <windows.h>

/**
 * https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-messageboxa
 int MessageBoxA(
  HWND   hWnd,
  LPCSTR lpText,
  LPCSTR lpCaption,
  UINT   uType
);
 */
typedef int(__stdcall *msgbox)(HWND, LPCSTR, LPCSTR, UINT);

int main(){
    HMODULE hModule = ::LoadLibraryA("User32.dll");
    msgbox f = NULL;
    if (hModule != NULL){
        f = reinterpret_cast<msgbox>(::GetProcAddress(hModule, "MessageBoxA"));
        if( f != NULL){
            (*f)(NULL, "Hello there from User32.dll" , "I was dynamically loaded!" , MB_OK);
        }
    ::FreeLibrary(hModule);
    }
}
```

C++ For Windows

Demos

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

Let’s use Ghidra to take a look at what our compiler produced!

Dynamically Loading via LoadLibrary

name (40)	blacklist (1)	group (8)	ordinal (0)	library (2)
DeleteCriticalSection	-	synchronization	-	kernel32.dll
EnterCriticalSection	-	synchronization	-	kernel32.dll
InitializeCriticalSection	-	synchronization	-	kernel32.dll
LeaveCriticalSection	-	synchronization	-	kernel32.dll
GetStartupInfoA	-	reckoning	-	kernel32.dll
VirtualProtect	x	memory	-	kernel32.dll
VirtualQuery	-	memory	-	kernel32.dll
malloc	-	memory	-	msvcrt.dll
memcpy	-	memory	-	msvcrt.dll
fwrite	-	file	-	msvcrt.dll
Sleep	-	execution	-	kernel32.dll
TlsGetValue	-	execution	-	kernel32.dll
SetUnhandledExceptionFilter	-	exception	-	kernel32.dll
FreeLibrary	-	dynamic-library	-	kernel32.dll
GetProcAddress	-	dynamic-library	-	kernel32.dll
LoadLibraryA	-	dynamic-library	-	kernel32.dll
GetLastError	-	diagnostic	-	kernel32.dll
__C_specific_handler	-	-	-	kernel32.dll
__getmainargs	-	-	-	msvcrt.dll
__initenv	-	-	-	msvcrt.dll
__iob_func	-	-	-	msvcrt.dll
__lconv_init	-	-	-	msvcrt.dll
__set_app_type	-	-	-	msvcrt.dll
__setusermatherr	-	-	-	msvcrt.dll
_acmdln	-	-	-	msvcrt.dll
_amsg_exit	-	-	-	msvcrt.dll
_cexit	-	-	-	msvcrt.dll
_commode	-	-	-	msvcrt.dll

Win32 API

We will mostly leverage documented functions from the Windows API

The function definitions are well documented

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

Strings in Windows

In C, strings aren’t official types. They are simply char pointers to data that is null terminated. These strings are typically UTF-8 encoded

This is great when we need the memory footprint of strings to be small, but has two problems

- Doesn’t allow for random access
- Doesn’t support all characters (Chinese, Russian...etc)
    - Why is that a problem for us :-)?

Strings in Windows Continued

- C/C++ runtimes for windows supports both UTF-8 and UTF-16 (Unicode) characters
- It is generally preferable to use UTF-16 characters with a few exceptions
- Each character set has its own set of strings functions (strlen, strcpy, strcat...etc) and wslen, wspy,wscat...etc for so called wide-character c strings

Expansion macros

- Win32 API has a convenience macro for most functions it defines that require strings as arguments.
- The suffix will be either an A for ANSI or W for wide character Unicode
- For example, GetComputerNameA will return an ANSI string, GetComputerNameW will return a UNICODE string

TCHAR type

Similarly, Windows provides the TCHAR macro to expand to either char or wchar_t type

- Chars are 1 byte each (8 bits)
- Wchars are 2 bytes each (16 bits)

String literals

By default, string literals are ANSI c strings, but by appending an L to the beginning of the literal, the compiler will define it as a wide character string literal

If you want it to work for Either, the TEXT() macro can be used

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.

Coding conventions for this class

- Windows API functions are prefixed by a double colon
- Type names use Pascal casing (Example PascalCasing)
- Private functions/elements in a class start with an underscore
- Function names are also Pascal casing
- We will use the C++ standard library for common types such as vectors/strings
- Try to keep functions shorter than 20 lines of code

C++ usage in this class

This is not a class on C++ or C. You are expected to know the basic of one or both. However, we will make use of the following features of C++ as we develop code

- Nullptr: Useful for NULL pointer
- auto: Type inference
- new/delete: memory management
- Smart pointers: memory management
- Classes
- Scoping
- Templates
- Constructors

General Advice for Strings

- For this class, you are more than welcome to only use the Unicode functions.
