DEFEATING ENCRYPTION BY USING UNICORN ENGINE WORKSHOP

MANTRA
INFORMATION SECURITY

Balázs Bucsay

Founder & CEO of
Mantra Information Security

https://mantrainfosec.com

BIO / BALÁZS BUCSAY

MANTRA
INFORMATION SECURITY

- Trainer of this course
- Originally from Hungary, living in London
- Over two decades of offensive security experience
- Started learning assembly at the age of 13
- Software Reverse Engineer
- 15 years of research and consultancy
- Certifications: OSCE, OSCP, OSWP; Prev: GIAC GPEN, CREST CCT Inf

BIO / BALÁZS BUCSAY

MANTRA
INFORMATION SECURITY

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

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

BIO / BALÁZS BUCSAY

MANTRA
INFORMATION SECURITY

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

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

MANTRA INFORMATION SECURITY

MANTRA
INFORMATION SECURITY

- Boutique consultancy approach
- Decades of experience and excellence
  - Training delivery (Software Reverse Engineering training)
  - 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 finding a bug until it gets fixed

https://mantrainfosec.com

GROUND RULES

- Please silence your phones
- Take phone calls outside of the room and preferably in breaks
- Interaction is encouraged, please ask as many questions as you'd like
- The workshop might be heavy at some points, let's stop and recap what is missing

INTRODUCTION TO THE WORKSHOP

- We are going to learn:
  - A bit of theory behind Unicorn Engine
  - How to script the API
  - How to execute code platform independently
  - How to map memory
  - How to pass parameters to functions
  - How to debug issues with our scripts

MANTRA
INFORMATION SECURITY

INTRODUCTION TO THE WORKSHOP

- Prerequisite
  - Proficiency in coding in a programming language
  - Familiarity with Assembly language
  - Competence in using Linux operating systems
  - A computer capable of x64 virtualization
  - VMWare Player installed on their computer

MANTRA
INFORMATION SECURITY

SOFTWARE REVERSE ENGINEER COURSES

MANTRA
INFORMATION SECURITY

- In case you are interested in the full training:
  - Software Reverse Engineering training
- Multiple courses (pick your level):
  - Day 01-03: Beginner level (from scratch)
  - Day 04-05: Intermediate level
  - Day 06-10: Advanced level
- Find me after the workshop

SETUP

- Install VMWare Player
  - Distributed with other materials
  - Next, next, finish - install MS VC Redistributable if required
  - Reboot if required
  - Select FREE option - Non-commercial use only
- Open Virtual Machine in VMWare Player
- Select Linux and click on "Play virtual machine"
- Make sure you have the slides in PDF format
- Make sure you click on COPIED not moved if VMWare Player asks

MANTRA
INFORMATION SECURITY

UBUNTU LINUX VM

- Ubuntu Desktop VM, FOSS
- All necessary tools installed for this workshop
- Challenges and solutions are also on the VM
- Feel free to use this VM after the course for as long as you want

VIRTUAL MACHINE SECURITY

MANTRA
INFORMATION SECURITY

- Credentials to log into the VMs:
  - Username: training
  - Password: training
- Feel free to change the password - make sure you remember it
- The network interface is set to NAT - no incoming connections
- Vanilla configuration, not hardened, might need security updates as well
- Please do not update during the workshop - could block you

UNDERSTANDING AND FOLLOWING THE MATERIAL

- Reverse Engineering is a complex skill that requires low-level knowledge
- Don't worry if you don't get everything for the first time
- Lots of back and forth
- Check the slides if you need to clear-up something
- Feel free to ask questions instead of lagging behind

UNICORN ENGINE

- Quick theory and then we start with the real deal
- Including:
  - Learning the capabilities of the Unicorn API (Python)
  - Loading and running code
  - Calling functions
  - Hooking execution
  - Passing function parameters
  - etc.

QEMU

- QEMU is a generic and open-source machine emulator and virtualizer
- Stands for Quick EMUlator
- It is capable to emulate multiple other architectures including:
  - x86/x64
  - ARM
  - PowerPC
  - RISC-V
  - ...
- User-mode emulation: runs a binary, emulates with minimal environment
- System emulation: emulates a whole 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 an environment and run code
  - Supports: C, Python, Java, Go, .NET, Rust, ...
- Easy way to execute and debug code

UNICORN ENGINE AS A SOLUTION

- Think of a scenario where you have a specific machine code
- This might be part of a program or just a snippet of code
- Without having the right hardware, how would you execute it?
- Without having a skeleton program, how would you execute it?
- Unicorn Engine allows to execute snippets on ***any*** architecture

MANTRA
INFORMATION SECURITY

TOOLS TO USE:

