DEFEATING ENCRYPTION BY USING UNICORN ENGINE WORKSHOP

MANTRA
INFORMATION SECURITY

Balázs Bucsay

Founder & CEO of

Mantra Information Security

https://mantrainfosec.com

BEFORE WE START

- Be efficient! Do not waste time on setup please.
- Ensure you have
  - A Github account
  - JDK & Ghidra installed (see repository)
  - Presentation: https://tinyurl.com/2f4xv326
- Lab repository: https://github.com/mantrainfosec/unicorn-workshop-labs/

BIO / BALÁZS BUCSAY

- Trainer of this course
- Over two decades of offensive security experience
- 15+ years of research and consultancy
- Started learning assembly at the age of 13
- Software Reverse Engineer
- Certifications: OSCE, OSCP, OSWP (Prev: GIAC GPEN, CREST CCT Inf)

BIO / BALÁZS BUCSAY

- Previously developed:
  - Exploits for Windows & Linux applications
  - Shellcode payloads for multiple architectures
  - Kernel driver exploits

- Frequent speaker at IT-Security conferences:
  - US - Washington DC, Atlanta, Honolulu
  - Europe - UK, Belgium, Norway, Austria, Hungary...
  - APAC - Australia, Singapore, Philippines

BIO / BALÁZS BUCSAY

- Hobbies:
  - Travelling (been to 75+ countries)
  - Hiking, kayaking, cycling
  - IT Security
- Passionate about learning from others
- Kayaked the length of the Thames (300km)

- Twitter: @xoreipeip
- Mantra on Twitter: @mantrainfosec
- Linkedin: https://www.linkedin.com/in/bucsayb/

BIO / BALÁZS BUCSAY

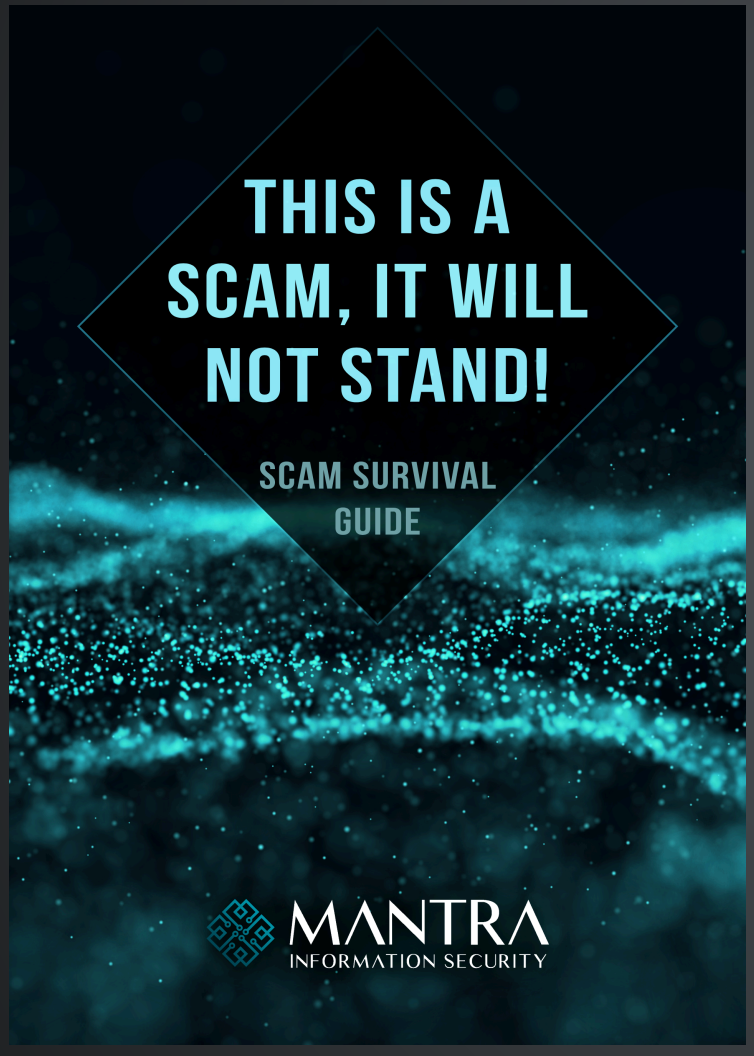
MANTRA
INFORMATION SECURITY

MANTRA INFORMATION SECURITY

- London based boutique consultancy
- Decades of experience and excellence
  - Specialised training delivery (Software Reverse Engineering, ...)
  - Cloud, CI/CD, Kubernetes reviews
  - Red Teaming, EASM, Infrastructure testing
  - Web application and API assessments
  - Reverse-engineering, embedded devices and exploit development
  - ...
- Full stack consultancy: from identifying vulnerabilities to implementing fixes

https://mantrainfosec.com

BOOK: THIS IS A SCAM! IT WILL NOT STAND!



MANTRA
INFORMATION SECURITY

- Empower others to protect themselves
- 11 gripping and educational real-world stories
- Download and share it freely
- Absolutely free of charge
- Perfect for a non-technical audience

https://mantrainfosec.com/scambook

GROUND RULES

- Please silence your phones
- **Interaction is encouraged** feel free to ask as many questions as you'd like
- The workshop may get intense at times
  - Don't hesitate to stop and ask for a recap if needed

INTRODUCTION TO THE WORKSHOP

- We will cover:
  - The theory behind Unicorn Engine
  - Scripting API
  - Executing code, platform-independently
  - Mapping memory
  - Passing parameters to functions
  - Debugging issues with our scripts

INTRODUCTION TO THE WORKSHOP

- Prerequisites
  - Proficiency in a programming language
  - Familiarity with Assembly language
  - Competence in using Linux operating systems
  - A Github account
  - JDK & Ghidra installed

SOFTWARE REVERSE ENGINEER COURSES

- If you are interested in the full trainings:
  - Software Reverse Engineering training
- Choose your level:
  - Day 01-03: Beginner level (from scratch)
  - Day 04-05: Intermediate level
  - Day 06-10: Advanced level
- Feel free to find me after the workshop for more details.

