Writing Config Extractors

Navigating the challenges in extracting malware artifacts

Botconf 2024

About Us

Souhail Hammou

Souhail is a senior malware reverse engineer with the Intel 471 Malware Intelligence team. He’s actively involved in reverse engineering malware and developing tools such as extractors and network protocol emulators to track malware and botnet activities.

Miroslav Stampar, PhD

Miroslav is a senior security engineer with the Intel 471 Malware Intelligence team. He's actively involved in lots of “miscellaneous” stuff.

Agenda

- Introduction
- Lab setup - TLP:AMBER+STRICT
- *Part I* General methodologies - TLP:AMBER+STRICT
- *Part II* Regular expressions for code: RisePro hands-on - TLP:GREEN
- *Part III* Using Unicorn and Capstone: Emotet hands-on - TLP:GREEN
- Conclusion and Appendices - TLP:GREEN

Introduction

- A malware's configuration is one of its most valuable assets.
- Samples typically come with a static configuration, sometimes minimal, that is set by the operator(s) before distribution.
  - Not expected to change unless explicitly updated or extended remotely.
- A malware config holds:
  - Command-and-Control (C2) servers.
  - Encryption keys and other configuration parameters.

INTEL471

Introduction

- Being able to extract elements of a malware's config allows defenders to:
    - Publish IOCs.
    - Emulate the malware's communications protocol (more IOCs).
    - Thwart or deeply inconvenience malware operators.
- For that reason, malware developers will:
    - Use obfuscation/encryption to conceal the config.
    - React to OSINT articles and config extractor scripts.

Lab setup

N/A in TLP:WHITE

Part I - General methodologies

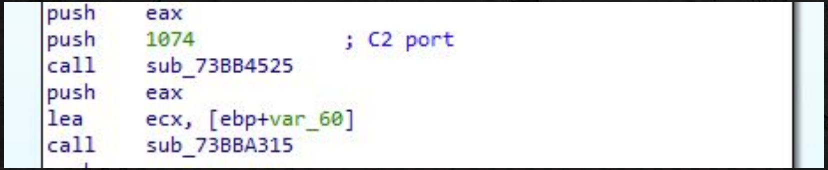
N/A in TLP:WHITE

Part II - Regular expressions for code

- When and why you need to use regular expressions?
- Limitations of using regex to match code.
- Introducing Coderex.
- 2 Hands-on tasks.

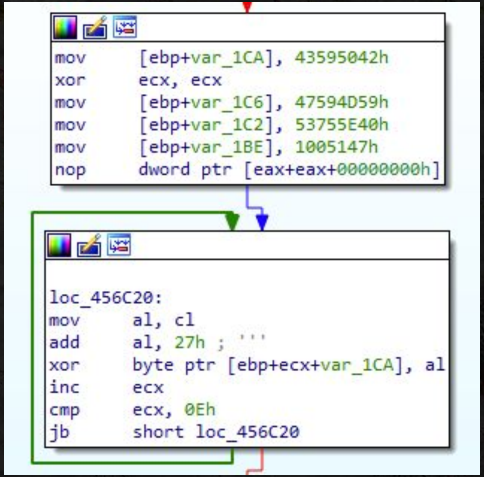
Regular expressions

- Sooner or later you will need to locate code in malware samples.
- For example:
    - To defeat encrypted stack strings.
    - To extract constants in the code.
    - To emulate instructions.



```
push    eax
push    1074            ; C2 port
call    sub_73BB4525
push    eax
lea     ecx, [ebp+var_60]
call    sub_73BBA315
```

Backconnect C2 TCP Port (Socks5SystemZ)



```
mov     [ebp+var_1CA], 43595042h
xor     ecx, ecx
mov     [ebp+var_1C6], 47594D59h
mov     [ebp+var_1C2], 53755E40h
mov     [ebp+var_1BE], 1005147h
nop     dword ptr [eax+eax+00000000h]
```

```
loc_456C20:
mov     al, cl
add     al, 27h ; '''
xor     byte ptr [ebp+ecx+var_1CA], al
inc     ecx
cmp     ecx, 0Eh
jb      short loc_456C20
```

Encrypted Stack Strings (RisePro)

INTEL471

Why regex?

- Wildcard opcodes using a '.' operator

  `re.compile(rb'\xB0.\xC3', re.DOTALL)        # MOV AL, XXh + RETN`

- Alternation operator:

  `rb'\x75.|\x0F\x85....'        # jnz short OR jnz near`

- Ranges:

  `rb'[\x50-\x57]'        # PUSH r32`

Why regex?

- Wildcard an arbitrary number of instructions:

  `rb'.{0,120}?'` *# Lazy match between 0 and 120 bytes of code*

- Groups allow capturing data, especially at arbitrary offsets in dynamic buffers:

  `rb'\xE8.{8,128}?\x68(?P<tcp_port>..\x00\x00).{1,6}?\xE8'` *# PUSH tcp_port*

E8 CALL	8 to 128 bytes	**68 (?? ?? 00 00) push (tcp_port)**	1 to 6 bytes	E8 CALL

Lazy mode

- Consume pattern as few times as possible and expand to yield the shortest match.

```
.{8,64}?\xE8
```

- Allows to retrieve the first occurrence in code.
- In most cases you’ll want to use lazy quantifiers when wildcarding a range of bytes.

Lazy mode

```
rb‘\xE8.{8,128}?\x68(?P<tcp_port>..\x00\x00).{1,6}?\xE8’
```

```
E8 FB FF FF 00          call   0x1000000
8D 4A 20                lea    ecx, [edx+0x20]
8B 41 04                mov    eax,DWORD PTR [ecx+0x4]
56                      push   esi
8B 31                   mov    esi,DWORD PTR [ecx]
68 BB 01 00 00          push   0x1bb
68 01 00 00 00          push   0x1
56                      push   esi
E8                      call
```

Lazy mode

```
rb‘\xE8.{8,128}?\x68(?P<tcp_port>..\x00\x00).{1,6}?\xE8’
```

```
E8 FB FF FF 00        call   0x1000000
8D 4A 20              lea    ecx, [edx+0x20]
8B 41 04              mov    eax,DWORD PTR [ecx+0x4]
56                    push   esi
8B 31                 mov    esi,DWORD PTR [ecx]
68 BB 01 00 00        push   0x1bb
68 01 00 00 00        push   0x1
56                    push   esi
E8                    call
```

Lazy mode

```
rb‘\xE8.{8,128}?\x68(?P<tcp_port>..\x00\x00).{1,6}?\xE8’
```

```
E8 FB FF FF 00          call   0x1000000
8D 4A 20                lea     ecx, [edx+0x20]
8B 41 04                mov     eax,DWORD PTR [ecx+0x4]
56                      push   esi
8B 31                   mov     esi,DWORD PTR [ecx]
68 BB 01 00 00          push   0x1bb
68 01 00 00 00          push   0x1
56                      push   esi
E8                      call
```

INTEL471

Lazy mode

```
rb'\xE8.{8,128}?\x68(?P<tcp_port>..\x00\x00).{1,6}?\xE8'
```

```
E8 FB FF FF 00          call   0x1000000
8D 4A 20                lea     ecx, [edx+0x20]
8B 41 04                mov     eax,DWORD PTR [ecx+0x4]
56                      push   esi
8B 31                   mov     esi,DWORD PTR [ecx]
68 BB 01 00 00          push   0x1bb                ; Port 443 (decimal)
68 01 00 00 00          push   0x1
56                      push   esi
E8                      call
```

Lazy mode

```
rb‘\xE8.{8,128}?\x68(?P<tcp_port>..\x00\x00).{1,6}?\xE8’
```

```
E8 FB FF FF 00          call   0x1000000
8D 4A 20                lea     ecx, [edx+0x20]
8B 41 04                mov     eax,DWORD PTR [ecx+0x4]
56                      push   esi
8B 31                   mov     esi,DWORD PTR [ecx]
68 BB 01 00 00          push   0x1bb                 ; Port 443 (decimal)
68 01 00 00 00          push   0x1
56                      push   esi
E8                      call
```

Greedy mode

- Consume as much as possible then backtrack to yield longest match.

  `.{8,64}\xE8`

- Allows to retrieve the last occurrence.
- Can cause bugs when used incorrectly to extract artifacts.

Greedy mode

```
rb'\xE8.{8,128}\x68(?P<tcp_port>..\x00\x00).{1,6}\xE8'
```

```
E8 FB FF FF 00          call   0x1000000
8D 4A 20                lea    ecx, [edx+0x20]
8B 41 04                mov    eax,DWORD PTR [ecx+0x4]
56                      push   esi
8B 31                   mov    esi,DWORD PTR [ecx]
68 BB 01 00 00          push   0x1bb
68 01 00 00 00          push   0x1
56                      push   esi
E8                      call
```

Greedy mode

```
rb‘\xE8.{8,128}\x68(?P<tcp_port>..\x00\x00).{1,6}\xE8’
```

```
E8 FB FF FF 00          call   0x1000000
8D 4A 20                lea    ecx, [edx+0x20]
8B 41 04                mov    eax,DWORD PTR [ecx+0x4]
56                      push   esi
8B 31                   mov    esi,DWORD PTR [ecx]
68 BB 01 00 00          push   0x1bb
68 01 00 00 00          push   0x1
56                      push   esi
E8                      call
```

Greedy mode

```
rb‘\xE8.{8,128}\x68(?P<tcp_port>..\x00\x00).{1,6}\xE8’
```

```
E8 FB FF FF 00       call   0x1000000
8D 4A 20             lea    ecx, [edx+0x20]
8B 41 04             mov    eax,DWORD PTR [ecx+0x4]
56                   push   esi
8B 31                mov    esi,DWORD PTR [ecx]
68 BB 01 00 00       push   0x1bb
68 01 00 00 00       push   0x1
56                   push   esi
E8                   call
```

Port was skipped

Greedy mode

```
rb'\xE8.{8,128}\x68(?P<tcp_port>..\x00\x00).{1,6}\xE8'
```

```
E8 FB FF FF 00          call   0x1000000
8D 4A 20                lea    ecx, [edx+0x20]
8B 41 04                mov    eax,DWORD PTR [ecx+0x4]
56                      push   esi
8B 31                   mov    esi,DWORD PTR [ecx]
68 BB 01 00 00          push   0x1bb
68 01 00 00 00          push   0x1
56                      push   esi
E8                      call
```

