Virtual Deobfuscator

Removing virtualization obfuscations from malware – a DARPA Cyber Fast Track funded effort

Approved for Public Release, Distribution Unlimited
Overview

• What is virtualization obfuscations?
• Why we care
• What has been done?
• Solution
• Future work
• Source code/Questions
What is Virtualization Obfuscation

- Software protection
- Translation of a binary into randomly generated bytecode
- Bytecode is a new instruction set targeted typically for RISC based architecture VM which runs on x86
- Original binary is lost
Why we care

• Superior anti-reverse engineering technique
• Malware is using this technology to avoid detection and analysis

• Analysis
  – Static:
    • Disassemblers fail on new bytecode
  – Dynamic:
    • Difficult due to finding the boundaries between interpreter and translated original program
    • Vast numbers of instructions
Pain and Joy

• Slogging
  – Understand logic of bytecode
  – Custom disassembler

• Architecture specific?
  – <Sigh>
  – No ‘break once break everywhere’

• Automation would be nice...
What has been done

• Rotalume – Sharif
  – Dynamic approach

• Unpacking Virtualization Obfuscators – R. Rolles
  – A static approach

• University of Arizona (Kevin Coogan, Gen Lu Gen, and Saumya K. Debray)
  – Dynamic approach
Virtual Deobfuscator

- Developed in Python
- Uses a run trace
- Filters out VM interpreters logic
  - RISC pipeline
- Result: Bytecode interpretation (syntax and semantics)
- Architecture agnostic
- Recursive clustering
- PeepHole Optimization
Virtual Deobfuscator Flow

- Parse
- Runtrace
- Cluster Patterns
- ... Recursive Clustering ...
- Repackage Binary
- Peephole Optimization

Debugger/Malware Analysis Software

IDA Pro/Disassembler Analysis
Parser

• Parse run traces into a XML based database
  – OllyDbg 2.0
  – OllyDbg 1.0
  – Immunity
  – WinDbg
  – Source code available – so you can add your own
    • Hypervisor, hardware emulator, etc
Parser

• Creates a file called vd.xml
• > python VirtualDeobfuscator.py -i file.txt -d 1 -t verify.txt
Clustering

1st Pass
- A
- B
- C
- D
- E

2nd Pass
- AB
- AB
- C
- C
- E

3rd Pass
- ABAB
- ABAB
- C
- C
- E

4th Pass
- ABABC
- ABABC
- C
- C
- E

Cluster

004041E8 mov eax, 5A4Dh
004041ED cmp [ecx], ax
004041F0 call 0x401000
...

004040D0 push 14h
004040D2 push 408968h
004040D7 push 1h
...

VM interpreter instructions
Translated bytecode instruction
Translated bytecode instruction
Clustering

• Parse run trace
• Create clusters by grouping snippets of assembly instructions
• Create new clusters based off pattern matching
• Assign each cluster a notational name that reflects depth of cluster (i.e. A, B, AB, etc)
• Loop until no more clusters
c2_______#8

- 'c' - the processing round ("a", "b", "c", etc.) \([c = \text{round 3}]\)
- '2' - ascending integer, unique per round \([ID = 2]\)
- '_____' shows depth
- '#8' - number of instructions in a cluster \([size = 8]\)
- Example: c2_______#8
  - c = round 3, '2' = second cluster, '_____' = depth, '#8' = contains 8 ins
Cluster Sample

• > VirtualDeobfuscator.py -c -d 1

Loop 1

Loop 2

if (only)
{
  _asm { mov eax, 0xDEADBEEF }
  only = false;
}


Console output...what's all that about
Clustering Loop sample

.... (start up code)
004113D3    JMP SHORT 004113DE
Clusters

c1______#11

c2______#8

f1__________________#47

f1__________________#47

c1______#11

a21_#2

c2______#8

a21_#2

00411411     MOV EAX,DEADBEEF    ;EAX=DEADBEEF

00411427     MOV ESI,ESP    ;ESI=0018FE34

...  (wrap up code)

Sweet!
Clustering Sample – Code Virtualizer

OR AX, 0xC0A1 ; ax = DEAD – Original Code

... 42D6BC NOP 42D6BD JMP 0049E22D 49E22D PUSH OFFSET 0049D34B 49E232 JMP 00499130 k7______________________________#3508 499B7D MOV AX,WORD PTR SS:[ESP] 499B81 PUSH EAX 499B82 JMP 0049AC87 49AC87 PUSH ESP 49AC88 POP EAX 49AC89 JMP 0049D056 49D056 ADD EAX,4 49D05B ADD EAX,2 49D060 XCHG DWORD PTR SS:[ESP],EAX 49D063 POP ESP 49D064 OR WORD PTR SS:[ESP],AX 49D068 PUSHFD 49D069 JMP 004993DE k8______________________________#3196 ....