  - In fact, when most versions of Windows call FunctionA (ANSI), the inputs are converted to unicode and passed to FunctionW (Unicode)
- If your entry point is named main, it will default to ANSI
- If your entry point is named wmain, it will default to Unicode using VisualStudio but since we are using Zig c++ /g++ you must explicitly pass -municode as a compile flag
- If your entry point is named _tmain it will check whether or not UNICODE is defined
- You may only use one of the following from the standard c++ library:
  - cout
  - wcout
  - Calling both can lead to undefined behavior

We will revisit this when we talk about C2 Channels, as this will be a become a bit of a headache.

For more see https://docs.microsoft.com/en-us/windows/win32/learnwin32/working-with-strings

Highlight of Win32 Data Types

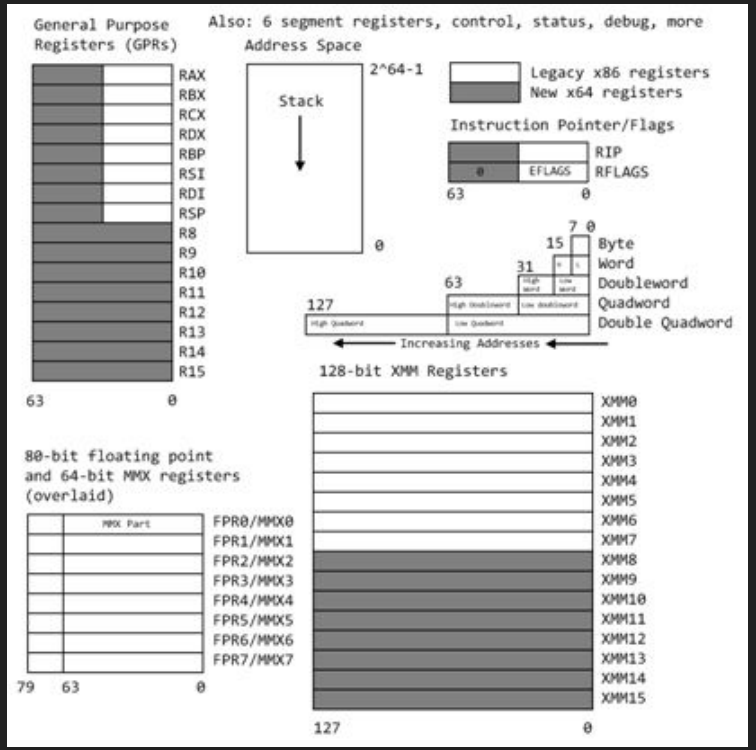
- DWORD (Double word)
- LPCSTR (Pointer to a c string (null terminated)
- LPCWSTR (Pointer to wide character c string (double null terminated)
- BYTE (8 bits)
- LPVOID (Pointer to any type)
- HANDLE (Handle object)

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

X86-64 Assembly Crash Course

- Assembly language is machine specific code that is executed directly by a processor
- X64 is an extension of 32bit x86
- It contains 16 general purpose registers: `RAX, RBX, RCX, RDX, RBP, RSI, RDI, and RSP, R8-R15`

Note: You are expected to be familiar with x86 assembly



Registers by Name

RAX - register a extended. Usually return register for integers

RBX - register b extended

RCX - register c extended

RDX - register d extended

RBP - register base pointer (start of stack) (32bit: EBP)

RSP - register stack pointer (current location in stack, growing downwards)

RSI - register source index (source for data copies)

RDI - register destination index (destination for data copies)

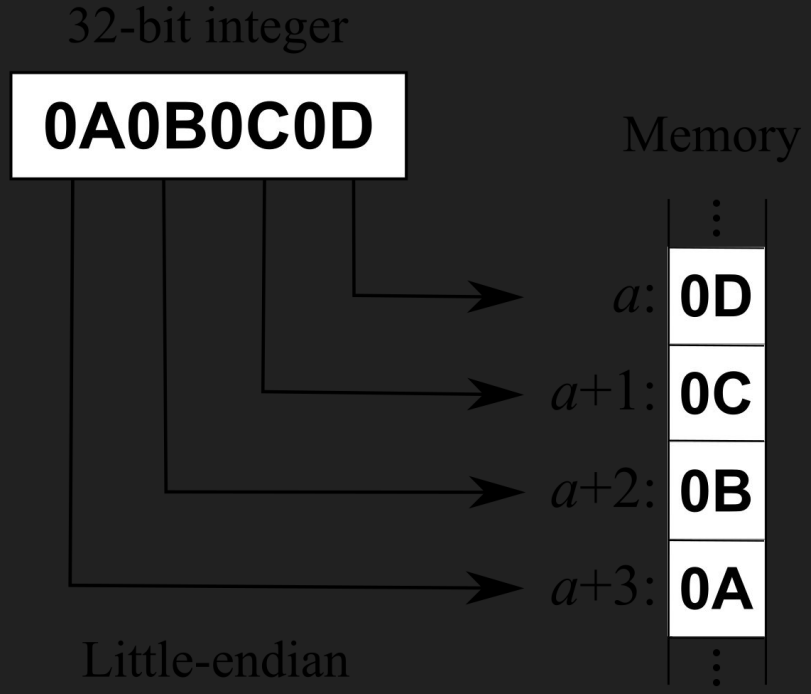
RIP - Instruction Pointer (pointer to the next instruction)

Replace the “R” with an “E” and you get the analogous instruction in x86.

