BlackBerry. Intelligent Security. Everywhere.

THE POWER AND PERILS OF BINARY EMULATION FOR MALWARE ANALYSIS

Anuj Soni
Principal Threat Researcher

But first…

- BlackBerry still exists
- I do not use a BlackBerry phone
- Malware Reverse Engineer
- SANS Author and Instructor
- Wannabe YouTuber
- Unicorn dad (3)

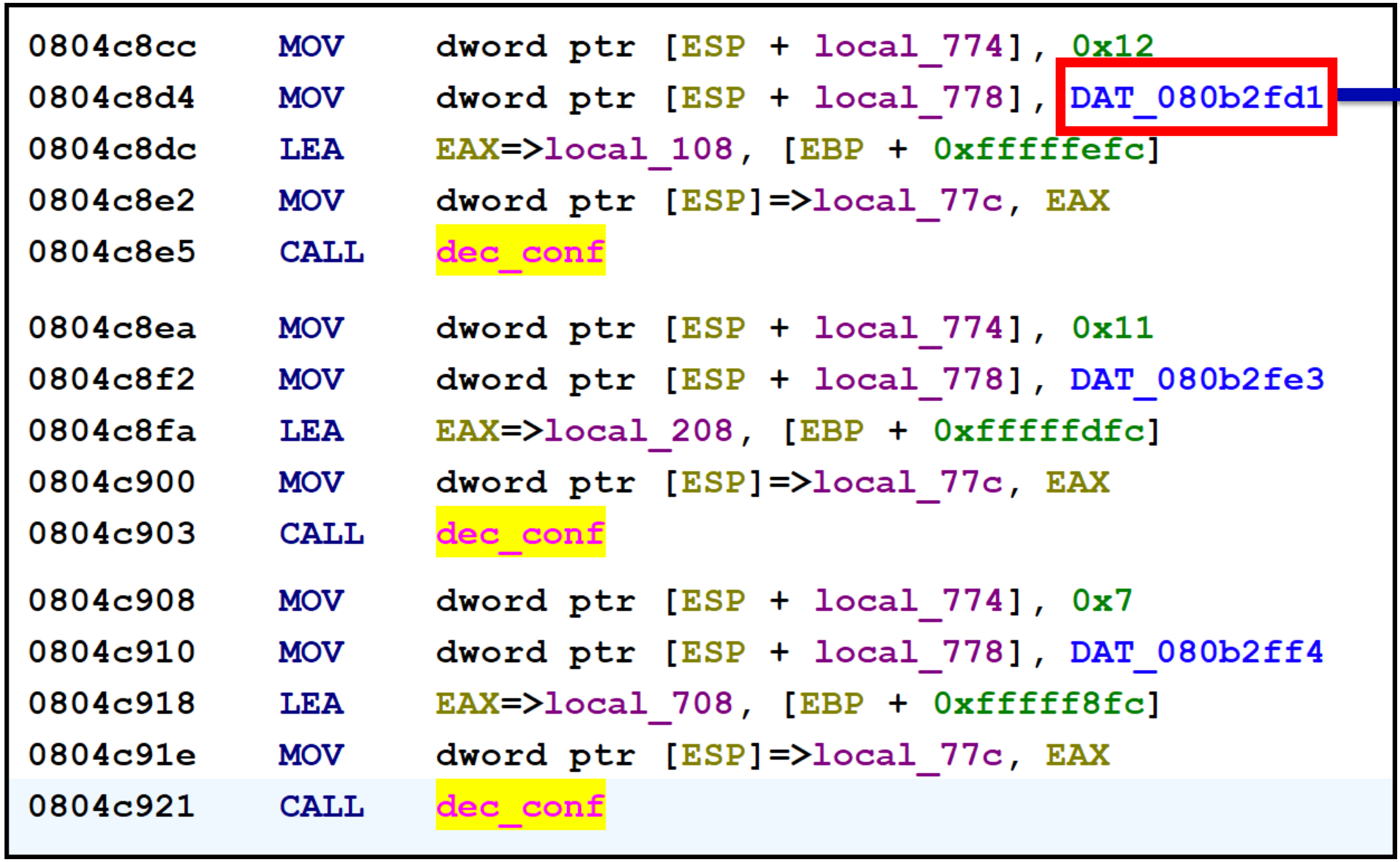
BlackBerry
BlackBerry Curve 3G 9300
The BlackBerry Curve 3G 9300, with original OS version 6.0, is a mid-range smartphone developed by Research in Motion and was the first Curve device to function on 3G networks.
FRONT COVER
LOUD SPEAKER
EARPIECE SPEAKER
KEYBOARD
LOGIC BOARD
BACK COVER
MICRO USB
BATTERY CONNECTOR
320 X 240 LCD SCREEN DISPLAY
TRACKPAD
CAMERA
HEADPHONE SOCKET
The 9300, and its CDMA counterpart, the Curve 9330, also contained some minor improvements over the 8500 series including a faster processor and the ability to upgrade the OS to BlackBerry 6. RIM BlackBerry Curve 3G is an addition to the popular BlackBerry Curve series of smartphones.

Buckle Up or Bail Out?

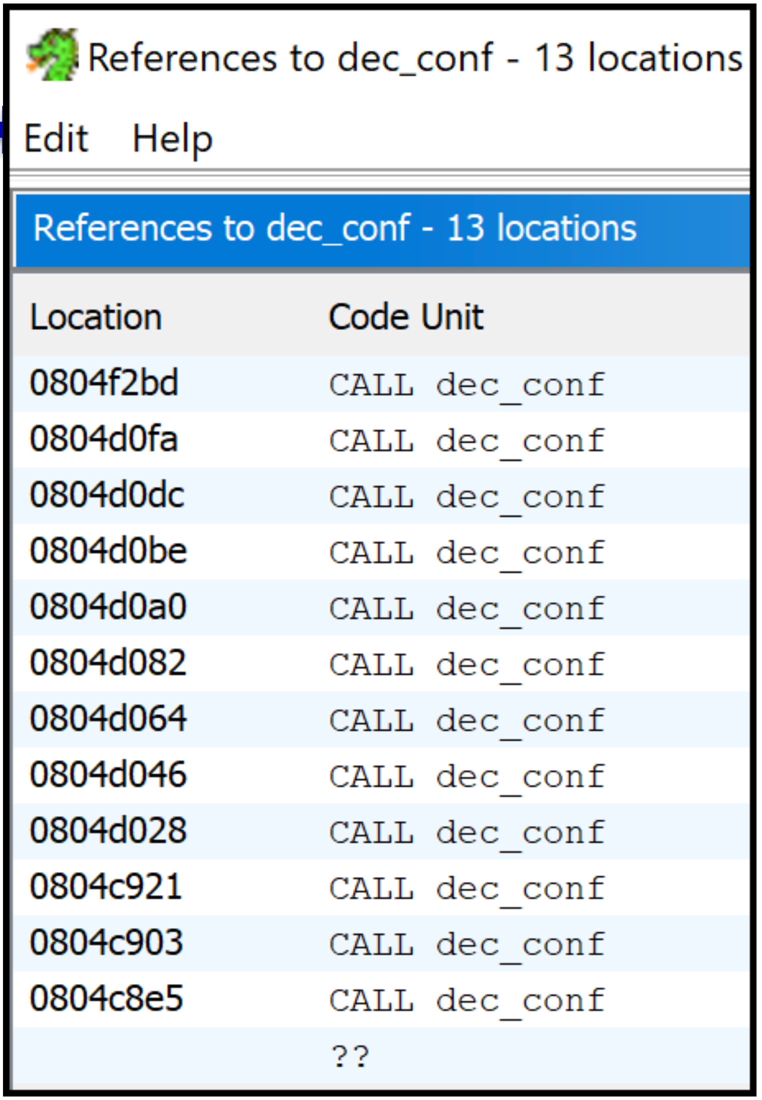
- What: Simulate execution of instructions, functions or program
- Why: Malware includes deobfuscation logic that is too time consuming or complex to implement.
- How: Unicorn, SpeakEasy, Qiling, and Dumpulator

CODE
AHEAD

Emulation Preview: XorDDoS (ELF)



```
0804c8cc    MOV    dword ptr [ESP + local_774], 0x12
0804c8d4    MOV    dword ptr [ESP + local_778], DAT_080b2fd1
0804c8dc    LEA    EAX=>local_108, [EBP + 0xfffffefc]
0804c8e2    MOV    dword ptr [ESP]=>local_77c, EAX
0804c8e5    CALL   dec_conf

0804c8ea    MOV    dword ptr [ESP + local_774], 0x11
0804c8f2    MOV    dword ptr [ESP + local_778], DAT_080b2fe3
0804c8fa    LEA    EAX=>local_208, [EBP + 0xfffffdfc]
0804c900    MOV    dword ptr [ESP]=>local_77c, EAX
0804c903    CALL   dec_conf

0804c908    MOV    dword ptr [ESP + local_774], 0x7
0804c910    MOV    dword ptr [ESP + local_778], DAT_080b2ff4
0804c918    LEA    EAX=>local_708, [EBP + 0xfffff8fc]
0804c91e    MOV    dword ptr [ESP]=>local_77c, EAX
0804c921    CALL   dec_conf
```



References to dec_conf - 13 locations

Edit Help

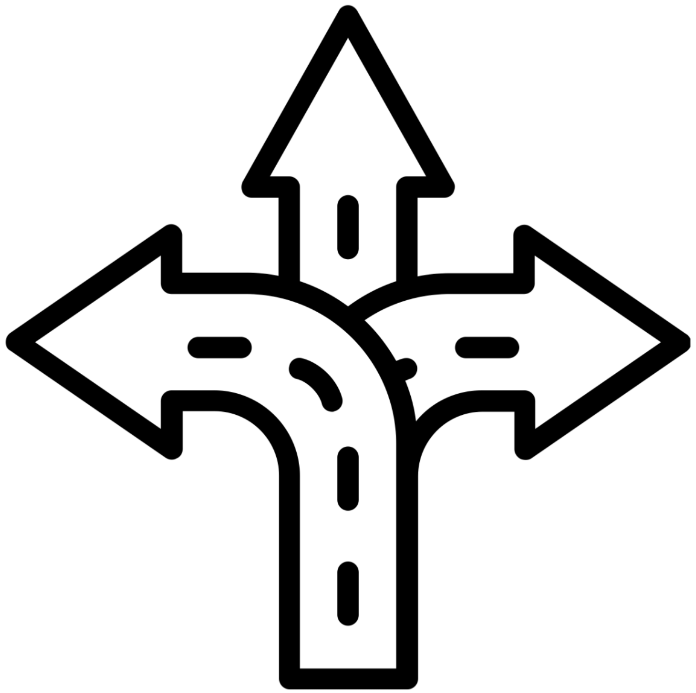
References to dec_conf - 13 locations

Location	Code Unit
0804f2bd	CALL dec_conf
0804d0fa	CALL dec_conf
0804d0dc	CALL dec_conf
0804d0be	CALL dec_conf
0804d0a0	CALL dec_conf
0804d082	CALL dec_conf
0804d064	CALL dec_conf
0804d046	CALL dec_conf
0804d028	CALL dec_conf
0804c921	CALL dec_conf
0804c903	CALL dec_conf
0804c8e5	CALL dec_conf
	??

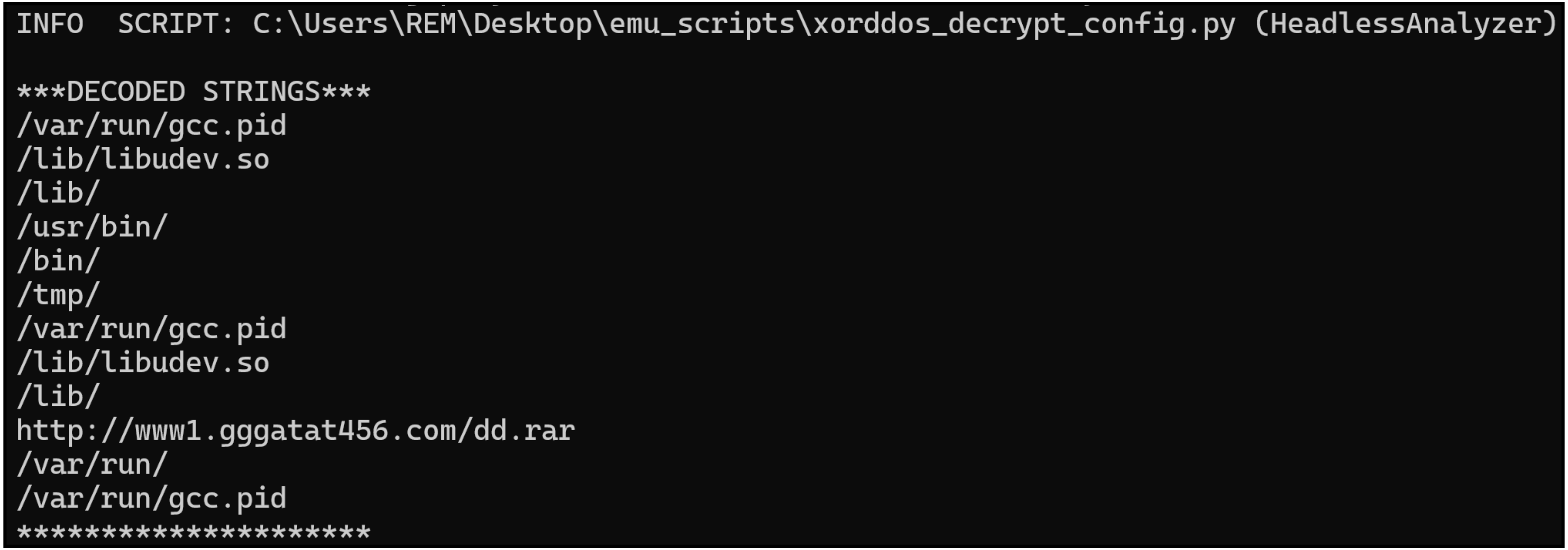
XorDDoS String Deobfuscation Options

1. Execute in Linux and view memory
2. Debug with GDB
3. Write Python script to extract and decode values

EMULATION



XorDDoS : Emulation Output



```
INFO  SCRIPT: C:\Users\REM\Desktop\emu_scripts\xorddos_decrypt_config.py (HeadlessAnalyzer)

***DECODED STRINGS***
/var/run/gcc.pid
/lib/libudev.so
/lib/
/usr/bin/
/bin/
/tmp/
/var/run/gcc.pid
/lib/libudev.so
/lib/
http://www1.gggatat456.com/dd.rar
/var/run/
/var/run/gcc.pid
*********************
```

Emulation Caveats

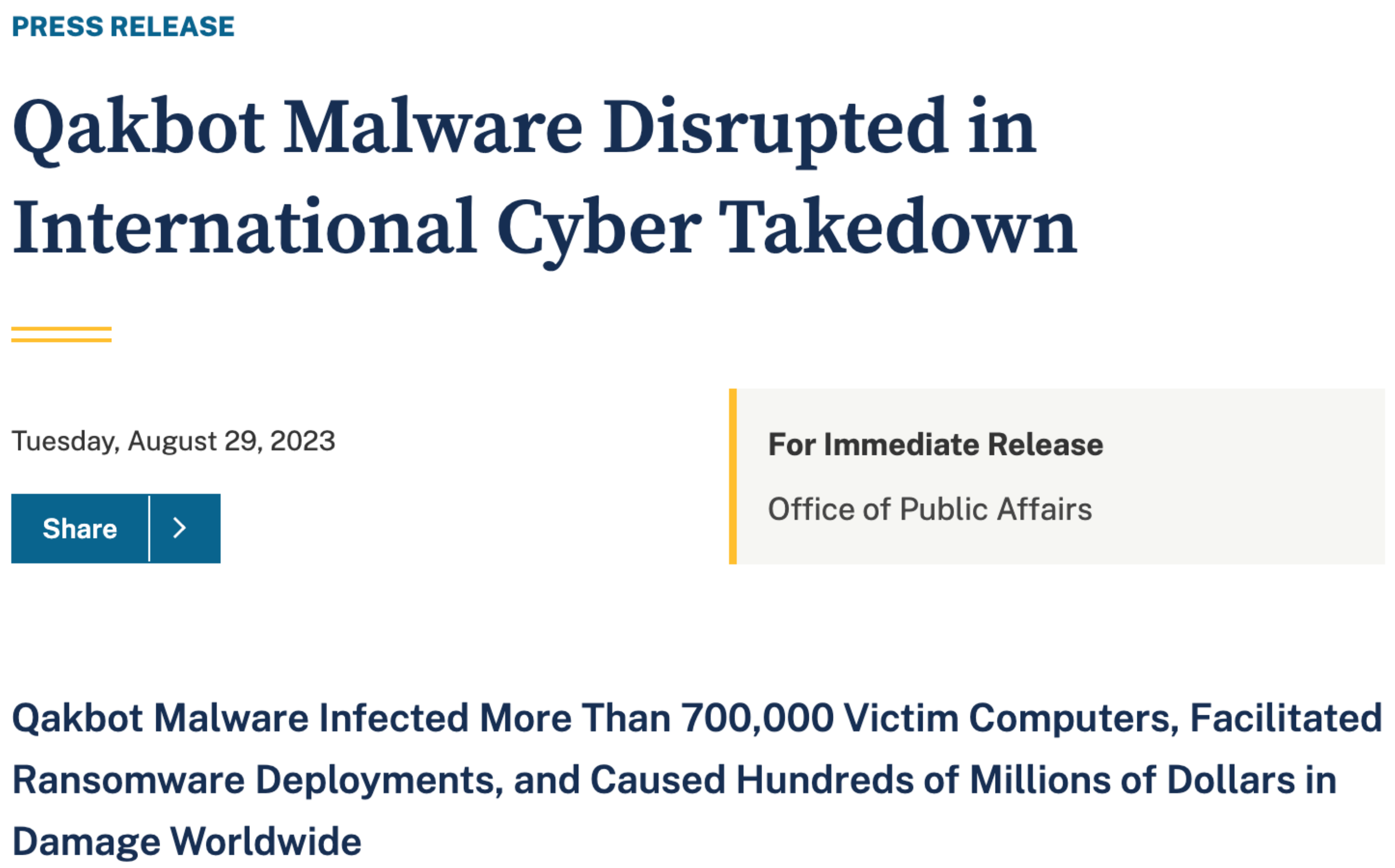
- Limited access to all resources/APIs within an operating system
- Performance is slower (vs. executing code in a VM)
- Prior (manual) analysis required
- Best suited for targeted execution of functions and instructions

