Tripoux: Reverse-Engineering Of Malware Packers For Dummies

Joan Calvet – j04n.calvet@gmail.com

Deepsec 2010
The Context (1)

• A lot of malware families use home-made packers to protect their binaries, following a standard model:

  - The unpacking code is automatically modified for each new distributed binary.
The Context (2)

• Usually people are only interested into the original binary:

1. It’s where the “real” malware behaviour is.

2. It’s hard to understand packers.
The Context (3)

• But developing an understanding of the unpacking code helps to:
  – Get an easy access to the original binary (sometimes “generic unpacking algorithm” fails..!)
  – Build signatures (malware writers are lazy and there are often common algorithms into the different packer’s instances)
  – Find interesting pieces of code: checks against the environment, obfuscation techniques,...
The Question

Why the **human analysis** of such packers is difficult, especially for beginners?
When trying to understand a packer, we cannot just sit and observe the API calls made by the binary:

- This is only a small part of the packer code
- There can be useless API calls (to trick emulators, sandboxes...)

We have to dig into the assembly code, that brings the first problem...
Problem 1: x86 Semantic

• The x86 assembly language is pretty hard to learn and manipulate.

• Mainly because of inexplicit side-effects and different operation semantics depending on the machine state (operands, flags):

  MOVSB

  Read ESI, Read EDI, Read [ESI], Write [EDI]
  
  If the DF flag is 0, the ESI and EDI register are **incremented**
  If the DF flag is 1, the ESI and EDI register are **decremented**
Problem 1: x86 Semantic

• When playing with standard code coming from a compiler, you only have to be familiar with a small subset of the x86 instruction set.

• But we are in a different world...
Problem 1: x86 Semantic

Example : Win32.Waledac’s packer
Problem 2: Amount Of Information

• Common packed binaries have several million instructions executed into the protection layers.

• Unlike standard code, we can not say that each of these line has a purpose.

• It’s often very hard to choose the right abstraction level when looking at the packed binary: “Should I really understand all these lines of code?”
Problem 2: Amount Of Information

Example: Win32.Swizzor’s packer
Problem 3: Absence Of (easily seen) High-Level Abstractions

• We like to “divide and conquer” complicated problems.

• In a standard binary:

```
push   esi
push   dword ptr [eax]
call   sub_1011990
test   eax, eax
jnz    loc_1011C7B
```

```
sub_1011990 proc near
    arg_0= dword ptr  8
    arg_4= dword ptr  0Ch
    push   ebp
    mov    ebp, esp
    mov    edx, [ebp+arg_0]
    mov    ecx, [ebp+arg_4]
    ...
```

This is a function! We can thus consider the code inside it as a “block” that shares a common purpose.
Problem 3: Absence Of (easily seen) High-Level Abstractions

• But in our world, we can have:

Win32.Swizzor’s packer
Problem 3: Absence Of (easily seen) High-Level Abstractions

• No easy way left to detect functions and thus divide our analysis in sub-parts.

• Also true for data: no more high-level structures, only a big array called memory.
The Good News

- Most of the time there is only one “interesting” path inside the protection layers (the one that actually unpacks the original binary).

- It’s pretty easy to detect that we have taken the “good” path: suspicious behaviour (network packets, registry modifications...) that indicate a successful unpacking.
Proposed Solution

• Let’s use this fact and adopt a **pure dynamic analysis approach**:
  – **Trace** the packed binary and collect the x86 side-effects (address problem 1)
  – Define an **intermediate representation** with some high level abstractions (address problem 3)
  – Build some **visualization tools** to easily navigate through the collected information (address problem 2)
Project Architecture

TRACER

Static instructions
Dynamic instructions
Program environment

CORE ENGINE

High level view
Execution details

Timeline
IDA Pro

TRACER

Static instructions
Dynamic instructions
Program environment

CORE ENGINE

High level view
Execution details

Timeline
IDA Pro
How to collect a maximum of information about the malware execution?

**STEP 1: THE TRACER**
Tracing Engine (1)

- **Pin**: dynamic binary instrumentation framework:
  - Insert arbitrary code (C++) in the executable (JIT compiler)
  - Rich library to manipulate assembly instructions, basic blocks, library functions...
  - Deals with self-modifying code

- Check it at [http://www.pintool.org/](http://www.pintool.org/)

- But what information do we want to gather at runtime?
1. Detailed description of the executed x86 instructions

- Binary code, address, size
- Instruction “type”:
  - (Un)Conditional branch
  - (In)Direct branch
  - Stack related
  - Throws an exception
  - API call
  - ...
- Data-flow information:
  - Memory access (@ + size)
  - Register access
- Flags access: read and possibly modified
2. Interactions with the operating system:

- The “official” way: API function calls
  
  • We only trace the malware code thanks to API calls detection (dynamically and statically linked libraries).
  
  • We dump the IN and OUT arguments of each API call, plus the return value, thanks to the knowledge of the API functions prototypes.

