Inside the Matrix, How to Build Transparent Sandbox for Malware Analysis

C.K Chen (Bletchley)
Who am I

- C.K Chen (陳仲寬)
- P.H.D Student in DSNS Lab, NCTU
- Research in
  - Reverse Engineering
  - Malware Analysis
  - Virtual Machine
About DSNS

- 謝續平教授
- 實驗室研究方向
  - 惡意程式分析
  - 虛擬機器
  - 數位鑑識
  - 網路安全
Outline

- VM for Malware Analysis
- Detect Security Utilities
- Out-of-Box Monitor
  - Emulation
  - Virtualization
- Malware Behavior Analysis
- Dynamic Taint Tracking
- Cloudebug
- Anti-VM
- Behavior Comparison to Detect Anti-VM
VM for Malware Analysis

- VM play an important role for nowadays for malware analysis
- Isolated Environment
- Fast Recovery
Reverse with VM

- **What we are doing everyday**
  - Automatic analysis malware:
    - Put monitor program into VM to keep track of malware
  - Reversing Malware
    - Put reversing tools(debugger, disassembler) into VM and reversing
Detect Security Utilities

- While your security utilities are placed in the same environment, it is possible for malware to detect it’s existence
  - KillAV
  - Anti-Debugger
Malware can check the existence of anti-virus, and then stop or bypass anti-virus

- Process Name
- If important function being hooked
- Read Process Memory

Any software in the same environment with malware can be detected
Anti-Debug

- To confuse analyst, malware employ anti-debug to detect or stop debug software

- Everything you put into VM expose the threat
  - File
  - Process
  - Registry
push offset exception_handler; set exception_handler
int 3h
pop dword ptr [ebp-0x38]
xor eax, eax
retn
mov eax, [esp-0x38]
except_handler:
inc dword ptr [eax-0x8]
set ContextRecord.EIP
mov eax, [esp-0x38]
except_handler:
inc dword ptr [eax-0x8]
set ContextRecord.EIP
mov eax, eax
add esp+4
xor eax, eax
retn
mov eax, dword ptr [esp+0xE]
EAX = (ContextRecord.EAX)
except_handler:
mov eax, dword ptr [esp+0x1c]
mov eax,eax; check the flag
jnt choke
jmp rt choke
test eax, eax: check the flag
add esp+4
xor eax, eax
retn
Anti-Debug Result
The “Ultimate Anti-Debugging” Reference


<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3. The Heap</td>
<td>15</td>
</tr>
<tr>
<td>4. Thread Local Storage</td>
<td>19</td>
</tr>
<tr>
<td>5. Anti-Step-Over</td>
<td>25</td>
</tr>
<tr>
<td>6. Hardware</td>
<td>29</td>
</tr>
<tr>
<td>A. Hardware breakpoints</td>
<td>29</td>
</tr>
<tr>
<td>B. Instruction Counting</td>
<td>30</td>
</tr>
<tr>
<td>C. Interrupt 3</td>
<td>34</td>
</tr>
<tr>
<td>D. Interrupt 0x2d</td>
<td>35</td>
</tr>
<tr>
<td>E. Interrupt 0x41</td>
<td>36</td>
</tr>
<tr>
<td>F. MOV SS</td>
<td>37</td>
</tr>
<tr>
<td>7. APIs</td>
<td>38</td>
</tr>
<tr>
<td>A. Heap functions</td>
<td>38</td>
</tr>
<tr>
<td>B. Handles</td>
<td>41</td>
</tr>
</tbody>
</table>
How can we do?

