# Inside VMProtect

Samuel Chevet

16 January 2015

# Agenda

- Describe what VMProtect is
- Introduce code virtualization in software protection
- Methods for circumvention
- VM logic

# Warning



- Some assumptions are made in this presentation
- Only few binaries have been studied
- Mostly 64 bits target

# Plan

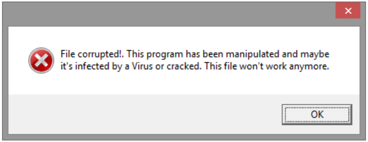
1. Introduction

# Software-based protection

- Content of the executable's sections is encrypted and/or compressed
- Append new code for decrypting/decompressing the sections
- Add all kinds of anti-debug, anti-vm, . . .
- Executable's entrypoint is redirected into this new code
- Execution is transferred back to the original entrypoint after decrypt/decomp

## Memory protection

- Allows protection of the file image in memory from any changes
- Integrity is checked before giving execution to the original entry point



# VMProtect

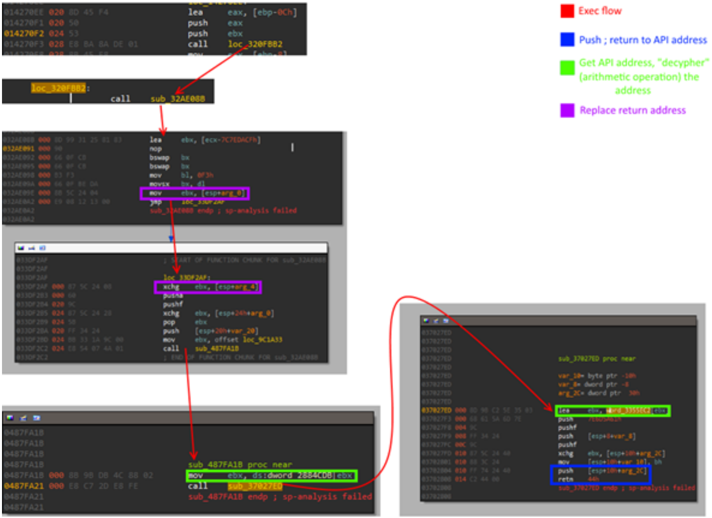
## Import protection

- All entries used by the original binary are removed from Import Table
- Append code redirection for API call
- Replace CALL DWORD PTR[@IAT] / CALL QWORD PTR[@IAT] (Encoded on 6 bytes)
- By CALL VMProtect.section (Encode on 5 bytes)

## 1 byte left: two variations

- Before: Fake push (Stack will be readjusted during redirection)
- After: Dead code (Increment the return address during redirection)

# VMProtect


Exec flow
Push ; return to API address
Get API address, "decypher" (arithmetic operation) the address
Replace return address

## Resource protection

- Encrypt resources: except icons, manifest and some other system types
- Hook:
  - LoadStringA/W
  - LdrFindResource_U
  - LdrAccessResource

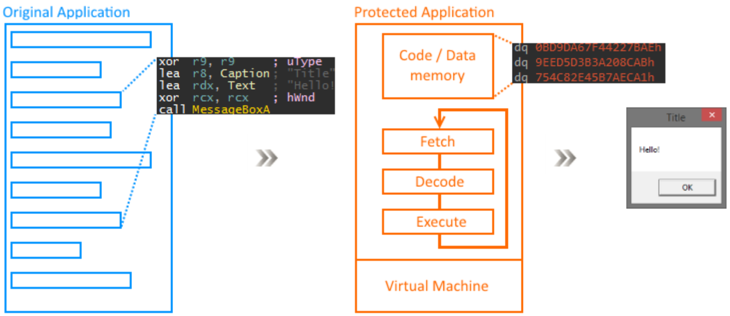
## License manager

- Track your sales online and manage serial numbers
- I have never worked on it

# Code virtualization

- In simple packer native code is simply encrypted and/or compressed
- Disassemble native code and compile it into proprietary bytecode
- Executed in a custom interpreter at run-time
- Interpreter: Fetch, Decode, Execute
- Original native code has disappeared
- Efficient way for anti-reverse engineering

# Code virtualization



# Code virtualization

- VM must fully reproduce correctly CPU instructions
- Save/Restore correctly the context of the application before/after emulation
- Care about correct result in EFLAGS|RFLAGS
- Any error in the emulation is not acceptable

# VMProtect

- VMProtect doesn't decrypt the code at all
- Native code is compiled into a proprietary **polymorphic** bytecode
- From one binary to another one, VM will not be the same
- Or even different VM inside the same binary!

