PinDemonium

a DBI-based generic unpacker for Windows executables

Sebastiano Mariani - Lorenzo Fontana - Fabio Gritti - Stefano D’Alessio
Malware Analysis

- **Static analysis**: Analyze the malware *without executing it*

- **Dynamic analysis**: Analyze the malware *while it is executed* inside a controlled environment
Malware Analysis

- **Static analysis**: Analyze the malware **without executing it**

- **Dynamic analysis**: Analyze the malware **while it is executed** inside a controlled environment

### Static Analysis

- Analysis of **disassembled code**
- Analysis of **imported functions**
- Analysis of **strings**
Maybe in a fairy tale...

What if the malware tries to hinder the analysis process?

---

**Packed Malware**

- Compress or *encrypt the original code* → Code and strings analysis impossible
- *Obfuscate the imported functions* → Analysis of the imported functions avoided
Packing Techniques

We can classify three packing techniques based on the location where the payload is unpacked:

- **Unpack on the Main Image**: The deobfuscated code is written inside a main Image section
- **Unpack on the Heap**: The deobfuscated code is written in a dynamically allocated memory area
- **Unpack inside remote process**: The deobfuscated code is injected in a remote process
Overriding the Main Image

Steps:

1. Start the execution of the decryption stub
Overriding the Main Image

**Steps:**

2. The decryption stub read data from an encrypted and decrypt it in place inside a main image section.
Overriding the Main Image

Steps:

3. At the end of the decryption phase, the **stub jumps into the first instruction of the decrypted section**.
Unpacking on the Heap

Steps:

1. Start the execution of the decryption stub
Steps:

2. The decryption stub read data from an encrypted main image section and decrypt it on a dynamically allocated memory area (heap)
Unpacking on the Heap

Steps:

3. At the end of the decryption phase the stub jumps into the first instruction of the decrypted section.
Steps:

1. Create remote legitimate process in a suspended state
Steps:

2. Unmap the legitimate code section of the process
Process Injection

Steps:

3. Allocates and writes the decrypted payload in the remote process memory space.
Process Injection

Steps:

4. Modify the thread context to execute code from the newly allocated area and resume the thread execution.
## Solutions

<table>
<thead>
<tr>
<th>Manual approach</th>
<th>Automatic approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Very time consuming</td>
<td>• Fast analysis</td>
</tr>
<tr>
<td>• Too many samples to be analyzed every day</td>
<td>• Scale well on the number of samples that has to be analyzed every day</td>
</tr>
<tr>
<td>• Adapt the approach to deal with different techniques</td>
<td>• Single approach to deals with multiple techniques</td>
</tr>
</tbody>
</table>
All hail

PinDemonium
What is a DBI?

Control Flow Graph

Basic Block

Trace

BB1

BB3

BB2

BB4

BB6

BB7

BB8
What is a DBI?

Trace is copied in the code cache
What is a DBI?

DBI provides the possibility to add user defined code after each:
- Instruction
- Basic Block
- Trace
What is a DBI?

DBI starts executing the program from the code cache.
How can an unpacker be generic?

Key idea

Exploit the functionalities of the DBI to identify the common behaviour of packers: they have to write new code in memory and eventually execute it.
Our stairway to heaven

Packed malware:
- Detect written and then executed memory regions
- Dump the process correctly
- Deobfuscate IAT
- Recognize the correct dump

Original malware
Our journey begins

We begin to build the foundation of our system
Detect WxorX memory regions

Concepts:

- **WxorX law broken**: instruction written by the program itself and then executed

- **Write Interval (WI)**: range of continuously written addresses

Idea:

Track each instruction of the program:

- **Write instruction**: get the target address of the write and update the **write interval** consequently.

- **All instructions**: check if the EIP is inside a write interval. If the condition is met then the **WxorX law** is broken.
Detect WxorX memory regions

Legend:

**EXEC**
Generic instruction

**WRITE**
Write instruction and its ranges

Current instr.