- Text editor (recommended Sublime)
- Terminal (recommended Terminator)
- Disassembler/Decompiler (recommended Ghidra)

- That is all we need

LAB: UNICORN ENGINE INTRO

- Execute the script: python3 ~/training/00start/00start.py
- Read the code in Sublime
- It creates an x86 environment and executes two instructions
- At the end, it prints the register values
- Let's take a look line by line

LAB: UNICORN ENGINE INTRO

- Imports Unicorn Engine and x86 constants

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

MANTRA
INFORMATION SECURITY

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
```

MANTRA
INFORMATION SECURITY

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 starts
- x86 architecture emulation is initialised
- 2 Megabyte memory is mapped at base address

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

MANTRA
INFORMATION SECURITY

LAB: UNICORN ENGINE INTRO

- The two instructions are written to 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 base address

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

- Emulation stops at the end of the code (2 bytes later)

LAB: UNICORN ENGINE INTRO

- Registers 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 executed
- The results were printed
- It did not need a skeleton program to execute the instructions
- We could write machine code directly in Python and execute it right away
- This might not seem to be useful, but Unicorn can do so much more!

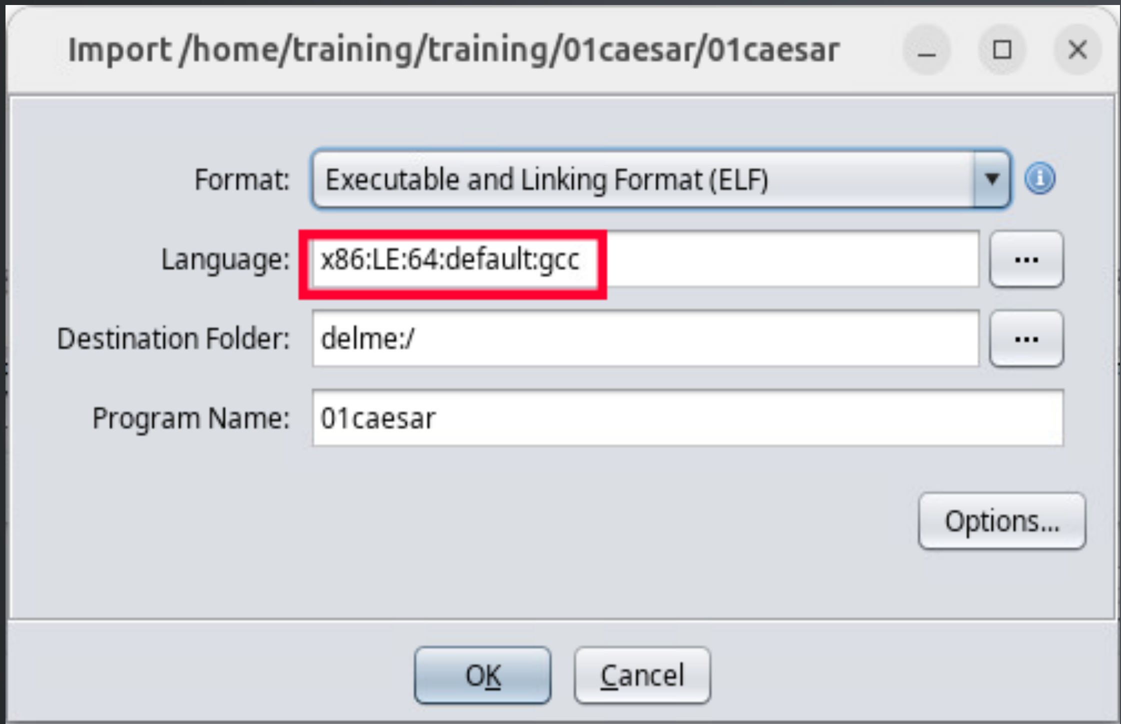
LAB: UNICORN ENGINE 01CAESAR

MANTRA
INFORMATION SECURITY

- Execute ~/training/01caesar/01caesar
- It prints the original and encoded string
- Use Ghidra to take a look at the decompiled code
  - Execute: ~/training/tools/ghidra_11.1.2_PUBLIC/ghidraRun
- Option 1: You could RE the **caesar_cipher()** function
- Option 2: You could look up what Caesar cipher is (and you should)
- Option 3: You could execute this function via Unicorn engine
- Let's go for the option 3

LAB: UNICORN ENGINE 01CAESAR

- Open the binary in Ghidra

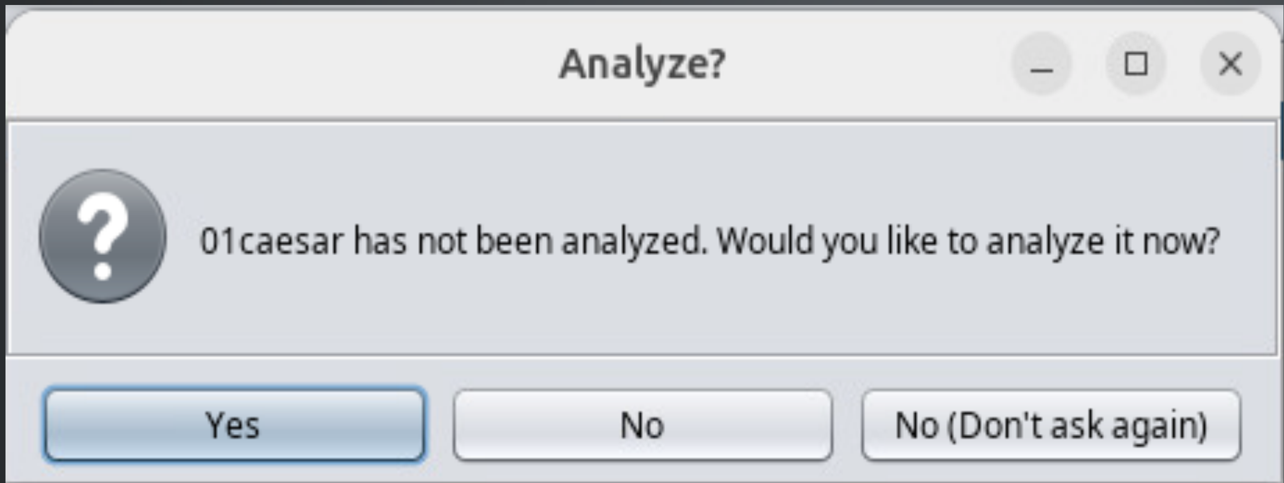


- Architecture: AMD64/Intel x64

MANTRA
INFORMATION SECURITY

LAB: UNICORN ENGINE 01CAESAR

- Analyze the code with Ghidra



LAB: UNICORN ENGINE 01CAESAR

- Find the function and check the decompiled code:

Decompile: caesar_cipher - (01caesar)