# Agenda



## Questions?

- What is the architecture of the virtual CPU generated?
- Is the VM generated randomly?
- VM bytecode obfuscated?
- Difficult to recontruct original bytecode?
- ...

# Plan

2 Internal

# Virtual machine architecture

- Virtualization obfuscator is Reduced Instruction Set Computing (RISC)
- One Complex Instruction Set Computing (CISC) instruction will be translated in multiple virtualized instructions

```
lea ecx, [ecx + ebx * 4 + 42]
```

## Translated into several virtual instructions

1. Fetch ebx
2. Multiply ebx by 4
3. Fetch ecx
4. Add theses two registers
5. Add 42
6. Store result in ecx

# Virtual machine architecture



- Language used by virtualization obfuscator is **Stack-Based**
- A stack machine implements registers with a stack
- The operands of the arithmetic logic unit (ALU) are always the top two registers of the stack
- Result from the ALU is stored in the top register of the stack
- Reconstruction original native code will involve removing stack machine feature

# Virtual machine context



- Before entering into the virtualization obfuscator, host's registeres and flags must be saved into VM's context structure

## VMProtect context structure

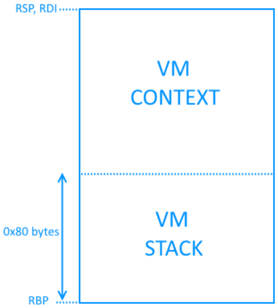
- 8/16 for VM-registers
- 2 for Relocation-Difference and SECURITY_CONSTANT
- 6 for temporal usage (mostly EFLAGS|RFLAGS)
- 0x80 bytes free for pushed variables

```
sub     esp, 0C0h ; 32bit
sub     rsp, 140h ; 64bit
```

# Interesting fact



- Register EDI|RDI holds VM context
- Register EBP|RBP holds VM stack

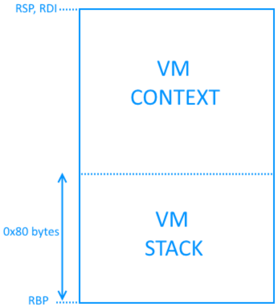



**Maximum value of RBP|EBP**

- 64 bit: RDI + 0xE0, 32 bit: EDI + 0x50
- If this value is reached, reserve more space on the stack and copy VM context and pushed variables

# Interesting fact

- Register EDI|RDI holds VM context
- Register EBP|RBP holds VM stack




## Maximum value of RBP|EBP

- 64 bit: RDI + 0xE0, 32 bit: EDI + 0x50
- If this value is reached, reserve more space on the stack and copy VM context and pushed variables

# VMProtect Context



- VM context is accessed by EDI|RDI (via mem location)
- Index register is EDI|RDI
- Index base is stored in opcode operands (can be encrypted, see later)
- From one VM to another, VM registers will not be stored at the same index!
- It makes VM context totally random

# Virtual machine implementations



## VM Loop

- Read the bytecode at instruction pointer
- Compute opcode handler
- Call the handler
- Can have two variations
  - Down-read VM-Bytes
  - Up-read VM-Bytes
- ESI|RSI: VM instruction pointer

# Bytecode encryption

## If encryption key is present

- Start of code virtualization depends on an encryption key
- VM Loop depends on this key to decrypt opcode
- Handler depends on this key to decrypt operands
- Key is updated during VM Loop **and** opcode handler execution
- Impossible to study code virtualization at a chosen point
- EBX|RBX holds the encryption key

# Logical & Arithmetic operations

- Some logical and arithmetic opcode handler must care of EFLAGS|RFLAGS
- Each of them has code to store them after the operation

```
; ... operation
pushfq
pop     qword ptr [rbp+0]
```

- After such handler, VM will call an handler to POP them in VM register
- **GUESS: there is VM opcode pairs**