Unicorn

- Multi-platform, multi-architecture CPU emulator framework
- A “CPU emulator” only emulates instructions.
- No awareness of Operating System or File Types
- Foundation for other emulators
  - Speakeasy
  - Dumpulator
  - Qiling

https://www.unicorn-engine.org/

Unicorn Example: Shellcode



PRESS RELEASE

Qakbot Malware Disrupted in International Cyber Takedown

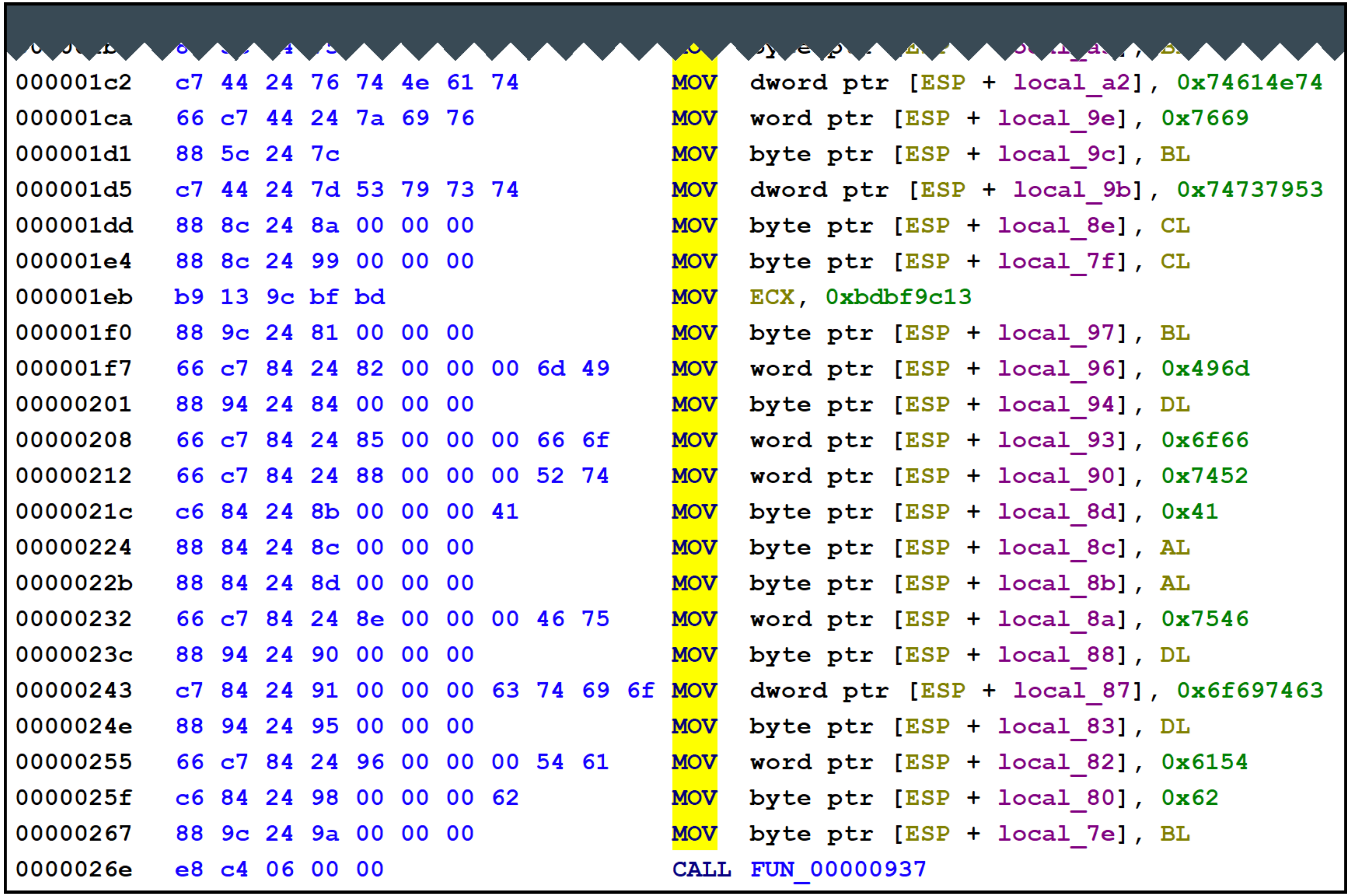
Tuesday, August 29, 2023

Share

For Immediate Release

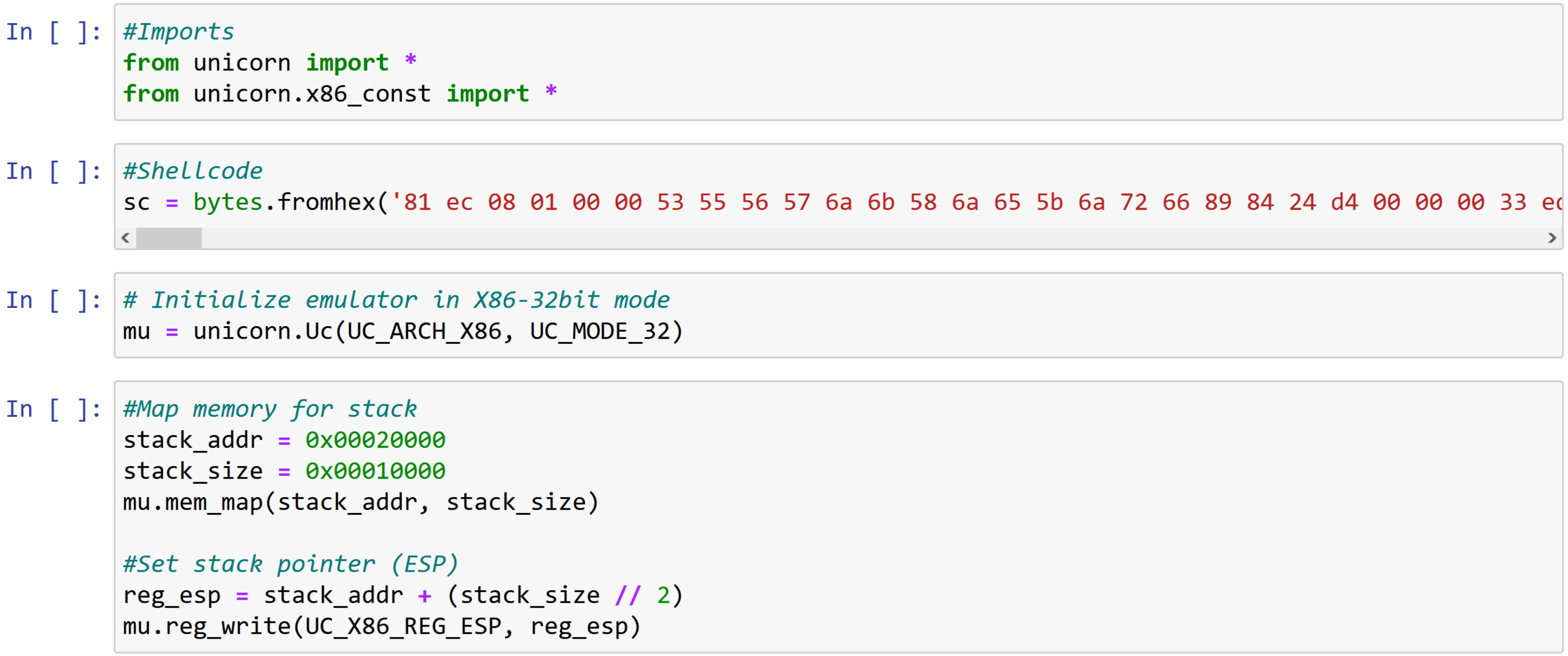
Office of Public Affairs

Qakbot Malware Infected More Than 700,000 Victim Computers, Facilitated Ransomware Deployments, and Caused Hundreds of Millions of Dollars in Damage Worldwide



```
000001c2   c7 44 24 76 74 4e 61 74          MOV   dword ptr [ESP + local_a2], 0x74614e74
000001ca   66 c7 44 24 7a 69 76             MOV   word ptr [ESP + local_9e], 0x7669
000001d1   88 5c 24 7c                      MOV   byte ptr [ESP + local_9c], BL
000001d5   c7 44 24 7d 53 79 73 74          MOV   dword ptr [ESP + local_9b], 0x74737953
000001dd   88 8c 24 8a 00 00 00             MOV   byte ptr [ESP + local_8e], CL
000001e4   88 8c 24 99 00 00 00             MOV   byte ptr [ESP + local_7f], CL
000001eb   b9 13 9c bf bd                   MOV   ECX, 0xbdbf9c13
000001f0   88 9c 24 81 00 00 00             MOV   byte ptr [ESP + local_97], BL
000001f7   66 c7 84 24 82 00 00 00 6d 49    MOV   word ptr [ESP + local_96], 0x496d
00000201   88 94 24 84 00 00 00             MOV   byte ptr [ESP + local_94], DL
00000208   66 c7 84 24 85 00 00 00 66 6f    MOV   word ptr [ESP + local_93], 0x6f66
00000212   66 c7 84 24 88 00 00 00 52 74    MOV   word ptr [ESP + local_90], 0x7452
0000021c   c6 84 24 8b 00 00 00 41          MOV   byte ptr [ESP + local_8d], 0x41
00000224   88 84 24 8c 00 00 00             MOV   byte ptr [ESP + local_8c], AL
0000022b   88 84 24 8d 00 00 00             MOV   byte ptr [ESP + local_8b], AL
00000232   66 c7 84 24 8e 00 00 00 46 75    MOV   word ptr [ESP + local_8a], 0x7546
0000023c   88 94 24 90 00 00 00             MOV   byte ptr [ESP + local_88], DL
00000243   c7 84 24 91 00 00 00 63 74 69 6f MOV   dword ptr [ESP + local_87], 0x6f697463
0000024e   88 94 24 95 00 00 00             MOV   byte ptr [ESP + local_83], DL
00000255   66 c7 84 24 96 00 00 00 54 61    MOV   word ptr [ESP + local_82], 0x6154
0000025f   c6 84 24 98 00 00 00 62          MOV   byte ptr [ESP + local_80], 0x62
00000267   88 9c 24 9a 00 00 00             MOV   byte ptr [ESP + local_7e], BL
0000026e   e8 c4 06 00 00                   CALL  FUN_00000937
```

Emulating Shellcode with Unicorn (1)



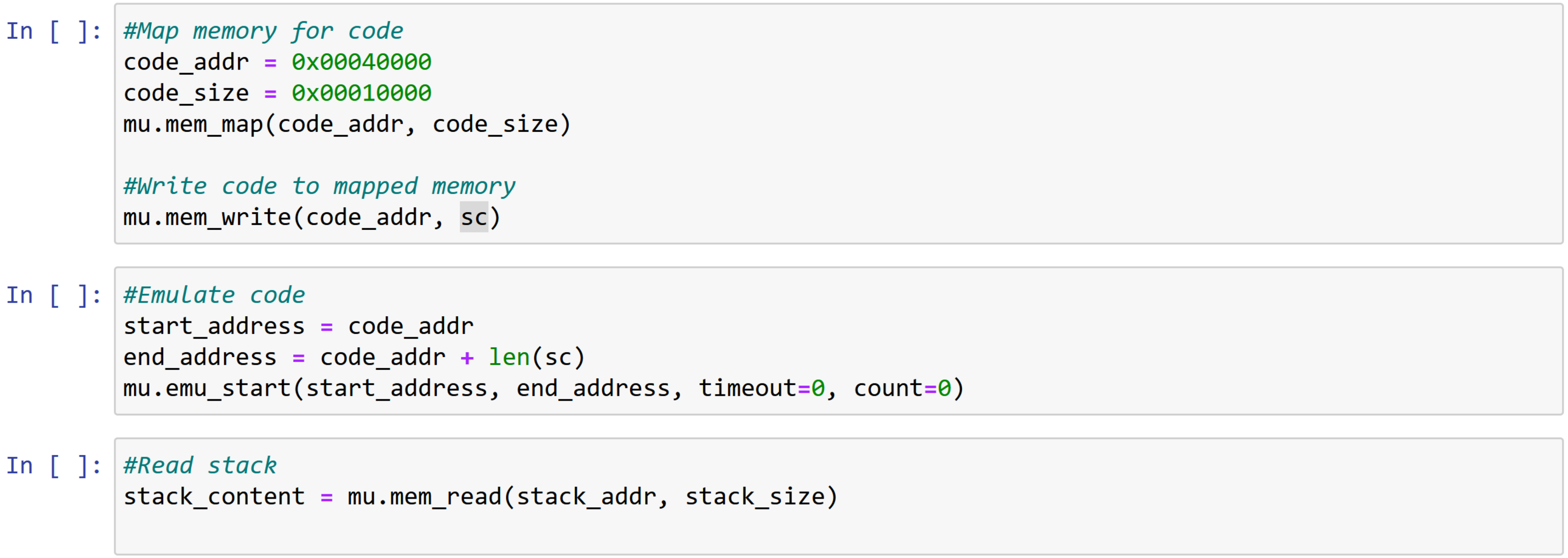
In [ ]:
```python
#Imports
from unicorn import *
from unicorn.x86_const import *
```

In [ ]:
```python
#Shellcode
sc = bytes.fromhex('81 ec 08 01 00 00 53 55 56 57 6a 6b 58 6a 65 5b 6a 72 66 89 84 24 d4 00 00 00 33 e
```

In [ ]:
```python
# Initialize emulator in X86-32bit mode
mu = unicorn.Uc(UC_ARCH_X86, UC_MODE_32)
```

In [ ]:
```python
#Map memory for stack
stack_addr = 0x00020000
stack_size = 0x00010000
mu.mem_map(stack_addr, stack_size)

#Set stack pointer (ESP)
reg_esp = stack_addr + (stack_size // 2)
mu.reg_write(UC_X86_REG_ESP, reg_esp)
```

Emulating Shellcode with Unicorn (2)



```
In [ ]: #Map memory for code
        code_addr = 0x00040000
        code_size = 0x00010000
        mu.mem_map(code_addr, code_size)

        #Write code to mapped memory
        mu.mem_write(code_addr, sc)
```

```
In [ ]: #Emulate code
        start_address = code_addr
        end_address = code_addr + len(sc)
        mu.emu_start(start_address, end_address, timeout=0, count=0)
```

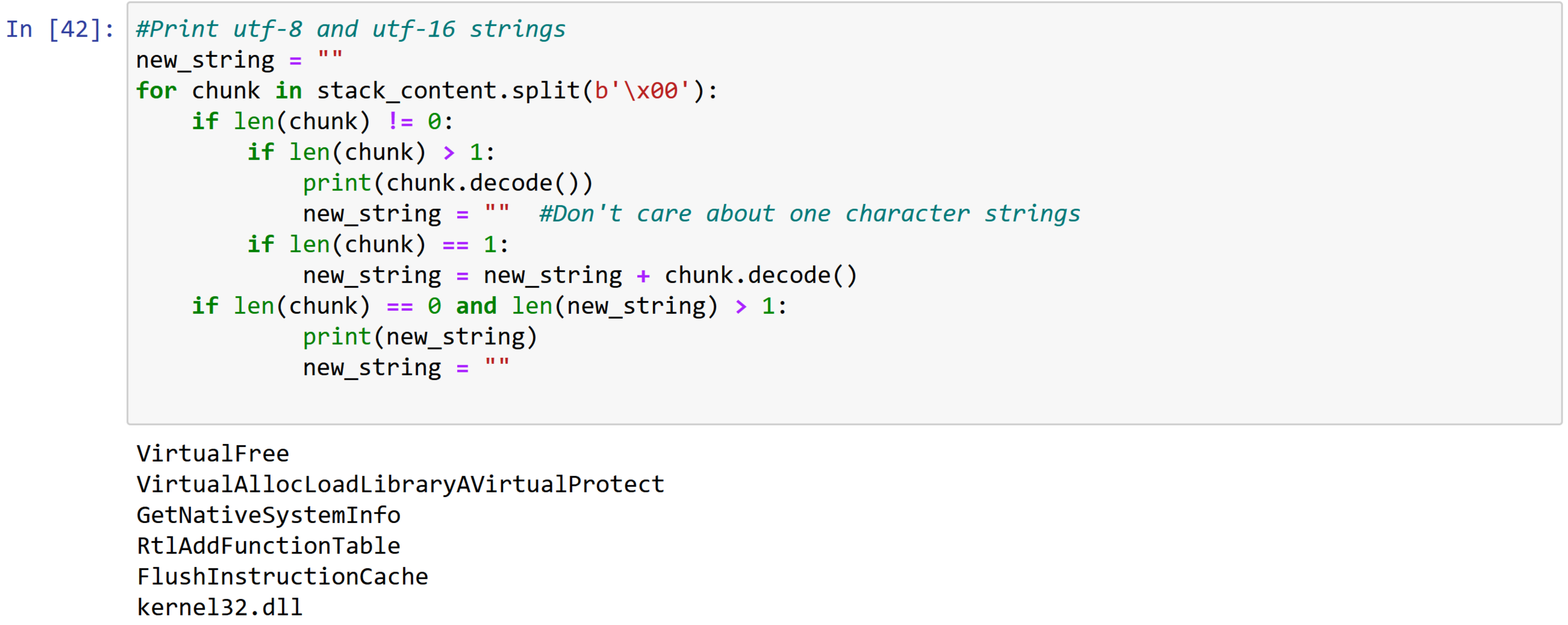
```
In [ ]: #Read stack
        stack_content = mu.mem_read(stack_addr, stack_size)
```

