Introduction to Modern Code Virtualization

by Nooby

This paper describes how code protection is done via "virtual machines" and techniques used in popular virtual machines, giving a considerable level of understanding of such virtual machines for readers from beginners to professionals.

Why Virtual Machines?

In the early years of software protection, techniques like obfuscation and mutation were developed, such methods insert junk codes into the original code flow, change the original instructions to their synonyms, replace constants with some calculations, insert conditional and unconditional branches and write random bytes in the hives between execution code(making sequential disassembling fail), etc. Example of such processing are included as example_obfuscated.exe.

Over time, the level of complexity introduced with such methods become insufficient due to the development of debugging tools(many now features run-time tracing) and average human skill. It is possible for an average skilled reverse engineer to manually or semi-automatically clean up the code to make it readable/alterable. Heavier obfuscation increases the size of protected code dramatically but brings little increase in complexity. People begin to seek a method of code protection against such brute force. By breaking down instructions into a execution loop with a set of reusable micro operations, the length of code being executed can be increased exponentially without growing much in size. Such loop code act like an "emulator" of the original code(aka. Interpreter), with takes in a flow of data(aka. Pesudo-code or P-code), do micro operations(aka. Handlers), which much like a "virtual machine" executes on its own instruction set. This grows into the term: code virtialization.

How Does a Virtual Machine Work?

We know that a real processor has registers, instruction decoders and execution logic. Virtual machines are about the same. The virtual machine entry code will collect context information from the real processor and store on its own context, the the execution loop will read P-Code and dispatch to the corresponding handler. And when the virtual machine exits, it will update real processor registers from its stored context.
For a quick example, there is a function being executed via a pseudo virtual machine:

Original Instructions:
```
    add eax, ebx
    retn
```

By transform it into virtualized code:
```
    push address_of_pcode
    jmp VMEntry
```

VMEntry:
```
    push all register values
    jmp VMLoop
```

```
VMLoop:
    fetch p-code from VMEIP
    dispatch to handler

VMInit:
    pop all register values into VMContext
    pop address_of_pcode into VMEIP
    jmp VMLoop

Add_EAX_EBX_Hander:
    do "add eax, ebx" on VMContext
    jmp VMLoop

VMRetn:
    restore register values from VMContext
    do "retn"
```

Note that a virtual machine does not and need not to emulate all x86 instructions, some can be executed as-is by the real processor, which takes the virtual machine to exit at certain point, point EIP to the raw instruction and then re-enter VM.

The actual virtual machine handlers are usually designed more generic as opposed to the example handler above. Usually the P-Code also determines operands. The "Add_EAX_EBX_Hander" can be defined as an "Add_Handler" which takes 2 parameters and produce a result. There will also be load/store register handlers with produces/saves parameters and results. By doing so makes the handlers more reusable so that tracing through such handlers without understanding the virtual machine architecture cannot not make a good understanding of original code. Now we see how it works on a stack-based virtual machine:

```
Add_Hander:
    pop REG                     ; REG = parameter2
    add [STACK], REG            ; [STACK] points to parameter1

GetREG_Handler:
    fetch P-Code for operand
    push VMCONTEXT[operand]     ; push value of REG on stack

SetREG_Handler:
    fetch P-Code for operand
    pop VMCONTEXT[operand]      ; pop value of REG from stack
```

The P-Code of above function will be:

```
    Init
    GetREG EBX
    GetREG EAX
    Add
    SetREG EAX
    Retn
```

What Modern Virtual Machines Do Against Reverse Engineering?

Code obfuscation and mutation are important to code virtualization, as the virtual machine Interpreters are directly exposed, they can help protecting the virtual machine against automated analysis tools. Heavily obfuscated virtual machine handlers can take a while to de-obfuscate without the knowledge of its underlying virtual machine architecture. Since some of the processor registers are not used(VMContext are stored separately and the virtual machine Interpreter can be designed to use only a few registers), they can be used as extra obfuscation. Virtual machine handlers can be designed to have as little operand/context dependency as possible. Furthermore, real stack pointer can be tracked within the VMContext, stack can be junked during interpreter loop. With these aspects, code obfuscation and mutation can be very effective. An example of such obfuscated virtual machine can be found in example_virtualized.exe

Now we know how the execution part of virtual machines are protected, let's continue to see what techniques are used during transforming instructions into P-Codes, which is the part of awesomeness in code virtualization.