<table>
<thead>
<tr>
<th>Start addr.</th>
<th>End addr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x401004</td>
<td>0x402000</td>
</tr>
<tr>
<td>0x402000</td>
<td>0x403000</td>
</tr>
<tr>
<td>0x412000</td>
<td>0x413000</td>
</tr>
</tbody>
</table>

Steps:

PinDemonium

Write set

execution

26
Detect WxorX memory regions

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x401000</td>
<td>0x402000</td>
<td>WRITE</td>
</tr>
<tr>
<td>0x402000</td>
<td>0x403000</td>
<td>WRITE</td>
</tr>
<tr>
<td>0x412000</td>
<td>0x413000</td>
<td>WRITE</td>
</tr>
</tbody>
</table>

Legend:
- EXEC: Generic instruction
- WRITE: Write instruction and its ranges

Steps:
- Start addr.: 0x401004
- End addr.: 0x425008
- Start addr.: 0x425004
- End addr.: 0x425000

Write set

PinDemonium
Steps:

1. The current instruction is a write, no WI present, create the new WI.
Detect WxorX memory regions

Legend:
- EXEC
- WRITE

Steps:
2. The current instruction is a write, the ranges of the write overlaps an existing WI, update the matched WI.
Detect WxorX memory regions

Steps:

3. The current instruction is a write, the ranges of the write don’t overlap any WI, create a new WI

Legend:
- EXEC
- WRITE
- Start addr.
- End addr.

Current instr.
0x401004 - 0x425008

Write set

- WRITE
0x412000 - 0x413000

Write interval 1
0x401000 - 0x403000

Write interval 2
0x412000 - 0x413000

Legend:
- Generic instruction
- Write instruction and its ranges
Detect WxorX memory regions

Steps:

4. The EIP of the current instruction is inside a WI

WxorX RULE BROKEN

Legend:

- EXEC: Generic instruction
- WRITE: Write instruction and its ranges

Current instr.:

PinDemonium

Write set

Write interval 1:
0x400000 - 0x403000

Write interval 2:
0x412000 - 0x413000

Start addr. - End addr.
Ok the core of the problem has been resolved...

... but we have just scratch the surface of the problem. Let’s collect the results obtained so far...
Dump the program correctly

Steps:

1. The execution of a written address is detected
Dump the program correctly

Steps:

2. PinDemonium get the addresses of the main module
Dump the program correctly

Steps:

3. PinDemonium dumps these memory range
Dump the program correctly

Steps:

4. Scylla to reconstruct the PE and set the Original Entry Point
Have we already finished?

Nope...
Unpacking on the heap

What if the original code is written on the heap?

Steps:
Unpacking on the heap

What if the original code is written on the heap?

Steps:

1. The execution of a written address is detected
2. PinDemonium get the addresses of the main module
3. PinDemonium dumps these memory range
4. Scylla to reconstruct the PE and set the Original Entry Point
Unpacking on the heap

The OEP doesn’t make sense!

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
<th>Type</th>
<th>Value</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Magic</td>
<td>000000F8</td>
<td>Word</td>
<td>010B</td>
<td>PE32</td>
</tr>
<tr>
<td>MajorLinkerVersion</td>
<td>000000FA</td>
<td>Byte</td>
<td>0A</td>
<td></td>
</tr>
<tr>
<td>MinorLinkerVersion</td>
<td>000000FB</td>
<td>Byte</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>SizeOfCode</td>
<td>000000FC</td>
<td>Dword</td>
<td>00003A00</td>
<td></td>
</tr>
<tr>
<td>SizeOfInitializedData</td>
<td>00000100</td>
<td>Dword</td>
<td>00003600</td>
<td></td>
</tr>
<tr>
<td>SizeOfUninitializedData</td>
<td>00000104</td>
<td>Dword</td>
<td>00000000</td>
<td></td>
</tr>
<tr>
<td>AddressOfEntryPoint</td>
<td>00000108</td>
<td>Dword</td>
<td>01E90000</td>
<td>Invalid</td>
</tr>
</tbody>
</table>
Unpacking on the heap

Solution

Add the heap memory range in which the WxorX rule has been broken as a new section inside the dumped PE!

1. Keep track of write-intervals located on the heap
2. Dump the heap-zone where the WxorX rule is broken
3. Add it as a new section inside the PE
4. Set the OEP inside this new added section
Unpacking on the heap

The OEP is correct!

