





Lecture 3:

C, C++ and Static Analysis Continued

Agenda for Today

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



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)
- 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 its dependencies, and declares them. If the PE loader cannot find the dependency, the program exits.

Explicit Linking: The programmer 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

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 services in the OS that are callable from user mode. 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.

The biggest difference being pointer types are 32-bit unsigned integers and on 64-bit are 64 bit unsigned integers

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

```
#include <windows.h>

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

Demo:
Compiling the example and linking
against User32.dll

Reading the docs

- Debugging your code for 8 hours can save you 5 minutes of reading the docs
 - I myself, routinely don't read the documentation and suffer for it. Be better than me. Learn from my mistakes. RTFM
- Example: <https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-messagebox>

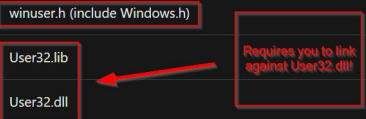
Syntax

C++

```
int MessageBoxA(  
    [in, optional] HWND    hWnd,  
    [in, optional] LPCSTR  lpText,  
    [in, optional] LPCSTR  lpCaption,  
    [in]           UINT    uType  
);
```

Requirements

Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-dialogbox-l1-1-0 (introduced in Windows 8)

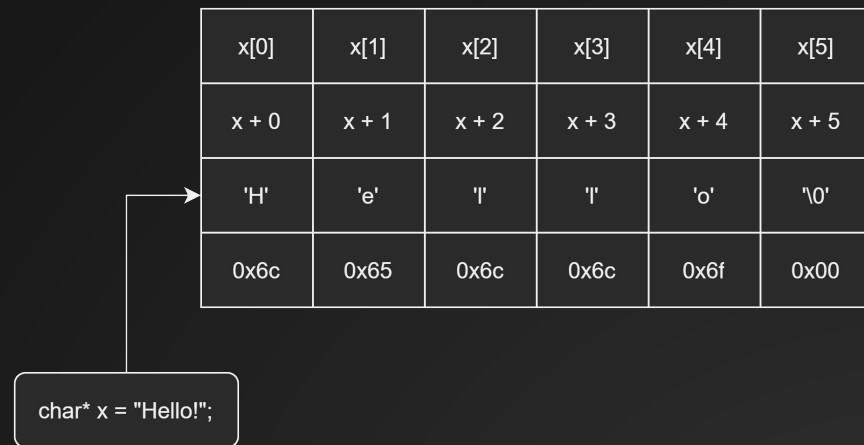


Strings in Windows

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 but not always! On Windows, they are ANSI

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)
 - This a problem for us. Why? :-)?



Strings in Windows Continued

- C/C++ runtimes for windows supports both ANSI and UNICODE UTF-16 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,,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
 - Also, `MessageBoxA` displays ANSI characters, where as `MessageBoxW` displays unicode!

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 depending on the encoding of the application, the TEXT() macro can be used

We will talk about this next time when we discuss WinMain vs wWinMain (take a guess at what it is used for :))

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 but I won't enforce this.

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
- (Pure) Virtual Functions: making our malware modular :)

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
 - I personally just stick to `printf/wprintf` though
- We will revisit this when we talk about C2 Channels, as this will become a bit of a headache.
- For more see <https://docs.microsoft.com/en-us/windows/win32/learnwin32/working-with-strings>

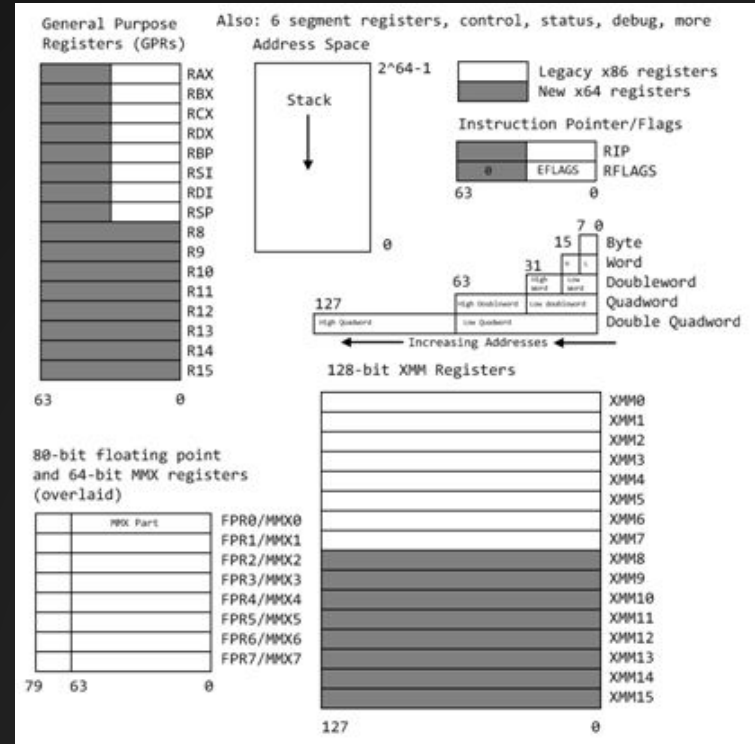
X86-64 Assembly Crash Course

Assembly language is machine specific code that is executed directly by a processor

X64 (amd64) 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)

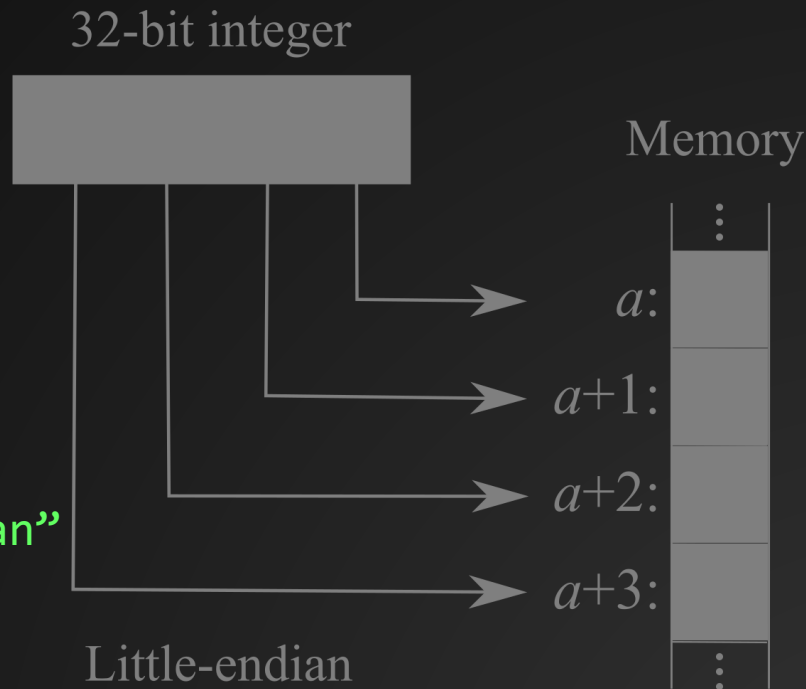
RIP - Instruction Pointer (pointer to the next instruction)

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>

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 (much) 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 is (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/