# VM Block Start



- Push all registers, and EFLAGS|RFLAGS
- **Order is totally random**
- Push SECURITY_CONSTANT
- Push Relocation-Difference
- Decrypt SECURITY_CONSTANT
- Store all pushed registers, flags & others into VM context
- **Index in VM context is totally random**

# VM Block End



- Push from VM context registers to stack
  - IF VM_EXIT
    - Pop all registers and EFLAGS|RFLAGS and return
  - ELSE
    - Encrypt SECURITY_CONSTANT
    - Push SECURITY_CONSTANT
    - Push Relocation-Difference
    - Jump to next VM_Block

# Internal registers



## What we know

- EBX|RBX: encryption key
- EDI|RDI: VM context
- ESI|RSI: VM instruction pointer
- EBP|RBP: VM stack
- EDX|RDX: arithmetic/result operation of handler address
- EAX|RAX: opcode value
- R13: relocation-difference
- R12: opcode handler table

# Plan

3 Analysis

# Dynamic analysis



- Now that we know how it "works"
- Before using symbolic execution to solve this problem
- We have to write an "intelligent code tracer"
- So we will be sure our symbolic execution is not buggy

# Dynamic analysis



- Trace full execution will take too much time
- Locate the VM Loop
- Inject DLL that setup a HBP on execution at VM Loop
- Store in DB:
    - VM Stack
    - VM Context
- Make a local WebService to output result (Diff between two states on VM_STACK, VM_CONTEXT)
- Initialize VM Context with default value

# Dynamic analysis

```
------------------------------------------------------------------------------- [REGISTERS]
         NUM : 0x0000000000000001
         RAX : 0x000000000000007E ; RSI : 0x00007FF621C67E05 ; RDX : 0x00007FF621C44852
         RDI : 0x0000003E4716F070 ; RBP : 0x0000003E4716F1C0 ; RBX : 0x0000000140087E62
         OP  : 0x7E ; SIZE_OPERAND : 0x0004 ; VMHANDLER : 0x0000000140064852
------------------------------------------------------------------------------- [VM_CONTEXT/REGISTERS]
0000003E4716F070 4141414141414141 4141414141414141 4141414141414141 4141414141414141
0000003E4716F090 4141414141414141 4141414141414141 4141414141414141 4141414141414141
0000003E4716F0B0 4141414141414141 4141414141414141 4141414141414141 4141414141414141
0000003E4716F0D0 00007FF4E1BE0000 4141414141414141 4141414141414141 4141414141414141
0000003E4716F0F0 4141414141414141 4141414141414141 4141414141414141 4141414141414141
0000003E4716F110 4141414141414141 4141414141414141 4141414141414141 4141414141414141
0000003E4716F130 4141414141414141 4141414141414141 4141414141414141 4141414141414141
0000003E4716F150 4141414141414141 4141414141414141 4141414141414141 4141414141414141
0000003E4716F170 4141414141414141 4141414141414141 4141414141414141 4141414141414141
0000003E4716F190 4141414141414141 4141414141414141 4141414141414141 4141414141414141
------------------------------------------------------------------------------- [RBP]