Incorrect port in group

tcp_port != 0x01

Greedy mode

```
rb'\xE8.{8,128}\x68(?P<tcp_port>..\x00\x00).{1,6}\xE8'
```

```
E8 FB FF FF 00        call   0x1000000
8D 4A 20              lea    ecx, [edx+0x20]
8B 41 04              mov    eax,DWORD PTR [ecx+0x4]
56                    push   esi
8B 31                 mov    esi,DWORD PTR [ecx]
68 BB 01 00 00        push   0x1bb
68 01 00 00 00        push   0x1
56                    push   esi
E8                    call
```

Incorrect port in group

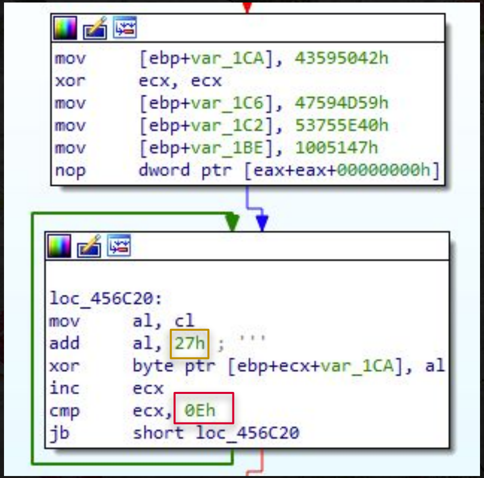
tcp_port != 0x01

Lazy vs. Greedy

- Opt for lazy mode when regex is FP prone:
    - The pattern after the range quantifier is too generic.
- The choice of which mode heavily depends on the specific use-case.

Regex: hands-on

- Stack string decryption loop in the RisePro stealer.
- We want to match and extract the XOR key and string length.
- Decryption process:
  - Add the current index to the initial XOR key: value is 0x27 (changes between samples).
  - XOR byte on stack, increment the index and compare the length.



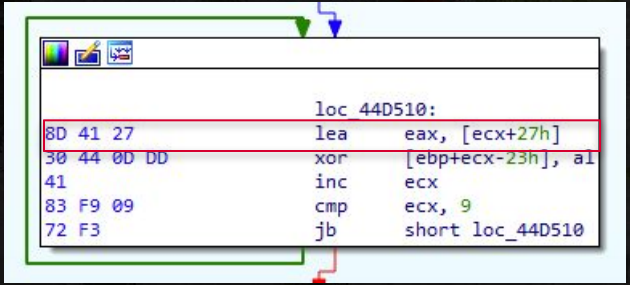
```
mov     [ebp+var_1CA], 43595042h
xor     ecx, ecx
mov     [ebp+var_1C6], 47594D59h
mov     [ebp+var_1C2], 53755E40h
mov     [ebp+var_1BE], 1005147h
nop     dword ptr [eax+eax+00000000h]
```

```
loc_456C20:
mov     al, cl
add     al, 27h ; '''
xor     byte ptr [ebp+ecx+var_1CA], al
inc     ecx
cmp     ecx, 0Eh
jb      short loc_456C20
```

Key = 0x27, Length = 14

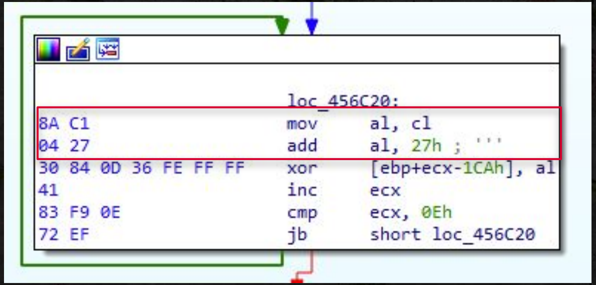
Regex: hands-on

- Decryption process:
  - Add the current index to the initial XOR key: value is 0x27 (changes between samples).
  - XOR byte on stack, increment the index and compare the length.
- Samples has 2 equivalent variants of how the key is calculated: LEA vs. MOV + ADD



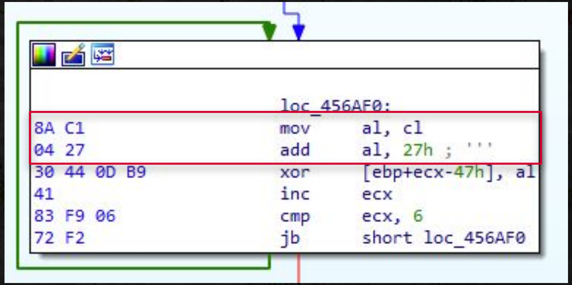
```
                     loc_44D510:
8D 41 27             lea     eax, [ecx+27h]
30 44 0D DD          xor     [ebp+ecx-23h], al
41                   inc     ecx
83 F9 09             cmp     ecx, 9
72 F3                jb      short loc_44D510
```

Length == 9



```
                     loc_456C20:
8A C1                mov     al, cl
04 27                add     al, 27h ; '''
30 84 0D 36 FE FF FF xor     [ebp+ecx-1CAh], al
41                   inc     ecx
83 F9 0E             cmp     ecx, 0Eh
72 EF                jb      short loc_456C20
```

Length == 14

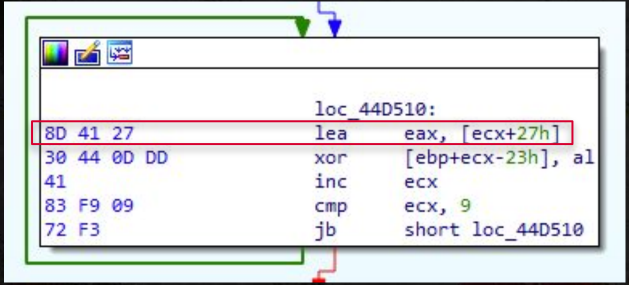


```
                     loc_456AF0:
8A C1                mov     al, cl
04 27                add     al, 27h ; '''
30 44 0D B9          xor     [ebp+ecx-47h], al
41                   inc     ecx
83 F9 06             cmp     ecx, 6
72 F2                jb      short loc_456AF0
```

Length == 6

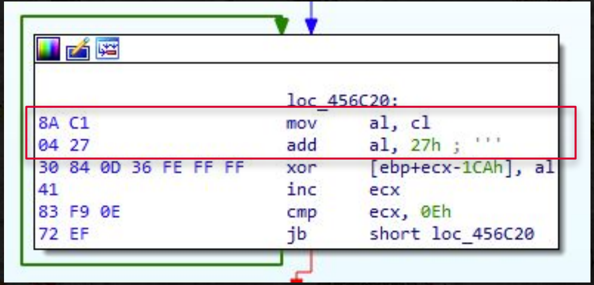
Regex: hands-on

```
re.compile(
        (
                rb’(\x8A\xC1\x04|\x8D\x41)(?P<key>.)’
```



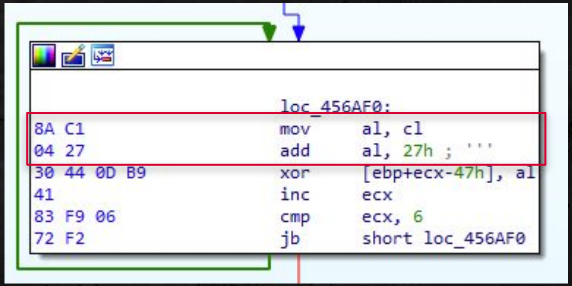
```
                          loc_44D510:
8D 41 27                  lea     eax, [ecx+27h]
30 44 0D DD               xor     [ebp+ecx-23h], al
41                        inc     ecx
83 F9 09                  cmp     ecx, 9
72 F3                     jb      short loc_44D510
```

Length == 9



```
                          loc_456C20:
8A C1                     mov     al, cl
04 27                     add     al, 27h ; '''
30 84 0D 36 FE FF FF      xor     [ebp+ecx-1CAh], al
41                        inc     ecx
83 F9 0E                  cmp     ecx, 0Eh
72 EF                     jb      short loc_456C20
```

Length == 14

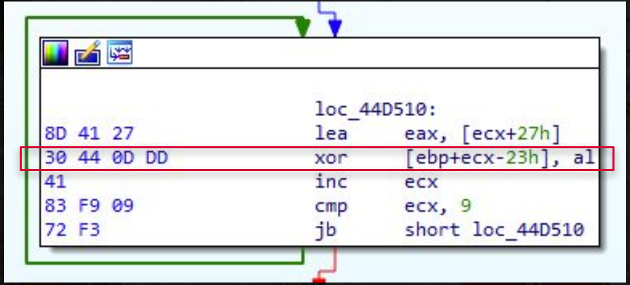


```
                          loc_456AF0:
8A C1                     mov     al, cl
04 27                     add     al, 27h ; '''
30 44 0D B9               xor     [ebp+ecx-47h], al
41                        inc     ecx
83 F9 06                  cmp     ecx, 6
72 F2                     jb      short loc_456AF0
```

Length == 6

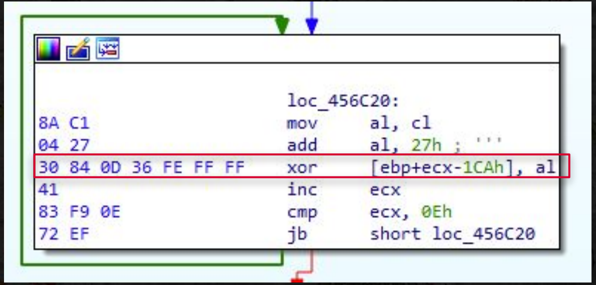
Regex: hands-on

```
re.compile(
        (
                rb'(\x8A\xC1\x04|\x8D\x41)(?P<key>.)'
                rb'\x30(\x44\x0D.|\x84\x0D....)'
```



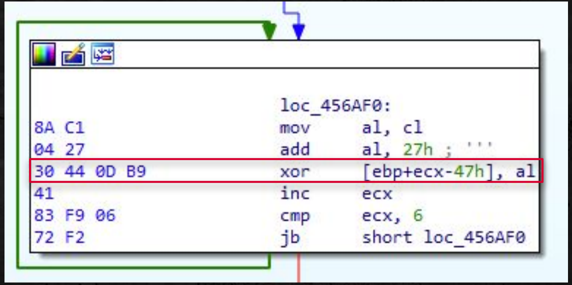
```
                         loc_44D510:
8D 41 27                 lea     eax, [ecx+27h]
30 44 0D DD              xor     [ebp+ecx-23h], al
41                       inc     ecx
83 F9 09                 cmp     ecx, 9
72 F3                    jb      short loc_44D510
```