Refresher: Types

- Byte: 8 bits
- Word: 2 bytes (16 bits)
- Double Word (Dword) 32 bits
- Quadword: 64 bits
- Double Quadword: 128 bits

Intel stores bytes as “Little Endian”



Declaration Mapping

C/C++ Type	Storage	Size (bits)
char	Byte	8
short	Word	16
int	Dword	32
unsigned int	Dword	32
long	Quadword	64
long long	Quadword	64
char *	Quadword	64
int *	Quadword	64
float	Dword	32
double	Quadword	64

Common Instructions

Opcode	Meaning	Opcode	Meaning
MOV	Move to/from/between memory and registers	AND/OR/XOR/NOT	Bitwise operations
CMOV*	Various conditional moves	SHR/SAR	Shift right logical/arithmetic
XCHG	Exchange	SHL/SAL	Shift left logical/arithmetic
BSWAP	Byte swap	ROR/ROL	Rotate right/left
PUSH/POP	Stack usage	RCR/RCL	Rotate right/left through carry bit
ADD/ADC	Add/with carry	BT/BTS/BTR	Bit test/and set/and reset
SUB/SBC	Subtract/with carry	JMP	Unconditional jump
MUL/IMUL	Multiply/unsigned	JE/JNE/JC/JNC/J*	Jump if equal/not equal/carry/not carry/ many others
DIV/IDIV	Divide/unsigned	LOOP/LOOPE/LOOPNE	Loop with ECX
INC/DEC	Increment/Decrement	CALL/RET	Call subroutine/return
NEG	Negate	NOP	No operation
CMP	Compare	CPUID	CPU information

Calling Conventions

- Calling conventions are *conventions* used to invoke functions. This includes how to pass arguments to the function, and how to get returned objects.
- Functions defined do not need to abide by this convention
  - In fact, weird calling conventions is a common obfuscation technique
- The Windows API has a standardized calling convention for most of its functions.
  - Note that WinAPI expands to `__stdcall`

Windows x64 Calling Convention: Simple case

- RCX, RDX, R8, R9 are used for integer, and pointer arguments in that order from left to right
- Additional arguments are pushed to the stack (left to right)
- The return value (integer/pointer) is stored in RAX if it is 64 bits or less
- Return values > 64 bits (structs, classes) will have stack space allocated by the *caller*.
  - This case is a bit more complicated and will be covered when we start reverse engineering C++ objects
- RAX, RCX, RDX, R8, R9, R10, and R11 are **volatile**

Src: https://software.intel.com/content/www/us/en/develop/articles/introduction-to-x64-assembly.html

Example: MessageBox

```
; Chris Lomont 2009 www.lomont.org
extrn ExitProcess: PROC   ; external  functions in system libraries
extrn MessageBoxA: PROC
.data
caption db '64-bit hello!', 0
message db 'Hello World!', 0
.code
Start PROC
  sub     rsp, 28h       ; shadow space, aligns stack
  mov     rcx,  0        ; hWnd = HWND_DESKTOP
  lea     rdx, message ;  LPCSTR lpText
  lea     r8,  caption ;  LPCSTR lpCaption
  mov     r9d,  0        ; uType = MB_OK
  call    MessageBoxA  ; call  MessageBox API function
  mov     ecx, eax       ;  uExitCode = MessageBox(...)
  call ExitProcess
Start ENDP
End
```

How are we doing to learn x64 in this class?

