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

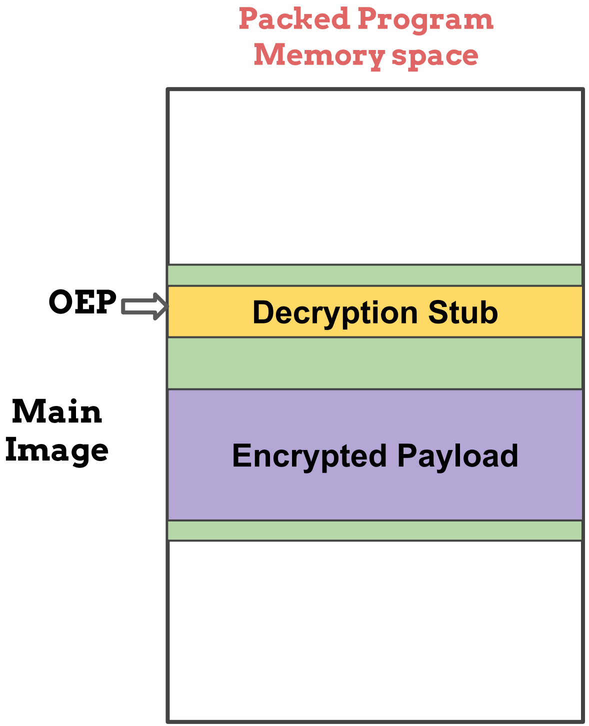
- Compress or encrypt the original code → Code and strings analysis impossible
- Obfuscate the imported functions → Analysis of the imported functions avoided

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

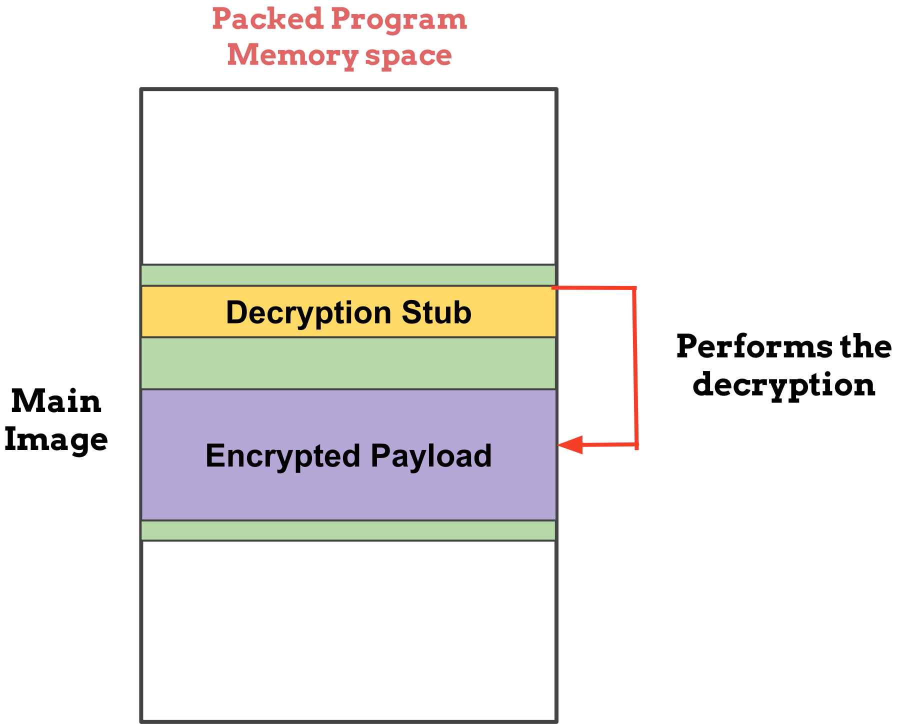
Overriding the Main Image



Steps:

1. Start the execution of the decryption stub

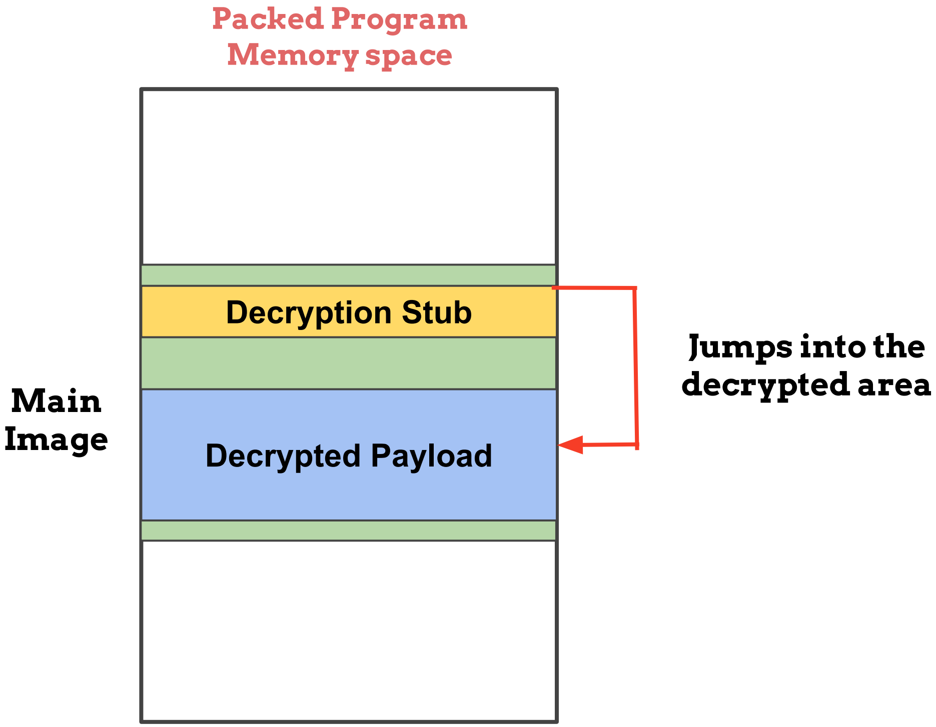
Overriding the Main Image



Steps:

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

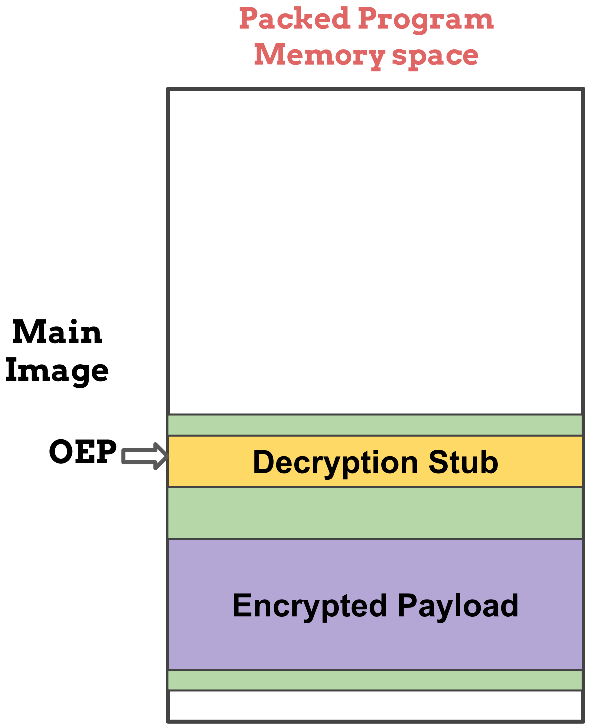
Overriding the Main Image



Steps:

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

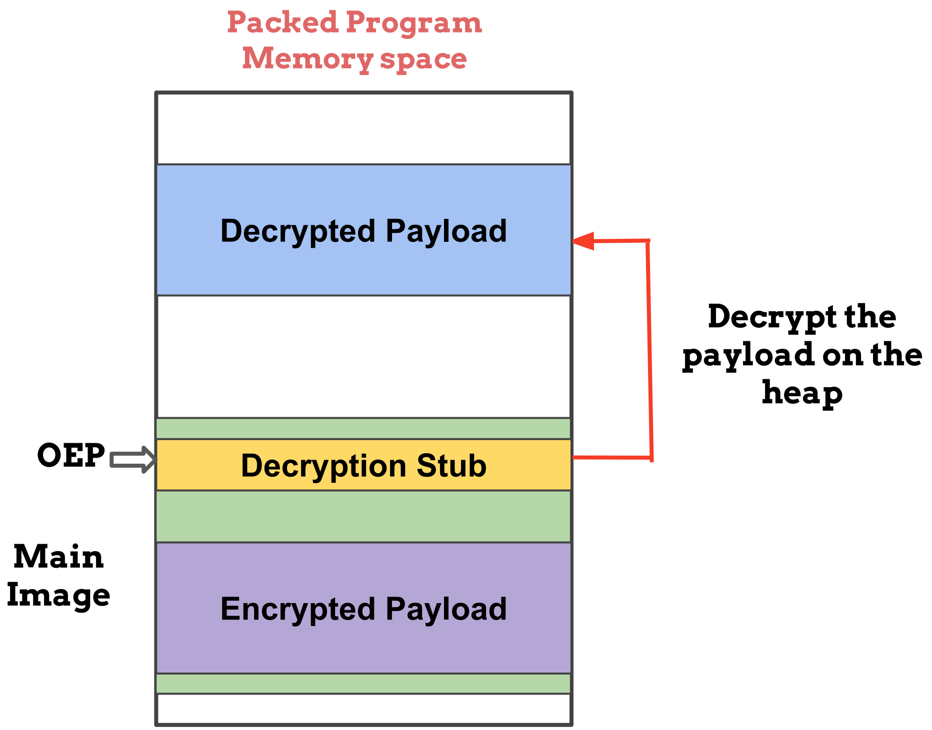
Unpacking on the Heap



Steps:

1. Start the execution of the decryption stub

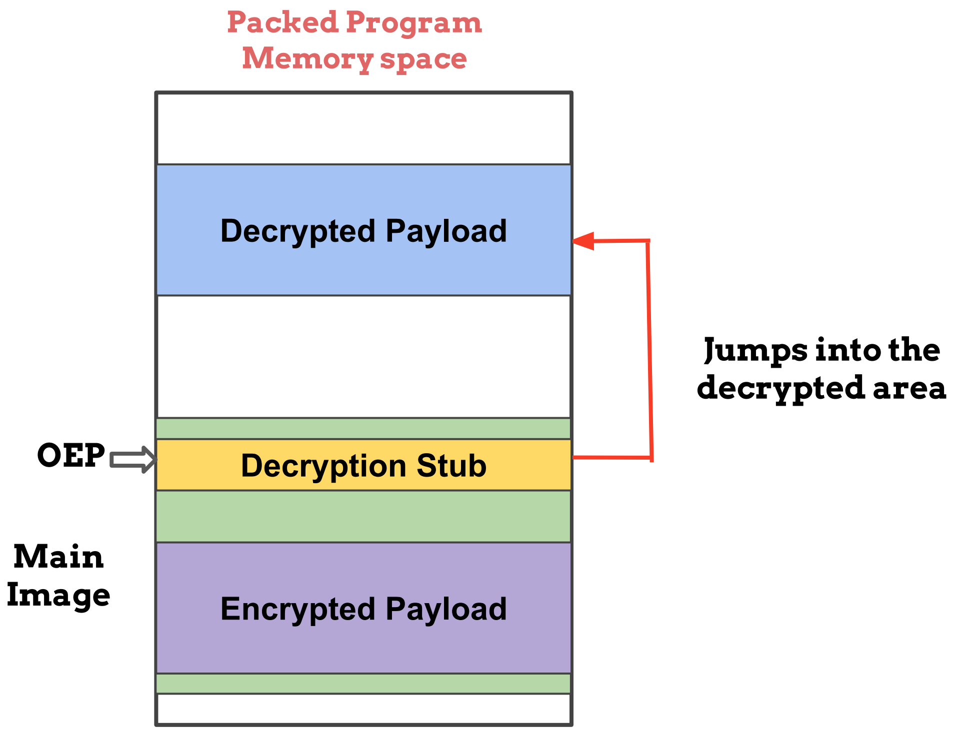
Unpacking on the Heap



Steps:

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

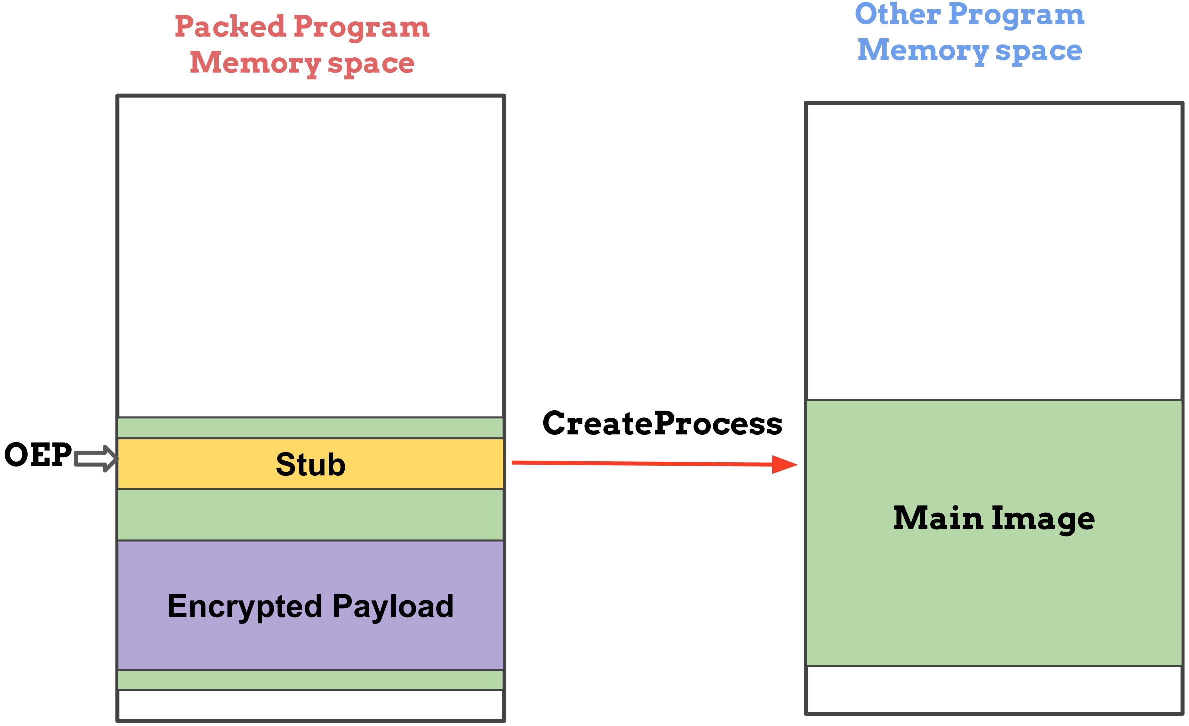
Unpacking on the Heap



Steps:

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

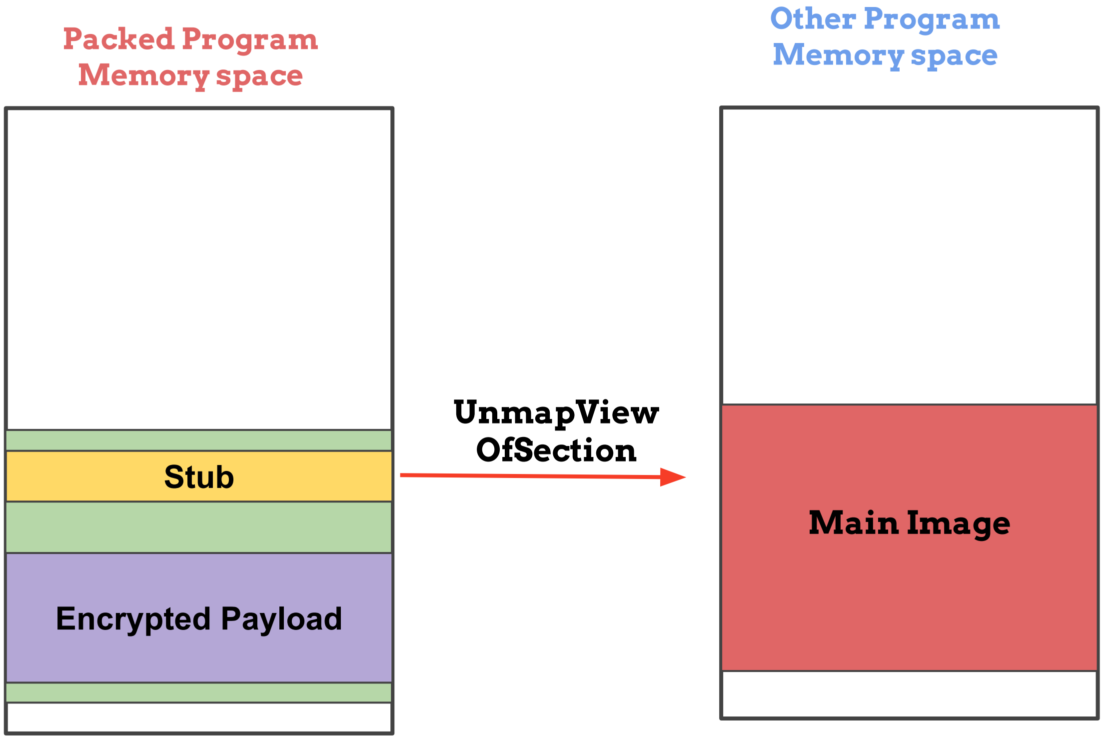
Process Injection



Steps:

1. Create remote legitimate process in a suspended state

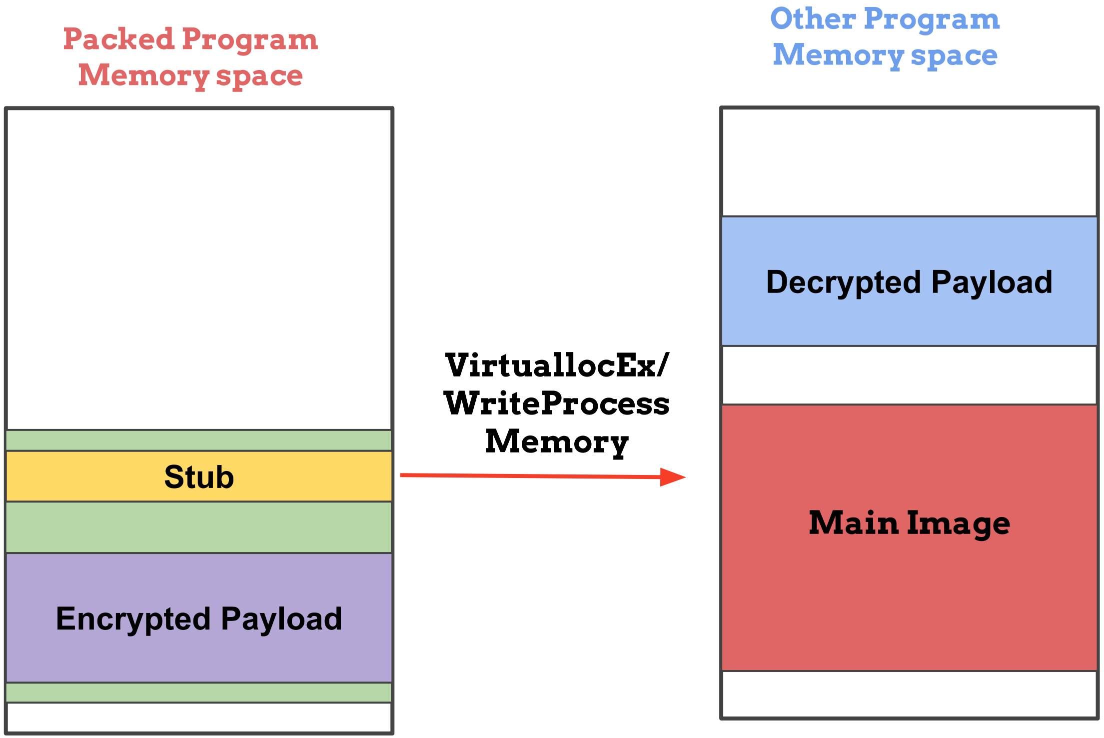
Process Injection



Steps:

2. Unmap the legitimate code section of the process

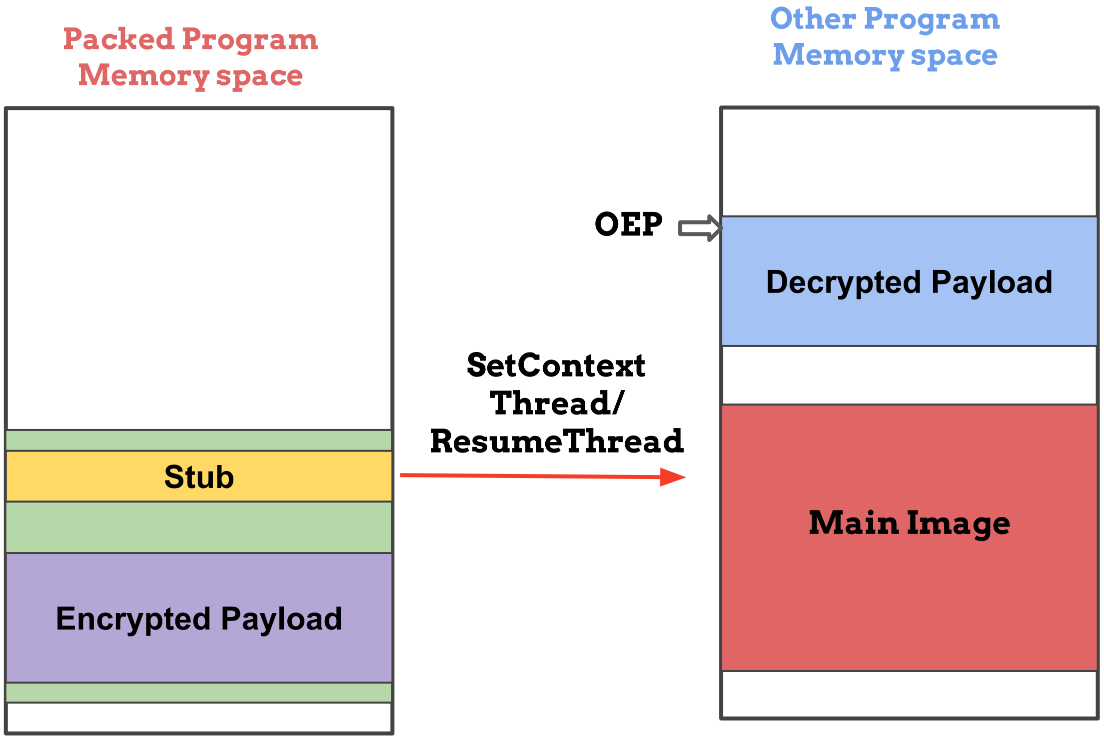
Process Injection



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 are and resume the thread execution