Length == 9



```
                         loc_456C20:
8A C1                    mov     al, cl
04 27                    add     al, 27h ; '''
30 84 0D 36 FE FF FF     xor     [ebp+ecx-1CAh], al
41                       inc     ecx
83 F9 0E                 cmp     ecx, 0Eh
72 EF                    jb      short loc_456C20
```

Length == 14



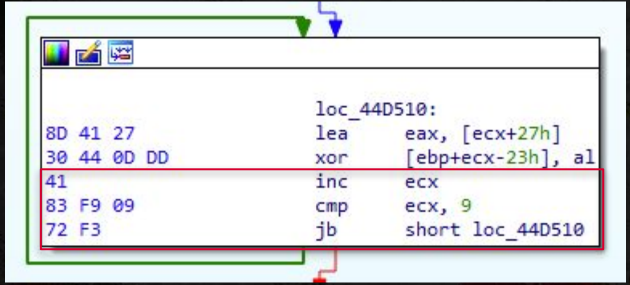
```
                         loc_456AF0:
8A C1                    mov     al, cl
04 27                    add     al, 27h ; '''
30 44 0D B9              xor     [ebp+ecx-47h], al
41                       inc     ecx
83 F9 06                 cmp     ecx, 6
72 F2                    jb      short loc_456AF0
```

Length == 6

INTEL471

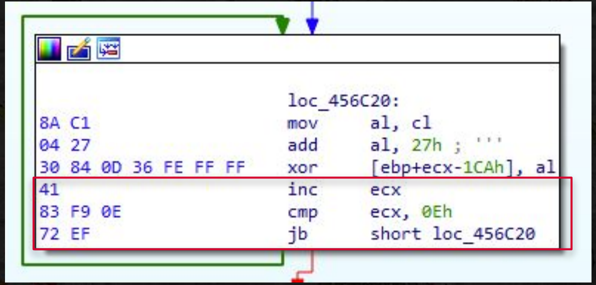
Regex: hands-on

```
re.compile(
    (
        rb'(\x8A\xC1\x04|\x8D\x41)(?P<key>.)'
        rb'\x30(\x44\x0D.|\x84\x0D....)'
        rb'\x41'
        rb'\x83\xF9(?P<len>.)'
        rb'\x72'
    ), re.DOTALL)
```



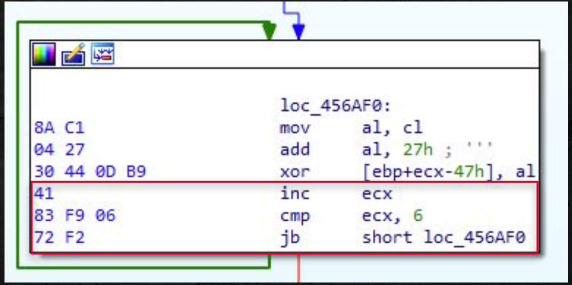
```
                    loc_44D510:
8D 41 27            lea     eax, [ecx+27h]
30 44 0D DD         xor     [ebp+ecx-23h], al
41                  inc     ecx
83 F9 09            cmp     ecx, 9
72 F3               jb      short loc_44D510
```

Length == 9



```
                          loc_456C20:
8A C1                     mov     al, cl
04 27                     add     al, 27h ; '''
30 84 0D 36 FE FF FF      xor     [ebp+ecx-1CAh], al
41                        inc     ecx
83 F9 0E                  cmp     ecx, 0Eh
72 EF                     jb      short loc_456C20
```

Length == 14

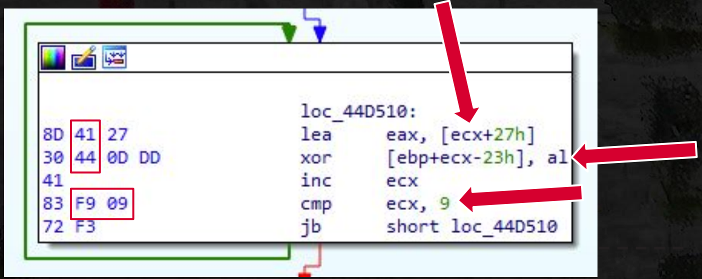


```
                          loc_456AF0:
8A C1                     mov     al, cl
04 27                     add     al, 27h ; '''
30 44 0D B9               xor     [ebp+ecx-47h], al
41                        inc     ecx
83 F9 06                  cmp     ecx, 6
72 F2                     jb      short loc_456AF0
```

Length == 6

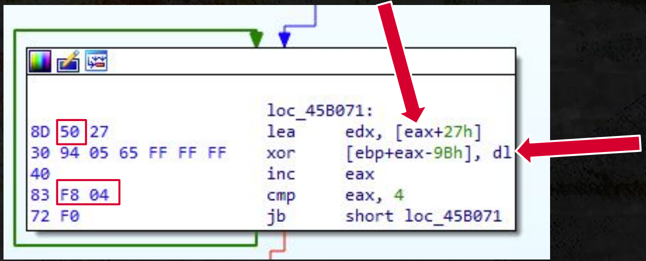
Regex: challenges

- The regular expression we wrote does not match all decryption loops.
  - Some loops use registers we did not take into account.
  - Regex needs to be improved to match these blocks.
- Code is volatile. Regex is better suited for text.

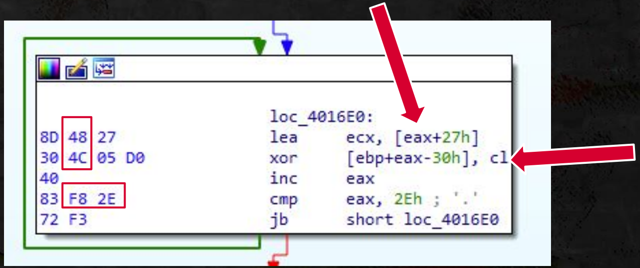


```
                      loc_44D510:
8D 41 27              lea     eax, [ecx+27h]
30 44 0D DD           xor     [ebp+ecx-23h], al
41                    inc     ecx
83 F9 09              cmp     ecx, 9
72 F3                 jb      short loc_44D510
```

Example we worked on



```
                               loc_45B071:
8D 50 27                       lea     edx, [eax+27h]
30 94 05 65 FF FF FF           xor     [ebp+eax-9Bh], dl
40                             inc     eax
83 F8 04                       cmp     eax, 4
72 F0                          jb      short loc_45B071
```



```
                               loc_4016E0:
8D 48 27                       lea     ecx, [eax+27h]
30 4C 05 D0                    xor     [ebp+eax-30h], cl
40                             inc     eax
83 F8 2E                       cmp     eax, 2Eh ; '.'
72 F3                          jb      short loc_4016E0
```

Other loops

Regex: challenges

- The x86 and x86-64 instruction sets are tricky. For example:
  - A lot of instructions support a memory operand as either src or dest.
  - But never two memory operands.
  - Requires the instruction to be encoded two different ways.
  - When both operands are registers, the two encodings become equivalent.
  - The choice of which encoding to use is left to the assembler.
- A variance to consider.

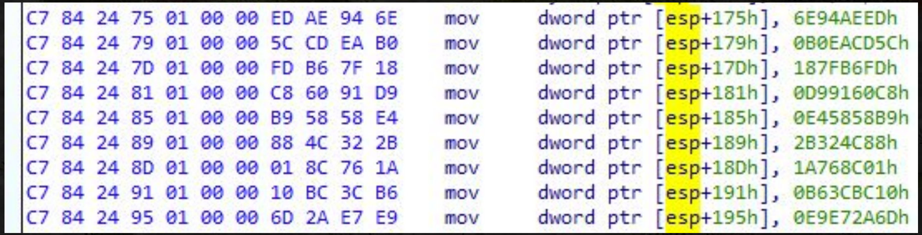
```
89 c6      mov    esi, eax      ; MOV r/m32 , r32
8b f0      mov    esi, eax      ; MOV r32   , r/m32
```

Regex: challenges

- Stack-frame indexing:
    - Usually by the stack-frame base pointer **EBP** e.g. [EBP-0x0C].
    - But ESP could be used as a base e.g. [ESP+0x0C].
        - Frame-Pointer Omission optimization.
- An additional variance to consider.

```
C7 45 98 94 98 89 91     mov     dword ptr [ebp-68h], 91899894h
C7 45 9C 8F AB AE AC     mov     dword ptr [ebp-64h], 0ACAEAB8Fh
C7 45 A0 B8 95 80 A7     mov     dword ptr [ebp-60h], 0A78095B8h
C7 45 A4 B0 B3 AE A2     mov     dword ptr [ebp-5Ch], 0A2AEB3B0h
C7 45 A8 B5 96 AF B0     mov     dword ptr [ebp-58h], 0B0AF96B5h
C7 45 AC A3 AD AA B7     mov     dword ptr [ebp-54h], 0B7AAADA3h
C7 45 B0 AA B0 B6 B4     mov     dword ptr [ebp-50h], 0B4B6B0AAh
C7 45 B4 7E 93 91 B1     mov     dword ptr [ebp-4Ch], 0B191937Eh
C7 45 B8 A3 84 67 A8     mov     dword ptr [ebp-48h], 0A86784A3h
```

Pikabot (**EBP**-based frame indexing)

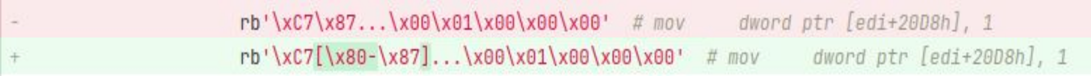


```
C7 84 24 75 01 00 00 ED AE 94 6E     mov     dword ptr [esp+175h], 6E94AEEDh
C7 84 24 79 01 00 00 5C CD EA B0     mov     dword ptr [esp+179h], 0B0EACD5Ch
C7 84 24 7D 01 00 00 FD B6 7F 18     mov     dword ptr [esp+17Dh], 187FB6FDh
C7 84 24 81 01 00 00 C8 60 91 D9     mov     dword ptr [esp+181h], 0D99160C8h
C7 84 24 85 01 00 00 B9 58 58 E4     mov     dword ptr [esp+185h], 0E45858B9h
C7 84 24 89 01 00 00 88 4C 32 2B     mov     dword ptr [esp+189h], 2B324C88h
C7 84 24 8D 01 00 00 01 8C 76 1A     mov     dword ptr [esp+18Dh], 1A768C01h
C7 84 24 91 01 00 00 10 BC 3C B6     mov     dword ptr [esp+191h], 0B63CBC10h
C7 84 24 95 01 00 00 6D 2A E7 E9     mov     dword ptr [esp+195h], 0E9E72A6Dh
```

Pikabot (**ESP**-based frame indexing)

Regex: challenges

- What seemed to be a trivial regex to write would end up:
    - Taking time to find variants and to anticipate any future code changes.
    - Gathering technical debt.