Can we move analysis tools outside the vm?
Out-of-Box Monitor

- Is it possible to monitor program behavior outside the VM
  - Out-of-Box Hooking
  - Virtual Machine Introspection

- How can we monitor the program’s behavior outside the VM

- Virtual Machine Types
  - Emulation
  - Virtualization
Emulation-based VM

- Emulation-based VM
- QEMU, Hydra, Bochs
- Interpreter, Dynamic Translation

Translation

Add monitor code here!
Monitor Based on Emulation

- Temu
- TTAnalyzer
  - Now, it become Anubis
- MBA
  - Develop by us!
Identify Process

- The first step of Out-of-box monitor is to identify process we want to check
Monitor Execution Trace

- Then we would like to monitor execution of process

- Helper function

- Monitor Execution Trace
  - Add helper function when each instruction translate
Malware Behavior Analyzer

- MBA (Malware Behavior Analyzer)
  - MBA run sample in the qemu and extract it’s behavior
  - Produce readable report for analysts
  - Monitor binary，前後比較，內外比對

- What MBA trace
  - File
  - Registry
  - Network
  - MBR
  - SSDT
  - …
Report of MBA(1)

- Analysis file: cad9d083ab6de63b9ddbb08fb0fc64ad
- It's classify to TR/Inject.126976.5 by AntiVir
Report of MBA(1)

Analysis file: cad9d083ab6de63b9ddbb08fb0fc64ad

Modified Files

==== Files tainted ====
/Documents and Settings/dsns/NTUSER.DAT
/Documents and Settings/dsns/NTUSER.DAT.LOG*
/Documents and Settings/dsns/桌面/
cad9d083ab6de63b9ddbb08fb0fc64ad.exe
/WINDOWS/system32/config/software
/WINDOWS/system32/config/software.LOG
/WINDOWS/system32/inetser/inetser.exe
Report of MBA(2)

- Analysis file: cad9d083ab6de63b9dcb08fb0fc64ad
- Network Packets

<table>
<thead>
<tr>
<th>Packet tainted</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>-&gt; 168.95.1.1 , UDP 1026 -&gt; 53</td>
<td>(v 0x01 0x00 0x00 0x01 0x00 0x00 0x00 0x00 0x00 0x05 gsmof 0x06 seed01 0x03 com 0x02 tw 0x00 0x00 0x01 0x00 0x01</td>
</tr>
<tr>
<td>-&gt; 50.115.42.145 , TCP 1027 -&gt; 443</td>
<td>0x02 0x04 0x05 0xb4 0x01 0x01 0x04 0x02</td>
</tr>
<tr>
<td>-&gt; 50.115.42.145 , TCP 1027 -&gt; 443</td>
<td>0x02 0x04 0x05 0xb4 0x01 0x01 0x04 0x02</td>
</tr>
<tr>
<td>-&gt; 50.115.42.145 , TCP 1027 -&gt; 443</td>
<td>0x02 0x04 0x05 0xb4 0x01 0x01 0x04 0x02</td>
</tr>
<tr>
<td>-&gt; 50.115.42.145 , TCP 1027 -&gt; 443</td>
<td>0x02 0x04 0x05 0xb4 0x01 0x01 0x04 0x02</td>
</tr>
<tr>
<td>-&gt; 50.115.42.145 , TCP 1027 -&gt; 443</td>
<td>0x02 0x04 0x05 0xb4 0x01 0x01 0x04 0x02</td>
</tr>
</tbody>
</table>
Report of MBA(3)

Analysis file: cad9d083ab6de63b9ddbb08fb0fc64ad

Modified Registries

====== Registry tainted ======
/WINDOWS/system32/config/SOFTWARE/Microsoft/Active Setup/Installed Components/
{181E2749-8F28-E14F-ECEF-F89FC5739401} StubPath REG_SZ c:\windows\system32\inetsrv\inetsr.exe
/WINDOWS/system32/config/SOFTWARE/Microsoft/Cryptography/RNG Seed REG_BINARY
/Documents and Settings/dsns/ntuser.dat/Software/Microsoft/Windows/ShellNoRoam/MUICache C:\DOCUME~1\dsns\LOCALS~1\Temp\anyexe.bat REG_SZ anyexe

Created Process

====== Process tainted ======
cad9d083ab6de63, 904
svchost.exe, 876
As my experience, this demo will make my pc halt for a while, so we leave it to end of presentation.
Dynamic Taint Tracking

Dynamic taint tracking is a useful tool for binary analysis:
- Precise track influence data of certain event
- Eliminate unrelated event/data

Concept of Infection

Data = readFile(private)
EncData = encrypt(Data)
Prefix = some string
Send(Prefix)
Send(Data)
Close()
Taint Source

- **What we want to track**
  - File
  - Network
  - Executables
  - ....