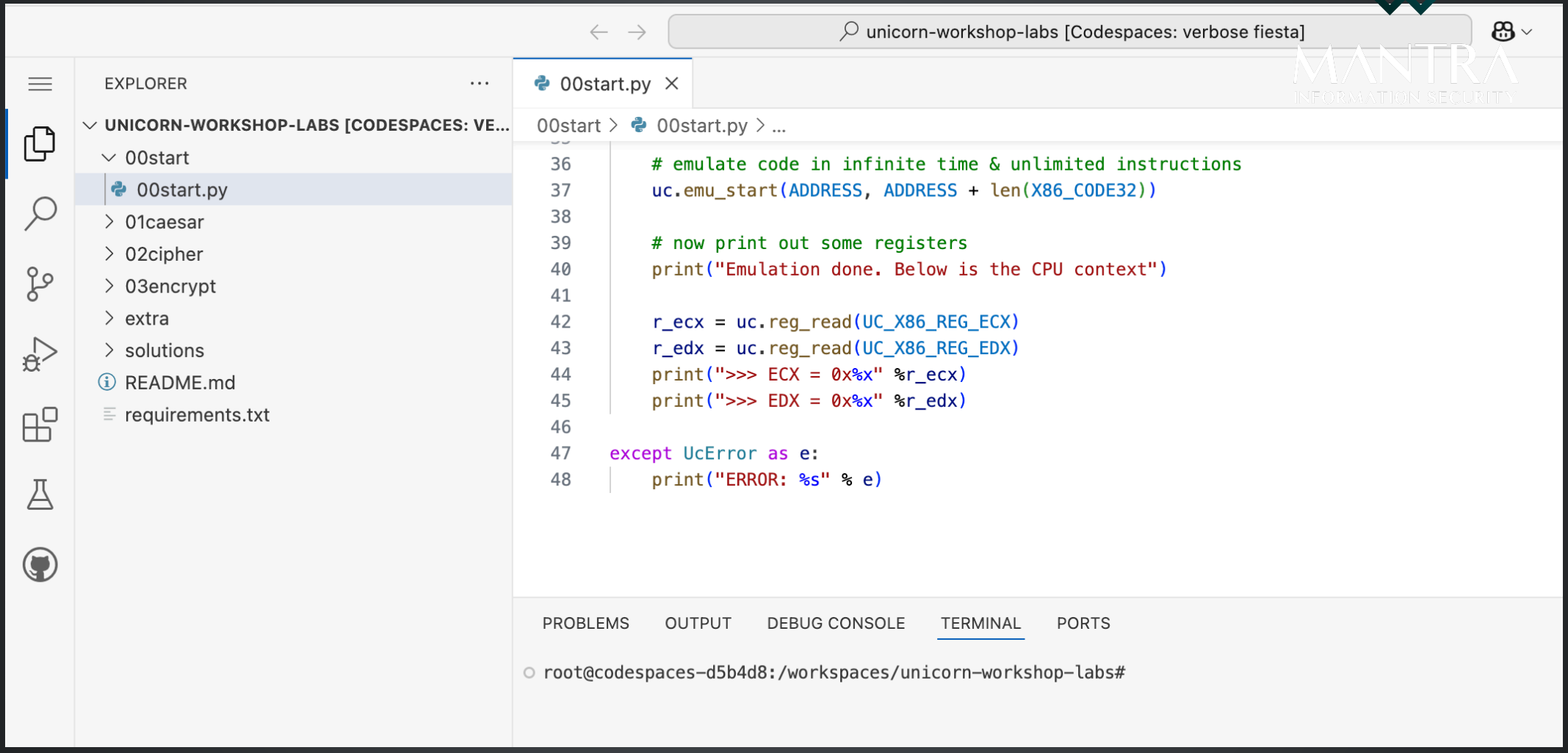
SETUP

- Log into your Github account
- Open the lab's repository:
  - https://github.com/mantrainfosec/unicorn-workshop-labs
  - 