Emulating Shellcode with Unicorn (3)

In [ ]:

```python
#Print utf-8 and utf-16 strings
new_string = ""
for chunk in stack_content.split(b'\x00'):
    if len(chunk) != 0:
        if len(chunk) > 1:
            print(chunk.decode())
            new_string = ""  #Don't care about one character strings
        if len(chunk) == 1:
            new_string = new_string + chunk.decode()
    if len(chunk) == 0 and len(new_string) > 1:
            print(new_string)
            new_string = ""
```

Emulating Shellcode with Unicorn (4)



```
In [42]: #Print utf-8 and utf-16 strings
         new_string = ""
         for chunk in stack_content.split(b'\x00'):
             if len(chunk) != 0:
                 if len(chunk) > 1:
                     print(chunk.decode())
                     new_string = ""  #Don't care about one character strings
                 if len(chunk) == 1:
                     new_string = new_string + chunk.decode()
             if len(chunk) == 0 and len(new_string) > 1:
                     print(new_string)
                     new_string = ""
```

```
VirtualFree
VirtualAllocLoadLibraryAVirtualProtect
GetNativeSystemInfo
RtlAddFunctionTable
FlushInstructionCache
kernel32.dll
```

Speakeasy

- Windows only (user and kernel mode)
- Performs Windows API emulation
- Access as
  - Python library
  - standalone command line tool

https://github.com/mandiant/speakeasy

Dumpulator

- Windows only
- Performs syscall emulation (vs. API emulation)
  - Good: Less syscalls vs. APIs
  - Less good: Minimal documentation, more challenging to implement
- Requires generating minidump file
  - Good: Full process memory is available
  - Less good: Need to execute program and capture
- Other benefits: tracing execution

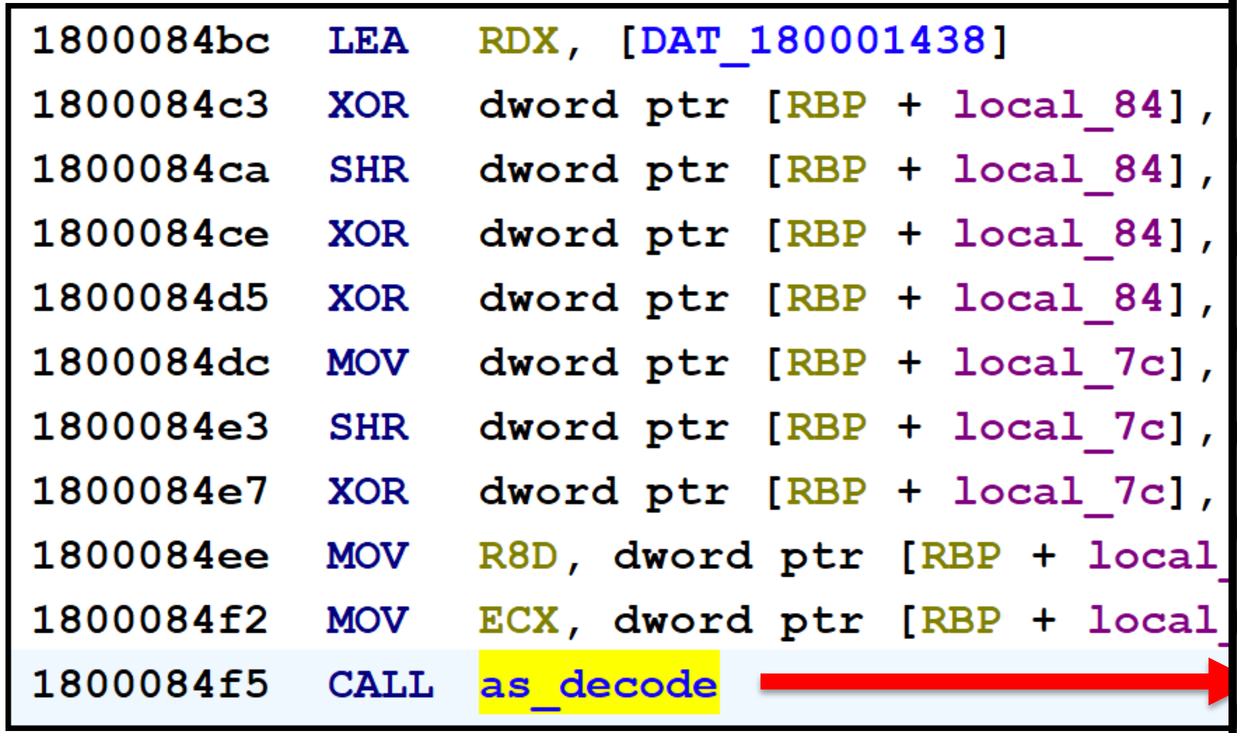
Qiling

- Cross platform: Windows, MacOS, Linux, BSD, UEFI, DOS
- Cross architecture: X86, X86_64, Arm, Arm64, MIPS, 8086
- Operating System and file type (e.g., PE) aware
- API emulation

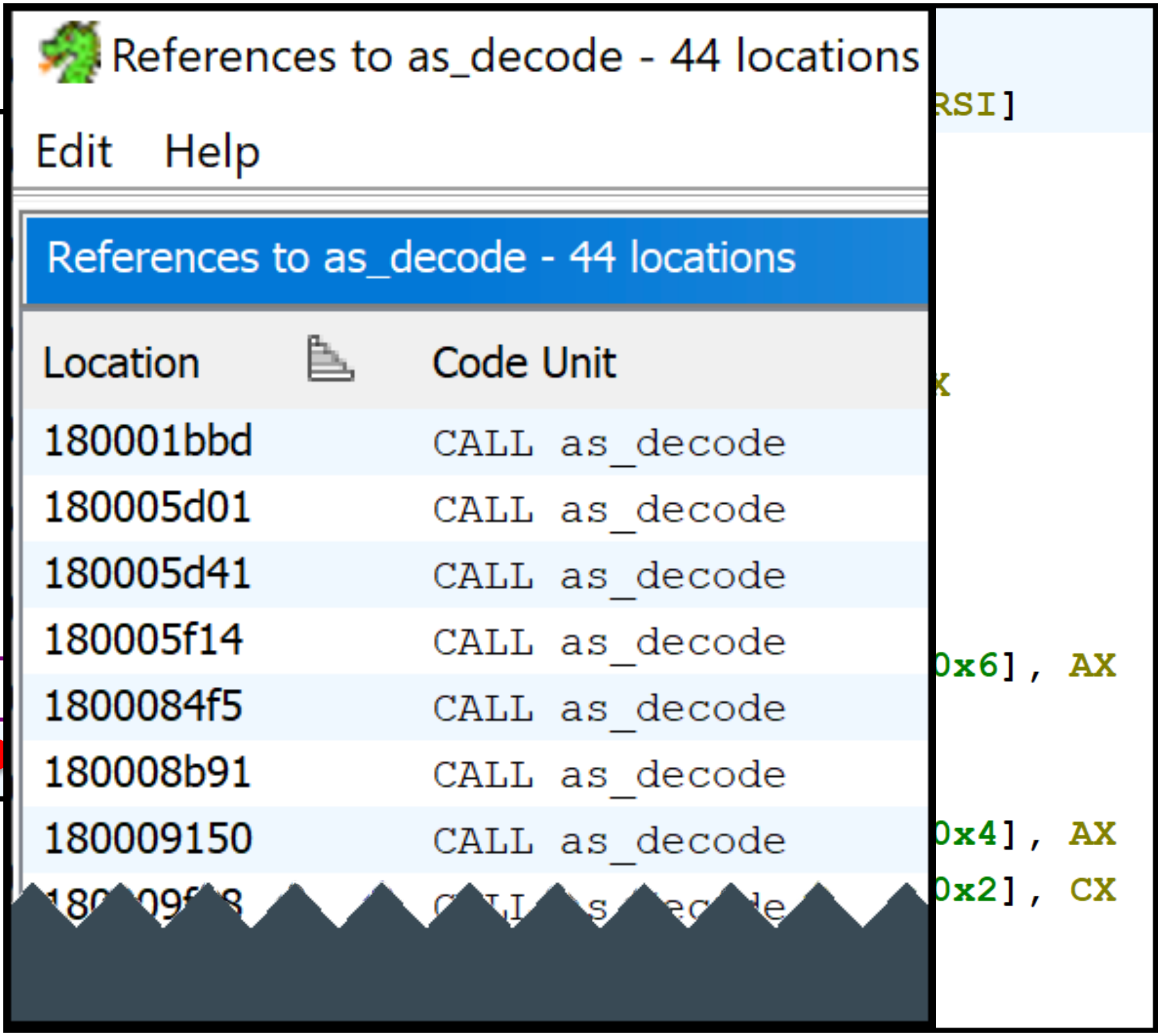
QILING.IO

https://github.com/qilingframework/qiling

Qiling Example: Emotet



```
1800084bc   LEA    RDX, [DAT_180001438]
1800084c3   XOR    dword ptr [RBP + local_84],
1800084ca   SHR    dword ptr [RBP + local_84],
1800084ce   XOR    dword ptr [RBP + local_84],
1800084d5   XOR    dword ptr [RBP + local_84],
1800084dc   MOV    dword ptr [RBP + local_7c],
1800084e3   SHR    dword ptr [RBP + local_7c],
1800084e7   XOR    dword ptr [RBP + local_7c],
1800084ee   MOV    R8D, dword ptr [RBP + local_
1800084f2   MOV    ECX, dword ptr [RBP + local_
1800084f5   CALL   as_decode
```



References to as_decode - 44 locations

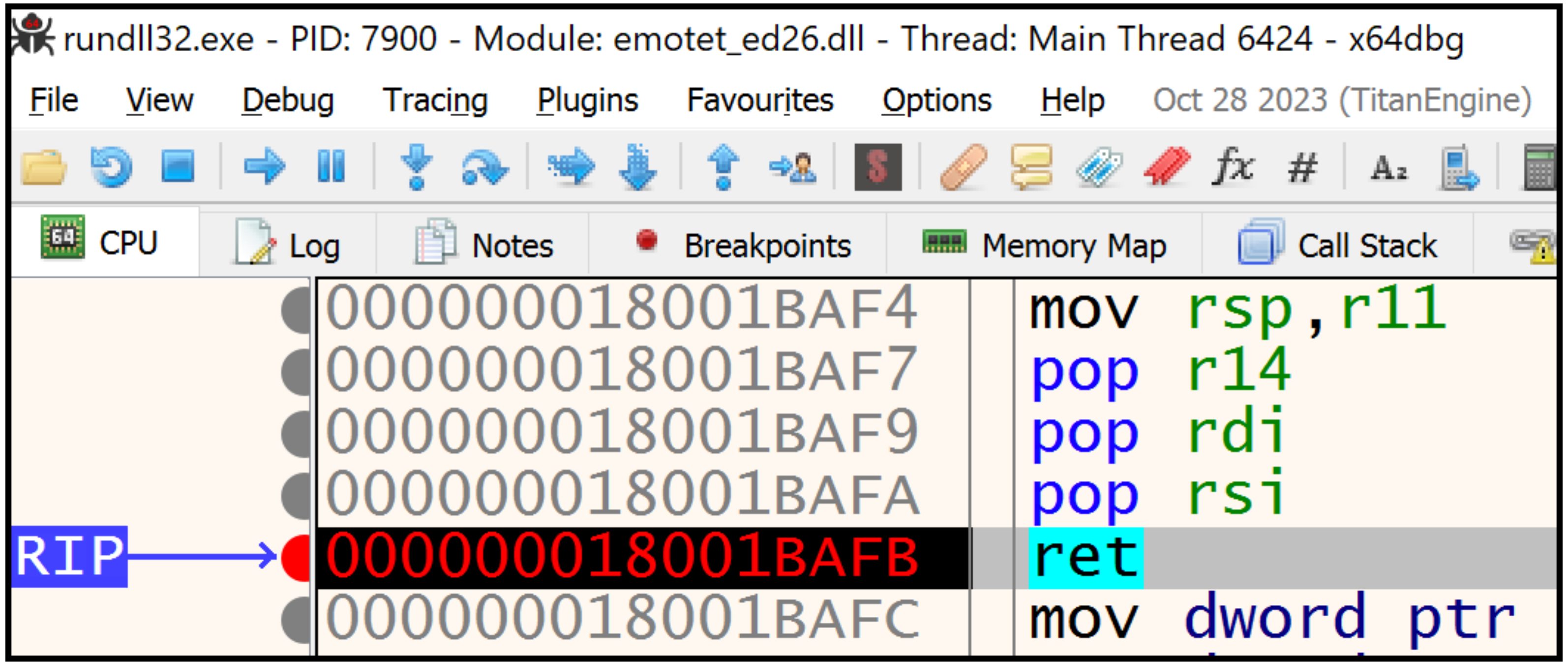
Edit Help

References to as_decode - 44 locations

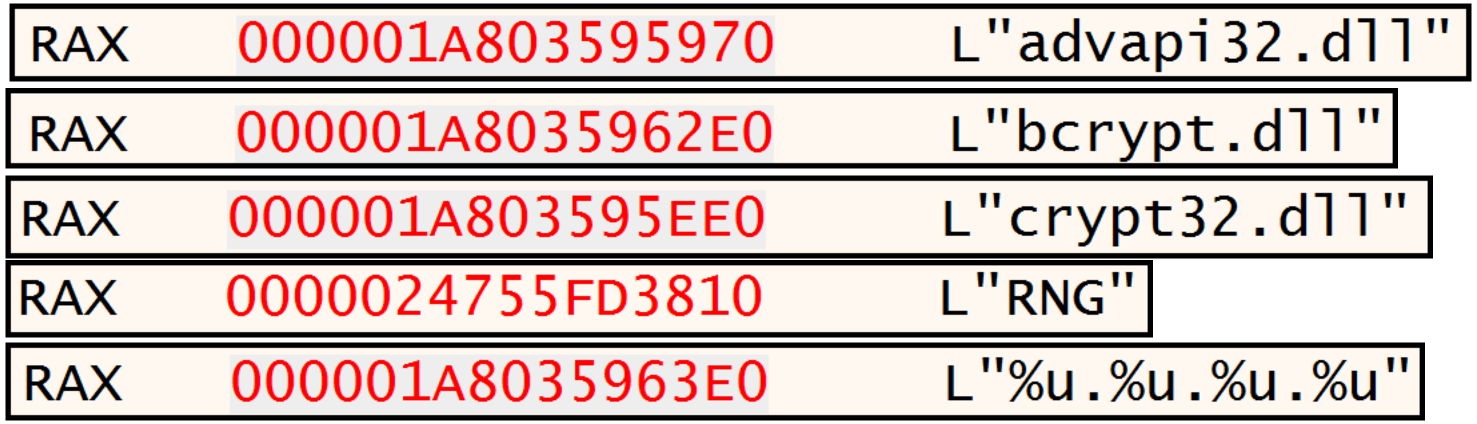
Location	Code Unit
180001bbd	CALL as_decode
180005d01	CALL as_decode
180005d41	CALL as_decode
180005f14	CALL as_decode
1800084f5	CALL as_decode
180008b91	CALL as_decode
180009150	CALL as_decode

```
RSI]
X
0x6], AX
0x4], AX
0x2], CX
```

