Workshop: Analysis of Virtualization-based Obfuscation

Tim Blazytko
@mr_phrazer
tim@blazytko.to
https://synthesis.to

https://t.me/learningnets

Personal Details

binary security researcher, co-founder of *emproof GmbH* and former PhD student

- **research:** code deobfuscation, fuzzing and root cause analysis
- **full-time:** design and evaluation of obfuscation techniques
- **freelancing:** reverse engineering and trainings

https://t.me/learningnets

Today

- basics of VM-based obfuscation
- manual analysis
- symbolic execution to guide manual analysis
- writing an SE-based disassembler

https://t.me/learningnets

Slides, Code and Samples

`https://github.com/mrphrazer/r2con2021_deobfuscation`

https://t.me/learningnets

Virtual Machine Basics

https://t.me/learningnets

```
mov ecx, [esp+4]
xor eax, eax
mov ebx, 1

__secret_ip:
  mov  edx, eax
  add  edx, ebx
  mov  eax, ebx
  mov  ebx, edx
  loop __secret_ip

mov eax, ebx
ret
```

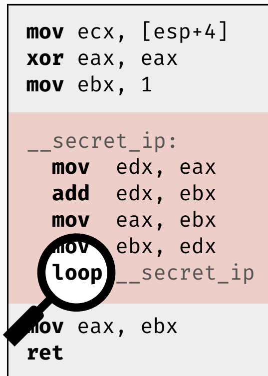
https://t.me/learningnets

```
mov ecx, [esp+4]
xor eax, eax
mov ebx, 1

__secret_ip:
  mov  edx, eax
  add  edx, ebx
  mov  eax, ebx
  mov  ebx, edx
  loop __secret_ip

mov eax, ebx
ret
```

https://t.me/learningnets



https://t.me/learningnets

Virtual Machines

```
mov ecx, [esp+4]
xor eax, eax
mov ebx, 1

__secret_ip:
   mov   edx, eax
   add   edx, ebx
   mov   eax, ebx
   mov   ebx, edx
   loop  __secret_ip

mov eax, ebx
ret
```

made-up instruction set

```
__bytecode:     vld  r1
   vld  r0      vpop r2
   vpop r1      vldi #1
   vld  r2      vld  r3
   vld  r1      vsub r3
   vadd r1      vld  #0
   vld  r2      veq  r3
   vpop r0      vbr0 #-0E
```

https://t.me/learningnets

Virtual Machines

```
mov ecx, [esp+4]
xor eax, eax
mov ebx, 1

__secret_ip:
  push __bytecode
  call vm_entry

mov eax, ebx
ret
```

made-up instruction set

```
__bytecode:
  db 54 68 69 73 20 64 6f
  db 65 73 6e 27 74 20 6c
  db 6f 6f 6b 20 6c 69 6b
  db 65 20 61 6e 79 74 68
  db 69 6e 67 20 74 6f 20
  db 6d 65 2e de ad be ef
```

https://t.me/learningnets

```
mov ecx, [esp+4]
xor eax, eax
mov ebx, 1

__secret_ip:
  push __bytecode
  call vm_entry

mov eax, ebx
ret
```

made-up instruction set

```
__bytecode:
  db 54 68 69 73 20 64 6f
  db 65 73 6e 27 74 20 6c
  db 6f 6f 6b 20 6c 69 6b
  db 65 20 61 6e 79 74 68
        6e 67 20 74 6f 20
        65 2e de ad be ef
```

https://t.me/learningnets

Core Components

VM Entry/Exit	Context Switch: native context ⇔ virtual context
VM Dispatcher	Fetch–Decode–Execute loop
Handler Table	Individual VM ISA instruction semantics

- **Entry** Copy native context (registers, flags) to VM context.
- **Exit** Copy VM context back to native context.

- Mapping from native to virtual registers is often 1:1.