0000003E4716F1B8 FFFFFFFFE4018D37 0000003E633F5A69 0000000000000000 0000000000000346
0000003E4716F1D8 0000003E47201B61 0000000000000000 00007FF4E1BE0000 0000003E4716F2C0
0000003E4716F1F8 0000003E47201BC0 0000003E4716F250 0000000000000018 0000003E47201CC8
0000003E4716F218 00007FF621BE0000 0000003E4716F1D0 0000000000000000 0000000000000000
0000003E4716F238 0000003E4716F1D0 0000003E4716F580 00000000D8469C9F 00000000653C2F8A
0000003E4716F258 00000000630B3AEA 0000003E4716F580 0000003E47201B61 0000000000000000
0000003E4716F278 0000003E4716F2C0 0000000000000206 0000003E4716F260 00007FF621BE0000
0000003E4716F298 00007FF621CE467E 0000000000000000 0000000000000002 2C237DA929FCF51C
```

```
------------------------------------------------------------------------------- [REGISTERS]
         NUM : 0x0000000000000002
         RAX : 0x00000000000000A3 ; RSI : 0x00007FF621C67E00 ; RDX : 0x00007FF621C4205C
         RDI : 0x0000003E4716F070 ; RBP : 0x0000003E4716F1B8 ; RBX : 0x00000001240A0B3A
         OP  : 0xA3 ; SIZE_OPERAND : 0x0000 ; VMHANDLER : 0x000000014006205C
------------------------------------------------------------------------------- [VM_CONTEXT/REGISTERS]
0000003E4716F070 4141414141414141 4141414141414141 4141414141414141 4141414141414141
0000003E4716F090 4141414141414141 4141414141414141 4141414141414141 4141414141414141
0000003E4716F0B0 4141414141414141 4141414141414141 4141414141414141 4141414141414141
0000003E4716F0D0 00007FF4E1BE0000 4141414141414141 4141414141414141 4141414141414141
0000003E4716F0F0 4141414141414141 4141414141414141 4141414141414141 4141414141414141
0000003E4716F110 4141414141414141 4141414141414141 4141414141414141 4141414141414141
0000003E4716F130 4141414141414141 4141414141414141 4141414141414141 4141414141414141
0000003E4716F150 4141414141414141 4141414141414141 4141414141414141 4141414141414141
0000003E4716F170 4141414141414141 4141414141414141 4141414141414141 4141414141414141
0000003E4716F190 4141414141414141 4141414141414141 4141414141414141 4141414141414141
------------------------------------------------------------------------------- [RBP]
0000003E4716F1B8 0000000000000217 0000003E4740E7A0 0000000000000000 0000000000000346
0000003E4716F1D8 0000003E47201B61 0000000000000000 00007FF4E1BE0000 0000003E4716F2C0
0000003E4716F1F8 0000003E47201BC0 0000003E4716F250 0000000000000018 0000003E47201CC8
0000003E4716F218 00007FF621BE0000 0000003E4716F1D0 0000000000000000 0000000000000000
0000003E4716F238 0000003E4716F1D0 0000003E4716F580 00000000D8469C9F 00000000653C2F8A
0000003E4716F258 00000000630B3AEA 0000003E4716F580 0000003E47201B61 0000000000000000
0000003E4716F278 0000003E4716F2C0 0000000000000206 0000003E4716F260 00007FF621BE0000
0000003E4716F298 00007FF621CE467E 0000000000000000 0000000000000002 2C237DA929FCF51C
```