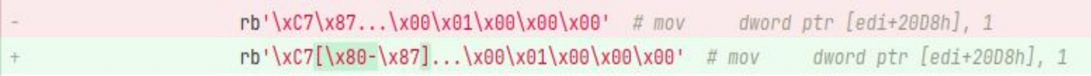


```
-                rb'\xC7\x87...\x00\x01\x00\x00\x00'  # mov      dword ptr [edi+20D8h], 1
+                rb'\xC7[\x80-\x87]...\x00\x01\x00\x00\x00'  # mov      dword ptr [edi+20D8h], 1
```

Example commit 1: Builds appeared that would use other registers than EDI

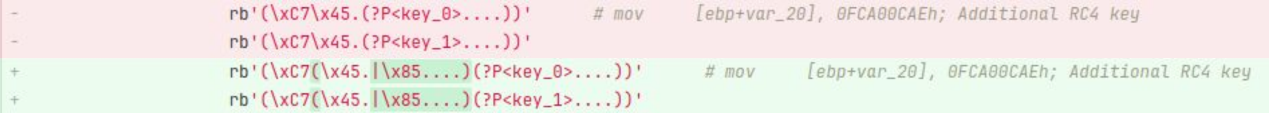
Regex: challenges

- What seemed to be a trivial regex to write would end up:
  - Taking time to find variants and to anticipate any future code changes.
  - Gathering technical debt.



```
-                rb'\xC7\x87...\x00\x01\x00\x00\x00'  # mov     dword ptr [edi+20D8h], 1
+                rb'\xC7[\x80-\x87]...\x00\x01\x00\x00\x00'  # mov     dword ptr [edi+20D8h], 1
```

Example commit 1: Builds appeared that would use other registers than EDI

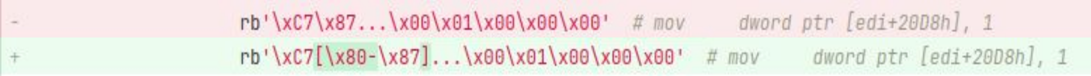


```
-                rb'(\xC7\x45.(?P<key_0>....))'      # mov     [ebp+var_20], 0FCA00CAEh; Additional RC4 key
-                rb'(\xC7\x45.(?P<key_1>....))'
+                rb'(\xC7(\x45.|\x85....)(?P<key_0>....))'      # mov     [ebp+var_20], 0FCA00CAEh; Additional RC4 key
+                rb'(\xC7(\x45.|\x85....)(?P<key_1>....))'
```

Example commit 2: In some samples, functions had a larger variable space.

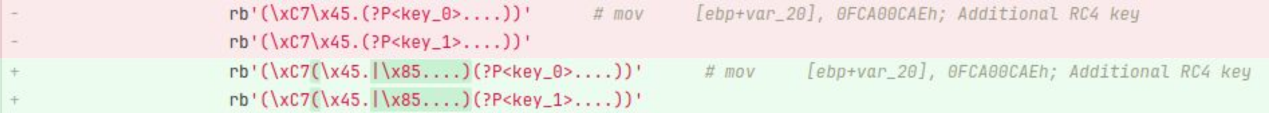
Regex: challenges

- What seemed to be a trivial regex to write would end up:
    - Taking time to find variants and to anticipate any future code changes.
    - Gathering technical debt.



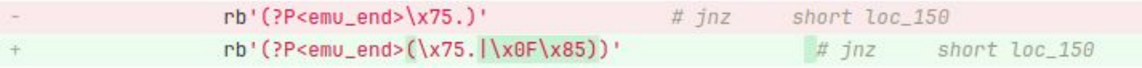
```
-               rb'\xC7\x87...\x00\x01\x00\x00\x00'  # mov     dword ptr [edi+20D8h], 1
+               rb'\xC7[\x80-\x87]...\x00\x01\x00\x00\x00'  # mov     dword ptr [edi+20D8h], 1
```

Example commit 1: Builds appeared that would use other registers than EDI



```
-               rb'(\xC7\x45.(?P<key_0>....))'      # mov     [ebp+var_20], 0FCA00CAEh; Additional RC4 key
-               rb'(\xC7\x45.(?P<key_1>....))'
+               rb'(\xC7(\x45.|\x85....)(?P<key_0>....))'      # mov     [ebp+var_20], 0FCA00CAEh; Additional RC4 key
+               rb'(\xC7(\x45.|\x85....)(?P<key_1>....))'
```

Example commit 2: In some samples, functions had a larger variable space.



```
-               rb'(?P<emu_end>\x75.)'              # jnz     short loc_150
+               rb'(?P<emu_end>(\x75.|\x0F\x85))'              # jnz     short loc_150
```

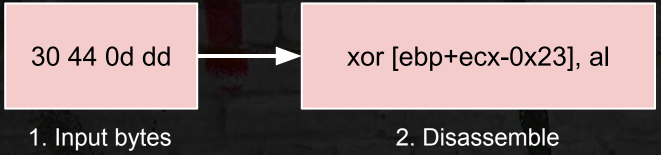
Example commit 3: The branch target could be farther. ‘JNZ near’ is possible.

Introducing Coderex

- Experimental tool we developed to tackle these challenges.
- Generates generic regular expression given a stream of code (x86 or x86-64).
- Relies on the **iced** assembler/disassembler (https://github.com/icedland/iced).
- Released today on: https://github.com/intel471/coderex

Introducing Coderex

- Experimental tool we developed to tackle these challenges.
- Generates generic regular expression given a stream of code (x86 or x86-64).
- Relies on the **iced** assembler/disassembler (https://github.com/icedland/iced).
- Released today on: https://github.com/intel471/coderex



30 44 0d dd → xor [ebp+ecx-0x23], al

1. Input bytes

2. Disassemble

Introducing Coderex

- Experimental tool we developed to tackle these challenges.
- Generates generic regular expression given a stream of code (x86 or x86-64).
- Relies on the **iced** assembler/disassembler (https://github.com/icedland/iced).
- Released today on: https://github.com/intel471/coderex

30 44 0d dd

1. Input bytes

xor [ebp+ecx-0x23], al

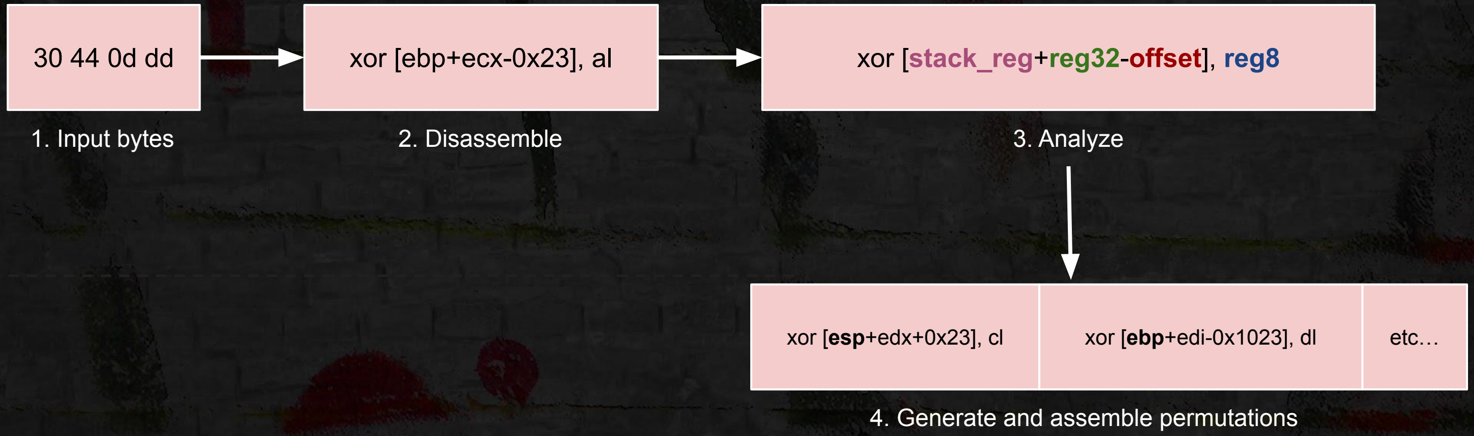
2. Disassemble

xor [**stack_reg**+**reg32**-**offset**], **reg8**

3. Analyze

Introducing Coderex

- Experimental tool we developed to tackle these challenges.
- Generates generic regular expression given a stream of code (x86 or x86-64).
- Relies on the **iced** assembler/disassembler (https://github.com/icedland/iced).
- Released today on: https://github.com/intel471/coderex