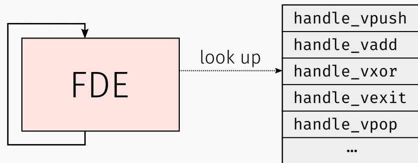
https://t.me/learningnets

Virtual Machines

Core Components

VM Entry/Exit	Context Switch: native context ⇔ virtual context
VM Dispatcher	Fetch–Decode–Execute loop
Handler Table	Individual VM ISA instruction semantics

1. Fetch and decode instruction
2. Forward virtual instruction pointer
3. Look up handler for opcode in handler table
4. Invoke handler

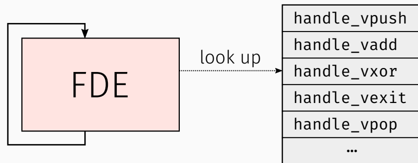


https://t.me/learningnets

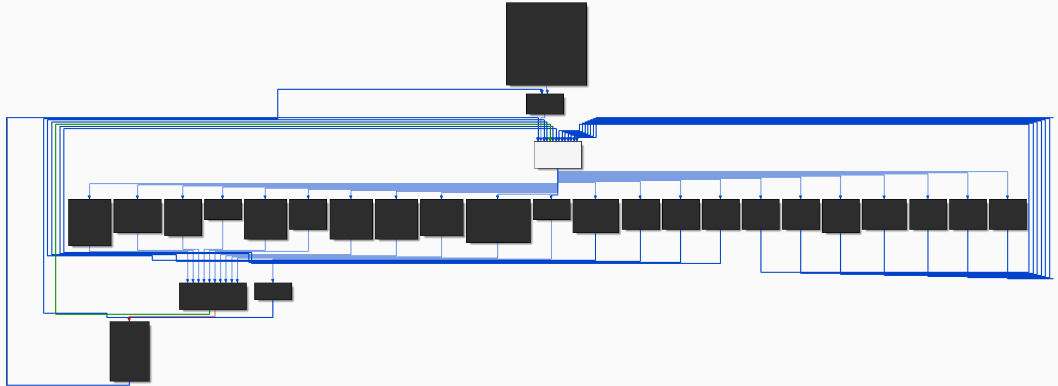
Virtual Machines

Core Components	
VM Entry/Exit	Context Switch: native context ⇔ virtual context
VM Dispatcher	Fetch–Decode–Execute loop
Handler Table	Individual VM ISA instruction semantics

- Table of function pointers indexed by opcode
- One handler per virtual instruction
- Each handler decodes operands and updates VM context

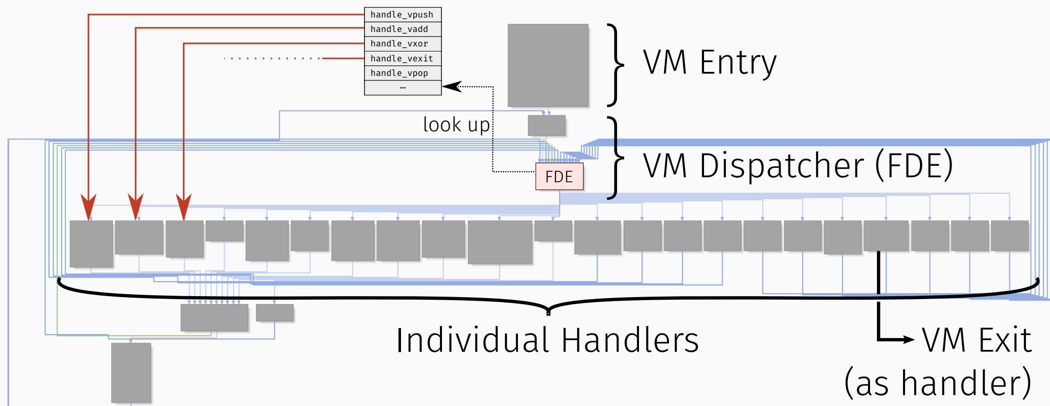


https://t.me/learningnets



https://t.me/learningnets

Virtual Machines



https://t.me/learningnets

Data Structures

- bytecode
  - array of bytes that **encodes** the protected code
  - will be **interpreted** by the virtual machine