Solutions

black hat USA 2016

Manual approach

- Very time consuming
- Too many samples to be analyzed every day
- Adapt the approach to deal with different techniques

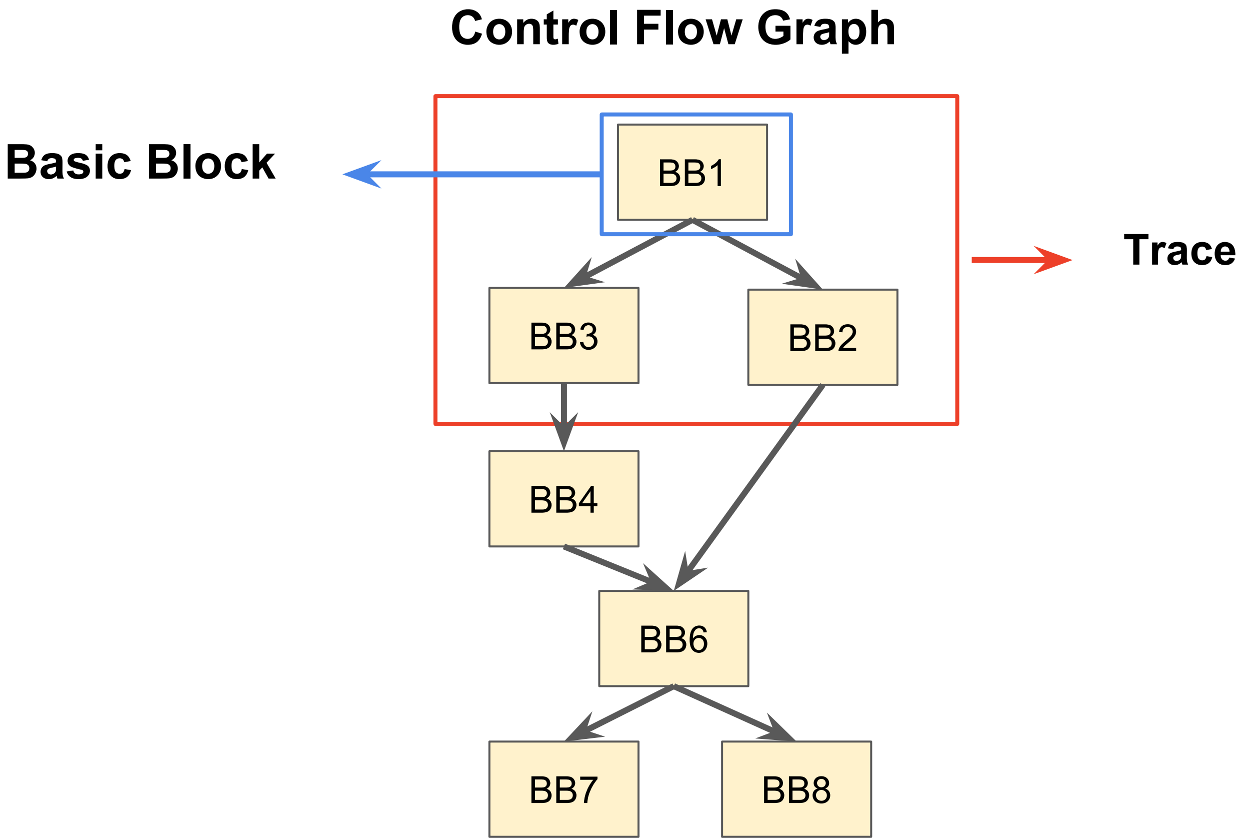
Automatic approach

- Fast analysis
- Scale well on the number of samples that has to be analyzed every day
- Single approach to deals with multiple techniques