A lot of instructions folded up in k7 cluster. This cluster likely represents the interpreter's loading of the emulator, loading of bytecode, simulated CPU pipeline (prefetch, decode, execute). 3,508 ins worth.

Starting area for unique translation

GOLDEN! AX becomes DEAD
Step 1: A Deeper Dive - Internals

- Create Frequency Graph - freq_graph[]

<table>
<thead>
<tr>
<th>cluster</th>
<th>line numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>4113D3</td>
<td>[13]</td>
</tr>
<tr>
<td>4113D5</td>
<td>[44, 77, 115, 148]</td>
</tr>
<tr>
<td>4113D8</td>
<td>[45, 78, 116, 149]</td>
</tr>
<tr>
<td>4113DB</td>
<td>[46, 79, 117, 150]</td>
</tr>
<tr>
<td>4113DE</td>
<td>[14, 47, 80, 118, 151]</td>
</tr>
</tbody>
</table>

This instruction at 4113d5 occurs on lines 44, 77, etc. It is the beginning of a basic block.

A new basic block begins.

```
004113D5  loc_4113D5:
004113D5      mov    eax, [ebp+var_20]
004113D8      add    eax, 1
004113DB      mov    [ebp+var_20], eax
004113DE      loc_4113DE:
004113DE      cmp    [ebp+var_20], 4
```
Step 2: Compress Basic Blocks

- Window size - window[] - A table of window sizes for each cluster with an cluster id
- Only done once

<table>
<thead>
<tr>
<th>cluster</th>
<th>window size</th>
<th>new cluster id</th>
</tr>
</thead>
<tbody>
<tr>
<td>4113A1</td>
<td>[(1,</td>
<td>4113A1)</td>
</tr>
<tr>
<td>4113A3</td>
<td>[(1,</td>
<td>4113A3)</td>
</tr>
<tr>
<td>....</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4113D3</td>
<td>[(1,</td>
<td>4113D3)</td>
</tr>
<tr>
<td>4113D5</td>
<td>[(3,</td>
<td>a16_#3)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cluster</th>
<th>line numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>4113D3</td>
<td>[13]</td>
</tr>
<tr>
<td>4113D5</td>
<td>[44, 77, 115, 148]</td>
</tr>
<tr>
<td>4113D8</td>
<td>[45, 78, 116, 149]</td>
</tr>
<tr>
<td>4113DB</td>
<td>[46, 79, 117, 150]</td>
</tr>
<tr>
<td>4113DE</td>
<td>[14, 47, 80, 118, 151]</td>
</tr>
</tbody>
</table>

Our new cluster with size 3
Step 3: Greedy Clustering

- Greedy refs cluster list, then iterates through this list looking for more matches
- Recursive

4113A0
a_a1_#2 <- a_a1_#2 + a_a2_#3 match - will become new cluster b1___#5
a_a2_#3
a_a1_#2 <- a_a1_#2 + a_a2_#3 match - will become cluster b1___#5
a_a2_#3
a_a1_#2 <- no match, but could be another match for a1,a3
a_a3_#8
Step 4: Back tracing

- Optional – Testing purposes
  - Verify clustering is working

<table>
<thead>
<tr>
<th>b2_______#22</th>
<th>a333_#5</th>
<th>a169_#17</th>
</tr>
</thead>
<tbody>
<tr>
<td>b3_________#6</td>
<td>a179_#4</td>
<td>a263_#2</td>
</tr>
<tr>
<td>b4__________#10</td>
<td>a747_#7</td>
<td>a162_#3</td>
</tr>
<tr>
<td>b5__________#7</td>
<td>a55_#2</td>
<td>a456_#</td>
</tr>
</tbody>
</table>

Round B

| a55_#2   | 419C46 | 419C48 |
| a456_#5  | 41C2E0 | 41C2E2 | 41C2E5 | 41C2E8 | 41C2EA |
| a601_#4  | 41CCE3 | 41CCE4 | 41CCE5 | 41CCE7 |
| a78_#2   | 419D09 | 419D0B |