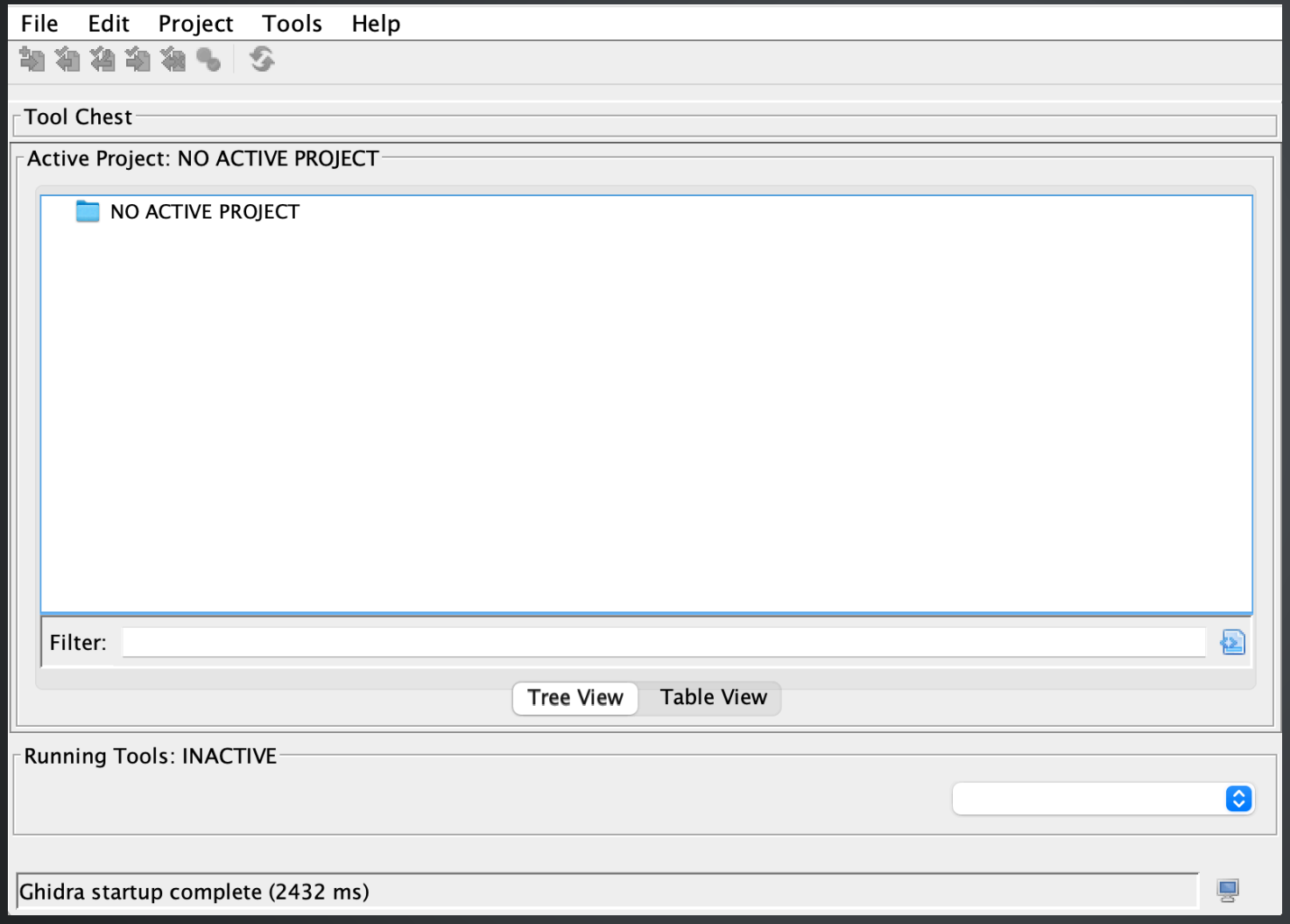
 the repository
  - Click on the 

 button
  - Select the "Codespaces" tab
  - Click on "Create codespace on main"
  - Don't forget to clone it to your computer as well!
  - Download the JDK and Ghidra as outlined in the README
- Ensure you have the slides in PDF format
- Remember to delete your codespace after the workshop!

GHITHUB CODESPACES



GHIDRA



MANTRA
INFORMATION SECURITY

UNDERSTANDING AND FOLLOWING THE MATERIAL

- Reverse Engineering is a complex skill that demands a strong understanding of low-level concepts
- Don't worry if you don't understand everything right away
- Expect plenty of back and forth as we work through the material
- If something is not clear, refer to the slides for clarification (or raise your hands!)
- Feel free to ask questions if you are unsure, rather than falling behind.

UNICORN ENGINE

- Quick theory, then we will dive into hands-on practice, including:
  - Exploring the capabilities of the Unicorn API (Python)
  - Loading and executing code
  - Calling functions
  - Hooking execution
  - Passing function parameters
  - And more.

QEMU

- QEMU is a generic and open-source machine emulator and virtualizer
- Stands for Quick EMUlator
- It can emulate multiple architectures including:
  - x86/x64
  - ARM
  - PowerPC
  - RISC-V
  - ...
- QEMU supports two types of emulation:
  - User-mode emulation: runs a binary with minimal environment
  - System emulation: emulates a entire system including peripherals
- Supports Windows, macOS, Linux and other UNIX operating systems

UNICORN ENGINE

- Next Generation CPU Emulator
- Based on QEMU
- It is capable to emulate code (multiple architecture)
- Provides an API for programming languages to create environments and run code
  - Supported languages include: C, Python, Java, Go, .NET, Rust, ...
- Offers an easy way to execute and debug code

UNICORN ENGINE AS A SOLUTION

- Imagine you have a specific machine code
- This could be part of a program or just a code snippet

- Without the right hardware, how would you execute it?
- Without a full program, how would you run it?
- Unicorn Engine enables you to execute code snippets on ***any*** architecture.

TOOLS TO USE:

- A web browser for Github/codespaces
  - Terminal
  - Text editor
- Disassembler/Decompiler (recommended Ghidra)

- That is all we need

LAB: UNICORN ENGINE INTRO

- Execute the script: python3 00start/00start.py
- Read the code in Github Codespaces
- It creates an x86 environment and executes two instructions
- At the end, it prints the register values
- Let's go through it line by line

LAB: UNICORN ENGINE INTRO

- Imports Unicorn Engine and x86 constants

```
from unicorn import *
from unicorn.x86_const import *
```

LAB: UNICORN ENGINE INTRO

- Creates a binary string with two bytes
- These two bytes are x86 (Intel/AMD) machine code
- INC ECX and DEC EDX

```
X86_CODE32 = b"\x41\x4a" # INC ecx; DEC edx
```

LAB: UNICORN ENGINE INTRO

- Creates a variable, which will be used later as base address

```
ADDRESS = 0x1000000
```

- Just a "random" address

LAB: UNICORN ENGINE INTRO

- This is where the interesting part begins
- x86 architecture emulation is initialised
- 2 Megabytes memory are mapped at the base address

```
uc = Uc(UC_ARCH_X86, UC_MODE_32)
uc.mem_map(ADDRESS, 2 * 1024 * 1024)
```

LAB: UNICORN ENGINE INTRO

- The two instructions are written at the base address
- ECX register is set to **0x1234**
- EDX register is set to **0x7890**

```
uc.mem_write(ADDRESS, X86_CODE32)

uc.reg_write(UC_X86_REG_ECX, 0x1234)
uc.reg_write(UC_X86_REG_EDX, 0x7890)
```

LAB: UNICORN ENGINE INTRO

- Emulation starts at the base address

```
uc.emu_start(ADDRESS, ADDRESS + len(X86_CODE32))
```

- Emulation stops at 2 bytes after, at the end of the code.