```c

void caesar_cipher(long param_1,int param_2,int param_3)

{
  int iVar1;
  int local_c;

  for (local_c = 0; local_c < param_3; local_c = local_c + 1) {
    if ((*(char *)(param_1 + local_c) < 'A') || ('Z' < *(char *)(param_1 + local_c))) {
      if (('`' < *(char *)(param_1 + local_c)) && (*(char *)(param_1 + local_c) < '{')) {
        iVar1 = param_2 + *(char *)(param_1 + local_c) + -0x61;
        *(char *)(param_1 + local_c) = (char)iVar1 + (char)(iVar1 / 0x1a) * -0x1a + 'a';
      }
    }
    else {
      iVar1 = param_2 + *(char *)(param_1 + local_c) + -0x41;
      *(char *)(param_1 + local_c) = (char)iVar1 + (char)(iVar1 / 0x1a) * -0x1a + 'A';
    }
  }
  return;
}

```

MANTRA INFORMATION SECURITY

LAB: UNICORN ENGINE 01CAESAR

MANTRA
INFORMATION SECURITY

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

```c
void caesar_cipher(char *str, int offset, int length)
{
    for (int i = 0; i < length; ++i)
    {
        if (str[i] >= 'A' && str[i] <= 'Z')
    {
            str[i] = (str[i] - 'A' + offset) % 26 + 'A';
        }
    else if (str[i] >= 'a' && str[i] <= 'z')
    {
            str[i] = (str[i] - 'a' + offset) % 26 + 'a';
        }
    }
}
```

LAB: UNICORN ENGINE 01CAESAR

- Finally, check the assembly code. 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
```

MANTRA
INFORMATION SECURITY

LAB: UNICORN ENGINE 01CAESAR

- 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 Sublime: ~/training/01caesar/01skeleton.py
- Identify the [HERE] insertion points that you need to figure out
- When properly configured, the code will run and print the same as the program does
- You can execute the script by: python3 ~/training/01caesar/01skeleton.py

LAB: UNICORN ENGINE 01CAESAR

MANTRA INFORMATION SECURITY

- The insertion points should be filled out in logical order
- This might not be sequential at all
- First the binary needs to be read into memory
  - The path of the binary needs to be specified at line 6
- Then set a base address at line 10
  - This address can be anything, but it's better to have space before and after
  - Let's stick to 0x100000

LAB: UNICORN ENGINE 01CAESAR

MANTRA
INFORMATION SECURITY

- 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 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 put 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