30 44 0d dd

1. Input bytes

xor [ebp+ecx-0x23], al

2. Disassemble

xor [**stack_reg**+**reg32**-**offset**], **reg8**

3. Analyze

xor [**esp**+edx+0x23], cl

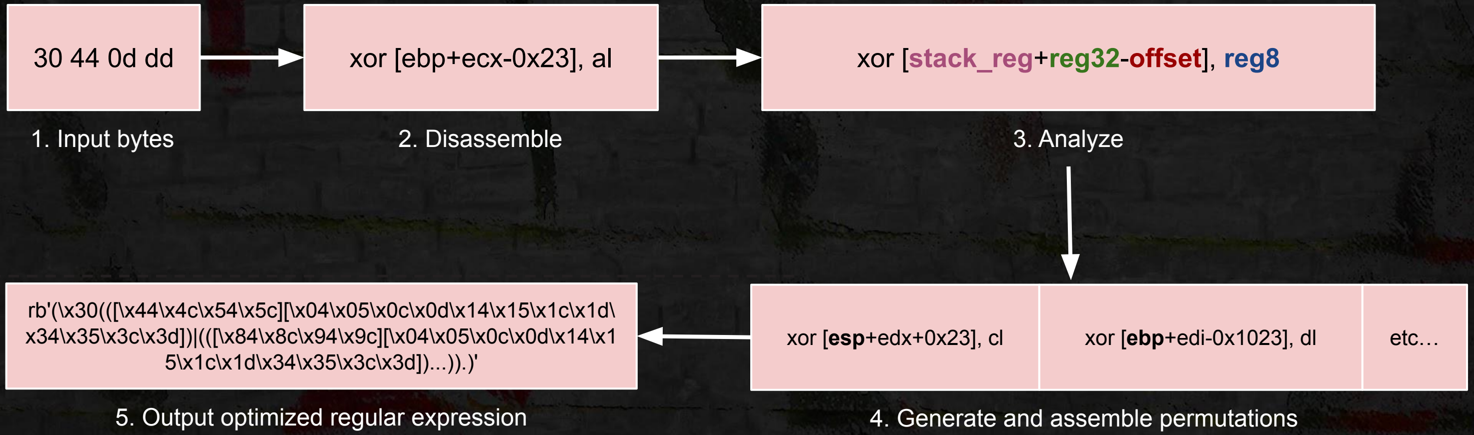
xor [**ebp**+edi-0x1023], dl

etc…

4. Generate and assemble permutations

Introducing Coderex

- Experimental tool we developed to tackle these challenges.
- Generates generic regular expression given a stream of code (x86 or x86-64).
- Relies on the **iced** assembler/disassembler (https://github.com/icedland/iced).
- Released today on: https://github.com/intel471/coderex



30 44 0d dd

1. Input bytes

xor [ebp+ecx-0x23], al

2. Disassemble

xor [**stack_reg**+**reg32**-**offset**], **reg8**

3. Analyze

xor [**esp**+edx+0x23], cl

xor [**ebp**+edi-0x1023], dl

etc…

4. Generate and assemble permutations

```
rb'(\x30(([\x44\x4c\x54\x5c][\x04\x05\x0c\x0d\x14\x15\x1c\x1d\x34\x35\x3c\x3d])|((([\x84\x8c\x94\x9c][\x04\x05\x0c\x0d\x14\x15\x1c\x1d\x34\x35\x3c\x3d])...)).)'
```

5. Output optimized regular expression

Coderex: Demo

- Generic x86 regular expression for:
  - 21 C0 and eax, eax

- Visualize the regular expression using: https://regexper.com/
- Assemble a few variants from the regex.
  - Online assembler/disassembler: https://defuse.ca/online-x86-assembler.htm
  - Notice that:
    - Coderex detected that the source and destination are the same.
    - Both r/m32 and m32/r encodings are generated.

Coderex

- In some cases we'd want to modify the regexes:
    - To add in groups.
    - Wildcard offsets.
- We need to understand the assumptions the tool makes.

Immediates

- Immediate values are preserved.

```
regex = re.compile(
        (
                # mov eax,1337h ; '.{5}'
                rb'([\xb8-\xbb\xbe\xbf]\x37\x13\x00\x00)'

                # add eax,10h ; '.{3}'
                rb'(\x83[\xc0-\xc3\xc6\xc7]\x10)'

        ), re.DOTALL
)
```

Branching

- Call targets are wildcarded.
- For relative jumps, both *short (1 byte)* and *near (4 bytes)* variants are generated.

```
# call 01001400h
rb'(\xe8....)'


# jmp near ptr 01001400h
rb'(((\xe9...)|\xeb).)'
```

Memory (Direct)

- Direct memory accesses are wildcarded.

```
# mov dword ptr ds:[1000h],1337h ; '.{10}'
rb'(\xc7\x05....\x37\x13\x00\x00)'


# call dword ptr ds:[2000h] ; '.{6}'
rb'(\xff\x15....)'


# jmp dword ptr ds:[3000h] ; '.{6}'
rb'(\xff\x25....)'
```

Memory (Displacement)

- Displacements are trickier to handle.
- Base address assumed for code/memory:
  - 0x100000 by default.
  - Only displacements above or equal will be wildcarded.

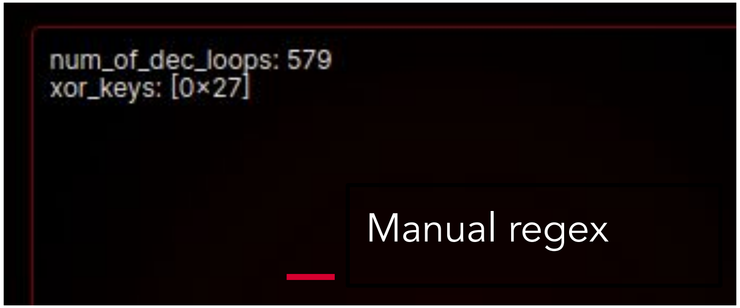
```
$ coderex -c “8d 34 b5 00 11 00 00” -a x86
regex = re.compile(
        (
                # lea esi,[esi*4+1100h] ; '.{7}'
                rb'(\x8d(\x04\x85|\x0c\x8d|\x14\x95|\x1c\x9d|\x34\xb5|\x3c\xbd)\x00\x11\x00\x00)'
        ), re.DOTALL
)
```

Memory (Displacement)

- For code that accesses low memory addresses. For example:
  - Shellcode loaded into a low address segment in IDA Pro.
- Override default base address using ‘-d’.

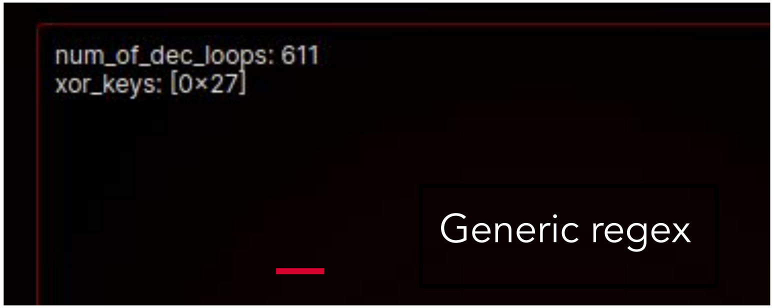
```
$ coderex -c “8d 34 b5 00 11 00 00“ -a x86 -d 0x1000
regex = re.compile(
        (
                # lea esi,[esi*4+1100h] ; '.{7}'
                rb'(\x8d(\x04\x85|\x0c\x8d|\x14\x95|\x1c\x9d|\x34\xb5|\x3c\xbd)....)'
        ), re.DOTALL
)
```

Coderex: hands-on



```
num_of_dec_loops: 579
xor_keys: [0×27]
```

Manual regex



```
num_of_dec_loops: 611
xor_keys: [0×27]
```

Generic regex

Part III - Unicorn and Capstone.

- Objective: write an Emotet config extractor for C2 servers.

- Introduce the Unicorn emulation engine:
    - To write config extractor for a single C2.
    - 1 Hands-on task.

- Introduce the Capstone disassembler engine:
    - To improve the extractor to automatically extract all C2s.
    - 2 Hands-on tasks.

Unicorn engine

- Lightweight and multi-architecture CPU emulator framework based on QEMU.
- Interprets machine instructions within a software-based context to replicate the behavior of a CPU.
- Actively maintained and widely adopted by the malware community.
- Easy to install: “pip install unicorn”

Unicorn

The Ultimate CPU emulator

www.unicorn-engine.org

Using Unicorn

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

- unicorn module:
    - Contains the "Uc" emulation class.
- x86_const module:
    - Defines x86 and x86-64 constants for registers and instructions.

```
# X86 registers

UC_X86_REG_INVALID = 0
UC_X86_REG_AH = 1
UC_X86_REG_AL = 2
UC_X86_REG_AX = 3
…
UC_X86_REG_RAX = 35
```

```
# X86 instructions
UC_X86_INS_INVALID = 0
UC_X86_INS_AAA = 1
UC_X86_INS_AAD = 2
UC_X86_INS_AAM = 3
UC_X86_INS_AAS = 4
UC_X86_INS_FABS = 5
…
```

Using Unicorn

```
# code to be emulated
X86_CODE32 = b"\x41\x4a" # INC ecx; DEC edx

# memory address where emulation starts
ADDRESS = 0x1000000
```

- Initialize the instructions to emulate.
    - 41 INC ECX
    - 4A DEC EDX
- Setup address where they will be written to in the emulator’s memory.

Using Unicorn

```
# Initialize emulator in X86-32bit mode
mu = Uc(UC_ARCH_X86, UC_MODE_32)
```

- Instantiate an Unicorn emulator object “Uc” for the target architecture.
  - UC_ARCH_X86: The x86 CPU architecture.
  - UC_MODE_32: The 32-bit CPU mode.
    - UC_MODE_64 to emulate 64-bit code.

Using Unicorn

```
# map 2MB memory for this emulation
mu.mem_map(ADDRESS, 2 * 1024 * 1024)

# write machine code to be emulated to memory
mu.mem_write(ADDRESS, X86_CODE32)
```

- In a fresh emulator, all memory is unmapped.
  - Raises an exception when accessed.
- Call “mem_map” to map memory with:
  - The memory address to map in the address space.
  - The memory size in bytes aligned to page boundary (4096).
  - The permissions, by default RWX.
- Memory can then be written to using mem_write.

Using Unicorn

```
# initialize machine registers
mu.reg_write(UC_X86_REG_ECX, 0x1234)

mu.reg_write(UC_X86_REG_EDX, 0x7890)
```

- Initialize registers affected by calling “reg_write”:
    - ECX = 0x1234
    - EDX = 0x7890
- All registers are initially 0.

Using Unicorn

```
# emulate code in infinite time & unlimited instructions
mu.emu_start(ADDRESS, ADDRESS + len(X86_CODE32))
```

- Call “emu_start” with the arguments:
    - Start address of the code: 0x1000000.
    - End address: 0x1000002.
- “emu_start” synchronously emulates the code.
    - Only returns when the end address is reached or an exception occurs.
    - Can get stuck indefinitely or for a long time.