# Dynamic analysis



```
------------------------------------------------------------------------ [REGISTERS]
        NUM : 0x0000000000000264
        RAX : 0x00000000000000A3 ; RSI : 0x00007FF621C6784E ; RDX : 0x00007FF621C4205C
        RDI : 0x0000003E4716F070 ; RBP : 0x0000003E4716F270 ; RBX : 0xB9602DA22A8FDF42
        OP  : 0xA3 ; SIZE_OPERAND : 0x0000 ; VMHANDLER : 0x000000014006205C
------------------------------------------------------------------------ [VM_CONTEXT/REGISTERS]
0000003E4716F070 0000003E4716F390 0000003E4716F580 0000003E4716F1D0 D4602AF4CF66059B
0000003E4716F090 0000000000000206 0000003E4716F1D0 0000003E4740E7A0 0000000000000206
0000003E4716F0B0 0000000000000002 2C237DA929FCF51C 00007FF4E1BE0000 00007FF621BE0000
0000003E4716F0D0 0000003E47201CC8 0000000000000202 00007FF4E1BE0000 0000000067800019
0000003E4716F0F0 0000000000000018 0000000000000000 3E747C74452767BD 0000003E4716F250
0000003E4716F110 0000000000000A02 0000000000000202 0000000000000000 00007FF621C64735
0000003E4716F130 4141414141414141 4141414141414141 4141414141414141 4141414141414141
0000003E4716F150 4141414141414141 4141414141414141 4141414141414141 4141414141414141
0000003E4716F170 4141414141414141 4141414141414141 4141414141414141 4141414141414141
0000003E4716F190 4141414141414141 4141414141414141 4141414141414141 4141414141414141
------------------------------------------------------------------------ [RBP]
0000003E4716F1B8 0000000000000217 0000003E4740E7A0 0000003E47201BC0 0000003E4716F1D0
0000003E4716F1D8 0000003E47201B61 0000000000000346 0000003E4716F580 0000000000000000
0000003E4716F1F8 0000000000000000 00007FF621BE0000 0000003E4716F250 0000003E4716F2C0
0000003E4716F218 0000000000000018 00007FF4E1BE0000 00000282CF66059B 0000000000000246
0000003E4716F238 0000000000000202 0000000000000213 0000000000000216 0000000000000206
0000003E4716F258 0000003E4716F290 0000003E4716F580 0206003E47201B61 0000000000000206
0000003E4716F278 00007FF621C3E08B F606157A247A99EC 00007FF621C64735 00000000B36DCE64
0000003E4716F298 00007FF621CE467E 0000000000000000 0000000000000002 2C237DA929FCF51C
```

```
------------------------------------------------------------------------ [REGISTERS]
        NUM : 0x0000000000000265
        RAX : 0x000000000000006E ; RSI : 0x00007FF621C6784D ; RDX : 0x00007FF621C40B56
        RDI : 0x0000003E4716F070 ; RBP : 0x0000003E4716F270 ; RBX : 0xB9602DA22A8FDF2C
        OP  : 0x6E ; SIZE_OPERAND : 0x0001 ; VMHANDLER : 0x0000000140060B56
------------------------------------------------------------------------ [VM_CONTEXT/REGISTERS]
0000003E4716F070 0000003E4716F390 0000003E4716F580 0000003E4716F1D0 D4602AF4CF66059B
0000003E4716F090 0000000000000206 0000003E4716F1D0 0000003E4740E7A0 0000000000000206
0000003E4716F0B0 0000000000000002 2C237DA929FCF51C 00007FF4E1BE0000 00007FF621BE0000
0000003E4716F0D0 0000003E47201CC8 0000000000000202 00007FF4E1BE0000 0000000067800019
0000003E4716F0F0 0000000000000018 0000000000000000 3E747C74452767BD 0000003E4716F250
0000003E4716F110 0000000000000206 0000000000000202 0000000000000000 00007FF621C64735
0000003E4716F130 4141414141414141 4141414141414141 4141414141414141 4141414141414141
0000003E4716F150 4141414141414141 4141414141414141 4141414141414141 4141414141414141
0000003E4716F170 4141414141414141 4141414141414141 4141414141414141 4141414141414141
0000003E4716F190 4141414141414141 4141414141414141 4141414141414141 4141414141414141
------------------------------------------------------------------------ [RBP]
0000003E4716F1B8 0000000000000217 0000003E4740E7A0 0000003E47201BC0 0000003E4716F1D0
0000003E4716F1D8 0000003E47201B61 0000000000000346 0000003E4716F580 0000000000000000
0000003E4716F1F8 0000000000000000 00007FF621BE0000 0000003E4716F250 0000003E4716F2C0
0000003E4716F218 0000000000000018 00007FF4E1BE0000 00000282CF66059B 0000000000000246
0000003E4716F238 0000000000000202 0000000000000213 0000000000000216 0000000000000206
0000003E4716F258 0000003E4716F290 0000003E4716F580 0206003E47201B61 0000000000000206
0000003E4716F278 00007FF621C3E08B F606157A247A99EC 00007FF621C64735 00000000B36DCE64
0000003E4716F298 00007FF621CE467E 0000000000000000 0000000000000002 2C237DA929FCF51C
```