The best way to get more comfortable reading assembly is to keep reading assembly.

You do not have to write assembly in this course, but you will have to read it. The decompiler offered by Ghidra will often make mistakes, and incorrectly reflect what the program is doing. The disassembled program (usually) the ground truth

Looking at Assembly:

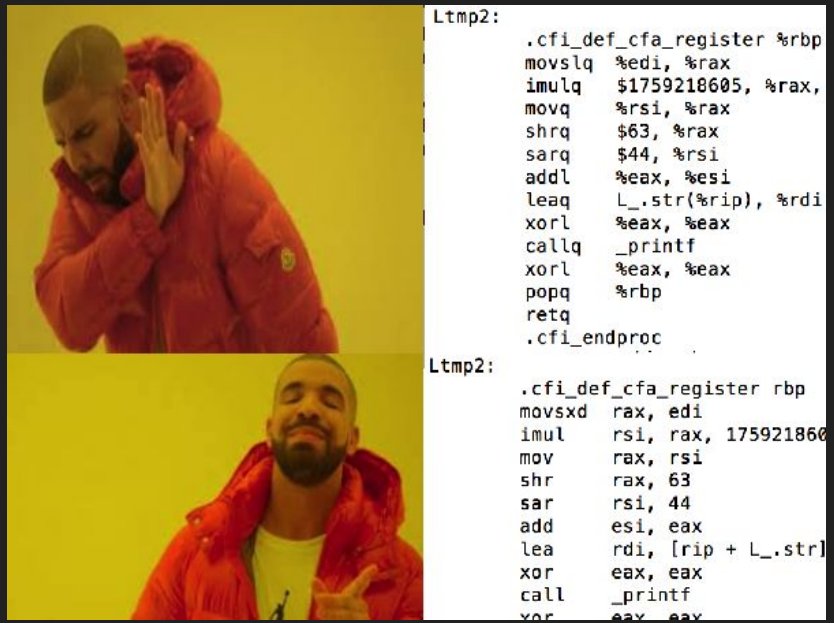
We will use Intel Flavor assembly syntax. Why? Because it is better.

Why is it better? Because that happens to be the convention I learned.

Examples:

- Basic MessageBox
- MessageBox Thread
- Crackme (Relevant to homework)
- Troll.exe

Src: https://www.reddit.com/r/ProgrammerHumor/comments/56fjm5/att_vs_intel_syntax/



Processes

Memory Layout

- The Windows OS divides its Virtual Address space into two portions: kernel and Userland*
- Memory here grows “downward” from higher address spaces to lower address spaces
- On x64, the upper half is reserved for kernel space and the bottom half is reserved for processes

Processes

Basic container for threads.

Nothing is executed outside of the context of a process.

- You don’t “run a processes”
- You run threads which are managed by a processes
- Processes are containers, and there is no such thing (to my knowledge) as code running outside of a process

Threads

Unit of execution contained within a process

- Actual entire that executes code
- Thread Local storage (TLS) provides a way to store data shared by other threads in the process

More on threads in later lectures.

Process Creation

Somewhat complicated. We will simplify it for this class, than dive into it next class after we talk about handles.

- Kernel opens the image (executable file) and verifies it is the correct format
- The kernel creates a new process kernel object and a thread kernel object
- The kernel maps the image to an address space, as well as ntdll.dll
  - Note this gets mapped to just about every type of process
- The creator process notifies Windows subsystem process (Csrss.exe) that a new process and thread have been created
- From the kernel's perspective, the process is created at this point
- Some magic happens, imports are resolved and after all the required LLs are declared, we reach the entry point and the program starts

How to Create a Process

```
BOOL CreateProcessA(
  LPCSTR                lpApplicationName,
  LPSTR                 lpCommandLine,
  LPSECURITY_ATTRIBUTES lpProcessAttributes,
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  BOOL                  bInheritHandles,
  DWORD                 dwCreationFlags,
  LPVOID                lpEnvironment,
  LPCSTR                lpCurrentDirectory,
  LPSTARTUPINFOA        lpStartupInfo,
  LPPROCESS_INFORMATION lpProcessInformation
);
```

Basic Information of a Process

- Name: Usually the executable name. This is NOT a unique identifier
- Process ID (PID: Unique ID of a process. PIDS are reused after a process terminates
- Status: Running, Suspended, Not Responding
- Username: the user who is running the process. It also includes the primary token that holds the security context for the user
- Session ID: Session number under which the process executes. Session 0 is for system processes and services. Session 1 and higher are used for interactive logins.