Using Unicorn

```
# emulate code in infinite time & unlimited instructions
mu.emu_start(ADDRESS, ADDRESS + len(X86_CODE32))
```

- “emu_start” accepts 1 of these 2 arguments to control the emulator’s execution.
  - “count”: A number that limits the individual instructions to execute.
  - “timeout”: The maximum runtime of the emulation in nanoseconds.
    - 10 milliseconds: 10 * UC_MILISECOND_SCALE
    - 10 seconds: 10 * UC_SECOND_SCALE

Using Unicorn

```
# emulate code in infinite time & unlimited instructions
mu.emu_start(ADDRESS, ADDRESS + len(X86_CODE32))
```

- Emulation stops at whichever comes first:
  - The instruction pointer reached the end address.
  - The “count” or “timeout” has expired.
  - An exception occured.

Using Unicorn

```
# now print out some registers
print("Emulation done. Below is the CPU context")

r_ecx = mu.reg_read(UC_X86_REG_ECX)
r_edx = mu.reg_read(UC_X86_REG_EDX)
print(">>> ECX = 0x%x" %r_ecx)
print(">>> EDX = 0x%x" %r_edx)
```

- After emulation:
    - ECX = 0x1234 + 1 = 0x1235
    - EDX = 0x7890 - 1 = 0x788f

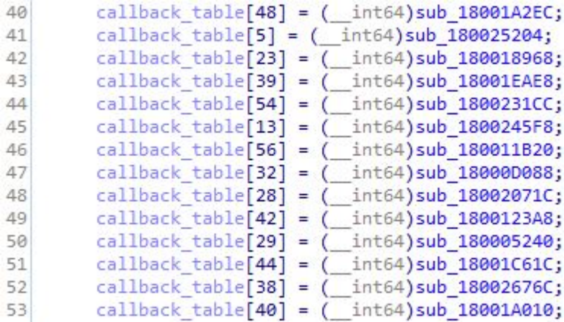
Using Unicorn

- To learn more about the engine's capabilities, find examples at:
    - https://github.com/unicorn-engine/unicorn/blob/master/bindings/python/sample_x86.py

Case-study: Emotet

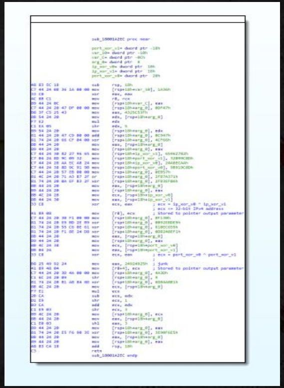
- In May 2022, Emotet updated its C2 config storage method.
    - Callback table of 64 functions.
    - Each function decodes and returns the C2 IP and port.
    - The C2 can be a decoy.
    - The table is walked until a functional C2 is found.

- Callback functions:
    - 1 basic block.
    - Mostly junk code.
    - Uses XOR to calculate the C2 IP and port.



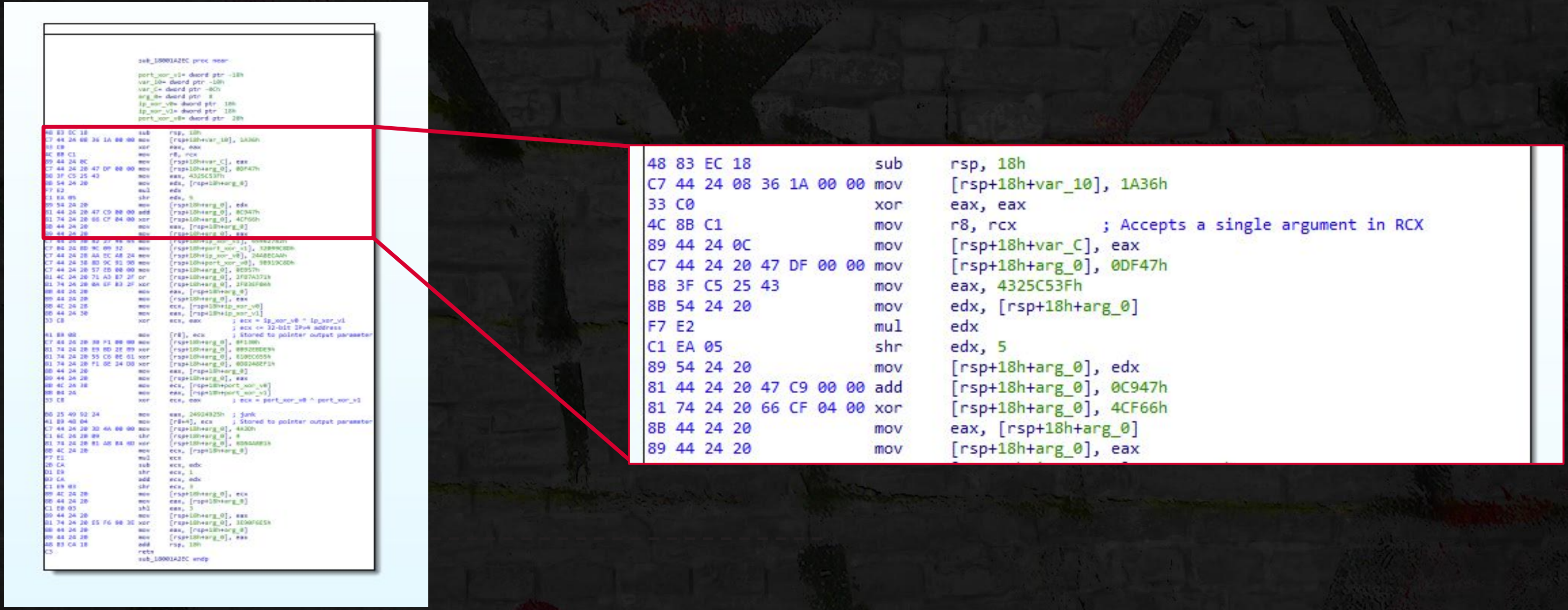
```
    callback_table[48] = (__int64)sub_18001A2EC;
    callback_table[5] = (__int64)sub_180025204;
    callback_table[23] = (__int64)sub_180018968;
    callback_table[39] = (__int64)sub_18001EAE8;
    callback_table[54] = (__int64)sub_1800231CC;
    callback_table[13] = (__int64)sub_1800245F8;
    callback_table[56] = (__int64)sub_180011B20;
    callback_table[32] = (__int64)sub_18000D088;
    callback_table[28] = (__int64)sub_18002071C;
    callback_table[42] = (__int64)sub_1800123A8;
    callback_table[29] = (__int64)sub_180005240;
    callback_table[44] = (__int64)sub_18001C61C;
    callback_table[38] = (__int64)sub_18002676C;
    callback_table[40] = (__int64)sub_18001A010;
```

Callback table being initialized



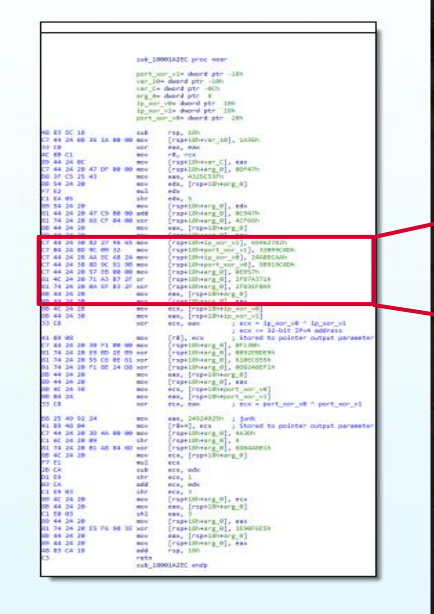
Callback function

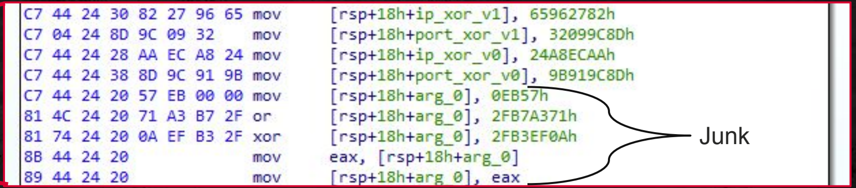
Callback function



```
48 83 EC 18                sub     rsp, 18h
C7 44 24 08 36 1A 00 00 mov     [rsp+18h+var_10], 1A36h
33 C0                      xor     eax, eax
4C 8B C1                   mov     r8, rcx          ; Accepts a single argument in RCX
89 44 24 0C                mov     [rsp+18h+var_C], eax
C7 44 24 20 47 DF 00 00 mov     [rsp+18h+arg_0], 0DF47h
B8 3F C5 25 43             mov     eax, 4325C53Fh
8B 54 24 20                mov     edx, [rsp+18h+arg_0]
F7 E2                      mul     edx
C1 EA 05                   shr     edx, 5
89 54 24 20                mov     [rsp+18h+arg_0], edx
81 44 24 20 47 C9 00 00 add     [rsp+18h+arg_0], 0C947h
81 74 24 20 66 CF 04 00 xor     [rsp+18h+arg_0], 4CF66h
8B 44 24 20                mov     eax, [rsp+18h+arg_0]
89 44 24 20                mov     [rsp+18h+arg_0], eax
```

Callback function

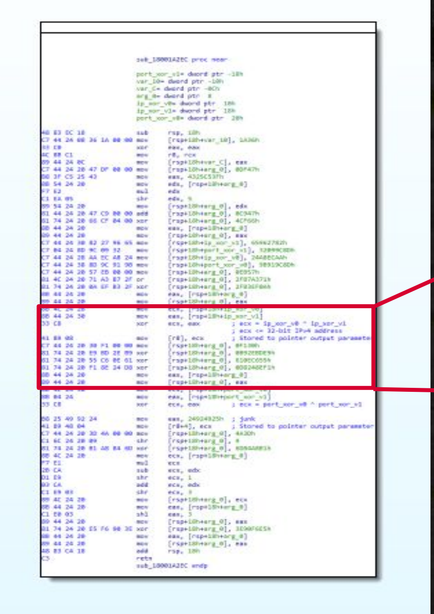


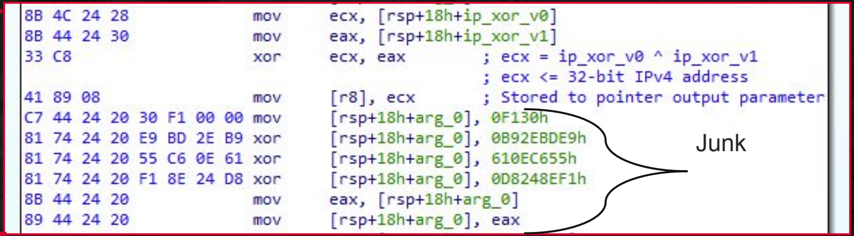


```
C7 44 24 30 82 27 96 65 mov     [rsp+18h+ip_xor_v1], 65962782h
C7 04 24 8D 9C 09 32    mov     [rsp+18h+port_xor_v1], 32099C8Dh
C7 44 24 28 AA EC A8 24 mov     [rsp+18h+ip_xor_v0], 24A8ECAAh
C7 44 24 38 8D 9C 91 9B mov     [rsp+18h+port_xor_v0], 9B919C8Dh
C7 44 24 20 57 EB 00 00 mov     [rsp+18h+arg_0], 0EB57h
81 4C 24 20 71 A3 B7 2F or      [rsp+18h+arg_0], 2FB7A371h
81 74 24 20 0A EF B3 2F xor     [rsp+18h+arg_0], 2FB3EF0Ah
8B 44 24 20             mov     eax, [rsp+18h+arg_0]
89 44 24 20             mov     [rsp+18h+arg_0], eax
```