All hail

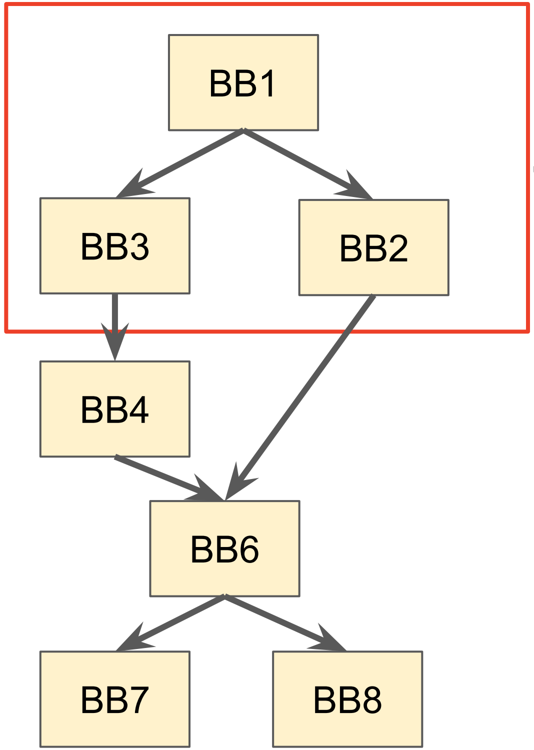
PinDemonium

What is a DBI?



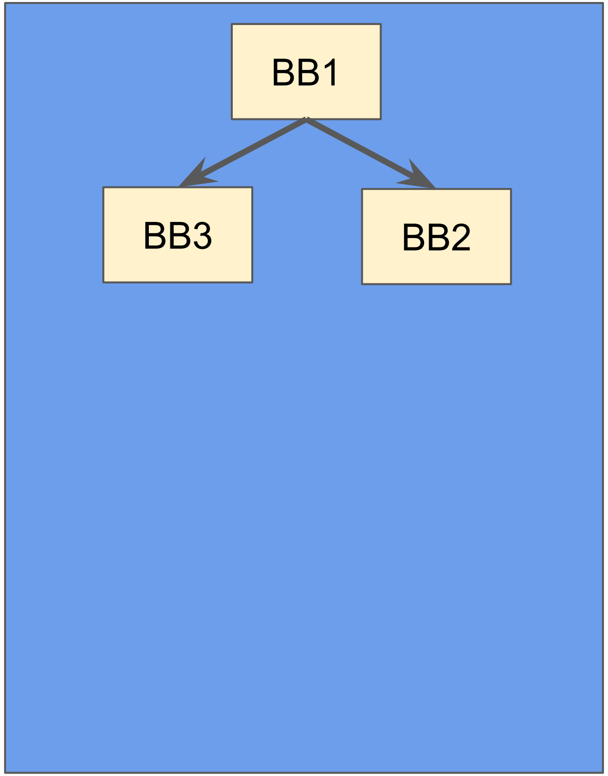
Control Flow Graph
Basic Block
Trace
BB1
BB3
BB2
BB4
BB6
BB7
BB8

What is a DBI?