- **Private in previous example**
Taint Propagate

- How to propagate taint tag
  - These rules describe how data flow corresponding to each behavior
Taint Sink

- Where we want to check taint status
  - Send() in our example

Example
- Taint Source : sensitiveFile
- Taint Sink : sendPkt()

```python
Content = readFile("sensitiveFile");
Encode = ""
For i in content :
    encode<= encode + i ^ ff
sendPkt(encode)
```
Other Application of Taint

- Detect software vulnerabilities and identify possible exploit
  - If EIP tainted while program running
  - Crax use Taint/Concolic Execution to produce exploit for software testing
    - There are the talk in HITCON PLG by SQLab student
- Detect sensitive data leak
- Detect key logger
Cloudebug - A User-interactive Malware Analysis Platform

- Deploy as the web service
  - Analysis malware without environment setting

- Transparent System
  - Out-of-box Monitor
  - Out-of-box Debugging

- Advanced Analysis Capability
  - Taint

- User Friendly
  - Javascript API
Demo
Detect Virtual Machine Environment

- Types and samples of anti-vm technique
  - Hardware Characteristic Checks
  - Timing Checks
  - Emulation Bug Checks
Environment Characteristic Checks

- Hardware specification used to detect virtualization platform
- Files
- Registry
- Process
- Device Name

```assembly
xor eax, eax
cpuid
cmp ecx, 444d4163h
jne exit
mov eax, 80000000h
cpuid
cmp eax, 2
jb exit

mov eax, 80000002h
cpuid
cmp eax, 554d4551h
je $ ;detected
```
Timing Checks

Timing difference between physical machine and virtual machine can be used to detect VM

...  
0x4012ce: rdtsc  
0x4012d0: mov [0x404060], %eax  
0x4012d5: rdtsc  
0x4012d7: mov [0x404070], %eax  
0x4012dc: mov %edx, [0x404060]  
0x4012e2: mov %eax, [0x404070]  
0x4012e7: sub %eax, %edx  
0x4012e9: cmp %eax, 0xff  
0x4012ee: jle 0x4012fe  
...
Emulation Bug Checks

⚠️ Instruction emulated by software may be inconsistent to physical machine

\[
\begin{align*}
\text{mov byte ptr es:[1004h], 5} \\
\text{mov al, fs:[1000h]} \\
\text{inc ax} \\
\text{cmpxchg8b fs:[1000h]} \\
\text{jmp $} \\
\end{align*}
\]
What is Transparent VM

- Guideline from Ether
  - Higher Privilege
  - No Non-privileged Side Effects
    - Any privilege instruction are back to vmm and emulated by software
  - Identical Basic Instruction Execution Semantics
    - 16 rep prefix instruction will make qemu crash
  - Transparent Exception Handling
  - Identical Measurement of Time
Is it possible to build Transparent VM

- Construct transparent analysis VM platform
  - It is extremely hard to implement a transparent system
  - Difficult to verify the completeness
  - Large amount of analysis tool is not based on transparent platform

- How can we do if we don’t have such transparent VM
Behavior Comparison to Detect Virtual Machine Awareness

- Hybrid Emulation & Virtualization to detect Anti-VM malware
- Anti-vm technique is hard to detect all the vm platform in one instruction
  - The code coverage diverge in different VM system
- How to hunt anti-vm malware
  - Execute program in multiple VM system(or physical one if possible)
  - Construct code coverage
  - Compare if there are something different
Virtualization-based VM