# Static analysis

- Now we can know the VM context at any point (perfect for debugging)
- We want to be able to reconstruct original bytecode
- Automate task
- Use metasm framework (`https://github.com/jjyg/metasm`)

# Metasm

- Ruby open source framework
- Assembler, disassembler, compiler, linker, . . .
- Description of the semantics for each instruction
- Allowing us to compute the semantic of a set of instructions
- **code_binding**

## code_binding example

```
rax => (byte ptr [rsi-1]&0ffffff00h)|((((((byte ptr [rsi-1]>>1)&7fffff80h)|(((((byte ptr [r
rdx => qword ptr [rbp]&0ffffffffffffffffh
rbx => (rbx&0ffffffffffffff00h)|((rbx-((((((byte ptr [rsi-1]>>1)&7fffff80h)|(((((byte ptr [
rbp => (rbp+8)&0ffffffffffffffffh
rsi => (rsi-1)&0ffffffffffffffffh
```

- Just need to replace (inject) inside expression the known value, so expression can be reduced
  - RSI: bytecode_ptr
  - RBX: encryption key
  - RBP: VM_stack

# VM's symbolic internal

- VM symbolic is **huge**
- All VM registers must implem each size of operand (byte, word, dword, qword)
- VM context contains lot of internals registers

```
vm_symbolism = {
    :rax => :opcode,
    :rbx => :vmkey,
    :rsi => :bytecode_ptr,
    :rbp => :vm_stack,
    Indirection[[:vm_stack], 8, nil] => :QWORD_OP_1,
    ...
    Indirection[[:vm_stack, :+, 0x8], 8, nil] => :QWORD_OP_02,
    Indirection[[:rdi], 8, nil] => :qword_vm_r0,
    Indirection[[:rdi, :+, 0x8], 1, nil] => :byte_vm_r0,
    ...
    Indirection[[:rdi, :+, 0x8], 8, nil] => :qword_vm_r1,
    ...
    Indirection[[:rdi, :+, 0x18], 8, nil] => :qword_vm_r3,
    ...
```

# Static analysis



- Setup start context of the VM
- Disassemble and compute semantic of the current opcode handler
- Compute next state with solved semantics
- Loop if not VM_EXIT

## Problem

- We need to know the end address for the code_binding
- Check list of basic block, if one basic block match the check on maximum value of VM_STACK (EBP) => STOP ADDR
- If not we are back to the VM_LOOP or RETN (VM_EXIT)

# Static analysis

- Setup start context of the VM
- Disassemble and compute semantic of the current opcode handler
- Compute next state with solved semantics
- Loop if not VM_EXIT

## Problem

- We need to know the end address for the code_binding
- Check list of basic block, if one basic block match the check on maximum value of VM_STACK (EBP) => STOP ADDR
- If not we are back to the VM_LOOP or RETN (VM_EXIT)

Samuel Chevet

## VM Context requirement at start

- Bytecode pointer start address: arg_00 of VM_ENTRY (constant unfolding can be applied on it)
- Key stored in EBX|RBX necessary to decrypt bytcode is equal to original PE ImageBase + RVA of bytecode pointer
- Opcode handler table (normally stored in r12)

- With our dynamic analysis we know those 3 parameters at any point!

# Static analysis



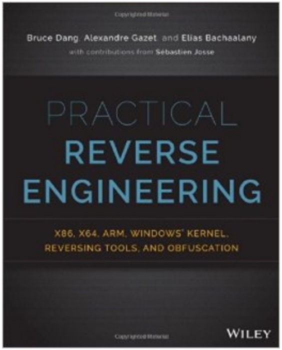
- Remove native register / not interesting VM register from solved binding
  - Keep only operation on EDI|RDI or VM_STACK
- Thanks to the RISC architecture and stack-based language
- Check if VM_STACK has been incremented or decremented

```
[+] solved binding
  qword ptr [rsp] => 140087e62h
  dword ptr [rsp-8] => dword ptr [rsp-8]+0e4018d37h
  QWORD_IMM => 0ffffffffe4018d37h # Indirection[[:vm_stack, :+, -0x8], 8, nil] => :QWORD_IMM
  virt_rax => 0ffffffffe4018d37h
  vm_stack => (vm_stack-8)&0ffffffffffffffffh
  bytecode_ptr => 140087e01h
########### After Remove register
  QWORD_IMM => 0ffffffffe4018d37h
  vm_stack => (vm_stack-8)&0ffffffffffffffffh
[DISAS]: PUSH 0XFFFFFFFFE4018D37
```

# Static analysis



- With that we can start to disassemble the whole bytecode VM
- Check "Pratical Reverse Engineering" (Chapter 5) for complete example on how to use metasm



# Plan



4 VM Logic

# VM Logic

## Reminder

- Langage used is stack-based
- Next opcode after logical or arithmetic operation will store EFLAGS|RFLAGS inside VM context

- For all the following slides we will use the following syntax:
  - QWORD_OP_1: [RBP + 0] ; operand 01
  - QWORD_OP_2: [RBP + 8] ; operand 02