Instruction Decomposition

Logical Instructions

In the approach of increased complexity and reusabiliy, logical operations can be broken into operation like NAND/NOR, according to the following:

NOT(X) = NAND(X, X) = NOR(X, X)
AND(X, Y) = NOT(NAND(X, Y)) = NOR(NOT(X), NOT(Y))
OR(X, Y) = NAND(NOT(X), NOT(Y)) = NOT(NOR(X, Y))
XOR(X, Y) = NAND(NAND(NOT(X), Y), NAND(X, NOT(Y))) = NOR(AND(X, Y), NOR(X, Y))

Arithmetic Instructions

Subtraction can be substituted by addition, with the overhead of EFLAGS calculation:

SUB(X, Y) = NOT(ADD(NOT(X), Y))

Taking the EFLAGS before the final NOT as A and the EFLAGS after the final NOT as B, the calculation is as follows:

EFLAGS = OR(AND(A, 0x815), AND(B, NOT(0x815)))   ; 0x815 masks OF, AF, PF and CF

Register Abstraction

Since a virtual machine can have more registers than an actual x86 processor, real processor registers can be dynamically mapped to virtual machine registers, the extra registers can be used to store intermediate values or simply be confusions. This also allows further obfuscation/optimization across instructions as described below.

Context Rotation

Due to register abstraction, different pieces of P-Code can have different register mappings, and such correspondence can be designed to change from time to time, making reverse engineering more difficult. The virtual machine only swaps the value on its context when the next piece of P-Code has different register mappings. When transforming instructions like XCHG, it can simply change the mappings of registers and not producing any P-Code. See the following example:

Original Instruction:

```
    xchg ebx, ecx
    add eax, ecx
```

P-Code Without Context Rotation:

Current Register Mappings

Real Registers	Virtual Registers
EAX	R0
EBX	R1
ECX	R2

```
    GetREG R2       ; R2 = ECX
    GetREG R1       ; R1 = EBX
    SetREG R2       ; ECX = value of EBX
    SetREG R1       ; EBX = value of ECX
    GetREG R2
    GetREG R0       ; R0 = EAX
    Add
    SetREG R0
```

P-Code With Context Rotation(exchage done during P-Code generation):

Before Exchange

Real Registers	Virtual Registers
EAX	R0
EBX	R1
ECX	R2

After Exchange

Real Registers	Virtual Registers
EAX	R0
EBX	R2
ECX	R1

```
    [Map R1 = ECX, R2 = EBX]        ; exchange
    GetREG R1       ; R1 = ECX
    GetREG R0       ; R0 = EAX
    Add
    SetREG R0       ; R0 = EAX
```

Such rotation can also be applied to the last SetREG operation, so that the result of addition will be written to another unused virtual machine register(i.e. R3), leaving the R0 with useless data. The following piece of P-Code operates on 3 virtual machine registers, makes it hard for reverse engineers to find its x86 equivalent.

P-Code With Context Rotation 2:

```
    [Map R1 = ECX, R2 = EBX]        ; exchange
    GetREG R1        ; R1 = ECX
    GetREG R0        ; R0 = EAX
    Add
    [Map R0 = Unused, R3 = EAX]     ; rotation
    SetREG R3        ; R3 = EAX
```

Register Aliasing

When processing assignment instructions, especially assignment between registers, it is possible to make temporary mappings between source and destination registers. Unless the source register is about to be changed(which forces a remapping or a GetREG & SetREG operation), this mapping can redirect read access to destination register to its source without actually perform the assignment. Take the following piece of code as an example:

Original Instructions:

```
    mov eax, ecx
    add eax, ebx
    mov ecx, eax
    mov eax, ebx
```

P-Code:

Current Register Mappings

Real Registers	Virtual Registers
EAX	R0
EBX	R1
ECX	R2

```
    [Make alias R0 = R2]
    GetREG R1        ; R1 = EBX
    GetREG R2        ; reading of R0 redirects to R2
    Add
    [R0(EAX) is being changed, since R0 is destination of an alias, just clear its alias]
    [Map R0 = Unused, R3 = EAX]     ; rotation
    SetREG R3        ; R3 = EAX
    [Make alias R2 = R3]
    GetREG R1
    [R3(EAX) is being changed, since R3 is source of an alias, we need to do the assignment]
    [Map R3 = ECX, R2 = EAX]        ; we can simplify the R2 = R3 assignment by rotation
    [Map R0 = EAX, R3 = Unused]     ; another rotation
    SetREG R0        ; R0 = EAX
```

Register Usage Analysis

Given the context of a set of instructions, it can determined that at some point the value of certain registers are changeable without affecting the program logic, and some overhead of EFLAGS calculations can be omitted. For example, a piece of code at 0x4069A8 in example.exe:

```
PUSH   EBP
MOV    EBP, ESP                          ; EAX|ECX|EBP|OF|SF|ZF|PF|CF
SUB    ESP, 0x10                         ; EAX|ECX|OF|SF|ZF|PF|CF
MOV    ECX, DWORD PTR [EBP+0x8]          ; EAX|ECX|OF|SF|ZF|PF|CF
MOV    EAX, DWORD PTR [ECX+0x10]         ; EAX|OF|SF|ZF|PF|CF
PUSH   ESI                               ; OF|SF|ZF|PF|CF
MOV    ESI, DWORD PTR [EBP+0xC]          ; ESI|OF|SF|ZF|PF|CF
PUSH   EDI                               ; OF|SF|ZF|PF|CF
MOV    EDI, ESI                          ; EDI|OF|SF|ZF|PF|CF
SUB    EDI, DWORD PTR [ECX+0xC]          ; OF|SF|ZF|PF|CF
ADD    ESI, -0x4                         ; ECX|OF|SF|ZF|PF|CF
SHR    EDI, 0xF                          ; ECX|OF|SF|ZF|PF|CF
MOV    ECX, EDI                          ; ECX|OF|SF|ZF|PF|CF
IMUL   ECX, ECX,0x204                    ; OF|SF|ZF|PF|CF
LEA    ECX, DWORD PTR [ECX+EAX+0x144]    ; OF|SF|ZF|PF|CF
MOV    DWORD PTR [EBP-0x10], ECX         ; OF|SF|ZF|PF|CF
MOV    ECX, DWORD PTR [ESI]              ; ECX|OF|SF|ZF|PF|CF
DEC    ECX                               ; OF|SF|ZF|PF|CF
TEST   CL, 0x1                           ; OF|SF|ZF|PF|CF
MOV    DWORD PTR [EBP-0x4],  ECX
JNZ    0x406CB8
```

Analysis in comments shows the unused register/flag state before the instruction. This information is used to generate register rotations, EFLAGS calculation omission and junk operations which makes generated P-Code even harder to analyze.

Other P-Code Obfuscations & Optimizations

Const Encryption

Intermediate values and constants from the original instructions can be transformed into calculated results during run-time, thus decrease the chance of constants being directly exposed in P-Codes.

Stack Obfuscation

The virtual machine's stack can be obfuscated by pushing/writing random values due to the fact that the real ESP can be calculated/tracked from VMContext.

Multiple Virtual Machine Interpreters

It is possible to use multiple virtual machines to execute one series of P-Code. On certain points, a special handler leading to another interpreter loop is executed. The P-Code data after such points are processed in a different virtual machine. These virtual machines need only to share the intermediate run-time information such as register mappings on switch points. Tracing such P-Code will need to analyze all virtual machine instances, which is considerably much more work.

References

VMProtect
http://vmpsoft.com/

Code Virtualizer
http://www.oreans.com/

Safengine
http://www.safengine.com/

ReWolf's x86 Virtualizer
http://rewolf.pl/stuff/x86.virt.pdf

OllyDBG
http://www.ollydbg.de/

VMSweeper
http://forum.tuts4you.com/topic/25077-vmsweeper/