Debug to Confirm Functionality

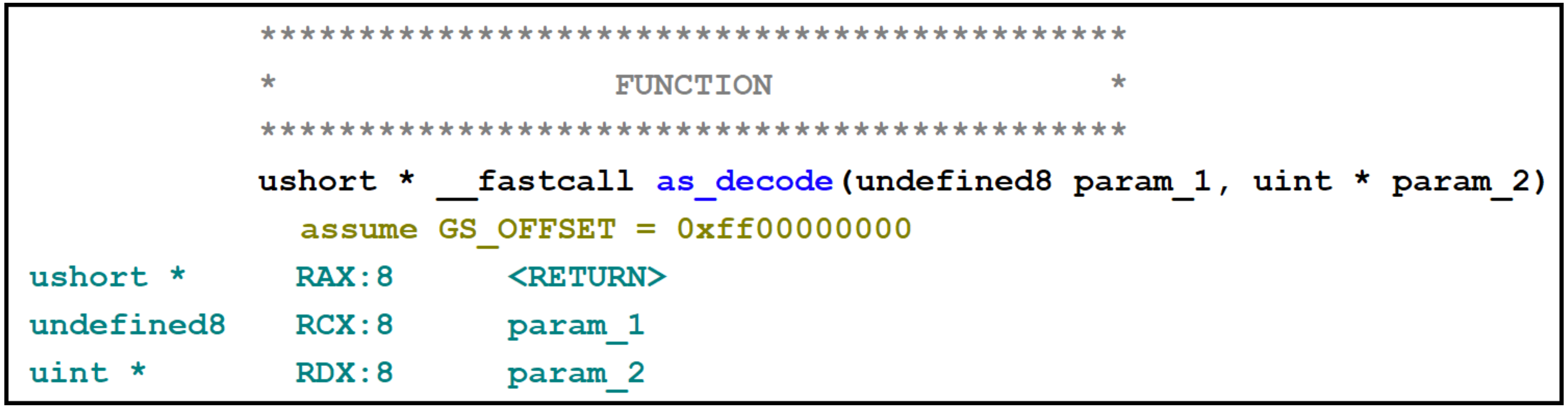

rundll32.exe - PID: 7900 - Module: emotet_ed26.dll - Thread: Main Thread 6424 - x64dbg
File View Debug Tracing Plugins Favourites Options Help Oct 28 2023 (TitanEngine)
CPU
Log
Notes
Breakpoints
Memory Map
Call Stack
RIP

```
000000018001BAF4   mov  rsp,r11
000000018001BAF7   pop  r14
000000018001BAF9   pop  rdi
000000018001BAFA   pop  rsi
000000018001BAFB   ret
000000018001BAFC   mov  dword ptr
```

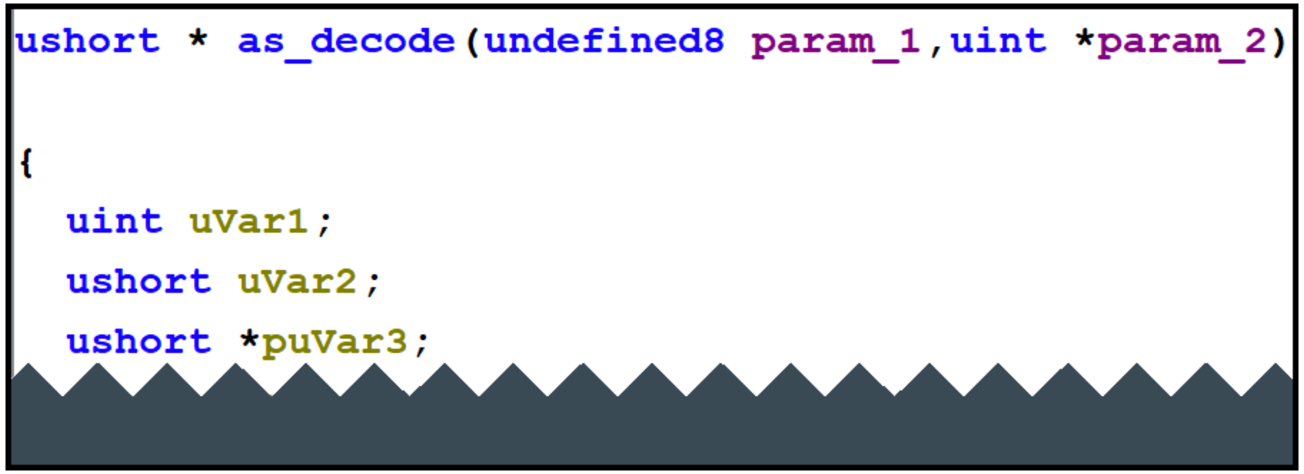


```
RAX    000001A803595970        L"advapi32.dll"
RAX    000001A8035962E0        L"bcrypt.dll"
RAX    000001A803595EE0        L"crypt32.dll"
RAX    0000024755FD3810        L"RNG"
RAX    000001A8035963E0        L"%u.%u.%u.%u"
```

Assess Function Arguments

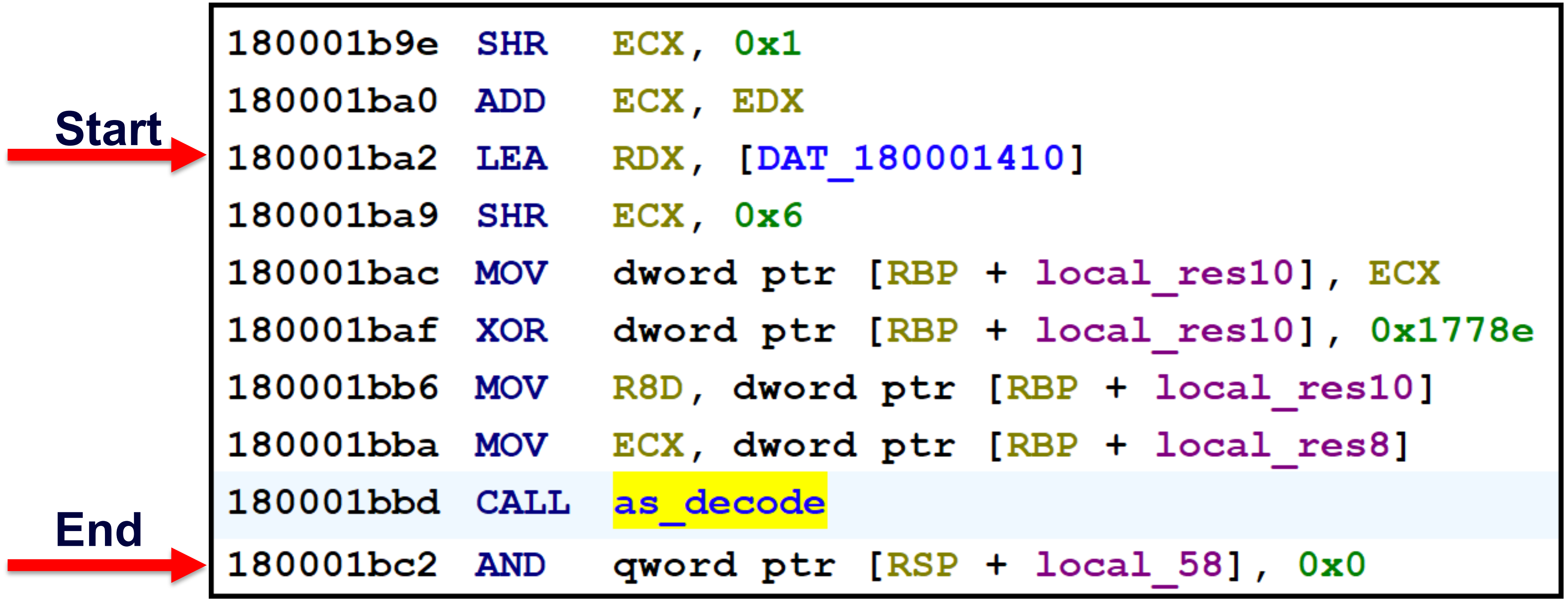


```
            *******************************************
            *                FUNCTION                 *
            *******************************************
            ushort * __fastcall as_decode(undefined8 param_1, uint * param_2)
              assume GS_OFFSET = 0xff00000000
ushort *      RAX:8      <RETURN>
undefined8    RCX:8      param_1
uint *        RDX:8      param_2
```

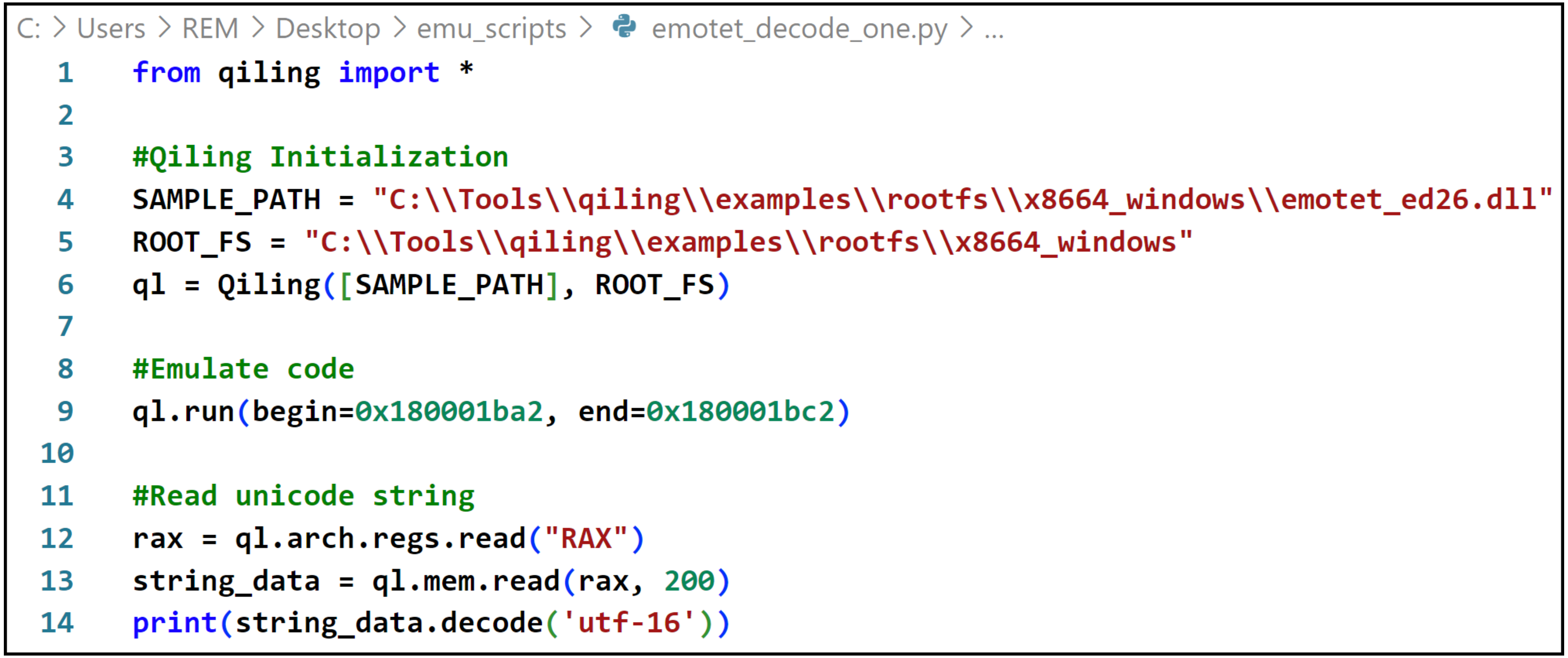


```
ushort * as_decode(undefined8 param_1,uint *param_2)

{
  uint uVar1;
  ushort uVar2;
  ushort *puVar3;
```

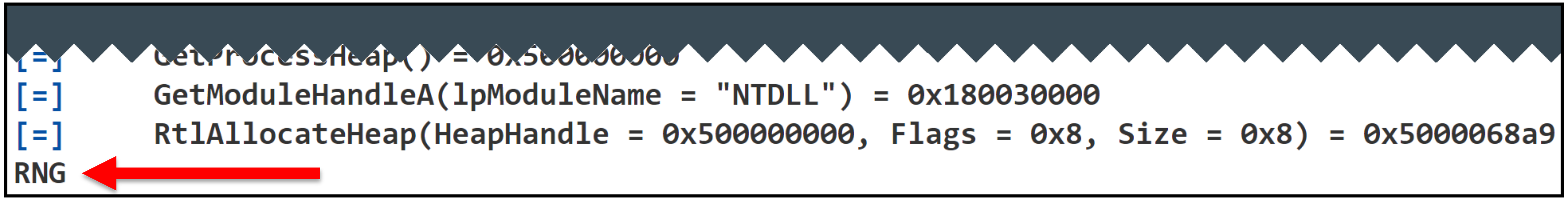
Example Start and End Addresses



```
180001b9e  SHR   ECX, 0x1
180001ba0  ADD   ECX, EDX
180001ba2  LEA   RDX, [DAT_180001410]
180001ba9  SHR   ECX, 0x6
180001bac  MOV   dword ptr [RBP + local_res10], ECX
180001baf  XOR   dword ptr [RBP + local_res10], 0x1778e
180001bb6  MOV   R8D, dword ptr [RBP + local_res10]
180001bba  MOV   ECX, dword ptr [RBP + local_res8]
180001bbd  CALL  as_decode
180001bc2  AND   qword ptr [RSP + local_58], 0x0
```


C: > Users > REM > Desktop > emu_scripts > emotet_decode_one.py > ...

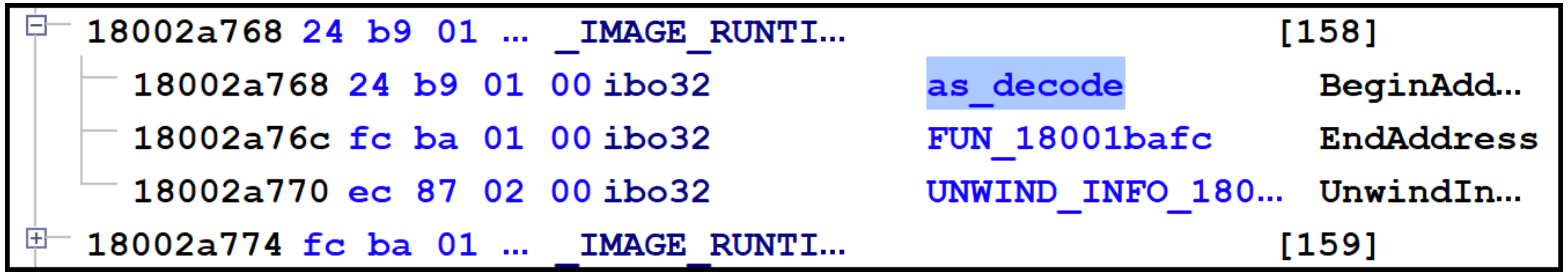
```python
from qiling import *

#Qiling Initialization
SAMPLE_PATH = "C:\\Tools\\qiling\\examples\\rootfs\\x8664_windows\\emotet_ed26.dll"
ROOT_FS = "C:\\Tools\\qiling\\examples\\rootfs\\x8664_windows"
ql = Qiling([SAMPLE_PATH], ROOT_FS)

#Emulate code
ql.run(begin=0x180001ba2, end=0x180001bc2)

#Read unicode string
rax = ql.arch.regs.read("RAX")
string_data = ql.mem.read(rax, 200)
print(string_data.decode('utf-16'))
```



```
[=]     GetProcessHeap() = 0x500000000
[=]     GetModuleHandleA(lpModuleName = "NTDLL") = 0x180030000
[=]     RtlAllocateHeap(HeapHandle = 0x500000000, Flags = 0x8, Size = 0x8) = 0x5000068a9
RNG
```

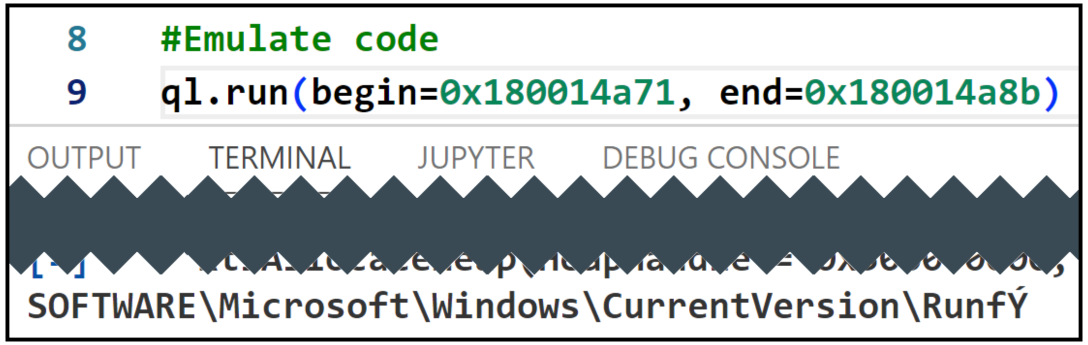
Decoding *All* Strings Presents Challenges

- Need a disassembler API to find references and traverse instructions
- Not all references are function calls



```
⊟ 18002a768 24 b9 01 ...  _IMAGE_RUNTI...                           [158]
    18002a768 24 b9 01 00 ibo32          as_decode           BeginAdd...
    18002a76c fc ba 01 00 ibo32          FUN_18001bafc       EndAddress
    18002a770 ec 87 02 00 ibo32          UNWIND_INFO_180...  UnwindIn...
⊞ 18002a774 fc ba 01 ...  _IMAGE_RUNTI...                           [159]
```