LAB: UNICORN ENGINE INTRO

- Register values are saved and printed in Python

```
r_ecx = uc.reg_read(UC_X86_REG_ECX)
r_edx = uc.reg_read(UC_X86_REG_EDX)

print(">>> ECX = 0x%x" %r_ecx)
print(">>> EDX = 0x%x" %r_edx)
```

UNICORN ENGINE INTRO

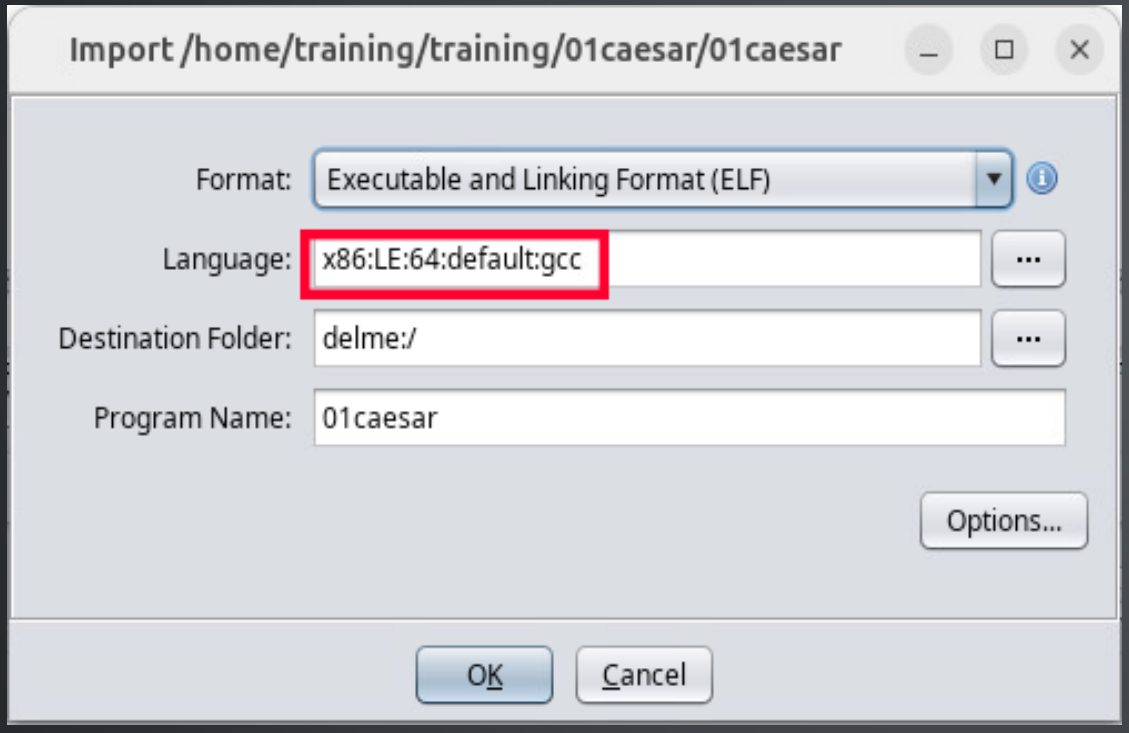
- The two instructions were executed
- The results were printed
- No skeleton program was needed to execute the instructions
- We could write machine code directly in Python and execute it immediately
- While this might seem simple, Unicorn can do so much more!

LAB: UNICORN ENGINE 01CAESAR

- Execute 01caesar/01caesar
  - It prints both the original and encoded strings
- Use Ghidra to take a look at the decompiled code
  - Start Ghidra on your computer: ghidra_XX.X.X_PUBLIC/ghidraRun
- Option 1: Reverse engineer the **caesar_cipher()** function
- Option 2: Research the Caesar cipher (and you should)
- Option 3: Execute the function via Unicorn Engine
- Let's go with Option 3