- The “unofficial” way: direct access to user land Windows structures like the PEB and the TEB:
  
  • We gather their base address at runtime (randomization!)
3. Output:

1: Dynamic instructions file

<table>
<thead>
<tr>
<th>Time</th>
<th>Address</th>
<th>Hash</th>
<th>Effects</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0x40100a</td>
<td>0x397cb40</td>
<td>RR_ebx_eax</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>WR_ebx</td>
</tr>
<tr>
<td>2</td>
<td>0x40100b</td>
<td>0x455e010</td>
<td>RM_419c51_1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RR_ebx</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2: Static instructions file

<table>
<thead>
<tr>
<th>Hash</th>
<th>Length</th>
<th>Type</th>
<th>W Flags</th>
<th>R Flags</th>
<th>Binary code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x397cb40</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>8D4</td>
<td>43</td>
</tr>
<tr>
<td>0x455e010</td>
<td>1</td>
<td>60</td>
<td>0</td>
<td>0</td>
<td>5E</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 3. Output:

#### 3: Program environment

<table>
<thead>
<tr>
<th>Type</th>
<th>Module name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOSH</td>
<td>ADVAPI32.DLL</td>
<td>77da0000</td>
</tr>
<tr>
<td>PE32H</td>
<td>ADVAPI32.DLL</td>
<td>77da00f0</td>
</tr>
<tr>
<td>PE32H</td>
<td>msvcrtdll</td>
<td>77be00e8</td>
</tr>
<tr>
<td>DOSH</td>
<td>DNSAPI.dll</td>
<td>76ed0000</td>
</tr>
<tr>
<td>PEB</td>
<td>0</td>
<td>7ffdc000</td>
</tr>
<tr>
<td>TEB</td>
<td>0</td>
<td>7ffdf000</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
STEP 2: THE CORE ENGINE
The Core Engine (1)

• Translate the tracer output into something usable.

• Set up some high-level abstractions onto the trace (Problem 3):
  – Waves
  – Loops
The Core Engine (2)

1. Waves:

• Represent a subset of the trace where there is no self-modification code:

  Two instructions $i$ and $j$ are in the same wave if $i$ doesn’t modify $j$ and $j$ doesn’t modify $i$.

• Easy to detect in the trace:
  – Store the written memory by each instruction.
  – If we execute a written instruction: end of the current wave and start of a new wave.
2. Loops:

• Instructions inside a loop have a common goal: memory decryption, research of some specific information, anti-emulation...

• Thus they are good candidate for abstraction!

• But how to detect loops?
The Core Engine (4)

2. Loops:

When tracing a binary, can we just define a loop as the repetition of an instruction?
2. Loops:

**(SIMPLIFIED) STATIC POINT OF VIEW**

- INSTRUCTION 1
- INSTRUCTION 2
- ...
- INSTRUCTION 3
- INSTRUCTION 4
- INSTRUCTION 5
- INSTRUCTION 6

**TRACE POINT OF VIEW**

<table>
<thead>
<tr>
<th>EXECUTED</th>
<th>TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSTRUCTION1</td>
<td>1</td>
</tr>
<tr>
<td>INSTRUCTION5</td>
<td>2</td>
</tr>
<tr>
<td>INSTRUCTION6</td>
<td>3</td>
</tr>
<tr>
<td>INSTRUCTION2</td>
<td>4</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>INSTRUCTION3</td>
<td>5</td>
</tr>
<tr>
<td>INSTRUCTION5</td>
<td>6</td>
</tr>
<tr>
<td>INSTRUCTION6</td>
<td>7</td>
</tr>
</tbody>
</table>

This is not a loop! So what’s a loop?
2. Loops:

What actually define the loop, is the back edge between instructions 3 and 1.
2. Loops:

• Thus we detect loops by looking for back edges inside the trace.

• Information collected about the loops:
  – Number of iterations
  – Read memory access
  – Write memory access
  – Multi-effects instructions (instructions with different effects at each loop turn)
The Core Engine (8)

• In addition to all the events gathered by the tracer (API calls, exceptions, system access...) the core engine also detects:
  – **Conditional or Indirect branch that always jump to the same target** (and that can thus be considered as unconditional direct branch)
Output:

1: High level view

[=> EVENT: API CALL <=]
[TIME: 36][@: 0x40121b]
[D_LoadLibraryA]
[A1:LPCSTR "shlwapi.dll"]
[RV:HMODULE 0x77f40000]

[=> EVENT: LOOP <=]
[START: 4cc620 - END: 4cc654]
[H: 0x21d21cd - T: 0x21d21ca]
| TURN : 2
| READ AREAS : [0x12feec-0x12fef3: 0x8 B]
| WRITE AREAS : [0x410992-0x410993: 0x2 B]
| DYNAMIC PROFILE : 0x21d21ed - 0x21d21ef

2: Full wave dumps

401070  55
401071  29d5
401073   4d
401074  89e5
...

How to avoid the Problem 2 and deal easily with all the collected information?
STEP 3 : VISUALIZATION PART
High-Level View Of The Execution

- Provide a big picture of the trace, plus some analysis tools.

- Build with the “Timeline” widget from the MIT:

  http://www.simile-widgets.org/timeline/
DEMO 1
Low-Level View Of The Execution

- When you need to dig into the code.

- Use IDA Pro (and IDA Python) to display the output of the core engine with the information gathered dynamically (one wave at time!).
DEMO 2
IDA fails to find all the JMP targets!

And so on for the next 6 basic blocs...

Example: Win32.Swizzor’s packer
DEMO 3
Work In Progress (1)

- Address the lack of high level abstraction for data by dynamic typing: (#Read, #Write, #Execution) for each memory byte

A loop inside the Swizzor’s packer

Allows some pretty efficient **heuristic rules**:

- The key is read 5 times because there are 5 decrypted areas by the loop.
- The decrypted areas are read 1 time and written 1 time.
- ...
Work In Progress (2)

• Define a real framework for trace manipulation:
  • Slicing
  • Data Flow
  • De-obfuscation
  • ...

• Allow the user to create his own abstractions on the trace (loops and waves are not always suitable!).

• Set up sandbox analysis to provide the visualization parts to the user?

• Test, test, test.
Thanks!

• Source code and binaries are available here: [http://code.google.com/p/tripoux/](http://code.google.com/p/tripoux/)

• This is really a 0.1 version of the project, any remark/advice is welcome!

• If you are interested, follow the updates @joancalvet

• Thanks to: Pierre-Marc Bureau, Nicolas Fallière and Daniel Reynaud.