- Some strings don't decode properly with existing code



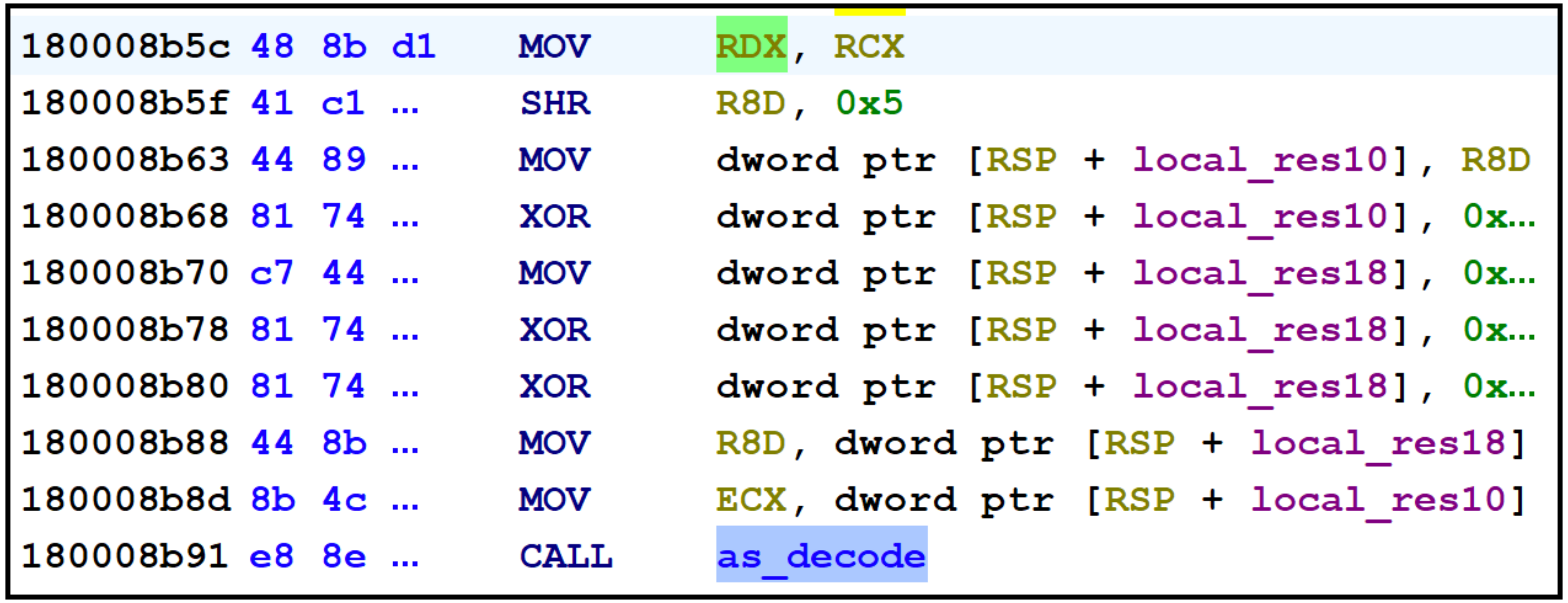
```
#Emulate code
ql.run(begin=0x180014a71, end=0x180014a8b)
```

OUTPUT   TERMINAL   JUPYTER   DEBUG CONSOLE

```
[redacted]
SOFTWARE\Microsoft\Windows\CurrentVersion\RunfÝ
```

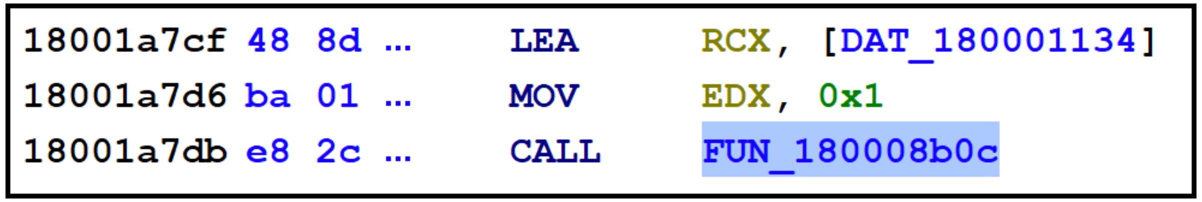
Decoding *All* Strings Presents Challenges (continued)

- In one case, the encrypted string is passed via a register

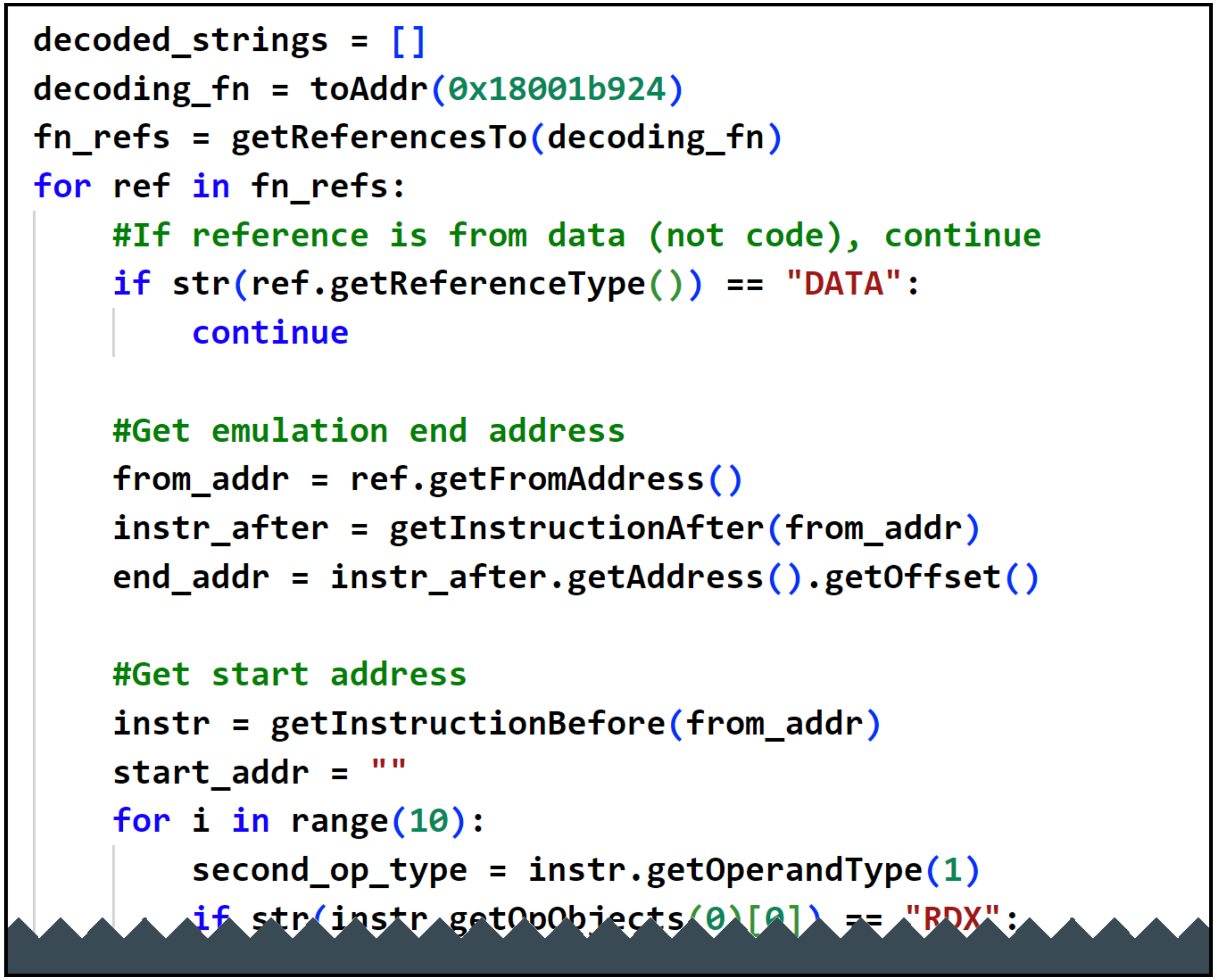


```
180008b5c 48 8b d1     MOV      RDX, RCX
180008b5f 41 c1 ...    SHR      R8D, 0x5
180008b63 44 89 ...    MOV      dword ptr [RSP + local_res10], R8D
180008b68 81 74 ...    XOR      dword ptr [RSP + local_res10], 0x...
180008b70 c7 44 ...    MOV      dword ptr [RSP + local_res18], 0x...
180008b78 81 74 ...    XOR      dword ptr [RSP + local_res18], 0x...
180008b80 81 74 ...    XOR      dword ptr [RSP + local_res18], 0x...
180008b88 44 8b ...    MOV      R8D, dword ptr [RSP + local_res18]
180008b8d 8b 4c ...    MOV      ECX, dword ptr [RSP + local_res10]
180008b91 e8 8e ...    CALL     as_decode
```

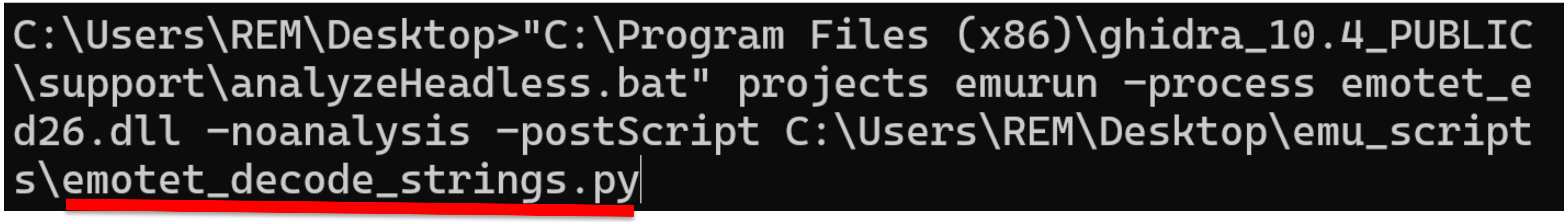
- This function is referenced multiple times



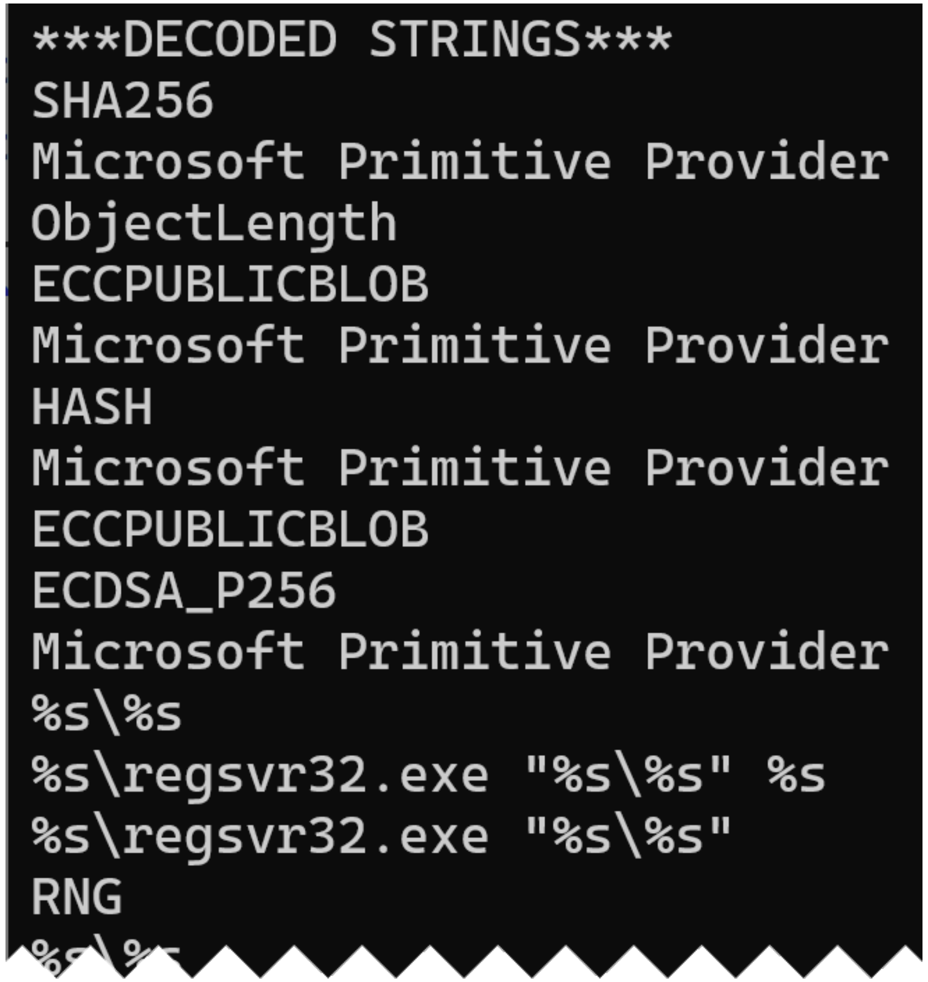
```
18001a7cf 48 8d ...    LEA      RCX, [DAT_180001134]
18001a7d6 ba 01 ...    MOV      EDX, 0x1
18001a7db e8 2c ...    CALL     FUN_180008b0c
```



```python
decoded_strings = []
decoding_fn = toAddr(0x18001b924)
fn_refs = getReferencesTo(decoding_fn)
for ref in fn_refs:
    #If reference is from data (not code), continue
    if str(ref.getReferenceType()) == "DATA":
        continue

    #Get emulation end address
    from_addr = ref.getFromAddress()
    instr_after = getInstructionAfter(from_addr)
    end_addr = instr_after.getAddress().getOffset()

    #Get start address
    instr = getInstructionBefore(from_addr)
    start_addr = ""
    for i in range(10):
        second_op_type = instr.getOperandType(1)
        if str(instr.getOpObjects(0)[0]) == "RDX":
```

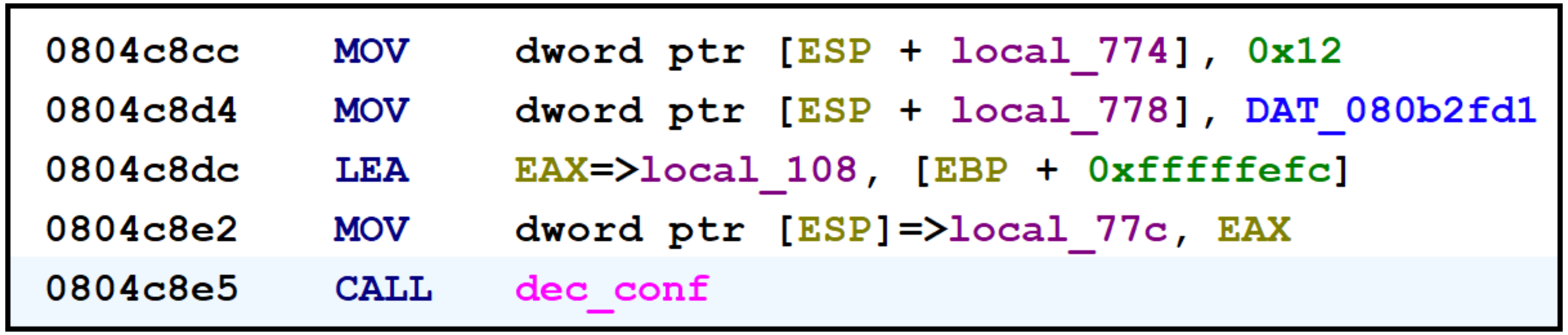


```
C:\Users\REM\Desktop>"C:\Program Files (x86)\ghidra_10.4_PUBLIC
\support\analyzeHeadless.bat" projects emurun -process emotet_e
d26.dll -noanalysis -postScript C:\Users\REM\Desktop\emu_script
s\emotet_decode_strings.py
```

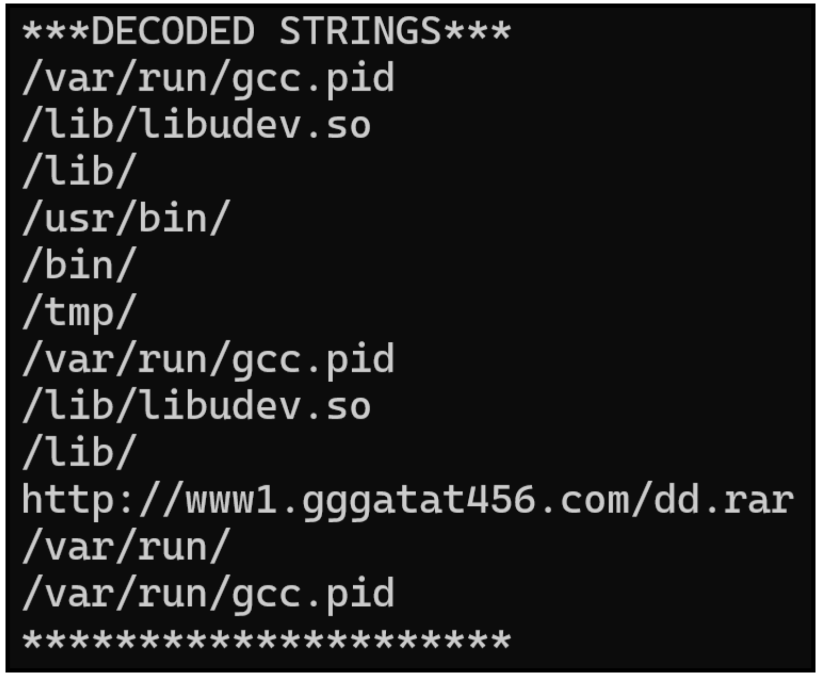


```
***DECODED STRINGS***
SHA256
Microsoft Primitive Provider
ObjectLength
ECCPUBLICBLOB
Microsoft Primitive Provider
HASH
Microsoft Primitive Provider
ECCPUBLICBLOB
ECDSA_P256
Microsoft Primitive Provider
%s\%s
%s\regsvr32.exe "%s\%s" %s
%s\regsvr32.exe "%s\%s"
RNG
%s\%s
```