LAB: UNICORN ENGINE 01CAESAR

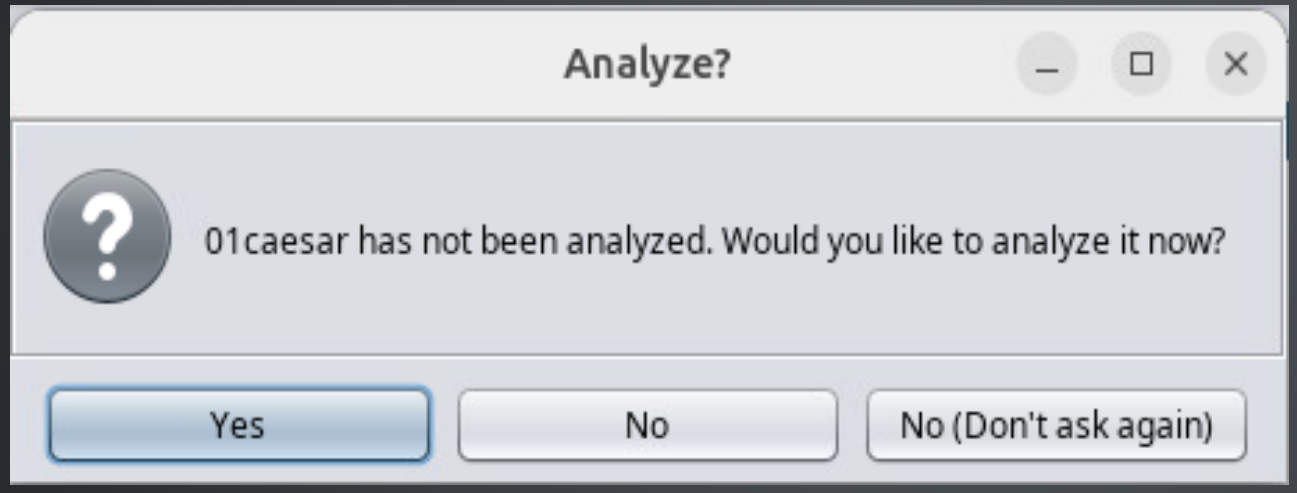
- Open the binary in Ghidra



- Architecture: AMD64/Intel x64

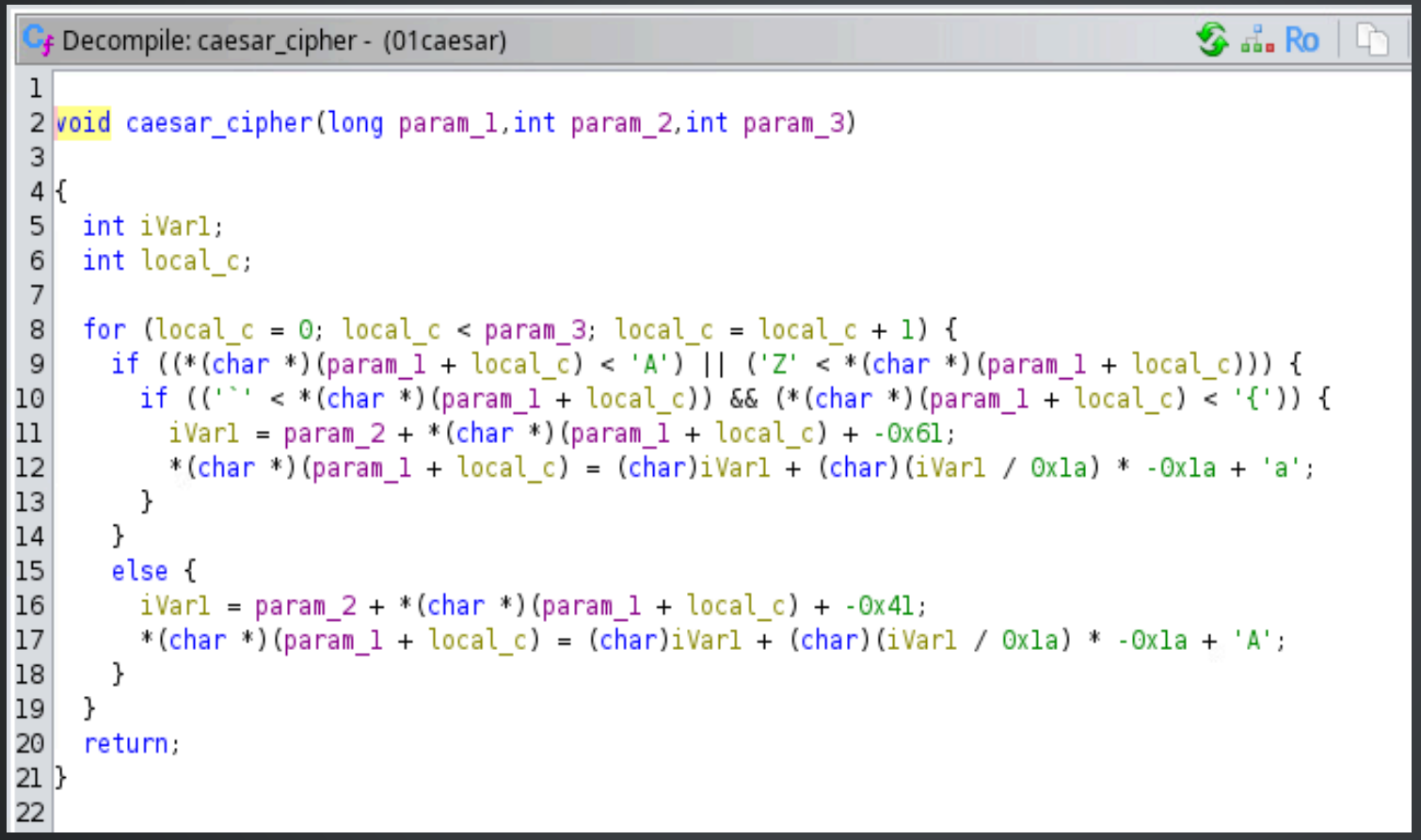
LAB: UNICORN ENGINE 01CAESAR

- Analyze the code with Ghidra



LAB: UNICORN ENGINE 01CAESAR

- Find the function and check the decompiled code:



MANTRA
INFORMATION SECURITY

LAB: UNICORN ENGINE 01CAESAR

- Compare it to the source code under 01caesar/01caesar.c

```
17  void caesar_cipher(char *str, int offset, int length)
18  {
19      for (int i = 0; i < length; ++i)
20      {
21          if (str[i] >= 'A' && str[i] <= 'Z')
22          {
23              str[i] = (str[i] - 'A' + offset) % 26 + 'A';
24          }
25      else if (str[i] >= 'a' && str[i] <= 'z')
26      {
27              str[i] = (str[i] - 'a' + offset) % 26 + 'a';
28          }
29      }
30  }
```

LAB: UNICORN ENGINE 01CAESAR

- Check the assembly code and make a note of the function's address:

```
                        caesar_cipher

00101189 f3 0f 1e fa     ENDBR64
0010118d 55              PUSH        RBP
0010118e 48 89 e5        MOV         RBP,RSP
00101191 48 89 7d e8     MOV         qword ptr [RBP + local_20],RDI
00101195 89 75 e4        MOV         dword ptr [RBP + local_24],ESI
00101198 89 55 e0        MOV         dword ptr [RBP + local_28],EDX
0010119b c7 45 fc        MOV         dword ptr [RBP + local_c],0x0
         00 00 00 00
001011a2 e9 f6 00        JMP         LAB_0010129d
         00 00
```

LAB: UNICORN ENGINE 01CAESAR

MANTRA
INFORMATION SECURITY

- Finally, make a note of the function's end address:

```
00101299 83 45 fc 01     ADD        dword ptr [RBP + local_c],0x1

                      LAB_0010129d
0010129d 8b 45 fc        MOV        EAX,dword ptr [RBP + local_c]
001012a0 3b 45 e0        CMP        EAX,dword ptr [RBP + local_28]
001012a3 0f 8c fe        JL         LAB_001011a7
         fe ff ff
001012a9 90              NOP
001012aa 90              NOP
001012ab 5d              POP        RBP
001012ac c3              RET
```