- XOR parameters are written to the stack.
    - IPv4 parts: ip_xor_v1, ip_xor_v2
    - Port parts: port_xor_v1, port_xor_v2

Callback function

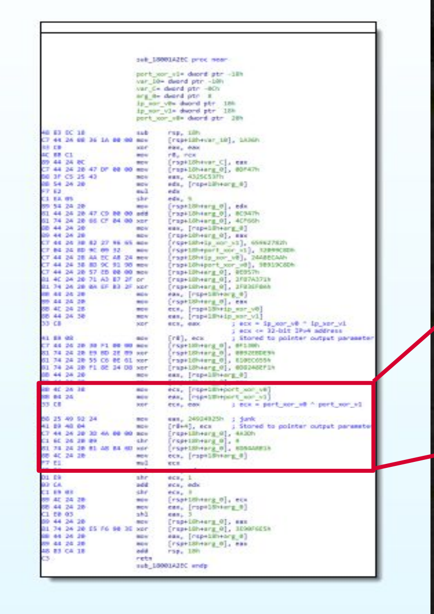


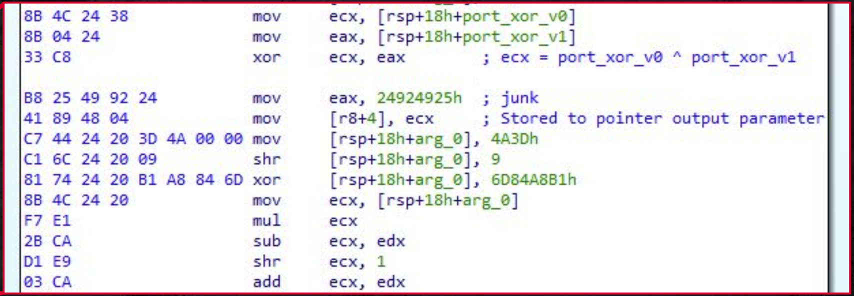


```
8B 4C 24 28              mov     ecx, [rsp+18h+ip_xor_v0]
8B 44 24 30              mov     eax, [rsp+18h+ip_xor_v1]
33 C8                    xor     ecx, eax          ; ecx = ip_xor_v0 ^ ip_xor_v1
                                                   ; ecx <= 32-bit IPv4 address
41 89 08                 mov     [r8], ecx         ; Stored to pointer output parameter
C7 44 24 20 30 F1 00 00 mov     [rsp+18h+arg_0], 0F130h
81 74 24 20 E9 BD 2E B9 xor     [rsp+18h+arg_0], 0B92EBDE9h
81 74 24 20 55 C6 0E 61 xor     [rsp+18h+arg_0], 610EC655h
81 74 24 20 F1 8E 24 D8 xor     [rsp+18h+arg_0], 0D8248EF1h
8B 44 24 20              mov     eax, [rsp+18h+arg_0]
89 44 24 20              mov     [rsp+18h+arg_0], eax
```

- ip_xor_v0 ^ ip_xor_v1 = IPv4 in network-byte order.
- Stored to the output argument in offset 0.

Callback function





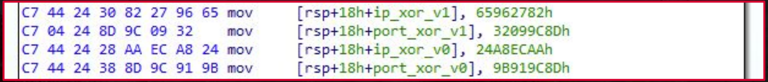
```
8B 4C 24 38             mov     ecx, [rsp+18h+port_xor_v0]
8B 04 24                mov     eax, [rsp+18h+port_xor_v1]
33 C8                   xor     ecx, eax        ; ecx = port_xor_v0 ^ port_xor_v1

B8 25 49 92 24          mov     eax, 24924925h  ; junk
41 89 48 04             mov     [r8+4], ecx     ; Stored to pointer output parameter
C7 44 24 20 3D 4A 00 00 mov     [rsp+18h+arg_0], 4A3Dh
C1 6C 24 20 09          shr     [rsp+18h+arg_0], 9
81 74 24 20 B1 A8 84 6D xor     [rsp+18h+arg_0], 6D84A8B1h
8B 4C 24 20             mov     ecx, [rsp+18h+arg_0]
F7 E1                   mul     ecx
2B CA                   sub     ecx, edx
D1 E9                   shr     ecx, 1
03 CA                   add     ecx, edx
```

- port_xor_v0 ^ port_xor_v1 = 32-bit integer
- Stored to the output argument in offset 4.
- Encodes two 16-bit values:
    - offset 0x04: 0 if it's a decoy C2, 1 if real.
    - offset 0x06: The port integer in little-endian.

Extraction options

- Use regular expressions:
  - Junk code => unpredictability.
    - Inserted at compile-time.
    - Could break or interfere with regex patterns.
  - Risks:
    - We could extract wrong XOR parts (junk code).
    - New samples could change the XOR to an ADD or SUB etc.
    - Extractor logic may yield wrong but valid IPv4 addresses.

Junk



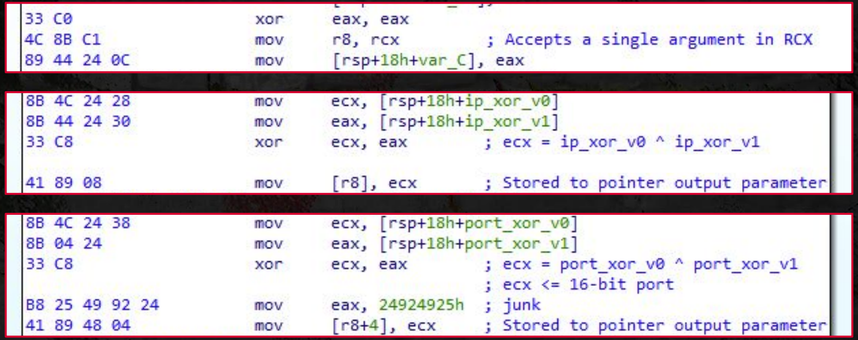
```
C7 44 24 30 82 27 96 65 mov     [rsp+18h+ip_xor_v1], 65962782h
C7 04 24 8D 9C 09 32    mov     [rsp+18h+port_xor_v1], 32099C8Dh
C7 44 24 28 AA EC A8 24 mov     [rsp+18h+ip_xor_v0], 24A8ECAAh
C7 44 24 38 8D 9C 91 9B mov     [rsp+18h+port_xor_v0], 9B919C8Dh
```

Extraction options

- Use Code Emulation:
    - Treat callbacks as gray-boxes:
        - Emulate the function's x64 instructions.
        - Read out the results from the emulator's memory.
    - Arithmetic operator changes will have no effect e.g. XOR, ADD, SUB

Emulating a callback: Hands-on

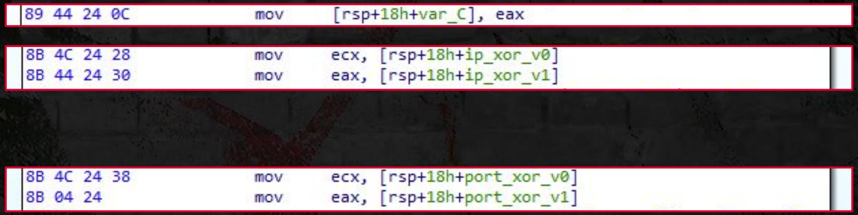
- We'll start by emulating a single function.
- We will automate the collection and emulation of all functions later on.



```
33 C0                  xor     eax, eax
4C 8B C1               mov     r8, rcx          ; Accepts a single argument in RCX
89 44 24 0C            mov     [rsp+18h+var_C], eax

8B 4C 24 28            mov     ecx, [rsp+18h+ip_xor_v0]
8B 44 24 30            mov     eax, [rsp+18h+ip_xor_v1]
33 C8                  xor     ecx, eax         ; ecx = ip_xor_v0 ^ ip_xor_v1

41 89 08               mov     [r8], ecx        ; Stored to pointer output parameter

8B 4C 24 38            mov     ecx, [rsp+18h+port_xor_v0]
8B 04 24               mov     eax, [rsp+18h+port_xor_v1]
33 C8                  xor     ecx, eax         ; ecx = port_xor_v0 ^ port_xor_v1
                                                ; ecx <= 16-bit port
B8 25 49 92 24         mov     eax, 24924925h   ; junk
41 89 48 04            mov     [r8+4], ecx      ; Stored to pointer output parameter
```

Emulating a callback: Hands-on

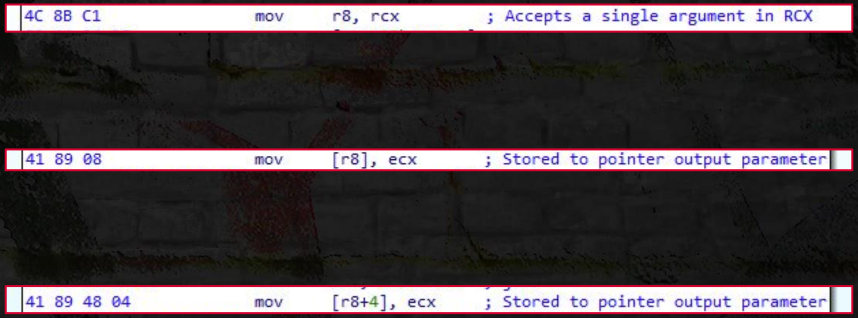
- Callbacks use the stack:
    - To store the artifact parts.
- We have to map memory for the stack in the emulator's memory.
- Set RSP to point somewhere in the middle.
    - Stack grows downwards.



```
89 44 24 0C              mov     [rsp+18h+var_C], eax

8B 4C 24 28              mov     ecx, [rsp+18h+ip_xor_v0]
8B 44 24 30              mov     eax, [rsp+18h+ip_xor_v1]

8B 4C 24 38              mov     ecx, [rsp+18h+port_xor_v0]
8B 04 24                 mov     eax, [rsp+18h+port_xor_v1]
```

Emulating a callback: Hands-on

- Callbacks expect a single argument:
    - Output memory location.
    - Size: 8 bytes.
- We have to map a memory region for the output argument.
- Set RCX to its address.



```
4C 8B C1                mov     r8, rcx          ; Accepts a single argument in RCX
```

```
41 89 08                mov     [r8], ecx        ; Stored to pointer output parameter
```

```
41 89 48 04             mov     [r8+4], ecx      ; Stored to pointer output parameter
```

Emulating a callback: Hands-on

- Emulate the function.
- Read the function's output from the passed in argument:
    - Offset <u>0x00</u>: IPv4.
    - Offset <u>0x04</u>: Is real or decoy C2?
    - Offset <u>0x06</u>: Port.
- Validate the IP and port and format into a C2:
    - e.g. "https://129.232.188.93:443"

0x00	0x04	0x06
IPv4 address (network-byte order aka big-endian)	Is Real C2? (little-endian)	Port (little-endian)
4 bytes	2 bytes	2 bytes

Extracting all controllers

- Find all callbacks and emulate each one of them.

