Creating Code Obfuscation Virtual Machines

VM Creation 101

Craig Smith
Senior Security Consultant
Neohapsis
What is a VM?

Not VMWare, VirtualPC, etc.

Our own Custom Emulator
• Our own CPU
• Our own Language
• Our own Compiler (P-Code)
Why go through all this trouble?

- Code Obfuscation
- Hide Functionality and Intellectual Property
- Increases Analysis and Reversing Time
- Anti-Dumping Method
VM Benefits and Tricks

- The Virtual CPU is specialized for your tasks
- Built in Encryption
- Hidden Anti-Debugging Techniques
- VM Self-Modifying Code (SMC)
- Library or System Call obfuscation
- VM Junk Code
Who uses this?

DRM Does
- Themedia
- VMProtect
- BD+ (Blue-Ray)

Virus, Spyware, Exploits can use this as well
- Made popular by Honeynet's SOTM32 (Nicolas Brulez)
Core Concepts

**P-Code**

| C8 | D2 | A1 | FF | 00 | 00 | 00 | D9 | CC | 09 |

**Handler Routine**

Virtual Registers

r1, r2, r3, etc...

- A0: POP VALUE INTO R1
- A1: PUSH VALUE INTO R1
- A2: POP VALUE INTO R2
- A3: PUSH VALUE INTO R2
- A4: POP VALUE INTO R3
- A5: PUSH VALUE INTO R3
- B0: POP VALUE INTO R4
- B1: PUSH VALUE INTO R4
Our VM Crackme

Design for our VM Crackme:

• Core App runs and prompts user for password
• Pass password to our VM
• VM Does math on the password to make a “Key”
• “Key” is returned and used to Decrypt JMP to GoodBoy message
• Bonus Nugget: Null key is returned if password is wrong
What do we need to get started?

How many registers you want? 9
How you want to control the program flow? EIP
How you going to handle memory? ESP
Macros or own custom language? Custom
What language will you write your compiler in? Ruby
Virtual CPU Register Layout

Four General Purpose Registers:
\( r1, r2, r3, r4 \)

Instruction Registers
\( IP, baseip \)

Stack Registers
\( SP, basesp \)

Flag Registers
\( flags \)
Our Virtual CPU Instruction Set

MOV r32, r32
MOV [r1], r32
MOV r1, [r1]
MOV r32, value
CMP r32, value
INC/DEC r32
AND/OR r1, value
XOR r32, r32
PUSH/POP r32
JMP (Relative / Direct) JE, JL, JG
CALL (r1 / value)
EXITVM
Virtual CPU Initialization

miniVm proc

pop ebx
pop eax ; Stack Argument
mov [stack],eax
pop eax ; P-Code
mov [ip],eax
mov [baseip],eax
pusha ; Save Registers
mov [flags],0 ; Init some regs
mov [stackp],0
call _core ; State Machine
popa ; Restore
push ebx ; Jump back to code
ret
Calling our CPU

- `invoke SendMessage, PasswordHandle, WM_GETTEXT, 20, addr hPassword`
- `mov eax, offset mystack`
- `mov [eax], offset hPassword`
- `push offset vmcode`
- `push offset mystack`
- `call miniVm`
Our Opcode Processor (State Machine)

_next_ip:
    mov ebx,[ip]
    xor eax,eax
    xor ecx,ecx
    xor edx,edx
    mov al,byte ptr [ebx] ; al = instructional opcode
    mov dl,al
    and dl,0Fh ; Major opcode command
    and al,0F0h ; Minor opcode command
    cmp al,0C0h ; MOV r32
    je _call_mov
... 
    call _inc_ip
    jmp _next_ip
Can be as complex or as simple as you want. Don't forget if your VM is small you can always use Macros instead.