https://t.me/learningnets

Data Structures

- bytecode
  - array of bytes that **encodes** the protected code
  - will be **interpreted** by the virtual machine
- virtual instruction pointer
  - points to the **current** instruction in the bytecode
  - **incremented** after each instruction by its size

https://t.me/learningnets

Data Structures

- bytecode
  - array of bytes that **encodes** the protected code
  - will be **interpreted** by the virtual machine
- virtual instruction pointer
  - points to the **current** instruction in the bytecode
  - **incremented** after each instruction by its size
- virtual stack pointer
  - points to the VM-internal **top of stack** (TOS)
  - modified by **vpush** and **vpop** instructions

https://t.me/learningnets

Virtual Machines

```
__vm_dispatcher:
  mov   bl, [rsi]
  inc   rsi
  movzx rax, bl
  jmp   __handler_table[rax * 8]
```

VM Dispatcher

`rsi` – virtual instruction pointer
`rbp` – VM context

https://t.me/learningnets

```
__vm_dispatcher:
  mov   bl, [rsi]
  inc   rsi
  movzx rax, bl
  jmp   __handler_table[rax * 8]
```

VM Dispatcher

`rsi` – virtual instruction pointer
`rbp` – VM context

```
__handle_vnor:
  mov   rcx, [rbp]
  mov   rbx, [rbp + 4]
  not   rcx
  not   rbx
  and   rcx, rbx
  mov   [rbp + 4], rcx
  pushf
  pop   [rbp]
  jmp   __vm_dispatcher
```

Handler performing `nor`
(with flag side-effects)

https://t.me/learningnets

Instruction Handler Arguments

Instruction handler can pass arguments through a stack or in registers.

- stack-based architecture
  - pop arguments from stack
  - push results onto stack
  - examples: JVM, CPython, WebAssembly, ...
- register-based architecture
  - pass arguments in virtual registers
  - store results in virtual registers
  - examples: Dalvik, Lua, LLVM, ...
- **hybrid** architectures possible

https://t.me/learningnets

- locate the bytecode that is interpreted by the VM
- understand the VM architecture/context
- reverse engineer the handler semantics
- reconstruct the VM control flow
- reconstruct the the high-level control flow

https://t.me/learningnets

Manual Analysis

https://t.me/learningnets

Goals

Get a better understanding of the VM:

- identify basic VM components and structures
- detect patterns in handlers
- recover handler semantics

https://t.me/learningnets

Sample

- function that implements iterative Fibonacci
- basic virtual machine protection generated with *Tigress*[1]
- virtual machine layout
  - stack-based virtual machine
  - virtual instruction and stack pointer
  - nested tree-based dispatching
  - 11 VM handlers

[1] `https://tigress.wtf/`

https://t.me/learningnets

Task #1: Identification of VM Components

Open the sample `vm_basic.bin` and start your analysis at `0x115a`.

- Locate the VM dispatcher.
- Locate the bytecode.
- Identify some basic blocks that implement handlers.
- What are the functions of `rdx` and `rcx`?

https://t.me/learningnets

Task #2: Recovering Handler Semantics I

Open the sample `vm_basic.bin` and analyze the handler at `0x11e1`.

- How fetches the handler its argument?
- What does it do with the argument?
- What else does the handler do?

https://t.me/learningnets

Task #3: Recovering Handler Semantics II

Open the sample `vm_basic.bin` and analyze the handler at `0x11a9`.

- How fetches the handler its arguments?
- What does it compute?
- What else does the handler do?

https://t.me/learningnets

Task #4: Recovering Handler Semantics IV

Open the sample `vm_basic.bin` and analyze the handler at `0x1281`.