```
Ltmp2:
    .cfi_def_cfa_register %rbp
    movslq %edi, %rax
    imulq $1759218605, %rax,
    movq %rsi, %rax
    shrq $63, %rax
    sarq $44, %rsi
    addl %eax, %esi
    leaq L_.str(%rip), %rdi
    xorl %eax, %eax
    callq _printf
    xorl %eax, %eax
    popq %rbp
    retq
    .cfi_endproc

Ltmp2:
    .cfi_def_cfa_register rbp
    movsxd rax, edi
    imul rsi, rax, 175921860
    mov rax, rsi
    shr rax, 63
    sar rsi, 44
    add esi, eax
    lea rdi, [rip + L_.str]
    xor eax, eax
    call _printf
    xor eax, eax
```

Using Ghidra to Look at our code

Using X64dbg to look at our code

Example: MessageBox

```
global _main
extern MessageBoxA ; Import external symbols from user32.dll
extern ExitProcess ; Import external symbols from kernel32.dll
SECTION .data
    message0 db "Hello world!", 0 ; null terminator
    message1 db "I am a message!", 0
SECTION .text
    sub rsp, 8
    xor rcx, rcx ; pass NULL for handle
    mov rdx, message0
    mov r8, message1
    xor r9, r9
    call MessageBoxA
    call ExitProcess
```

Processes

Processes

Nothing in userland 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

Memory Layout

- Each processes gets its own *Virtual Address Space*
- The Windows OS divides 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

Threads

Unit of execution contained within a process

- Actual entity that executes code

More on threads in later. For now, we will only deal with processes that have a single executing thread.

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

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.

Memory Management

C++ In a nutshell



Stack Memory

- Data structure built on top of our Virtual Address Space that allows us to Push, and pop values from a stack
- Whenever we invoke a function with *call* a new *stack frame* is created. We call this the function call stack
- This is a contiguous chunk of memory that acts as a working space for a function's duration
- Stack memory is temporary: once the function returns the memory is reclaimed
- Still error prone but is typically safer and faster than Heap allocations
