De Obfuscation of Virtualization
Obfuscated Software

Selvakumar Gopal Rajendran
Virtualization Obfuscated Malware Code

Time Constraint
Difficult to Reverse Engineer
  • Resistant to
    • Static Analysis
    • Dynamic Analysis
Current Approaches in Malware Analysis

Reverse Engineer byte code interpreter
Work out the logic

Why this doesn't work?
Works only when structure of the Interpreter is known

Our Approach

Identify Instructions that affect the behavior of Obfuscated code
Virtualization Obfuscators

Protect code against Reverse Engineering

Original Intel x86 Instructions  \rightarrow  Virtual Opcodes understood only by internal VM

Virtual Opcodes and Virtual Machine are unique for each application

Code Virtualizer
VM Protect
Reverse Engineering

What a Virtualization Obfuscator does?

Original Code (x86 CPU)

mov eax, ebx
push eax
add ecx, eax

Code Virtualization

Virtualizer

Imaginary CPU#1

ld f2, a
sub r1, r1, #8
bnez r1, l#3
...

Imaginary CPU#10

ld a
call tmp1
st b
...

Imaginary CPU#n

add 18 = t, r0
add 17 = 17, r0
cmp.eq p1 = o5, 17
...

#include <stdio.h>
#include "VirtualizerSDK.h"

void main()
{
    VIRTUALIZER_START // the area to protect starts here
    printf("Hello World");
    VIRTUALIZER_END    // end of area to protect
}
Dealing with Virtualization Obfuscation

Programs use system-calls to interact with OS
Malicious code too intends to interact with OS through system-calls

Identify Instructions which interact with OS
Remaining instructions are discarded

Dosen't attempt to recover the original instructions, but to capture behavior of relevant code
Overall Approach

Use a tracing tool to obtain (like QEMU)
• Address of the instruction executed
• Byte Sequence, Mnemonics, Operands etc
• Value of Machine Registers
Identify System calls and their arguments

not all system calls may be of interest like start up or exit

Analyse the instruction trace

analyse the flow of controls to the system calls
these instructions are known as relevant instructions

Build a subtrace from the relevant instructions

approximates the dynamic trace of original unobfuscated code
Use - definition Chain

Initially follow data dependency, consider control transfers separately
Variation of use-definition chains (ud)

link instructions that use a variable to instructions that define it

```
mov eax, [ecx + edx]
push eax
call print
```
Value based dependence

focus on the flow of values, rather than on the details of intermediate computation of address of those values

use(op)
    if op is a register r then \{r\}
    else if op specifies a memory address a then \{a\}
how to identify relevant instructions?

Initialize set $S$ to locations holding the arguments to the system calls (use ABI)

Scan back and process $I$ as
  - if $I$ defines $I$ belongs to $S$, mark $I$ as relevant
  - remove $I$ from $S$ and set of locations used by $I$ is added to $S$
  - continue until $S$ is empty or we reach beginning of trace

Works to find the nearest instruction that wrote to location $a$ but ignores how $a$ is computed.
Relevant Conditional control flow

Value based dependence analysis identifies instructions but not the control flow.

In x86 conditional statements are implemented by setting conditional code flags in the eflags register then executing the branch statement.

The target of the branch statement can be address given in the instruction or address of the next instruction in the code, depending on the value in eflags register.

Examine how target address is calculated in control flow statements to find when conditional control flow is occurring.
how conditional logic is implemented in virtualization obfuscated code?

not implemented the same way as in x86

VMProtect moves the value of flags register to any of the general purpose register for manipulation

there are practically no implementations without the use of eflags register

So we can assume that flags register is used

There are a lot of ways to use this value to implement conditional logic
Equational reasoning system

translate each instruction in the dynamic trace into equivalent set of equations

in dynamic trace, multiple equations may define the same register or memory location

to maintain original behavior of the trace, number the variables

variable appearing on the left side is numbered according to the order it appears in the trace
variable appearing on the right side is numbered according to the instruction that defined it
Translating Instructions into equations

/*l10 */mov ebx, 0x0
/*l11 */pop eax
/*l12*/add ebx, eax
/*l13 */pop eax
/*l14 */sub ebx, eax

ebx10 = 0x0
eax11 = ValueAt(M1000)
esp11 = esp8 + 4
ebx12 = ebx10 + eax11
eflags12 = Flag(ebx10 + eax11)
eax13 = ValueAt(M1004)
esp13 = esp11 + 4
ebx14 = ebx12 - eax13
eflags14 = Flag(ebx12 - eax13)
Target address calculation

We are concerned about the calculation of target address of control flow instructions.