- What does the handler check?
- Why does it branch?
- What does the handler do with `rdx` and `rax`?

https://t.me/learningnets

Lessons Learned

- `rdx` is virtual instruction pointer, `rcx` is virtual stack pointer
- handlers push and pop arguments from/onto the stack
- handlers update the virtual instruction and stack pointers
- handler `0x11e1` loads a constant from the bytecode and pushes it onto the stack
- handler `0x11a9` implements a stack-based addition
- handler `0x1281` implements a conditional branch

https://t.me/learningnets

Symbolic Execution

https://t.me/learningnets

Symbolic Execution

- computer algebra system for assembly code
- symbolic summaries of instructions, basic blocks and paths
- summaries provide detailed insights and reveal patterns
  - ⇒ supports manual VM analysis
- can be mixed with concrete values (dynamic/concolic execution)
- can automatically follow the execution flow (interactive emulator/debugger)
  - ⇒ dynamic VM disassembler

https://t.me/learningnets

Symbolic Execution

```
__handle_vnor:
  mov   rcx, [rbp]
  mov   rbx, [rbp + 4]
  not   rcx
  not   rbx
  and   rcx, rbx
  mov   [rbp + 4], rcx
  pushf
  pop   [rbp]
  jmp   __vm_dispatcher
```

Handler performing **nor**
(with flag side effects)

https://t.me/learningnets

Symbolic Execution

```
__handle_vnor:
• mov   rcx, [rbp]
  mov   rbx, [rbp + 4]
  not   rcx
  not   rbx
  and   rcx, rbx
  mov   [rbp + 4], rcx
  pushf
  pop   [rbp]
  jmp   __vm_dispatcher
```

Handler performing **nor**
(with flag side effects)

rcx ← [rbp]

https://t.me/learningnets

Symbolic Execution

```
__handle_vnor:
  mov   rcx, [rbp]
• mov   rbx, [rbp + 4]
  not   rcx
  not   rbx
  and   rcx, rbx
  mov   [rbp + 4], rcx
  pushf
  pop   [rbp]
  jmp   __vm_dispatcher
```

Handler performing **nor**
(with flag side-effects)

$$rcx \leftarrow [rbp]$$
$$rbx \leftarrow [rbp + 4]$$

https://t.me/learningnets

Symbolic Execution

```
__handle_vnor:
  mov   rcx, [rbp]
  mov   rbx, [rbp + 4]
• not   rcx
  not   rbx
  and   rcx, rbx
  mov   [rbp + 4], rcx
  pushf
  pop   [rbp]
  jmp   __vm_dispatcher
```

Handler performing `nor` (with flag side effects)

$$\begin{aligned} rcx &\leftarrow [rbp] \\ rbx &\leftarrow [rbp + 4] \\ rcx &\leftarrow \neg rcx = \neg [rbp] \end{aligned}$$

https://t.me/learningnets

Symbolic Execution

```
__handle_vnor:
  mov   rcx, [rbp]
  mov   rbx, [rbp + 4]
  not   rcx
• not   rbx
  and   rcx, rbx
  mov   [rbp + 4], rcx
  pushf
  pop   [rbp]
  jmp   __vm_dispatcher
```

Handler performing nor (with flag side effects)

$$\begin{aligned} rcx &\leftarrow [rbp] \\ rbx &\leftarrow [rbp + 4] \\ rcx &\leftarrow \neg\, rcx = \neg\, [rbp] \\ rbx &\leftarrow \neg\, rbx = \neg\, [rbp + 4] \end{aligned}$$

https://t.me/learningnets

Symbolic Execution

```
__handle_vnor:
  mov   rcx, [rbp]
  mov   rbx, [rbp + 4]
  not   rcx
  not   rbx
• and   rcx, rbx
  mov   [rbp + 4], rcx
  pushf
  pop   [rbp]
  jmp   __vm_dispatcher
```