Steps

- Step 1: Find possible callback functions using a regular expression.
- Step 2: Validate them by examining the disassembly.
  - Must have a single basic block.
    - No jumps.
  - Must not call to another function.
- Step 3: Emulate validated callbacks.

Steps

- Step 1: Find possible callback functions using a regular expression.

- Step 2: Validate them by examining the disassembly.
  - Must have a single basic block.
    - No jumps.
  - Must not call to another function.

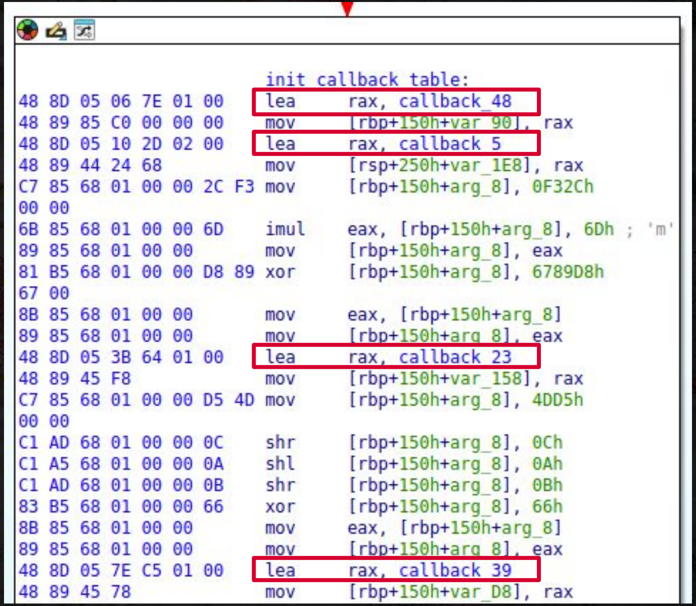
- Step 3: Emulate validated callbacks.

Steps

- Step 1: Find possible callback functions using a regular expression.
- Step 2: Validate them by examining the disassembly.
  - Must have a single basic block.
    - No jumps.
  - Must not call to another function.
- Step 3: Emulate validated callbacks ✅

Step 1: Find possible callbacks

- A LEA instruction loads the address of each callback function into a 64-bit register.
- Create a regex to match this pattern across the whole binary.
- We only want to process references to code.
  - Validate match and xref are in the same section.
- Will still match functions that are not C2 callbacks.
  - Filtered out in the next step.



```
                            init_callback_table:
48 8D 05 06 7E 01 00        lea     rax, callback_48
48 89 85 C0 00 00 00        mov     [rbp+150h+var_90], rax
48 8D 05 10 2D 02 00        lea     rax, callback_5
48 89 44 24 68              mov     [rsp+250h+var_1E8], rax
C7 85 68 01 00 00 2C F3 mov     [rbp+150h+arg_8], 0F32Ch
00 00
6B 85 68 01 00 00 6D        imul    eax, [rbp+150h+arg_8], 6Dh ; 'm'
89 85 68 01 00 00           mov     [rbp+150h+arg_8], eax
81 B5 68 01 00 00 D8 89 xor     [rbp+150h+arg_8], 6789D8h
67 00
8B 85 68 01 00 00           mov     eax, [rbp+150h+arg_8]
89 85 68 01 00 00           mov     [rbp+150h+arg_8], eax
48 8D 05 3B 64 01 00        lea     rax, callback_23
48 89 45 F8                 mov     [rbp+150h+var_158], rax
C7 85 68 01 00 00 D5 4D mov     [rbp+150h+arg_8], 4DD5h
00 00
C1 AD 68 01 00 00 0C        shr     [rbp+150h+arg_8], 0Ch
C1 A5 68 01 00 00 0A        shl     [rbp+150h+arg_8], 0Ah
C1 AD 68 01 00 00 0B        shr     [rbp+150h+arg_8], 0Bh
83 B5 68 01 00 00 66        xor     [rbp+150h+arg_8], 66h
8B 85 68 01 00 00           mov     eax, [rbp+150h+arg_8]
89 85 68 01 00 00           mov     [rbp+150h+arg_8], eax
48 8D 05 7E C5 01 00        lea     rax, callback_39
48 89 45 78                 mov     [rbp+150h+var_D8], rax
```

File: “IDBs/emotet/emotet_0.bin.i64”
Disassembly at 0x1800030DF

Step 2: Validating callbacks

- Step 1: Find possible callback functions using a regular expression.
- Step 2: Validate them by examining the disassembly.
  - Must have a single basic block.
    - No jumps.
  - Must not call to another function.
- Step 3: Emulate validated callbacks.

Capstone engine

- Lightweight multi-platform, multi-architecture disassembly framework.
- Intuitive and easy-to-use API to disassemble and analyze instructions.
- Actively maintained, used by projects such as Radare2.
- Easy to install: "pip install capstone"

Capstone
The Ultimate Disassembler

www.capstone-engine.org

Capstone engine

- Learn more about Capstone capabilities:
    - https://github.com/capstone-engine/capstone/blob/next/bindings/python/test_x86.py

Step 3: Emulate validated callbacks

- Already done that earlier in this section.

Conclusion

- Malware evolves and changes are unpredictable.
- The extractor will break but things can be done to minimize that:
  - Anticipate changes e.g. generic regex.
  - Rely on the static parts when possible. Example:
    - Check many versions, see what doesn't change in/around an area you want to locate.
    - Chances are future versions are the same.
- It is better for the extractor to register nothing than to yield wrong IOCs.
  - Perform sanity checks.
  - Log extensively to help with debugging.

Appendix 1: Mapping PE to Unicorn Memory

Emulating code that calls into other functions and/or accesses its data sections would require mapping the PE executable into the emulator's memory. Writing the raw PE file to memory, as it is on disk, would lead to invalid relative offsets and memory accesses since the alignment of PE sections in the physical file differs from their alignment in memory. To map a PE to Unicorn engine's memory:

```
# load PE file.
pe = pefile.PE(pe_path)

# Get memory-mapped PE image.
pe_img = pe.get_memory_mapped_image()

# Get the image base address.
img_base = pe.OPTIONAL_HEADER.ImageBase

# Initialize Unicorn
emulator = unicorn.Uc(unicorn.UC_ARCH_X86, unicorn.UC_MODE_32)

# Map memory for the PE at the image base. The image size is aligned to page boundary.
emulator.mem_map(img_base, (len(pe_img) + 0xfff) & ~0xfff)

# Write the memory-mapped image to the emulator's memory at the image base.
emulator.mem_write(img_base, pe_img)
```

Appendix 2: Unicorn engine hooks

The Unicorn engine is OS-agnostic, and so emulating code will inevitably lead to exceptions or undefined behavior. Luckily the engine offers useful hooks for the developer to correct the behavior or implement workarounds. The implementation details are largely tied to specific use-cases hence why these hooks are user-defined.

One notable hook - that we find ourselves registering a lot - is the invalid memory access hook. It is a function that gets invoked when the emulated instruction causes an invalid memory access. The hook would attempt to resolve this access by mapping the memory at the faulting address and returning ‘True’. In which case, the Unicorn engine would re-execute the instruction and continue emulation. On the other hand, emulation would stop when the hook returns ‘False’.

```
def uc_invalid_mem_access_hook(emulator, access_type, address, size, value, _user_data):
    # Align to previous page boundary and map e.g. address 0x20098 becomes 0x20000.
    emulator.mem_map(address & ~0xfff, 0x1000)
    return True

# Initialize the emulator and register the hook
emulator = unicorn.Uc(unicorn.UC_ARCH_X86, unicorn.UC_MODE_32)
emulator.hook_add(unicorn.UC_HOOK_MEM_INVALID, uc_invalid_mem_access_hook)
```

Appendix 2: Unicorn engine hooks

The code hook is another useful callback that you will usually need to implement. A major drawback is that it slows down the emulation speed considerably because the engine breaks out to invoke the user-defined function prior to the execution of every instruction.

The example below uses this hook to skip all call instructions in the emulated code with the help of Capstone.

```
# Initialize Capstone
cap_md = capstone.Cs(capstone.CS_ARCH_X86, capstone.CS_MODE_32)
cap_md.detail = True

def uc_code_hook(emulator, address, size, user_data):
    try:
        # Read and disassemble the instruction
        inst = next(cap_md.disasm(emulator.mem_read(address, size), 0))
    except StopIteration:
        return
    # If it's a CALL instruction skip it
    if inst.id == capstone.x86.X86_INS_CALL:
        emulator.reg_write(unicorn.x86_const.UC_X86_REG_EIP, address + size)

# Initialize Unicorn and register the code hook
emulator = unicorn.Uc(unicorn.UC_ARCH_X86, unicorn.UC_MODE_32)
emulator.hook_add(unicorn.UC_HOOK_CODE, uc_code_hook)
```

Thank you!

@Dark_Puzzle

@stamparm