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
<th>Type</th>
<th>Value</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>Magic</td>
<td>000000F8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Word</td>
<td>010B</td>
<td></td>
<td>PE32</td>
<td></td>
</tr>
<tr>
<td>MajorLinkerVersion</td>
<td>000000FA</td>
<td>Byte</td>
<td>0A</td>
<td></td>
</tr>
<tr>
<td>MinorLinkerVersion</td>
<td>000000FB</td>
<td>Byte</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>SizeOfCode</td>
<td>000000FC</td>
<td>Dword</td>
<td>00003A00</td>
<td></td>
</tr>
<tr>
<td>SizeOfInitializedData</td>
<td>00000100</td>
<td>Dword</td>
<td>00003600</td>
<td></td>
</tr>
<tr>
<td>SizeOfUninitializedData</td>
<td>00000104</td>
<td>Dword</td>
<td>00000000</td>
<td></td>
</tr>
<tr>
<td>AddressOfEntryPoint</td>
<td>00000108</td>
<td>Dword</td>
<td>0001A000</td>
<td>.heap</td>
</tr>
</tbody>
</table>
Unpacking on the heap

However, the dumped heap-zone can contain references to addresses inside other **not dumped** memory areas!
Unpacking on the heap

Solution

Dump all the heap-zones and load them in IDA in order to allow static analysis!

1. Retrieve all the currently allocated heap-zones
2. Dump these heap-zones
3. Create new segments inside the .idb for each of them
4. Copy the heap-zones content inside these new segments!
Unpacking on the heap

```
; DATA XREF: HEADER:00400204h

.head: 08410000 start:
.head: 08410008 add   eax, 1
.head: 0841000c add   eax, 2
.head: 08410010 mov   eax, dword ptr ds:aAAAA_0 ; "AAAA"
.head: 08410018 mov   eax, 22C00000h
.head: 08410020 call   eax
.head: 08410028 ;-----------------------------------------------

; Segement type: Regular
; Segement alignment "" can not be represented in assembly

seg021       segment para private "" use32
    assume cs:seg021
    ;org 22C00000h
    assume ss:nothing, es:nothing, fs:nothing, gs:nothing
    xor edx, edx
    push eax
```

Two down, two still standing!

Reverser we are coming for you! Let’s deobfuscate some imported functions...
Deobfuscate the IAT

Extended Scylla functionalities:

- **IAT Search**: Used Advanced and Basic IAT search functionalities provided by Scylla

- **IAT Deobfuscation**: Extended the plugin system of Scylla for IAT deobfuscation
One last step...

Too many dumps, too many programs making too many problems... Can’t you see? This is the land of confusion
Recognize the correct dump

We have to find a way to identify the correct dump

Idea

Give for each dump a “quality” index using the heuristics defined in our heuristics module

1. Entropy difference
Recognize the correct dump

We have to find a way to identify the correct dump

Idea

Give for each dump a “quality” index using the heuristics defined in our heuristics module

1. Entropy difference
2. Far jump
Recognize the correct dump

We have to find a way to identify the correct dump

Idea

Give for each dump a “quality” index using the heuristics defined in our heuristics module

1. Entropy difference
2. Far jump
3. Jump outer section
Recognize the correct dump

We have to find a way to identify the correct dump

Idea

Give for each dump a “quality” index using the heuristics defined in our heuristics module

1. Entropy difference
2. Far jump
3. Jump outer section
4. Yara rules
Yara Rules

Yara is executed on the dumped memory and a set of rules is checked for two main reasons:

Detecting Evasive code
- Anti-VM
- Anti-Debug

Identifying malware family
- Detect the Original Entry Point
- Identify some malware behaviours
Advanced Problems
You either die a hero or you live long enough to see yourself become the villain.

Exploit PIN functioning to break PIN

A.k.a. Self modifying trace
Self modifying trace

Steps:

Code Cache

Main module of target program

ins_1
ins_2
wrong_ins_3
ins_4
ins_5
Steps:

1. The trace is collected in the code cache
Self modifying trace

Steps:

2. Execute the analysis routine before the write
Steps:

3. The wrong instruction is patched in the main module
Self modifying trace

Steps:

4. The wrong ins_3 is executed

CRASH!
Solution
Self modifying trace

Steps:

List of written addresses

ins_1(write)
ins_2
crash_ins_3
ins_4

ins_1
ins_2
crash_ins_3
ins_4
ins_5
Self modifying trace

Steps:

1. Insert one analysis routine before each instruction and another one if the instruction is a write
Self modifying trace

Steps:

2. Execute the analysis routine before the write
Self modifying trace

Steps:

3. The crash_ins_3 is patched in the main module

CheckEipWritten()
MarkWrittenAddress()
ins_1 (write)
CheckEipWritten()
ins_2
CheckEipWritten()
crash_ins_3
CheckEipWritten()
ins_4

List of written addresses

crash_ins_3_addr

ins_1
ins_2

3

crash_ins_3

ins_3
ins_4
ins_5
Self modifying trace

Steps:

4. Check if crash_ins_3 address is inside the list

YES!
Self modifying trace

Steps:

5. **Stop** the execution

List of written addresses:

- crash_ins_3_addr

Steps:

- CheckEipWritten()
- MarkWrittenAddress()
- ins_1 (write)
- CheckEipWritten()
- ins_2
- CheckEipWritten()
- crash_ins_3
- CheckEipWritten()
- ins_4
- ins_1
- ins_2
- crash_ins_3
- ins_4
- ins_5
Self modifying trace

Steps:

6. Recollect the new trace
Are there other ways to break the WxorX rule?
Process Injection

Inject code into the memory space of a different process and then execute it

- Dll injection
- Reflective Dll injection
- Process hollowing
- Entry point patching
Solution
Process Injection

Identify remote writes to other processes by hooking system calls:

● NtWriteVirtualMemory
● NtMapViewOfSection

Identify remote execution of written memory by hooking system calls:

● NtCreateThreadEx
● NtResumeThread
● NtQueueApcThread
Finally for the SWAG!
Experiments

➔ **Test 1**: test our tool against the same binary packed with different known packers.

➔ **Test 2**: test our tool against a series of packed malware sample collected from VirusTotal.
Experiment 1: known packers

<table>
<thead>
<tr>
<th></th>
<th>Upx</th>
<th>FSG</th>
<th>Mew</th>
<th>mpress</th>
<th>PeCompact</th>
<th>Obsidium</th>
<th>ExePacker</th>
<th>ezip</th>
</tr>
</thead>
<tbody>
<tr>
<td>MessageBox</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>WinRAR</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Xcomp</th>
<th>PElock</th>
<th>ASProtect</th>
<th>ASPack</th>
<th>eXpressor</th>
<th>exe32packer</th>
<th>beropacker</th>
<th>Hyperion</th>
<th>PeSpin</th>
</tr>
</thead>
<tbody>
<tr>
<td>MessageBox</td>
<td>✓</td>
<td>⚠️</td>
<td>✓</td>
<td>✓</td>
<td>⚠️</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>WinRAR</td>
<td>✓</td>
<td>⚠️</td>
<td>✓</td>
<td>✓</td>
<td>⚠️</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

⚠️ Original code dumped but Import directory not reconstructed
# Experiment 2: wild samples

Number of packed (checked manually) samples

1066

<table>
<thead>
<tr>
<th></th>
<th>N°</th>
<th>% of all</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unpacked and working</td>
<td>519</td>
<td>49</td>
</tr>
<tr>
<td>Unpacked but Different behaviour</td>
<td>150</td>
<td>14</td>
</tr>
<tr>
<td>Unpacked but not working</td>
<td>139</td>
<td>13</td>
</tr>
<tr>
<td>Not unpacked</td>
<td>258</td>
<td>24</td>
</tr>
</tbody>
</table>
## Experiment 2: Wild samples

### Number of packed (checked manually) samples

<table>
<thead>
<tr>
<th></th>
<th>$N$</th>
<th>% of all</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unpacked and working</td>
<td>519</td>
<td>49</td>
</tr>
<tr>
<td>Unpacked but Different behaviour</td>
<td>150</td>
<td>14</td>
</tr>
<tr>
<td>Unpacked but not working</td>
<td>139</td>
<td>13</td>
</tr>
<tr>
<td>Not unpacked</td>
<td>258</td>
<td>24</td>
</tr>
</tbody>
</table>

63%
Limitations

- Performance issues due to the overhead introduced by PIN
- Packers which re-encrypt / compress code after its execution are not supported
- Evasion techniques are not handled
Conclusions

- Generic unpacker based on a DBI
- Able to reconstruct a working version of the original binary
- Able to deal with IAT obfuscation and dumping on the heap
Conclusions

17 common packers defeated

63% of random samples correctly unpacked (known and custom packers employed)
DEMO
The source code is available at

https://github.com/PINdemonium
That's all Folks!

Thank you!