Emulating ELF: XorDDos

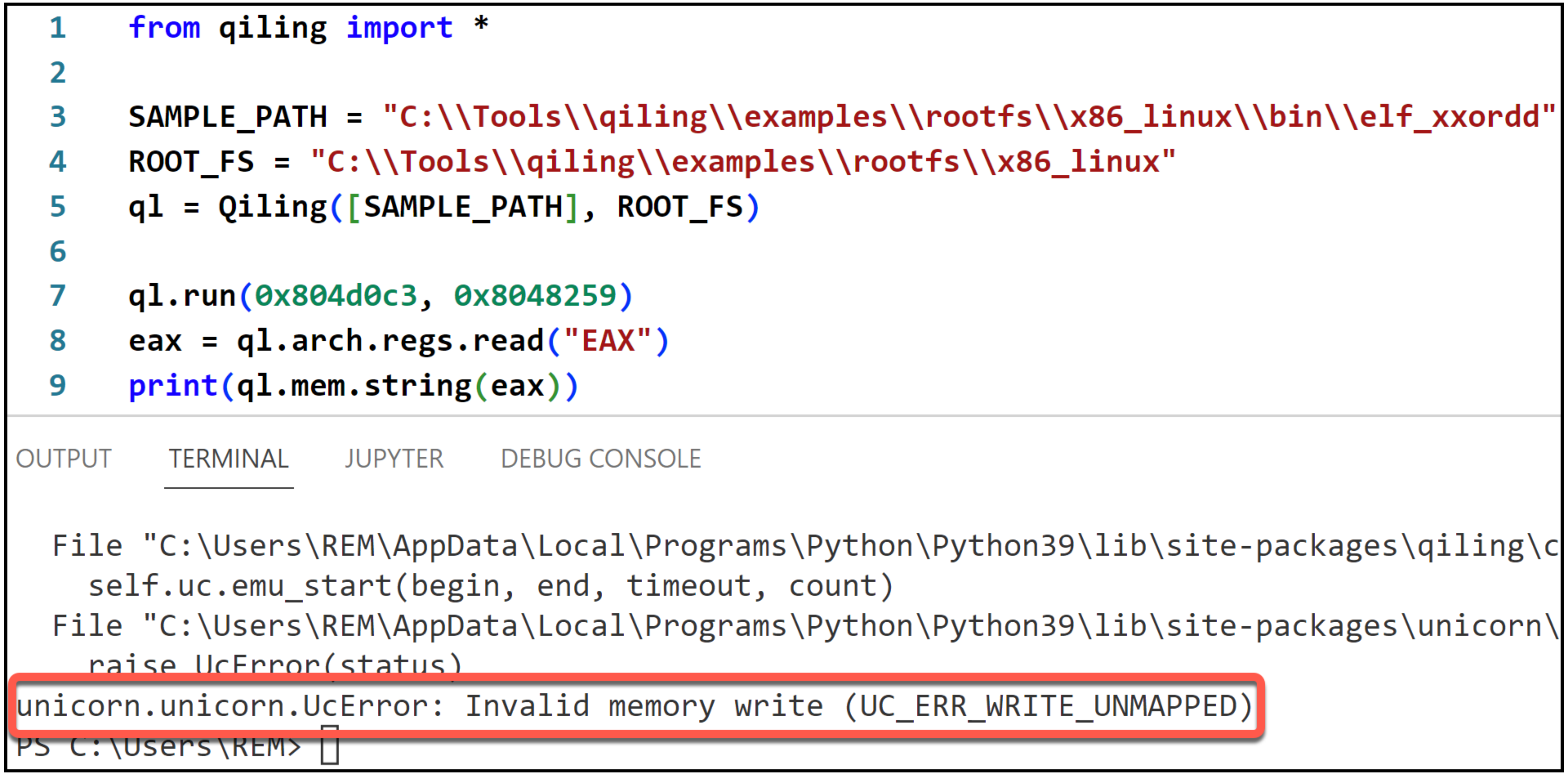


```
0804c8cc    MOV     dword ptr [ESP + local_774], 0x12
0804c8d4    MOV     dword ptr [ESP + local_778], DAT_080b2fd1
0804c8dc    LEA     EAX=>local_108, [EBP + 0xfffffefc]
0804c8e2    MOV     dword ptr [ESP]=>local_77c, EAX
0804c8e5    CALL    dec_conf
```



```
***DECODED STRINGS***
/var/run/gcc.pid
/lib/libudev.so
/lib/
/usr/bin/
/bin/
/tmp/
/var/run/gcc.pid
/lib/libudev.so
/lib/
http://www1.gggatat456.com/dd.rar
/var/run/
/var/run/gcc.pid
*********************
```

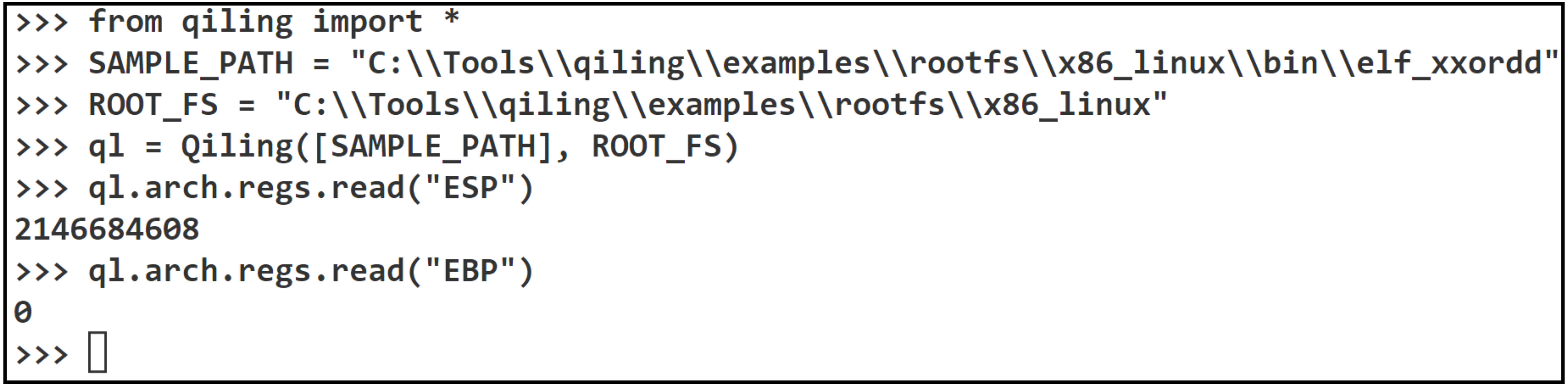
XorDDos Emulation Challenges



```python
from qiling import *

SAMPLE_PATH = "C:\\Tools\\qiling\\examples\\rootfs\\x86_linux\\bin\\elf_xxordd"
ROOT_FS = "C:\\Tools\\qiling\\examples\\rootfs\\x86_linux"
ql = Qiling([SAMPLE_PATH], ROOT_FS)

ql.run(0x804d0c3, 0x8048259)
eax = ql.arch.regs.read("EAX")
print(ql.mem.string(eax))
```

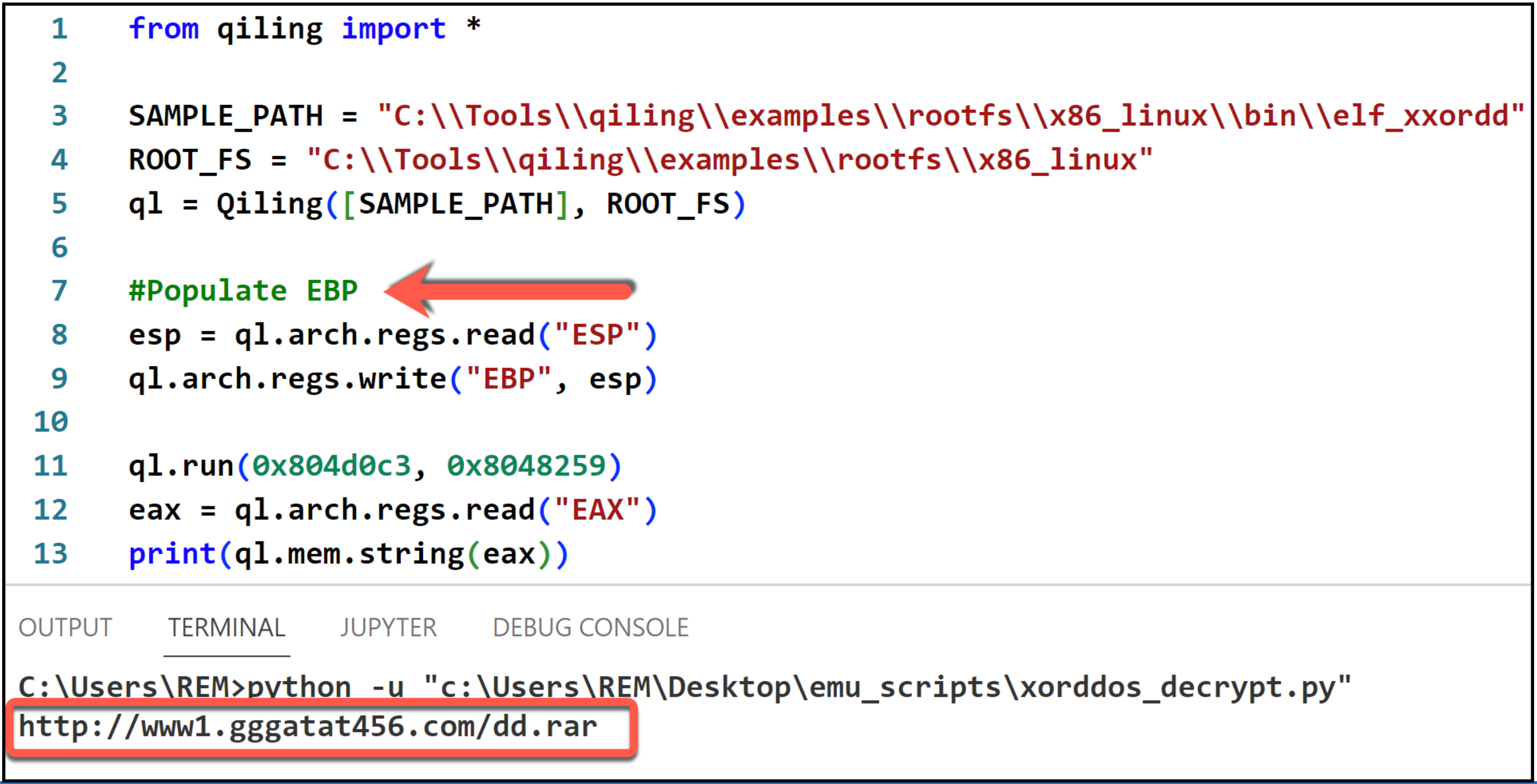
```
  File "C:\Users\REM\AppData\Local\Programs\Python\Python39\lib\site-packages\qiling\c
    self.uc.emu_start(begin, end, timeout, count)
  File "C:\Users\REM\AppData\Local\Programs\Python\Python39\lib\site-packages\unicorn\
    raise UcError(status)
unicorn.unicorn.UcError: Invalid memory write (UC_ERR_WRITE_UNMAPPED)
PS C:\Users\REM>
```

XorDDos Emulation Challenges



```
>>> from qiling import *
>>> SAMPLE_PATH = "C:\\Tools\\qiling\\examples\\rootfs\\x86_linux\\bin\\elf_xxordd"
>>> ROOT_FS = "C:\\Tools\\qiling\\examples\\rootfs\\x86_linux"
>>> ql = Qiling([SAMPLE_PATH], ROOT_FS)
>>> ql.arch.regs.read("ESP")
2146684608
>>> ql.arch.regs.read("EBP")
0
>>> 
```

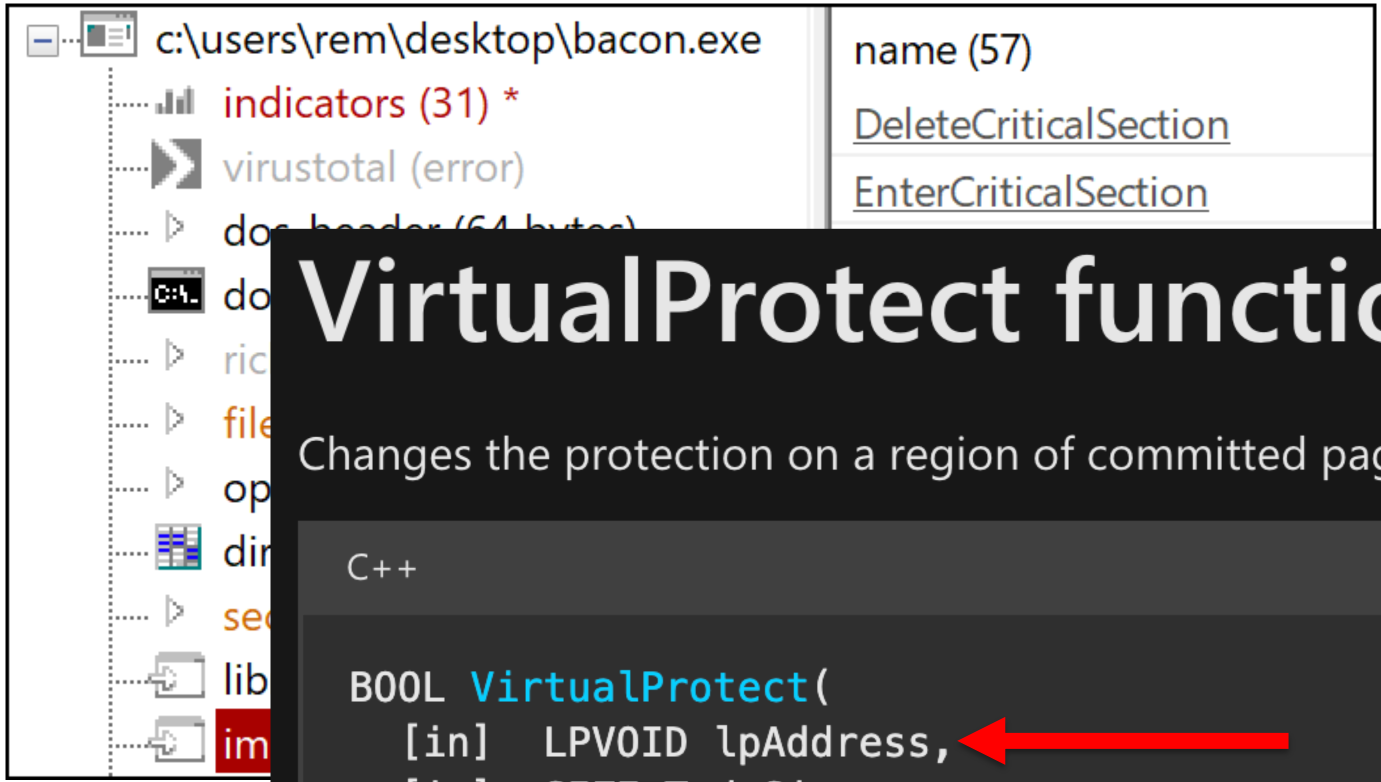
XorDDos Emulation Success



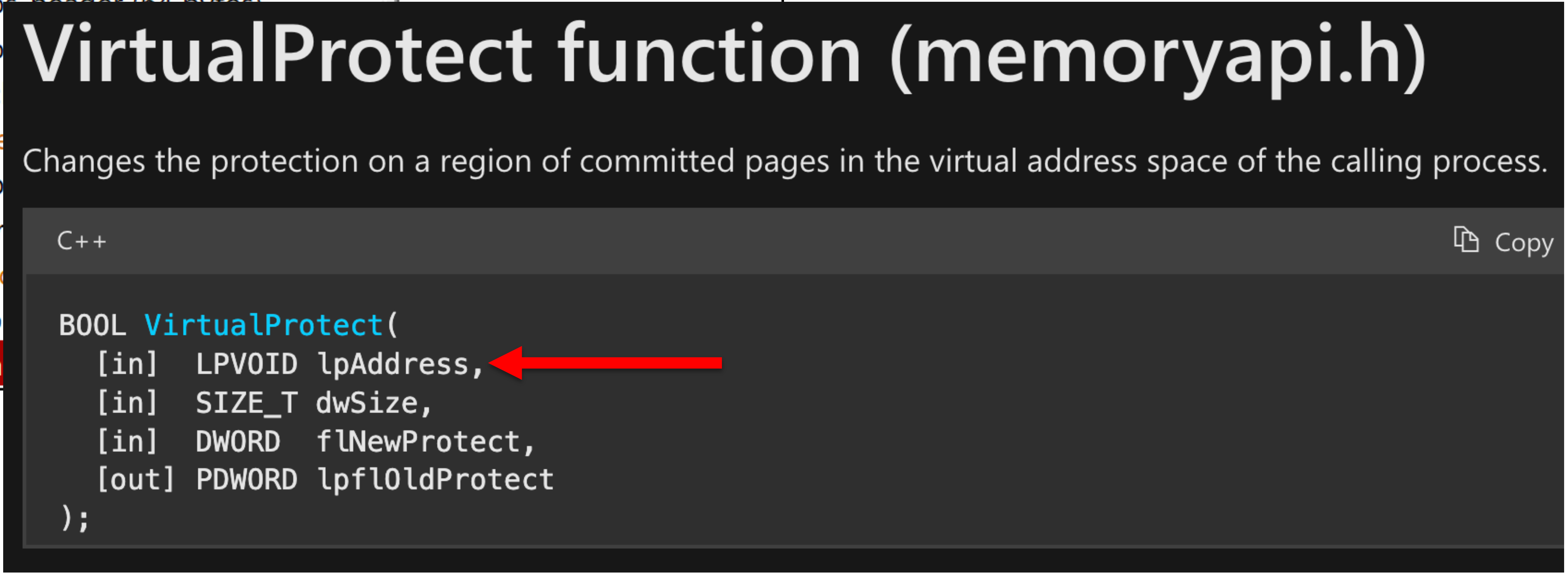
```python
from qiling import *

SAMPLE_PATH = "C:\\Tools\\qiling\\examples\\rootfs\\x86_linux\\bin\\elf_xxordd"
ROOT_FS = "C:\\Tools\\qiling\\examples\\rootfs\\x86_linux"
ql = Qiling([SAMPLE_PATH], ROOT_FS)

#Populate EBP
esp = ql.arch.regs.read("ESP")
ql.arch.regs.write("EBP", esp)

ql.run(0x804d0c3, 0x8048259)
eax = ql.arch.regs.read("EAX")
print(ql.mem.string(eax))
```

```
C:\Users\REM>python -u "c:\Users\REM\Desktop\emu_scripts\xorddos_decrypt.py"
http://www1.gggatat456.com/dd.rar
```