Handler performing `nor`
(with flag side-effects)

$$
\begin{aligned}
\text{rcx} &\leftarrow [\text{rbp}] \\
\text{rbx} &\leftarrow [\text{rbp} + 4] \\
\text{rcx} &\leftarrow \neg\, \text{rcx} = \neg\, [\text{rbp}] \\
\text{rbx} &\leftarrow \neg\, \text{rbx} = \neg\, [\text{rbp} + 4] \\
\text{rcx} &\leftarrow \text{rcx} \wedge \text{rbx} \\
&\qquad = (\neg\, [\text{rbp}]) \wedge (\neg\, [\text{rbp} + 4])
\end{aligned}
$$

https://t.me/learningnets

Symbolic Execution

```
__handle_vnor:
  mov   rcx, [rbp]
  mov   rbx, [rbp + 4]
  not   rcx
  not   rbx
• and   rcx, rbx
  mov   [rbp + 4], rcx
  pushf
  pop   [rbp]
  jmp   __vm_dispatcher
```

Handler performing **nor**
(with flag side effects)

$$
\begin{aligned}
\mathrm{rcx} &\leftarrow [\mathrm{rbp}] \\
\mathrm{rbx} &\leftarrow [\mathrm{rbp} + 4] \\
\mathrm{rcx} &\leftarrow \neg\,\mathrm{rcx} = \neg\,[\mathrm{rbp}] \\
\mathrm{rbx} &\leftarrow \neg\,\mathrm{rbx} = \neg\,[\mathrm{rbp} + 4] \\
\mathrm{rcx} &\leftarrow \mathrm{rcx} \wedge \mathrm{rbx} \\
&\qquad = (\neg\,[\mathrm{rbp}]) \wedge (\neg\,[\mathrm{rbp} + 4]) \\
&\qquad = [\mathrm{rbp}] \downarrow [\mathrm{rbp} + 4]
\end{aligned}
$$

https://t.me/learningnets

Symbolic Execution

```
__handle_vnor:
  mov   rcx, [rbp]
  mov   rbx, [rbp + 4]
  not   rcx
  not   rbx
  and   rcx, rbx
• mov   [rbp + 4], rcx
  pushf
  pop   [rbp]
  jmp   __vm_dispatcher
```

Handler performing **nor** (with flag side-effects)

$$
\begin{aligned}
rcx &\leftarrow [rbp] \\
rbx &\leftarrow [rbp + 4] \\
rcx &\leftarrow \neg\, rcx = \neg\, [rbp] \\
rbx &\leftarrow \neg\, rbx = \neg\, [rbp + 4] \\
rcx &\leftarrow rcx \wedge rbx \\
&\qquad = (\neg\, [rbp]) \wedge (\neg\, [rbp + 4]) \\
&\qquad = [rbp] \downarrow [rbp + 4] \\
[rbp + 4] &\leftarrow rcx = [rbp] \downarrow [rbp + 4]
\end{aligned}
$$

https://t.me/learningnets

Symbolic Execution

```
__handle_vnor:
  mov   rcx, [rbp]
  mov   rbx, [rbp + 4]
  not   rcx
  not   rbx
  and   rcx, rbx
  mov   [rbp + 4], rcx
• pushf
  pop   [rbp]
  jmp   __vm_dispatcher
```

Handler performing **nor** (with flag side effects)

$$
\begin{aligned}
\text{rcx} &\leftarrow [\text{rbp}] \\
\text{rbx} &\leftarrow [\text{rbp} + 4] \\
\text{rcx} &\leftarrow \neg\,\text{rcx} = \neg\,[\text{rbp}] \\
\text{rbx} &\leftarrow \neg\,\text{rbx} = \neg\,[\text{rbp} + 4] \\
\text{rcx} &\leftarrow \text{rcx} \wedge \text{rbx} \\
&\qquad = (\neg\,[\text{rbp}]) \wedge (\neg\,[\text{rbp} + 4]) \\
&\qquad = [\text{rbp}] \downarrow [\text{rbp} + 4] \\
[\text{rbp} + 4] &\leftarrow \text{rcx} = [\text{rbp}] \downarrow [\text{rbp} + 4] \\
\\
\text{rsp} &\leftarrow \text{rsp} - 4 \\
[\text{rsp}] &\leftarrow \text{flags}
\end{aligned}
$$

https://t.me/learningnets

```
__handle_vnor:
  mov   rcx, [rbp]
  mov   rbx, [rbp + 4]
  not   rcx
  not   rbx
  and   rcx, rbx
  mov   [rbp + 4], rcx
  pushf
• pop   [rbp]
  jmp   __vm_dispatcher
```