Determine if any part of the target address calculation depends on the value of the flag calculation.

Generate a simplified expression for the target address then check if it contains any call to flag operation.

Look out for branch statements that are always true or always false added for the purpose of obfuscation.
branch instructions for conditional flow

/*\10 */cmp ebx, eax
/*\11 */mov ebx, 0x0
/*\12 */mov eax, 0x10
/*\13 */jnz 10000

eflags10 = Flag(ebx7 cmp eax6 )
ebx11 = 0x0
eax12 = 0x10
eflags13 = eflags10
## Relevant Call Return Control Flow

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>call 5000 (^<em>/ \rightarrow f^</em>/)</td>
<td>1000</td>
<td>push 1008</td>
</tr>
<tr>
<td>1004</td>
<td>(^<em>/L^</em>/) mov ebx, [eax]</td>
<td>1004</td>
<td>jmp 5000 (^<em>/ \rightarrow f^</em>/)</td>
</tr>
<tr>
<td>1008</td>
<td>inc ebx</td>
<td>1008</td>
<td>(^<em>/L^</em>/) mov ebx, [eax]</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>100C</td>
<td>inc ebx</td>
</tr>
<tr>
<td>5000</td>
<td>(^<em>/f^</em>/) mov eax, [abc0]</td>
<td>5000</td>
<td>(^<em>/f^</em>/) mov eax, [abc0]</td>
</tr>
<tr>
<td>5004</td>
<td>ret (^<em>/ \rightarrow L^</em>/)</td>
<td>5004</td>
<td>ret (^<em>/ \rightarrow L^</em>/)</td>
</tr>
</tbody>
</table>

(a) no obfuscation

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>push 1008</td>
<td>1000</td>
<td>push 6000</td>
</tr>
<tr>
<td>1004</td>
<td>mov eax, 5000</td>
<td>1004</td>
<td>(^<em>/f^</em>/) mov eax, [abc0]</td>
</tr>
<tr>
<td>1008</td>
<td>jmp [eax] (^<em>/ \rightarrow f^</em>/)</td>
<td>1008</td>
<td>ret (^<em>/ \rightarrow L^</em>/)</td>
</tr>
<tr>
<td>100C</td>
<td>(^<em>/L^</em>/) mov ebx, [eax]</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td>inc ebx</td>
<td>6000</td>
<td>(^<em>/f^</em>/) mov ebx, [eax]</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>6004</td>
<td>inc ebx</td>
</tr>
<tr>
<td>5000</td>
<td>(^<em>/f^</em>/) mov eax, [abc0]</td>
<td>5004</td>
<td>pop ebx</td>
</tr>
<tr>
<td>5008</td>
<td>jmp [ebx] (^<em>/ \rightarrow L^</em>/)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(b) simple obfuscation

(c) another simple obfuscation

(d) extensive obfuscation
Call : a code address is saved at the call site
Return : the saved address is used for a control transfer at the re-turn point

function's return address is used as a necessary condition for function call/return pairs
Methodology for evaluating analysis

- generate original subtrace for source code and dynamic trace for obfuscated code
- generate relevant subtrace
- obfuscated subtrace and relevant subtrace are matched to original subtrace and scores are produced
- calculate relevant score and obfuscation score
Score Calculation

relevance score - % of instructions from original trace included in the subtrace
obfuscation score - % of instructions that are added by the obfuscator that are correctly excluded from the relevant subtrace
"Harris, when I said 'any questions' I was using only a figure of speech."