My First VM Compiler was in Perl (Back in 2004)

This one is in Ruby

• Object Oriented Core
• Simple method for adding Opcodes
• Easily expandable
• Portable
; Sample MiniVM Code
POP r1 ; Get String off of stack
MOV r1,[r1] ; Get DWORD
CMP r1,0x34333231 ; Cmp to "1234"
JE GoodPassword
MOV r1,0 ; Set Stack to NULL
PUSH r1
JMP ExitCode

GoodPassword:
MOV r1,1 ; Set Stack to 1 to show
; password was valid
PUSH r1

ExitCode:
EXIT ; Quit VM
MiniVMParser.rb:
class MiniVMParser < VMParse...

def define_opcodes
  ops = VMOpcodes.new
  ops.add("PUSH","r1",nil,"\x30")
  ops.add("PUSH","r2",nil,"\x31")
  ops.add("PUSH","r3",nil,"\x32")
  ops.add("CMP","r1","r2","\xd0")
  ops.add("CMP","r1",:value,"\xd8")
  ops.add("CMP","r2",:value,"\xd9")
  ops.add("MOV","eip",:value,"\xcc")

  ...etc...
miniVM Compiler
(c) 2007-2008 Neohapsis
Usage: minivmc [options]

Suggested Options:
-s, --source src  Source file to compile
-d, --destination dst  Destination file
-v, --verbose  Show opcodes per line
-o, --output style  Output style. [ Bin, C, MASM ]
-h, --help  Show this message
; dbx directive example
    MOV r2, msg
    MOV r3, 76 ; r3 holds the xorkey
    JMP code

msg:
    .dbx 76, '/etc/passwd',0

code:
    MOV r1, r2
    MOV r1, [r1]
    AND r1, 0x000000FFh
    XOR r1, r3
    CMP r1, 0
    JE done

...
Output of DBX directive

```
xxd -c 8 minivm.bin

00000000: c900 0000 0aca 0000 ..........  
00000008: 004c 6329 382f 633c .Lc)8/c<  
00000010: 2d3f 3f3b 284c b042 -??;(L.B  
00000018: 4000 0000 0002 d800 @.......  
00000020: 0000 0020 0000 0028 ... ...(  

XOR KEY

String
```
How to add your own directives

3 Steps:
• `@directives.add("mydirective")`
• Def `get_directive_size(tok)` (optional)
• Def `process_directive(tok, tokens)`

`tok.directive.cmd`
`tok.directive.line`
Our Crackme

Valid: ReCon 08;
Goal: To find more valid passwords
Tips for Debugging your VM

Debugging Techniques:
• Add INT 3 Breakpoints to your VM
• Break on the call handler table
• Minivmc -v
• View your Virtual Registers while you are debugging
Attacks against your VM

• Your VM Core must be decrypted in order to process your p-code
• It is very simply to use a signature to identify a VM processor
• Use traditional methods to try and protect your VM core.

Remember this is just obfuscation, not security. The goal is to quickly write code that takes a reverser much longer to analyze.
Example of our VM in IDA
Look Ma, I can mutate!

Self Modifying Code (SMC) example:

```assembly
MOV r1, mutate
ADD r1, eip ; Adjust for relative offset
MOV [r1], 0x21000000h ; 0x21h == JL opcode
MOV r1, 6
CMP r1, 5 ; 6 > 5
mutate:
    JG fakecode ; Appears to always goto fakecode
    ; After mutation becomes JL <some addr>
JMP realcode
```
Advanced Extensions

XOR Register coupled with .xorkey directive

All Register Operations first pass through the XOR register

Example:

```assembly
MOV xorkey, 76        ; where xorkey is a register
MOV r1, 1             ; 1 becomes 0x4Dh
MOV xorkey, r1        ; xorkey becomes 1
.xorkey 1
MOV r1, 1             ; 1 becomes 0
```

* Note included in this version
Shifting Operands

Similar to the xorkey register but used on the operand as follows:

• The CPU can be “seeded” on init with a value

• This value is used when parsing any operand byte (Example XOR)

• The compiler MUST know what the seed value is so it can write the appropriate opcode. Example: .seed 0x4c

• This seed value can change mid program

* Note included in this version
Where to get the Code

Neohapsis Labs (Blog)
VM, and Compiler
http://labs.neohapsis.com/

Crackme is here:
http://crackmes.de/

Email: craig.smith (@) neohapsis.com
Questions?