Handler performing **nor** (with flag side-effects)

https://t.me/learningnets

$$
\begin{aligned}
\text{rcx} &\leftarrow [\text{rbp}] \\
\text{rbx} &\leftarrow [\text{rbp}+4] \\
\text{rcx} &\leftarrow \neg\,\text{rcx} = \neg\,[\text{rbp}] \\
\text{rbx} &\leftarrow \neg\,\text{rbx} = \neg\,[\text{rbp}+4] \\
\text{rcx} &\leftarrow \text{rcx} \wedge \text{rbx} \\
&\quad = (\neg\,[\text{rbp}]) \wedge (\neg\,[\text{rbp}+4]) \\
&\quad = [\text{rbp}] \downarrow [\text{rbp}+4] \\
[\text{rbp}+4] &\leftarrow \text{rcx} = [\text{rbp}] \downarrow [\text{rbp}+4]
\end{aligned}
$$

$$
\begin{aligned}
\text{rsp} &\leftarrow \text{rsp} - 4 \\
[\text{rsp}] &\leftarrow \text{flags} \\
[\text{rbp}] &\leftarrow [\text{rsp}] = \text{flags} \\
\text{rsp} &\leftarrow \text{rsp} + 4
\end{aligned}
$$

```
__handle_vnor:
  mov   rcx, [rbp]
  mov   rbx, [rbp + 4]
  not   rcx
  not   rbx
  and   rcx, rbx
  mov   [rbp + 4], rcx
  pushf
  pop   [rbp]
• jmp   __vm_dispatcher
```

Handler performing **nor** (with flag side-effects)

https://t.me/learningnets

$$
\begin{aligned}
\text{rcx} &\leftarrow [\text{rbp}] \\
\text{rbx} &\leftarrow [\text{rbp} + 4] \\
\text{rcx} &\leftarrow \neg\,\text{rcx} = \neg\,[\text{rbp}] \\
\text{rbx} &\leftarrow \neg\,\text{rbx} = \neg\,[\text{rbp} + 4] \\
\text{rcx} &\leftarrow \text{rcx} \wedge \text{rbx} \\
&\qquad = (\neg\,[\text{rbp}]) \wedge (\neg\,[\text{rbp} + 4]) \\
&\qquad = [\text{rbp}] \downarrow [\text{rbp} + 4] \\
[\text{rbp} + 4] &\leftarrow \text{rcx} = [\text{rbp}] \downarrow [\text{rbp} + 4]
\end{aligned}
$$

$$
\begin{aligned}
\text{rsp} &\leftarrow \text{rsp} - 4 \\
[\text{rsp}] &\leftarrow \text{flags} \\
[\text{rbp}] &\leftarrow [\text{rsp}] = \text{flags} \\
\text{rsp} &\leftarrow \text{rsp} + 4
\end{aligned}
$$

Symbolic Execution on the Binary Level

- disassemble a given code location
- lift the disassembled code into an intermediate representation
  - free of side effects (explicit formulas for implicit flag and stack pointer updates)
  - common language for various architectures (x86, arm, mips, ...)
- pre-configure the symbolic state with concrete values (for concolic execution)
- symbolically execute the code starting at a given address

Today: Based on the *Miasm* reverse engineering framework[2]

[2] https://github.com/cea-sec/miasm

https://t.me/learningnets

Task #5: SE-based Handler Analysis I

Use `symbolic_execution.py` and analyze the handler at `0x11e1`.