LAB: UNICORN ENGINE 01CAESAR

- Open the following source in Codespaces: 01caesar/01skeleton.py
- Identify the [HERE] insertion points that need to be figured out
- When properly configured, the code will run and produce the same output as the program
- You can execute the script by running: python3 01caesar/01skeleton.py

LAB: UNICORN ENGINE 01CAESAR

- The insertion points should be filled out in logical order
- This may not necessarily be sequential
- First the binary needs to be read into memory
  - The path of the binary needs to be specified at line 17
- Then, set a base address at line 21
  - This address can be anything, but it's better to have space before and after
  - Let's stick with **0x1000000**
- Stack and Heap at line 22 & 23
  - These address can be anything, make sure these addresses do not overlap
  - Let's stick with **0x2000000** & **0x3000000** respectively

LAB: UNICORN ENGINE 01CAESAR

- Let's think about the memory layout
- We need to load the executable into memory
  - How big is the program?
  - Make a note of the size in Kb
- We need some memory for stack operations
  - Do we need a stack?
  - Check Ghidra, does the assembly interact with the stack?
  - Any ESP/RSP references? PUSH/POP?
- We will need some memory for heap
  - We need to pass a pointer to the function
  - The pointer should point to a string in memory
  - Where do you want to place that string?

LAB: UNICORN ENGINE 01CAESAR

- Filesize to be loaded

```
training@unicorn:~/training/01caesar$ ls -la
total 36
drwxrwxr-x 2 training training  4096 Oct  1 15:01 .
drwxrwxr-x 8 training training  4096 Sep  8 19:42 ..
-rwxrwxr-x 1 training training 16088 Sep  8 17:40 01caesar
-rw-rw-r-- 1 training training  1097 Sep  6 11:12 01caesar.c
-rw-rw-r-- 1 training training  1340 Sep  6 11:14 01skeleton.py
-rw-rw-r-- 1 training training    28 Sep  8 17:49 secret.txt
```

LAB: UNICORN ENGINE 01CAESAR

MANTRA
INFORMATION SECURITY

- Stack usage (POP/PUSH; RSP, RBP references)

```
                        caesar_cipher

00101189 f3 0f 1e fa     ENDBR64
0010118d 55              PUSH       RBP
0010118e 48 89 e5        MOV        RBP,RSP
00101191 48 89 7d e8     MOV        qword ptr [RBP + local_20],RDI
00101195 89 75 e4        MOV        dword ptr [RBP + local_24],ESI
00101198 89 55 e0        MOV        dword ptr [RBP + local_28],EDX
0010119b c7 45 fc        MOV        dword ptr [RBP + local_c],0x0
         00 00 00 00
```

LAB: UNICORN ENGINE 01CAESAR

- The memory layout we build should look something like this:

Address	Region
0x0100000	Program code starts
[...]	
0x0103FFF	Program ends starts
[...]	
0x0400000	Stack starts
[...]	
0x04FFFFF	Stack ends
[...]	
0x0500000	Heap* starts
[...]	
0x05FFFFF	Heap* ends

LAB: UNICORN ENGINE 01CAESAR

- Allocate/map more memory than you need (line 29, 30, 31)
  - Calculate it in bytes, 2MB should be enough
- Write the **string argument** to the **heap**, calculate address (line 40)
- Use the **same address** for reading at the end of the program (line 52)
- Look up the corresponding calling convention
  - What is used for the **first argument**? (line 41)
  - What is used for the **second argument**? (line 42)
  - What is used for the **third argument**? (line 43)
- Set the arguments (pointer for string, value for integers) (line 41, 42, 43)

LAB: UNICORN ENGINE 01CAESAR

- Calling Convention: **System V AMD64 ABI**

Argument register overview

Argument type	Registers
Integer/pointer arguments 1-6	RDI, RSI, RDX, RCX, R8, R9
Floating point arguments 1-8	XMM0 - XMM7
Excess arguments	Stack
Static chain pointer	R10

LAB: UNICORN ENGINE 01CAESAR

- Calculate the **stack** address and set the correct register (line 45)
- Use Ghidra to find the function's start and end addresses
  - Locate the **first instruction** of the function (line 49)
  - Locate the **last instruction** of the function (line 49)
- Note: The stack grows and shrinks with each instruction
  - Be aware that references might fall outside the stack bounds (eg. **EBP-0x10**)
- Protip: Set ESP/RSP to somewhere near the middle of the stack memory range.

LAB: UNICORN ENGINE 01CAESAR

- Run your script, did it print the correct string?
- Try changing the input string to something else
- You can run the function standalone without modifying the binary!
- This is especially useful if you don't want to reimplement the code

LAB: UNICORN ENGINE 01CAESAR

- All done? Well done!

```
training@unicorn:~/training/01caesar$ python3 01caesar.py
Encoded string: Khoor Xqlfruq!
training@unicorn:~/training/01caesar$ cat secret.txt
Jxkqox Fkclojxqflk Pbzrofqv
```

- Now, try to decode the content of secret.txt

- Having issues? Let's solve them together!