- A programmer usually does not need to worry about managing stack memory
- We need to know how much is needed at compile time!

Stack based Footguns

- Stack overflows
- Referencing reclaimed stack memory
- Mostly, just be careful when referencing strings/raw data
- You cannot return a string from a function safely!



Heap Allocation

Allows for dynamic allocation sizes. I.e., we don't need to know how much memory is needed at compile time

Heap memory is managed by a heap allocator.

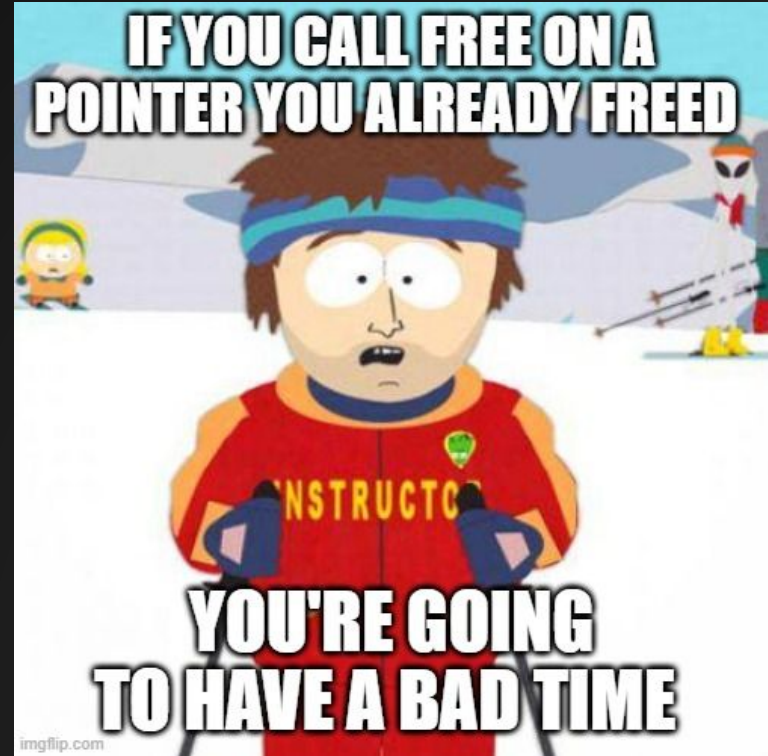
The programmer allocates with a call to `malloc` and frees the memory with a call to `free`

If the programmer forgets to call `free`, that memory is now unusable until the process terminates and we have introduced a *memory leak*

Footguns. So many footguns

If you try and free memory that has already been freed, you are going to get undefined behavior and possibly an exploitable bug.

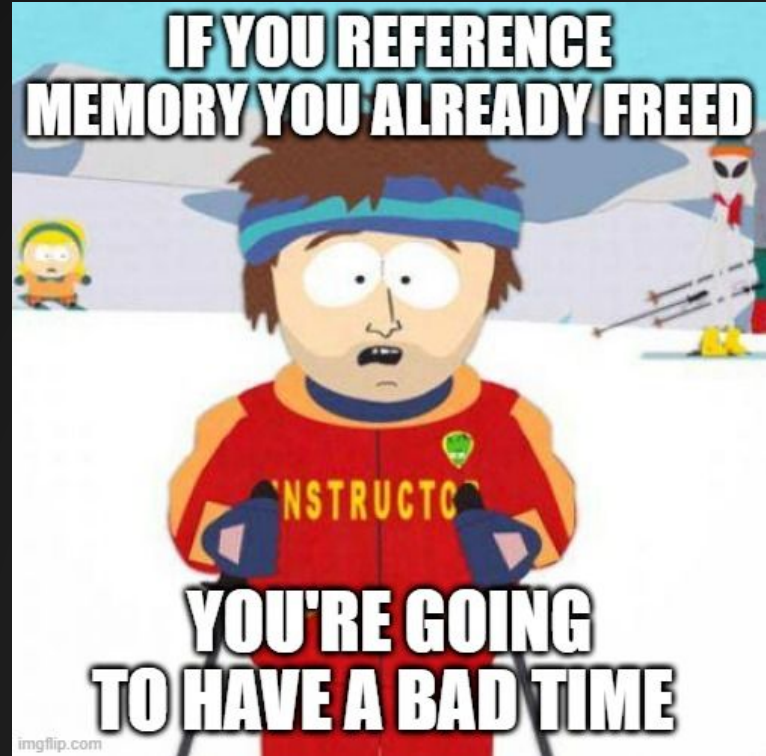
This is called a double free



Footguns. So many footguns

If you try and reference memory that has already been freed, you are going to get undefined behavior and possibly an exploitable bug.

This is called a use after free bug



Dead Squirrels



Uninitialized memory

An uninitialized variable is one that is declared, with no value set

For example, `DWORD x;`

Depending on the compiler, the contents of `x` could be completely random

This will lead to unexpected behavior

Memory Corruption

Allocating memory on the heap (malloc/new) without freeing it

This memory is no longer usable for allocation and from the perspective of the processes is used

Some compilers/runtimes will detect when allocations have gone out of scope

You should not rely on this. Anytime you call new/malloc, you should also call delete/free

Returning Stack Variables

If we have a pointer to memory in a stack frame, when we exit the function that memory might no longer be valid

By accessing it directly after the function it might look OK, but this is incredibly dangerous

In particular, the process will reclaim this memory and it will likely be filled with garbage.

Debugging Example

Examples Heap Example