- 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

MANTRA
INFORMATION SECURITY

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

Address	Region
0x0100000	Program code starts
[...]	
0x0103FFF	Program ends starts
0x0104000	Heap* starts
[...]	
0x0104FFF	Heap* ends
0x0105000	Stack starts
[...]	
0x0105FFF	Stack ends

LAB: UNICORN ENGINE 01CAESAR

MANTRA
INFORMATION SECURITY

- Allocate/map more memory than you need (line 16)
  - Calculate it in bytes, 2Mb should be enough
- Write the **string argument** to the **heap**, calculate address (line 25)
- Use the **same address** to read at the end of the program (line 37)
- Look up the corresponding calling convention
  - What is used for the **first argument**? (line 26)
  - What is used for the **second argument**? (line 27)
  - What is used for the **third argument**? (line 28)
- Set the arguments (pointer for string, value for integers) (line 26, 27, 28)

LAB: UNICORN ENGINE 01CAESAR

MANTRA
INFORMATION SECURITY

- 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 address for **stack** and set the right register (line 30)
- Look up the start and end addresses from Ghidra
  - Find the **first instruction** of the function (line 34)
  - Find the **last instruction** of the function (line 34)
- Note: Stack grows and shrinks by instruction
  - It might happen that a reference points outside of stack (eg. EBP-0x10)
- Protip: Set ESP/RSP in the middle of the stack memory range

LAB: UNICORN ENGINE 01CAESAR

- Run your script, did it print the right string?
- Change your input string to something else
- You can run the function standalone without modifying the binary!
- This comes really handy if you do not 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
- Any issues? Let's solve them together!

UNICORN ENGINE ERRORS

MANTRA
INFORMATION SECURITY

- Unicorn Engine might throw an error depending the issue we have
- In case 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)
- Make sure:
  - Enough memory was mapped
  - Right address was used

UNICORN ENGINE HOOKS

MANTRA INFORMATION SECURITY

- Hooks are special functions that can be registered before execution
- These functions are called at different scenarios:
  - UC_HOOK_CODE - Every instruction in a range
  - UC_HOOK_MEM_* - Memory related action
  - UC_HOOK_BLOCK - New block
  - UC_HOOK_INSN - Particular instruction
  - UC_HOOK_INTR - Interrupts and syscalls

UNICORN ENGINE HOOKS

MANTRA
INFORMATION SECURITY

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

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

- Where:
  - **uc**: initialized instance
  - **address**: current address
  - **size**: instruction or data size
  - **user_data**: optional user data

UNICORN ENGINE HOOKS

MANTRA
INFORMATION SECURITY

- 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

MANTRA
INFORMATION SECURITY

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

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

- Add this function to 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 starts:

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

- Add this function to 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$
```

MANTRA
INFORMATION SECURITY

LAB: UNICORN ENGINE BASICS

- We are capable to:
  - Create a Python script to use Unicorn Engine
  - Set and read registers values
  - Execute selected parts of code
  - Hook the code
  - Debug our own script

LAB: UNICORN ENGINE 02CIPHER

- Let's tackle a new challenge!
- This time, it's a substitution cipher
- Execute the binary: ~/training/02cipher/02cipher
- Load it into Ghidra and analyze it!
- If you're curious, check out the source code: ~/training/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 is the start and end address?
  - Does it use stack? Heap?
- Repeat this for the substitutionDecipher() as well

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

MANTRA
INFORMATION SECURITY

- Let's tackle another new challenge!
- This time, it's AES encryption (OpenSSL)
- Execute the binary: ~/training/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.py source (generate key)
  - We can modify the 03decrypt-skeleton.c source (decrypt file)

LAB: UNICORN ENGINE 03ENCRYPT

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

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 = 0xf;
```

LAB: UNICORN ENGINE 03ENCRYPT

MANTRA INFORMATION SECURITY

- 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 have?
  - What is the start and end address?
  - Does it use stack? 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()
  - seed should come from the encrypted file
  - print the key after the function returned

LAB: UNICORN ENGINE 03ENCRYPT

- After the key is generated:
  - use the 03decrypt-skeleton.c source to write your decryptor
  - specify the IV and the key in the source
  - compile it according to the instruction in the file header

```
// compile: gcc 03decrypt-skeleton.c -o 03decrypt -lssl -lcrypto
```

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

MANTRA
INFORMATION SECURITY

LAB: UNICORN ENGINE + CAPSTONE DISASSEMBLER

- Can you create a disassembler?
- Sounds more complex than it is
- Capstone is a lightweight multi-architecture disassembly framework
- 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

- 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

MANTRA
INFORMATION SECURITY

- 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

MANTRA
INFORMATION SECURITY

- 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: ~/training/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