MANTRA
INFORMATION SECURITY

UNICORN ENGINE ERRORS

- Unicorn Engine might throw an error depending on the issue we encounter
- If we dereference memory that was not mapped to the application:
  - Invalid memory write (UC_ERR_WRITE_UNMAPPED)
  - Invalid memory read (UC_ERR_READ_UNMAPPED)
  - Invalid memory fetch (UC_ERR_FETCH_UNMAPPED)
- To avoid these errors, ensure:
  - Enough memory was mapped
  - The correct address is used

UNICORN ENGINE HOOKS

- Hooks are special functions that can be registered before execution
- These functions are triggered at different scenarios:
  - **UC_HOOK_CODE** - Called on every instruction within a specified range
  - **UC_HOOK_MEM_*** - Triggered on memory-related actions
  - **UC_HOOK_BLOCK** - Invoked when a new block starts executing
  - **UC_HOOK_INSN** - Fires for a particular instruction
  - **UC_HOOK_INTR** - handles interrupts and syscalls

UNICORN ENGINE HOOKS

- Hooks need to be added **before** **emu_start()** is called
- Each hook follows a specific format::
  - **UC_HOOK_BLOCK** and **UC_HOOK_CODE**:

```
def hook_func(uc, address, size, user_data)
```

- Where:
  - **uc**: Initialized Unicorn instance
  - **address**: Current instruction address
  - **size**: Instruction or data size
  - **user_data**: Optional user data

UNICORN ENGINE HOOKS

- Memory hooking format:
  - **UC_HOOK_MEM_***:

```
def hook_mem_access(uc, access, address, size, value, user_data)
```

- Where:
  - ...
  - **access**: Access type (READ/WRITE/...)
  - **value**: Data value

LAB: UNICORN ENGINE MEMORY HOOK

- Let's add a memory hook that prints the access type and address
- Add this code **before** emulation being started:

```
uc.hook_add(UC_HOOK_MEM_WRITE, hook_mem_access)
uc.hook_add(UC_HOOK_MEM_READ,  hook_mem_access)
```

- Add this function **at the beginning** of the file:

```
def hook_mem_access(uc, access, address, size, value, user_data):
  print("[*] Memory access: {} at 0x{}, data size = {},
        data value = 0x{}"
  .format(access, address, size, value))
```

LAB: UNICORN ENGINE MEMORY HOOK

- Output:

```
[*] Memory access: 16 at 0x1064973, data size = 1, data value = 0x0
[*] Memory access: 16 at 0x1069044, data size = 4, data value = 0x0
[*] Memory access: 16 at 0x1069024, data size = 8, data value = 0x0
[*] Memory access: 16 at 0x1064973, data size = 1, data value = 0x0
[*] Memory access: 16 at 0x1069044, data size = 4, data value = 0x0
[*] Memory access: 17 at 0x1069044, data size = 4, data value = 0x14
[*] Memory access: 16 at 0x1069044, data size = 4, data value = 0x0
[*] Memory access: 16 at 0x1069016, data size = 4, data value = 0x0
[*] Memory access: 16 at 0x1069048, data size = 8, data value = 0x0
Encoded string: Khoor Xqlfruq!
training@re-training-linux:~/labs/day08/lab$
```

LAB: UNICORN ENGINE CODE HOOK

- Let's add a code hook that prints the **EIP/RIP** at every instruction
- Add this code **before** emulation being started:

```
uc.hook_add(UC_HOOK_CODE, hook_code, None,
  ADDRESS + 0x1189, ADDRESS + 0x12AC)
```

- Add this function **at the beginning** of the file:

```
def hook_code(uc, address, size, user_data):
  print("[*] Current RIP: 0x{}, instruction size = 0x{}"
  .format(address, size))
```

LAB: UNICORN ENGINE CODE HOOK

- Output:

```
[*] Current RIP: 0x1053230, instruction size = 0x3
[*] Current RIP: 0x1053233, instruction size = 0x2
[*] Current RIP: 0x1053235, instruction size = 0x2
[*] Current RIP: 0x1053337, instruction size = 0x4
[*] Current RIP: 0x1053341, instruction size = 0x3
[*] Current RIP: 0x1053344, instruction size = 0x3
[*] Current RIP: 0x1053347, instruction size = 0x6
[*] Current RIP: 0x1053353, instruction size = 0x1
[*] Current RIP: 0x1053354, instruction size = 0x1
[*] Current RIP: 0x1053355, instruction size = 0x1
Encoded string: Khoor Xqlfruq!
training@re-training-linux:~/labs/day08/lab$
```

LAB: UNICORN ENGINE BASICS

- At this point, we are capable to:
  - Create a Python script to use Unicorn Engine
  - Set and read registers values
  - Execute selected parts of the code
  - Hook into the execution
  - Debug our own script

LAB: UNICORN ENGINE 02CIPHER

- Let's tackle a new challenge!
- This time, it's a substitution cipher
- Execute the binary: 02cipher/02cipher
- Load it into Ghidra and analyze it!
- If you're curious, check out the source code: 02cipher/02cipher.c

LAB: UNICORN ENGINE 02CIPHER

- Find and decompile the main() function
- Retype uStack_78 (name might differ) to char[100]
- Make a note of the **strings** and copy them

LAB: UNICORN ENGINE 02CIPHER

- Find the substitutionCiper() function
  - How many arguments does it have?
  - What are its start and end address?
  - Does it use stack? Heap?
- Repeat the same steps for substitutionDecipher()

LAB: UNICORN ENGINE 02CIPHER

- Now that we have all the details we need, try to:
  - create a script for substitutionCiper() and the first string
  - print the output and compare it to the binary's output
  - create another script for substitutionDeciper() and 2nd string
- Feel free to use 02skeleton.py!

LAB: UNICORN ENGINE 03ENCRYPT

- Let's tackle another new challenge!
- This time, it's AES encryption (OpenSSL)
- Execute the binary: 03encrypt/03encrypt testfile
- It generated and printed its AES key and created the following file:
  - output_file.encrypted
  - Check its content with xxd!