Round A
Step 5: Last Clustering Step

- **New Clusters - new_cluster_lst[]**

<table>
<thead>
<tr>
<th>line number</th>
<th>new cluster id</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>004113D3</td>
</tr>
<tr>
<td>14</td>
<td>b1___#7</td>
</tr>
<tr>
<td>15</td>
<td>b2___#4</td>
</tr>
</tbody>
</table>

  VA if no cluster created

  Cluster ID

- From here repeat the steps until no more clusters
Step 6: Final Step

• **Final\_assembly.txt**

  4113D3 JMP SHORT 004113DE
  c1______#11
  f1___________________#47
  a21_#2
  411411 MOV EAX,DEADBEEF
  f1___________________#4

  What we are interested in

• **Last Cluster file (round\_cluster.txt)**

  4113D3
  c1______#11
  f1___________________#47
  a21_#2
  411411
  f1___________________#4
More on Formatting

- Run trace line number
- Current file line number of `final_assembly.txt`
- Line numbers of where this cluster is duplicated on
Chunking

- Grouping of instructions based on cluster
- Found in DIR ‘chunk_cluster’
- f34_____________#173_19.asm (19 is line num)
  - Not intended to be assembled (.asm) for color syntax in vi
- Can compare same clusters
VirtualDeobfuscator.py -c -d 1 -s 1300

So why create all these sections?
That is where our instructions of interest are at. After peephole optimization phase, we will have the interpreted instructions of the original program, and then we are laughing!
Final Tally

- BAC – Blood Alcohol Calculator (77 instructions)
- Protected with VMProtect and Code Virtualizer
- ~255,000 ins
- Sections = 40,000 ins
- Virtual Deobfuscator reduced run trace by 85%
  - ~90% reduction for VMProtect
- Why so much?
  - Code obfuscations! <sigh>
Code Obfuscations

MOV EBP,76732756 ;EBP=76732756
AND EBP,45421A6A ;EBP=44420242
ADD EBP,39C01533 ;EBP=7E021775
JMP 0041B02B
AND EBP,41EA266F ;EBP=40020665
XOR EBP,40020661 ;EBP=00000004

PUSH 100F
MOV DWORD PTR SS:[ESP],EAX

POP ECX
PUSH ECX

And many more...
Repackage Binary

- NASM (The Netwide Assembler) [http://www.nasm.us/](http://www.nasm.us/)
- Used to assemble ‘chunk_sections’ files
- Look for _nasm.asm (14_nasm.asm)
- Massaging run trace
  - Assembler needs either 'h' or '0x' added to hex numbers
  - Memory refs: e.g. MOV EDX, DWORD PTR DS:[EAX*4+__pioinfo]
  - I skip over control flow breaks such as (jmp, jxx, call, rets)
  - NASM does not support LODS, MOVS, etc (instead use LODSB)
  - I removed keywords such as OFFSET, PTR, SS:, DS:
  - ST(0), ST(1) - NASM chooses to call them st0, st1 etc
- > nasm -f win32 final_assembly_nasm.asm
PeepHole

• After binary repackaging, disassemble in IDA Pro
• Python plugin (VD_peephole.py) to remove code obfuscations
• Generates another ‘optimized’ assembly file
  – Run nasm again on the optimized file for analysis in IDA Pro or whatever disassembler you prefer
PeepHole (VD_peephole.py)

• Example of 5 instructions VM protected
  – ADD ESP, 4
  – LEA EAX, [drinks]
  – PUSH EAX
  – PUSH "%d"
  – SCANF

• Equated to 3,329 instructions

• After machine code deobfuscation – 359 instructions

• From here it was easy to hand remove code to see final equivalent instructions
Malware Analysis

- Win32.Klone.af – uses VMProtect along with NSPacker
- Able to reduce the .vmp0 section to 50 instructions
- Quickly determined:
  - Decrypt the compressed section of .nsp1 (to later be decompressed into dynamic memory)
  - Setup of local variables for VirtualAlloc
  - Setup dynamic memory for VirtualAlloc
  - Call VirtualAlloc
  - Finalize the resource section in .nsp1, so that NSPacker can decompress the newly decrypted compressed area of the malware
Future Work

• Machine code deobfuscation
  – This capability could filter out categories of obfuscation patterns never seen before

• Profiler
  – identify hot-spots
  – aid for quick program understanding
  – fixing bugs or to optimize code
  – clustering method could be a similar concept in lumping code and data flow into a more abstract representation of the actual program run trace
Where to get it

• Available from

• POC: Jason Raber
  – jason.raber@hexeffect.com
  – Phone: 937-430-1365

• The views expressed are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.” This is in accordance with DoDI 5230.29, January 8, 2009.