Extracting Second Stage Payloads


c:\users\rem\desktop\bacon.exe
indicators (31) *
virustotal (error)
dos-header (64 bytes)
do
ric
file
op
dir
sec
lib
im

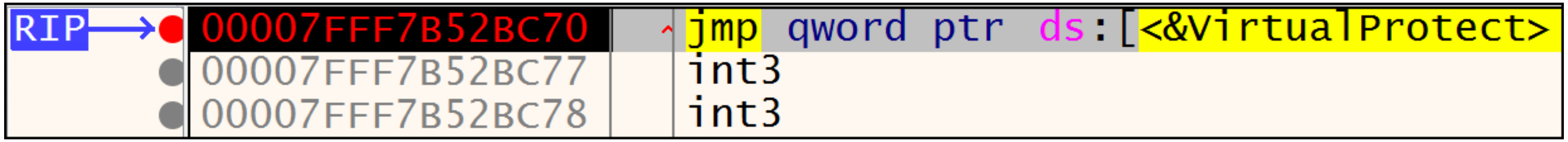
name (57)
DeleteCriticalSection
EnterCriticalSection


VirtualProtect function (memoryapi.h)

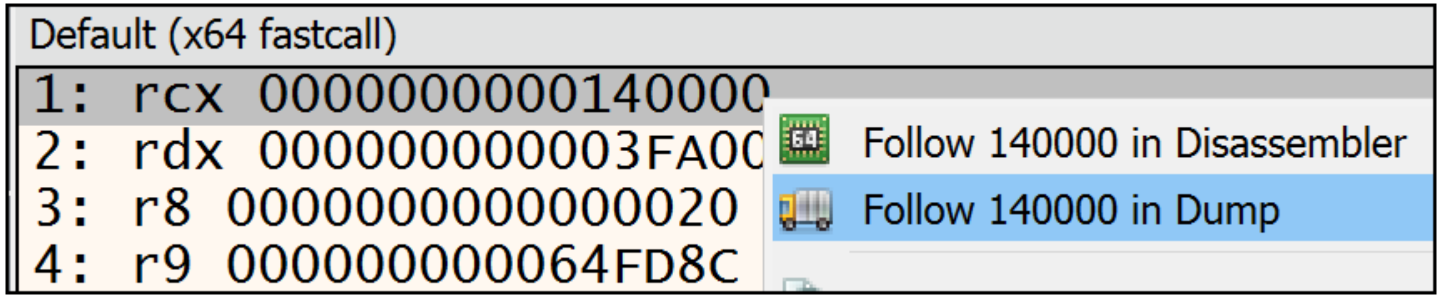
Changes the protection on a region of committed pages in the virtual address space of the calling process.

```cpp
BOOL VirtualProtect(
  [in]  LPVOID lpAddress,
  [in]  SIZE_T dwSize,
  [in]  DWORD  flNewProtect,
  [out] PDWORD lpflOldProtect
);
```

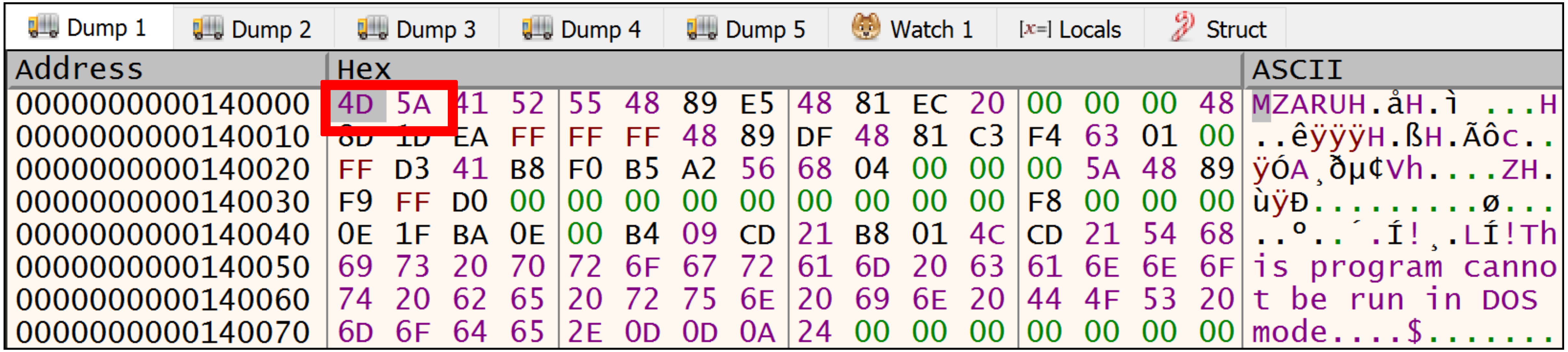
Debugging Bacon.exe



```
RIP → ● 00007FFF7B52BC70    jmp qword ptr ds:[<&VirtualProtect>
      ● 00007FFF7B52BC77    int3
      ● 00007FFF7B52BC78    int3
```



```
Default (x64 fastcall)
1: rcx 000000000014000
2: rdx 00000000003FA0C
3: r8  000000000000020
4: r9  00000000064FD8C
```

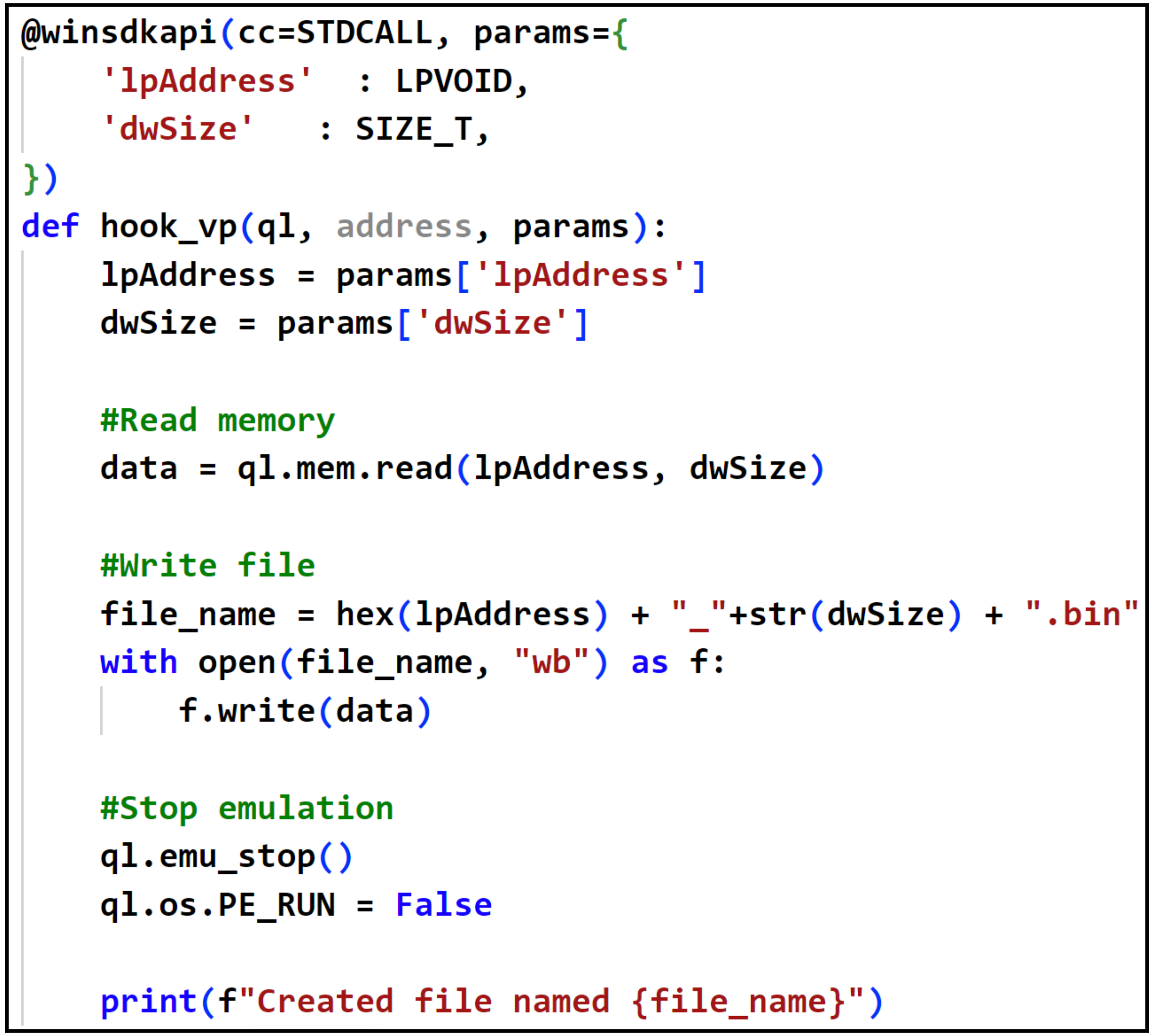


```
Address           Hex                                              ASCII
0000000000140000  4D 5A 41 52 55 48 89 E5 48 81 EC 20 00 00 00 48  MZARUH.åH.ì ...H
0000000000140010  8D 1D EA FF FF FF 48 89 DF 48 81 C3 F4 63 01 00  ..êÿÿÿH.ßH.Ãôc..
0000000000140020  FF D3 41 B8 F0 B5 A2 56 68 04 00 00 00 5A 48 89  ÿÓA¸ðµ¢Vh....ZH.
0000000000140030  F9 FF D0 00 00 00 00 00 00 00 00 00 F8 00 00 00  ùÿÐ.........ø...
0000000000140040  0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68  ..º..´.Í!¸.LÍ!Th
0000000000140050  69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F  is program canno
0000000000140060  74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20  t be run in DOS 
0000000000140070  6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00  mode....$.......
```

Deobfuscating Bacon.exe’s Second Stage

- In a debugger, we set a breakpoint on VirtualProtect
- Qiling can hook APIs *but* may encounter APIs not implemented
- **`ql.os.set_api("VirtualProtect", hook_vp)`**
- Intercept types (3rd parameter):
  - **`QL_INTERCEPT.CALL`**: Execute handler instead of API implementation (default).
  - **`QL_INTERCEPT.ENTER`**: Execute handler before API is called.
  - **`QL_INTERCEPT.EXIT`**: Execute handler on exit.

VirtualProtect Hook Implementation

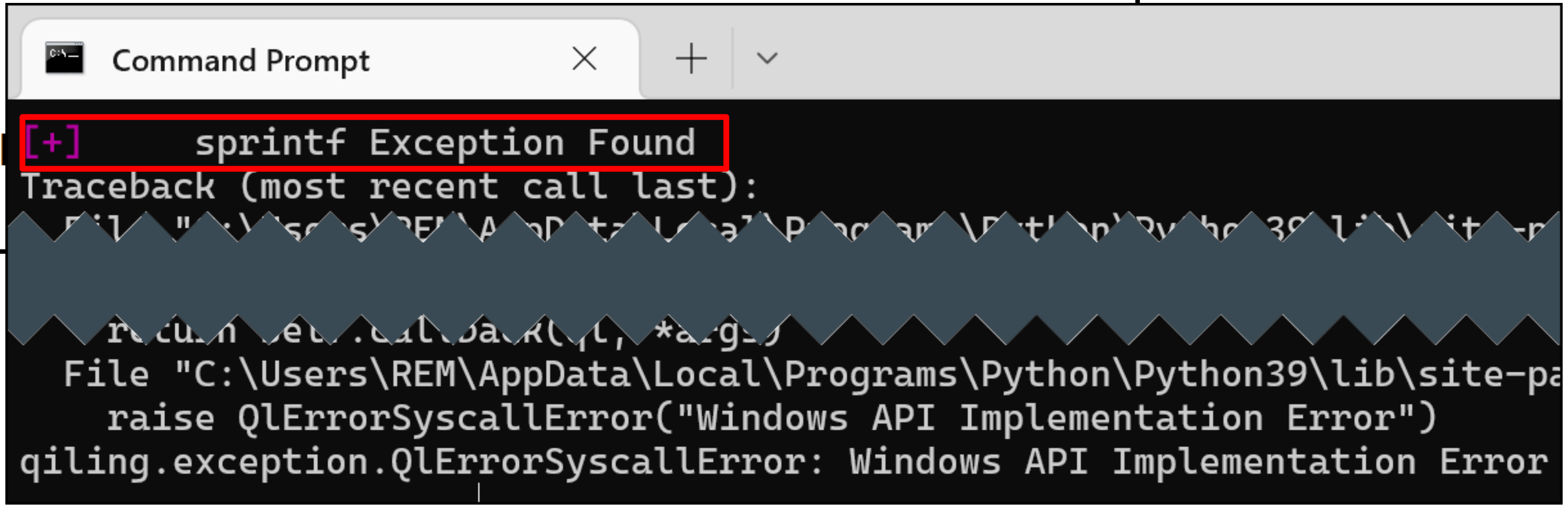


```python
@winsdkapi(cc=STDCALL, params={
    'lpAddress'  : LPVOID,
    'dwSize'    : SIZE_T,
})
def hook_vp(ql, address, params):
    lpAddress = params['lpAddress']
    dwSize = params['dwSize']

    #Read memory
    data = ql.mem.read(lpAddress, dwSize)

    #Write file
    file_name = hex(lpAddress) + "_"+str(dwSize) + ".bin"
    with open(file_name, "wb") as f:
        f.write(data)

    #Stop emulation
    ql.emu_stop()
    ql.os.PE_RUN = False

    print(f"Created file named {file_name}")
```

- Use **`@winsdkapi`** decorator for hooks
  - A *decorator* function takes another function as an argument
  - Include the calling convention and a dictionary of parameters
- Qiling memory methods:
  - **`ql.mem.read(address, size)`**
  - **`ql.mem.write(address, data)`**

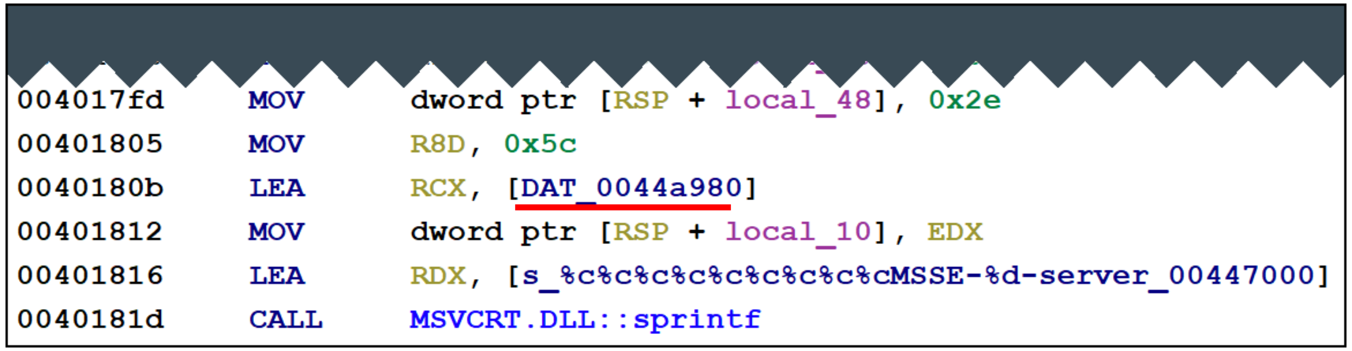
Debugging with Verbose Logging

```
from qiling import *
from qiling.const import QL_VERBOSE

SAMPLE_PATH = "C:\\Tools\\qiling\\examples\\rootfs\\x8664_windows\\bacon.exe"
ROOT_FS = "C:\\Tools\\qiling\\examples\\rootfs\\x8664_windows"
ql = Qiling([SAMPLE_PATH], ROOT_FS, verbose=QL_VERBOSE.DEBUG)

#VirtualProtect Handler Code
...

ql.os.set_api("VirtualProtect", 
ql.run()
```