# Disassembler



```
...
[DISAS]: PUSH 0xC2666C77B83B1153
[DISAS]: PUSH vm_r6
[DISAS]: PUSH 0x000000014014A631
[DISAS]: ADD QWORD_OP_1, QWORD_OP_2
[DISAS]: POP vm_r2
[DISAS]: MOV QWORD_OP_1, [QWORD_OP_1]
[DISAS]: ADD QWORD_OP_1, QWORD_OP_2
[DISAS]: POP vm_r14
...
```

# Optimization



- We need to remove stack machine "feature"
- Replace push ; pop by assignement statement
- Track stack pointer
- Check if the destination size match!

# VM Logic

- All push, pop with all different size & mem deref
- Add
- Div, Idiv
- Mul
- Rcl, Rcr
- Shl, Shr
- Shld, Shrd

# VM Logic



- Inside VM handlers, operation like AND|SUB|OR|NOT seems not supported
- In fact all those operations are managed by one handler "NOR" logical gate:

## Native semantic of this handler

```
NOT QWORD_OP_1
NOT QWORD_OP_2
AND QWORD_OP_1, QWORD_OP_2
MOV QWORD_OP_2, QWORD_OP_1
MOV QWORD_OP_1, RFLAGS
```

# VM Logic



- Lot of logical instruction will use this "NOR" logical gate handler:
  - NOT(OP_00) = NOR(OP_00, OP_00)
  - AND(OP_00, OP_01) = NOR(NOT(OP_00), NOT(OP_01))
  - XOR(OP_00, OP_01) = NOR(NOR(OP_00, OP_01), AND(OP_00, OP_01))
  - SUB(OP_00, OP_01) = NOR(ADD(OP_01, NOT(OP_01)))
  - . . .

# VM Logic



- VM_ADC(OP_00, OP_01) = VM_ADD(OP_00, (OP_01 + CARRY))
- VM_SUB(OP_00, OP_01) = VM_NOT(VM_ADD(B, VM_NOT(A)))
- VM_CMP = VM_SUB
- VM_NEG(OP_00) = VM_SUB(0, OP_00)
- . . .
- Original bytecode is sometimes converted to more than 50 VM opcodes . . .

# VM block entry



## VM block entry

```
POP REG ; relocation-difference
PUSH IMMEDIATE
ADD QWORD_OP_1, QWORD_OP_2 ; compute security const
POP REG ; flags
POP REG ; pop result
...
; POP ALL HOST REGISTER (SAVE CONTEXT)
...
```

# VM conditional jump

## VM jcc

1. Push two vm_offset
2. Push VM_stack
3. Convert EFLAGS|RFLAGS for adjustement 0 or 4|8
4. Adjust pointer from result (ADD operation)
5. Prepare to load next vm block from [VM_STACK]

- We will have to reconstruct JCC correctly

## VM CRC

- There is a special opcode for making CRC
- Op_01: Mem pointer, Op_02: Size
- Check VM integrity, executable integrity
- Collision :)

```
rcx = rax = 0;
for (i = 0; i < Size; i++) {
    rcx = rax
    rcx = rcx >> 0x19
    rax = (rax << 0x07) | rcx
    rax = (rax & 0xFFFFFF00) | (rax & 0xFF) ^ buf[i]
}
```

- Compared with SECURITY_CONSTANT
- Found the same checksum in all samples

# VM CPUID



## VM CPUID

- There is a special opcode for making CPUID instruction
- Op_01: Value
- Save 0x0C on VM_STACK (EBP) for storing eax, ebx, ecx, edx

# Listing opcodes

- Try to compute set of all list of opcodes to reconstruct the correct original one
- Really long task, I didn't have finish it at this time
  1. bored
  2. need more samples
- The mapping between the set of VM bytecode and original one will work directly on all binaries

# Plan

5 Conclusion

# Conclusion



- Always the same architecture: RISC + stack machine
- VirtualMachine are generated in a random way
- Difficult to make a static disassembler, prefer to use symbolic execution
- Before having the question: no toolz is going to be released
- VMProtect is a cool challenge (start by 64 bits binary, "obfuscation" is not difficult)

# Questions ?

Thank you for your attention