- Can you spot the virtual instruction pointer update?
- Try to locate the handler's core semantics.
- What else do you see?

Reminder: The handler loads a constant (bytecode) and pushes it onto the stack.

https://t.me/learningnets

Task #6: SE-based Handler Analysis II

Use `symbolic_execution.py` and analyze the handler at `0x11a9`.

- Can you spot the virtual instruction pointer update?
- Try to locate the handler's core semantics.
- Try to understand how the parameters are derived.

Reminder: The handler performs a stack-based addition.

https://t.me/learningnets

Lessons Learned

- `RDX = RDX + 0x1`
  - increment the virtual instruction pointer by 1
- `RDX = RDX + 0x5`
  - increment the virtual instruction pointer by 5
- `@32[RCX + 0x8] = @32[RDX + 0x1]`
  - load a constant from the bytecode and store it onto the stack
- `@32[RCX + 0xFFFFFFFFFFFFFFF8] = @32[RCX] + @32[RCX + 0xFFFFFFFFFFFFFFF8]`
  - pop to values from the stack, add them and push the result onto the stack

https://t.me/learningnets

Writing an SE-based Disassembler

https://t.me/learningnets

Overview

- up until now: manual analysis to get a basic VM understanding
  - VM components and structures
  - basic VM layout
  - handlers and (some) of their semantics
- next step: automated VM analysis
  - goal: SE-based disassembler
  - interactive approach between manual analysis and automation

https://t.me/learningnets

VM Deobfuscation Automation Primer

1. build a symbolic execution engine that automatically follows the execution flow
2. start SE at the VM entry
3. each time SE stops, check why and hardcode register/memory values (bytecode, ...)
4. if SE reaches VM exit, extend VM executor
   - add knowledge about handlers
   - dump values
   - reconstruct control-flow graph

https://t.me/learningnets

Task #7: Following the Execution Flow

Modify `follow_execution_flow.py` until the symbolic execution leaves the VM.

- Execute the script and check where it stops.
- Add more and more knowledge about the VM and re-run the script.
- Use multiple concrete inputs for the VM and derive their corresponding outputs.

https://t.me/learningnets

Task #8: Building a VM Disassembler

Modify `vm_disassembler.py` and enrich the disassembler output as much as possible.

- Start with the handlers you already know.
- Reverse engineer additional handlers and improve the disassembler output.
- If possible, dump intermediate values and add them to the output.

Hint: The handlers executed before conditional jumps are comparisons.

https://t.me/learningnets

Task #9: Reconstruction of VM Disassembly

Run `vm_disassembler_final.py`. Try to reconstruct the underlying algorithm.

- Have a look at the disassembly. Can you identify patterns?
- Try to simplify the disassembly. Can you omit certain instructions?
- Can you rewrite multiple instructions in shorter sequences?
- Try to map the VM disassembly to the original code.

Hint: The underlying algorithm implements an iterative Fibonacci calculation.

https://t.me/learningnets

Lessons Learned

- `goto` can be omitted
- `PUSH 0x0 ; PUSHPTR var_0x4 ; POPTOVAR`
  - `var_0x4 := 0`
- `PUSHPTR var_0x8 ; PUSHFROMVAR ; PUSHPTR var_0x4 ; PUSHFROMVAR ; ADD ; PUSHPTR ; POPTOVAR`
  - `var_0xc := var_0x8 + var_0x4`

https://t.me/learningnets

Conclusion

https://t.me/learningnets

Takeaways

- VM analysis can be time-consuming
- mixture of manual analysis and automation
- automation can be cumbersome to implement (API calls, external data, ...)
- way more advanced VMs exist, but approach stays the same

https://t.me/learningnets

Conclusion

Today:

- manual analysis of a VM
- writing an SE-based disassembler
- reconstruction of VM disassembly
- slides, code and samples:
  `https://github.com/mrphrazer/r2con2021_deobfuscation`

Reach out for questions or discussions:

Thank you very much for your active participation!

https://t.me/learningnets