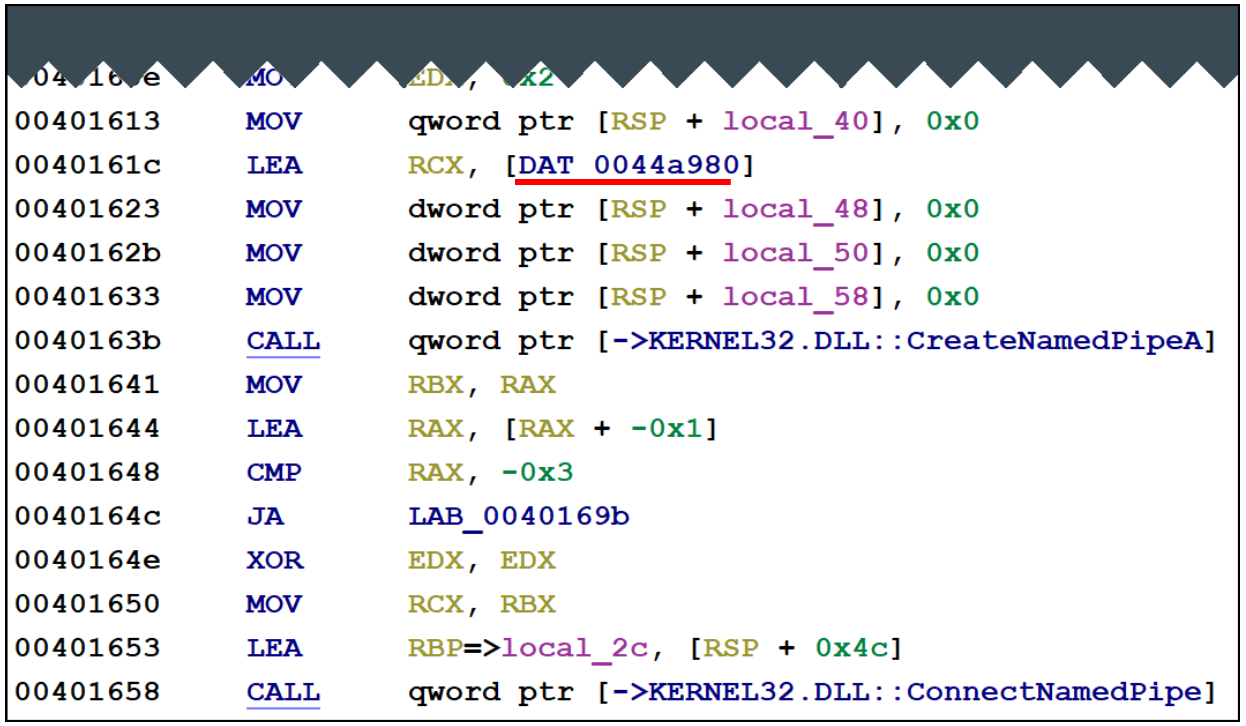


```
[+]     sprintf Exception Found
Traceback (most recent call last):
  [illegible]
    [illegible]
  File "C:\Users\REM\AppData\Local\Programs\Python\Python39\lib\site-pa
    raise QlErrorSyscallError("Windows API Implementation Error")
qiling.exception.QlErrorSyscallError: Windows API Implementation Error
```

Sprintf() Context



```
004017fd    MOV     dword ptr [RSP + local_48], 0x2e
00401805    MOV     R8D, 0x5c
0040180b    LEA     RCX, [DAT_0044a980]
00401812    MOV     dword ptr [RSP + local_10], EDX
00401816    LEA     RDX, [s_%c%c%c%c%c%c%c%c%cMSSE-%d-server_00447000]
0040181d    CALL    MSVCRT.DLL::sprintf
```

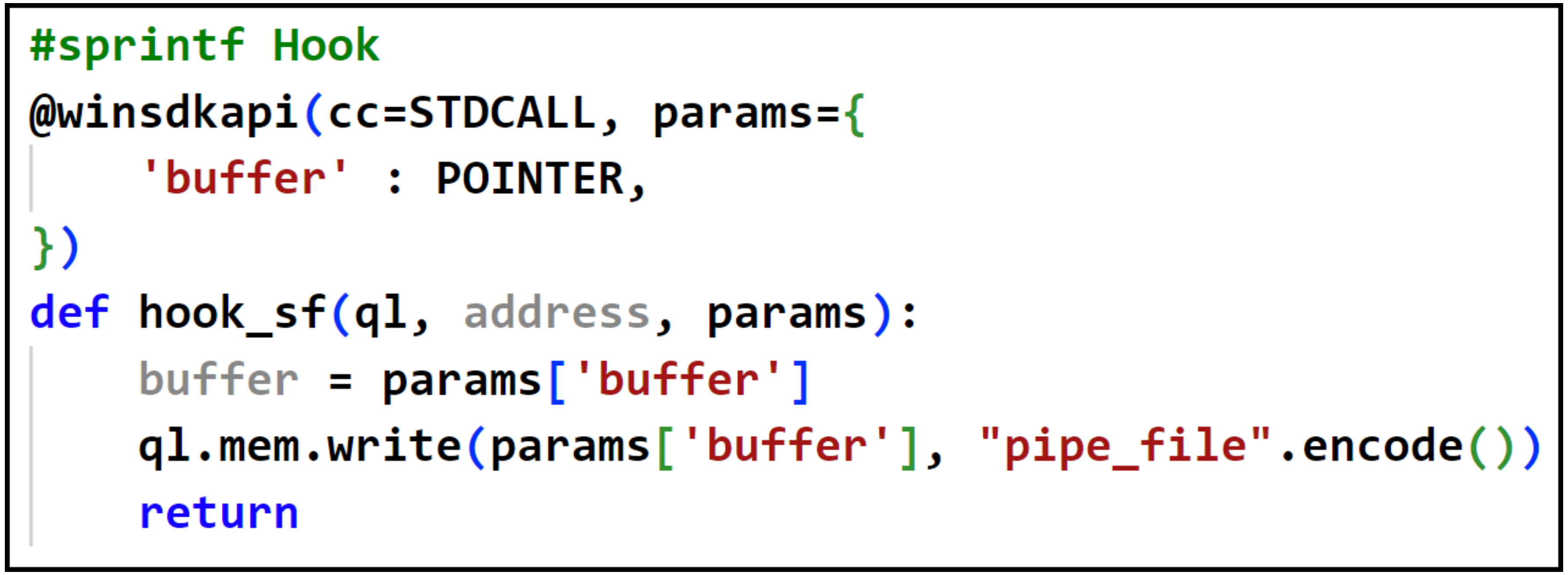


```
00401613    MOV     qword ptr [RSP + local_40], 0x0
0040161c    LEA     RCX, [DAT_0044a980]
00401623    MOV     dword ptr [RSP + local_48], 0x0
0040162b    MOV     dword ptr [RSP + local_50], 0x0
00401633    MOV     dword ptr [RSP + local_58], 0x0
0040163b    CALL    qword ptr [->KERNEL32.DLL::CreateNamedPipeA]
00401641    MOV     RBX, RAX
00401644    LEA     RAX, [RAX + -0x1]
00401648    CMP     RAX, -0x3
0040164c    JA      LAB_0040169b
0040164e    XOR     EDX, EDX
00401650    MOV     RCX, RBX
00401653    LEA     RBP=>local_2c, [RSP + 0x4c]
00401658    CALL    qword ptr [->KERNEL32.DLL::ConnectNamedPipe]
```

- `sprintf()` creates a formatted string
- **`CreateNamedPipeA()`** creates a pipe with that name
- **`ConnectNamedPipe()`** connects to the named pipe.
- Encoded content is written to the pipe
- Later, that content is read and decoded to produce a DLL

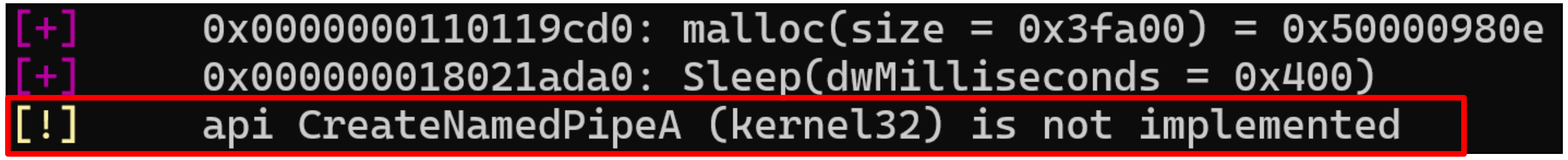
Sprintf() Hook

- Given the context of **`sprintf()`**, only the first parameter is necessary.
- Use **`ql.mem.write(address, data)`** to write a string to the buffer.



```
#sprintf Hook
@winsdkapi(cc=STDCALL, params={
    'buffer' : POINTER,
})
def hook_sf(ql, address, params):
    buffer = params['buffer']
    ql.mem.write(params['buffer'], "pipe_file".encode())
    return
```

Additional Hooks



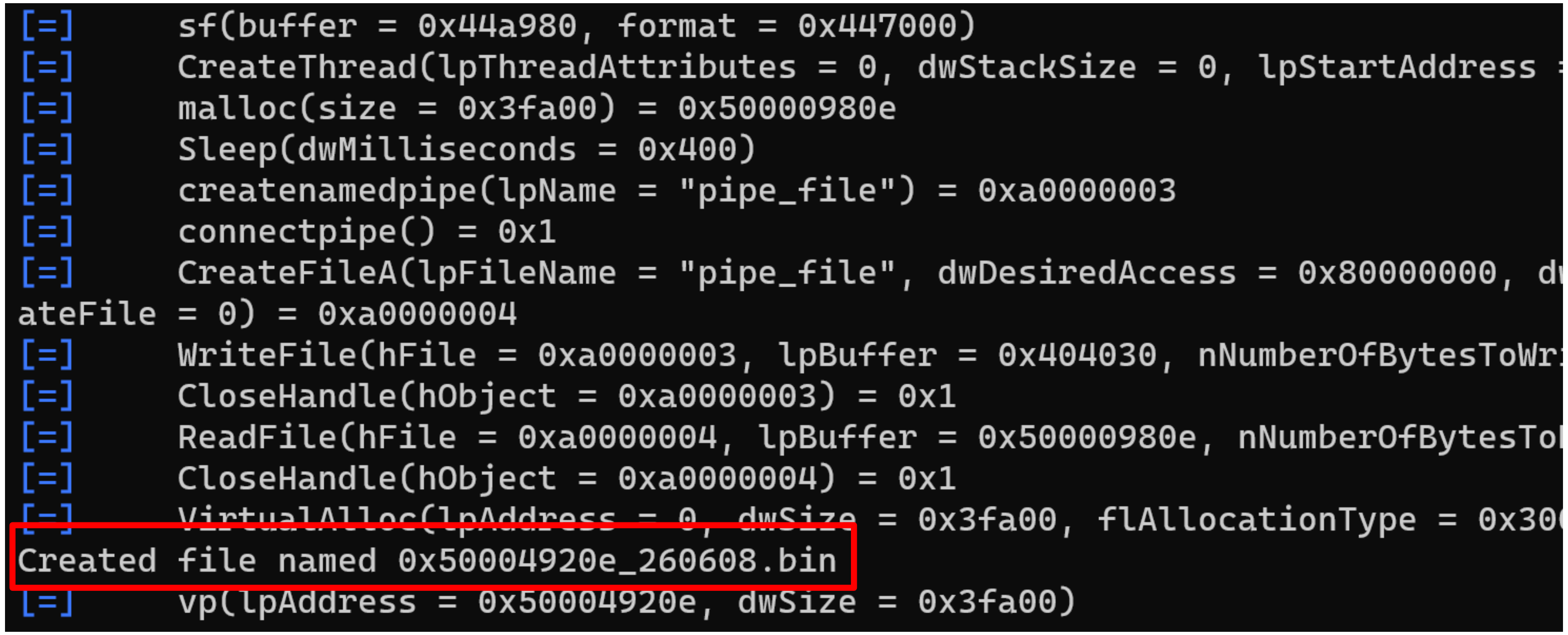
```
[+]     0x0000000110119cd0: malloc(size = 0x3fa00) = 0x50000980e
[+]     0x000000018021ada0: Sleep(dwMilliseconds = 0x400)
[!]     api CreateNamedPipeA (kernel32) is not implemented
```

- Interactions with a named pipe are like file interactions.
- Use Qiling's **`_CreateFile`** method to create a file on disk.

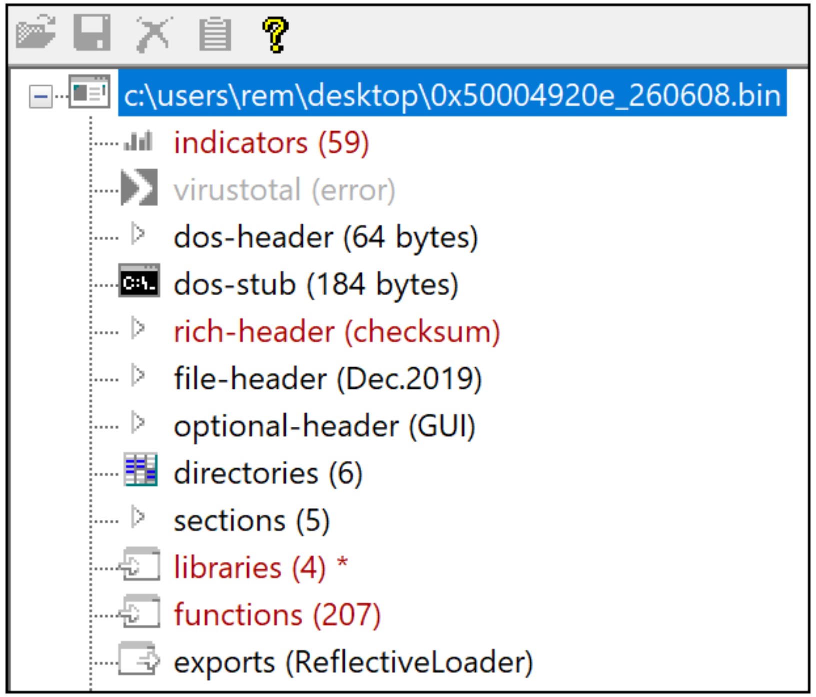
```
from qiling.os.windows.dlls.kernel32.fileapi import _CreateFile

#CreateNamedPipeA Hook
@winsdkapi(cc=STDCALL, params={
    'lpName' : LPCSTR
})
def hook_createnamedpipe(ql, address, params):
    filename = params['lpName']
    new_params = {}
    new_params['lpFileName'] =  filename
    new_params['dwDesiredAccess'] = (GENERIC_READ | GENERIC_WRITE)
    file_handle = _CreateFile(ql, address, new_params)
    return file_handle
```

Hook APIs and Execute Emulation

```
ql.os.set_api("sprintf", hook_sf)
ql.os.set_api("CreateNamedPipeA", hook_createnamedpipe)
ql.os.set_api("VirtualProtect", hook_vp)

ql.run()
```



```
[=]     sf(buffer = 0x44a980, format = 0x447000)
[=]     CreateThread(lpThreadAttributes = 0, dwStackSize = 0, lpStartAddress 
[=]     malloc(size = 0x3fa00) = 0x50000980e
[=]     Sleep(dwMilliseconds = 0x400)
[=]     createnamedpipe(lpName = "pipe_file") = 0xa0000003
[=]     connectpipe() = 0x1
[=]     CreateFileA(lpFileName = "pipe_file", dwDesiredAccess = 0x80000000, dw
ateFile = 0) = 0xa0000004
[=]     WriteFile(hFile = 0xa0000003, lpBuffer = 0x404030, nNumberOfBytesToWr
[=]     CloseHandle(hObject = 0xa0000003) = 0x1
[=]     ReadFile(hFile = 0xa0000004, lpBuffer = 0x50000980e, nNumberOfBytesTo
[=]     CloseHandle(hObject = 0xa0000004) = 0x1
[=]     VirtualAlloc(lpAddress = 0, dwSize = 0x3fa00, flAllocationType = 0x30
Created file named 0x50004920e_260608.bin
[=]     vp(lpAddress = 0x50004920e, dwSize = 0x3fa00)
```



c:\users\rem\desktop\0x50004920e_260608.bin
- indicators (59)
- virustotal (error)
- dos-header (64 bytes)
- dos-stub (184 bytes)
- rich-header (checksum)
- file-header (Dec.2019)
- optional-header (GUI)
- directories (6)
- sections (5)
- libraries (4) *
- functions (207)
- exports (ReflectiveLoader)

Closing Thoughts

- Emulation is a powerful option to automate malware analysis
- It can tackle complexity and facilitate scalability
- It works, but it isn't

- There are growing number of frameworks to choose from
- With each emulation script you write, the next one gets easier
- Unicorns aren't just for kids

Thank you

E-mail: anuj.soni@gmail.com

YouTube: https://youtube.com/@sonianuj

X: @asoni

BlackBerry. Intelligent Security. Everywhere.