- Virtualization-based VM
- KVM, XEN, ...
- Use hardware-assistant virtualization to improve the transparent and performance

Programming Logic

- Compare to emulation system which like sequential logic
- Hardware-Assistant Virtualization more like event-driven model
Virtualization-based VM

Remind how emulation works

How Virtualization Work
## Monitoring Scope

<table>
<thead>
<tr>
<th>VM</th>
<th>Emulator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Privilege instruction</td>
<td>Inst1</td>
</tr>
<tr>
<td>Privilege instruction</td>
<td>Inst2</td>
</tr>
<tr>
<td>Privilege instruction</td>
<td>Inst3</td>
</tr>
<tr>
<td>Privilege instruction</td>
<td>Inst 4</td>
</tr>
<tr>
<td>Privilege instruction</td>
<td>Inst 5</td>
</tr>
</tbody>
</table>
Monitor Based on Virtualization

- Ether(XEN)
- XENAccess(XEN)
- VMITools(XEN, KVM)
- Nitro(KVM)
- ....
Monitor System Call

- System call monitor can be implement with similar concept
  - Make the exception every time system call happened

- How system work
  - When system call happened by SYSENTER instruction
  - OS jump to privilege location defined by SYSENTER_EIP_MSR to handle system call

- Monitor system call
  - Replace value in SYSENTER_EIP_MSR to some invalid address
  - Implement exception handler to profile behavior and put correct value back
Monitor Instruction Trace

- While putting the program to execute in virtualization system, our software cannot directly monitor instruction trace.
- The instruction is directly run by CPU.
- Not going through VMM, which means not manipulated by software.

- Enforce debug exception triggered every instruction.
  - Setting Trap flag to enable debug trap in every instruction.
  - VMExit happened, and VMM gains control.
  - Therefore, we can use software to handle/profile the behavior.
Divergence Point Locator

- In our system, we use two VM system
  - Qemu
  - XEN
  - kvm
Partial Assembly Code of *rdtsc*

Timing Check

Assembly of *rdtsc* sample

...  
0x4012ce: rdtsc  
0x4012d0: mov [0x404060], %eax  
0x4012d5: rdtsc  
0x4012d7: mov [0x404070], %eax  
0x4012dc: mov %edx, [0x404060]  
0x4012e2: mov %eax, [0x404070]  
0x4012e7: sub %eax, %edx  
0x4012e9: cmp %eax, 0xff  
0x4012ee: jle 0x4012fe  
...
Result of *rdtsc* Timing Check

**Code block coverage of *rdtsc* sample**

<table>
<thead>
<tr>
<th>Executed Basic Blocks on KVM</th>
<th>Executed Basic Blocks on QEMU</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0x401260-0x40126a</td>
<td>0x401260-0x40126a</td>
</tr>
<tr>
<td>0x401446-0x401449</td>
<td>0x401446-0x401449</td>
</tr>
<tr>
<td>0x4012ba-0x4012ee</td>
<td>0x4012ba-0x4012f7</td>
</tr>
<tr>
<td>0x4012fe-0x401305(not executed on QEMU)</td>
<td>0x401850-0x401850</td>
</tr>
<tr>
<td>0x401850-0x401850</td>
<td></td>
</tr>
<tr>
<td>0x40130a-0x401311</td>
<td>0x40130a-0x401311</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Bypass Anti-VM in the Fly

- Once we know the location of Anti-VM, we can make the signature
- For runtime patch the executed process
- Make Anti-VM fails
Summary

- Out-of-box monitor to defense anti-debug
- Malware behavior analyzer
- Taint tracking
- Cloudebug
- Anti-vm
- Trace comparison to find out anti-vm
Demo

⚠️ Remember that we need to demo 😊
Q & A