LAB: UNICORN ENGINE 03ENCRYPT

- Our task is to decrypt the secret.enc file
- To speed up the process:
  - We can read the encryptor's source: 03encrypt.c
  - We can modify the 03skeleton-getkey.py source (to generate key)
  - We can modify the 03skeleton-decrypt.py source (to decrypt file)

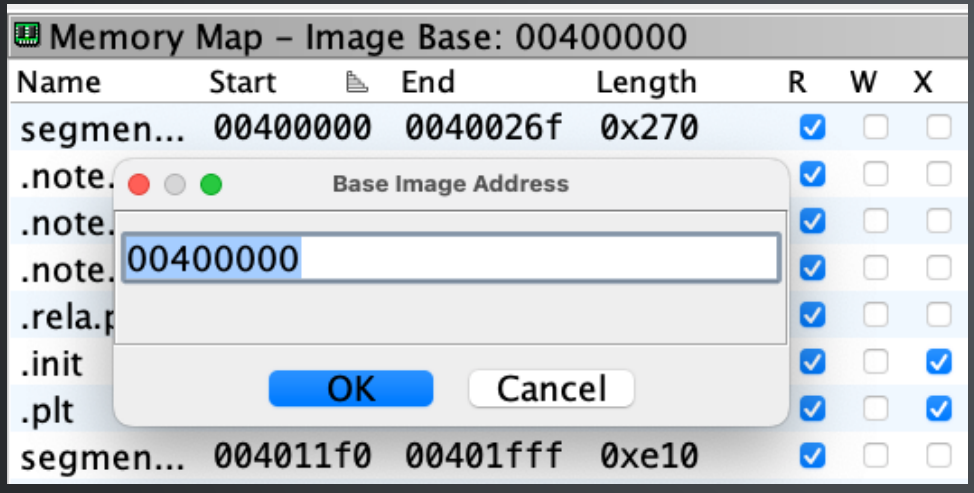
LAB: UNICORN ENGINE 03ENCRYPT

- The content of the encrypted file looks like this:
  - byte 0..7 - Seed
  - byte 8..X - Encrypted content
- The seed is used to generate the key
- Let's locate the key generation function

LAB: UNICORN ENGINE 03ENCRYPT

MANTRA
INFORMATION SECURITY

- Use the Display Memory Map to get the Base Image Address



LAB: UNICORN ENGINE 03ENCRYPT

- Make a note of the IV:

```
argc_local = argc;
argv_local = argv;
local_20 = *(long *)(in_FS_OFFSET + 0x28);
keySize = 0x10;
iv = "AAAABBBECCCCDDDD";
local_38 = 0x1;
```

LAB: UNICORN ENGINE 03ENCRYPT

- Function used to generate the key:

```
puVar3 = aesKey;
keySize_00 = keySize;
seed = seed_00;
*(undefined8 *)((long)pppcVar10 + -0x18) = 0x10184a;
generateAESKey(seed_00,puVar3,keySize_00);
*(undefined8 *)((long)pppcVar10 + -0x18) = 0x10185e;
printf("Generated AES Key: ");
```

LAB: UNICORN ENGINE 03ENCRYPT

- Find the generateAESKey() function
  - How many arguments does it take?
  - What are the start and end addresses?
  - Does it use stack? Or heap?
- What is the size of the binary?

LAB: UNICORN ENGINE 03ENCRYPT

- Now that we have all the details we need, try to:
  - use skeleton script for generateAESKey()
  - the seed should come from the encrypted file
  - print the key after the function returns

LAB: UNICORN ENGINE 03ENCRYPT

- After the key is generated:
  - use the 03skeleton-decrypt.py source to write your decryptor
  - specify the IV and the key in the source

FURTHER READING AND RESOURCES

- Tutorial for Unicorn
- Unicorn Engine Notes
- Unicorn Engine tutorial
- Unicorn Engine Reference (Unofficial)

AFTER THE WORKSHOP

- Well done! You've successfully covered the workshop material
- If you'd like to take your learning further:
  - Continue with the following slides and explore more challenges
  - Join us for the full SRE training: Software Reverse Engineering Training
  - Follow us on twitter: @MantraInfoSec

LAB: UNICORN ENGINE + CAPSTONE DISASSEMBLER

- Can you create a disassembler?
- Sounds more complex than it is!
- Capstone is a lightweight multi-architecture disassembly framework
- It provides an API for C, Python, Java, PowerShell, Rust, etc...
- Combining it with Unicorn Engine makes it really powerful
- Use 01caesar and 01caesar.py for this

LAB: UNICORN ENGINE + CAPSTONE DISASSEMBLER

MANTRA
INFORMATION SECURITY

- First capstone needs to be imported:

```
from capstone import *
```

- A Capstone instance needs to be created:

```
capmd = Cs(CS_ARCH_X86, CS_MODE_64)
```

LAB: UNICORN ENGINE + CAPSTONE DISASSEMBLER

- The following function needs to be added and the hook replaced:

```
def disas_single(data, address):
  for i in capmd.disasm(data, address):
    print("0x%x:\t%s\t%s" % (i.address, i.mnemonic, i.op_str))

def hook_code(uc, address, size, user_data):
  print("[*] Current RIP: 0x{}, instruction size = 0x{}".
    format(address, size))
  mem = uc.mem_read(address, size)
  disas_single(bytes(mem), address)
```

LAB: UNICORN ENGINE + CAPSTONE DISASSEMBLER

- Output:

```
[*] Current RIP: 0x1053217, instruction size = 0x3
0x101221:       mov     eax, dword ptr [rbp - 4]
[*] Current RIP: 0x1053220, instruction size = 0x3
0x101224:       movsxd  rdx, eax
[*] Current RIP: 0x1053223, instruction size = 0x4
0x101227:       mov     rax, qword ptr [rbp - 0x18]
[*] Current RIP: 0x1053227, instruction size = 0x3
0x10122b:       add     rax, rdx
[*] Current RIP: 0x1053230, instruction size = 0x3
0x10122e:       movzx   eax, byte ptr [rax]
[*] Current RIP: 0x1053233, instruction size = 0x2
```

LAB: OBFUSCATED CIPHER: UNICORN

- Find your target under: extra/05cipher-unicorn/05cipher-unicorn
- Executable deciphers a ciphered message
- Create a Unicorn script that deciphers the message from decipher.this
- External function might cause an issue. Patch them from Unicorn!
- Use **mem_write()** and assemblers to create machine code


MANTRA
INFORMATION SECURITY
MANTRA
INFORMATION SECURITY
Join us for the full SRE training:
Software Reverse Engineering Training
https://mantrainfosec.com