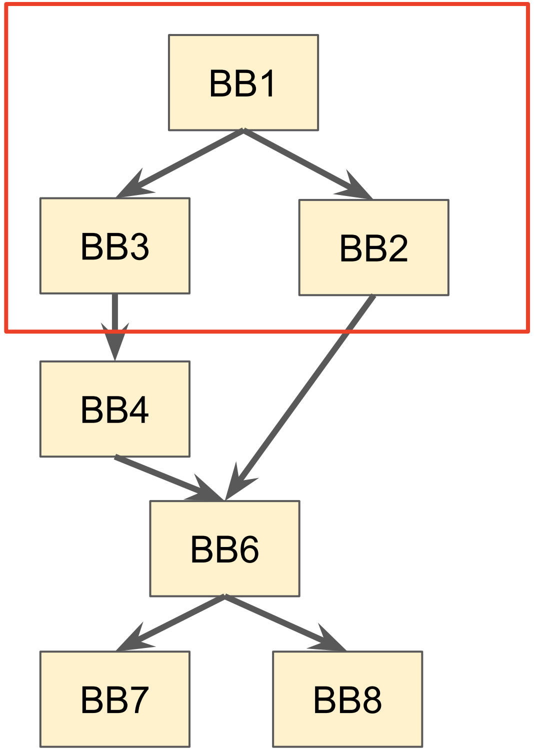


Trace is copied in the code cache

Code Cache



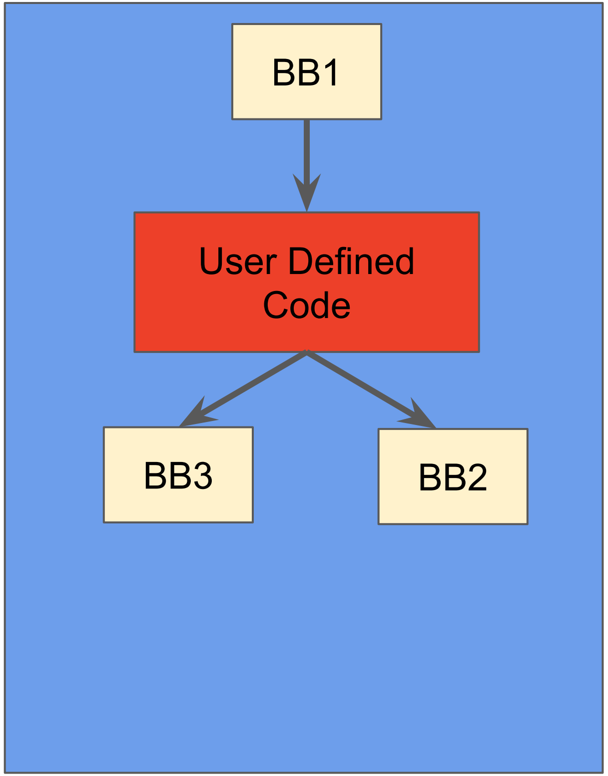
What is a DBI?



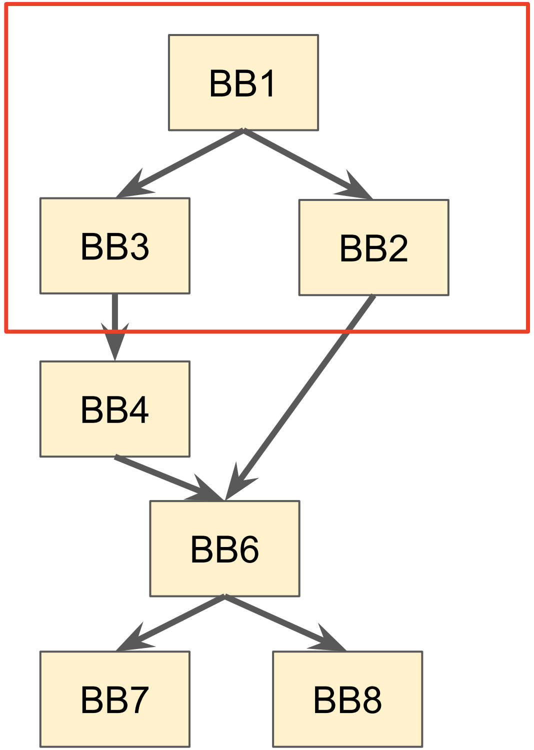
DBI provides the possibility to add user defined code after each:

- Instruction
- Basic Block
- Trace

Code Cache

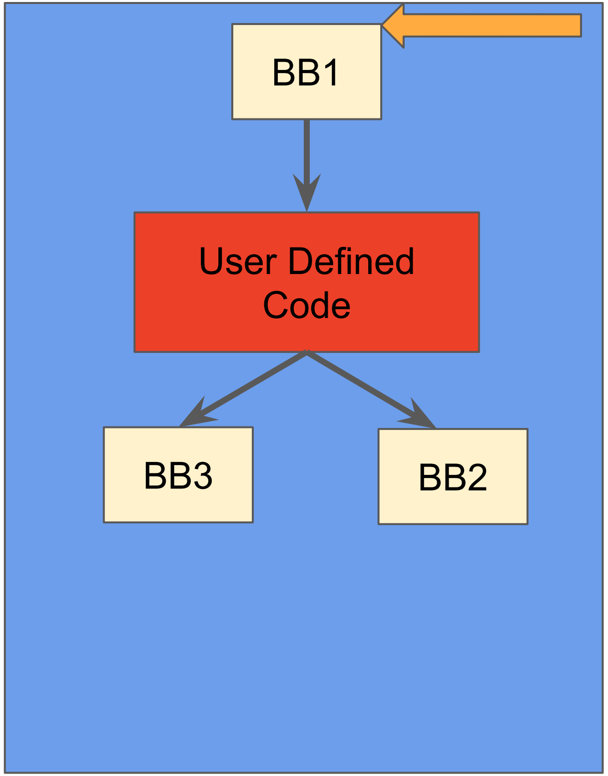


What is a DBI?



DBI starts executing the program from the code cache

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

black hat USA 2016

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

black hat USA 2016

0x401004	0x425008	0x425004	0x425000
EXEC	WRITE 0x412000 - 0x413000	WRITE 0x402000 - 0x403000	WRITE 0x401000 - 0x402000

Current instr.

PinDemonium

Write set

Legend:

EXEC — Generic instruction

WRITE Start addr. - End addr. — Write instruction and its ranges

Steps:

Detect WxorX memory regions

0x401004	0x425008	0x425004	0x425000
EXEC	WRITE 0x412000 - 0x413000	WRITE 0x402000 - 0x403000	WRITE 0x401000 - 0x402000

Current instr.

PinDemonium

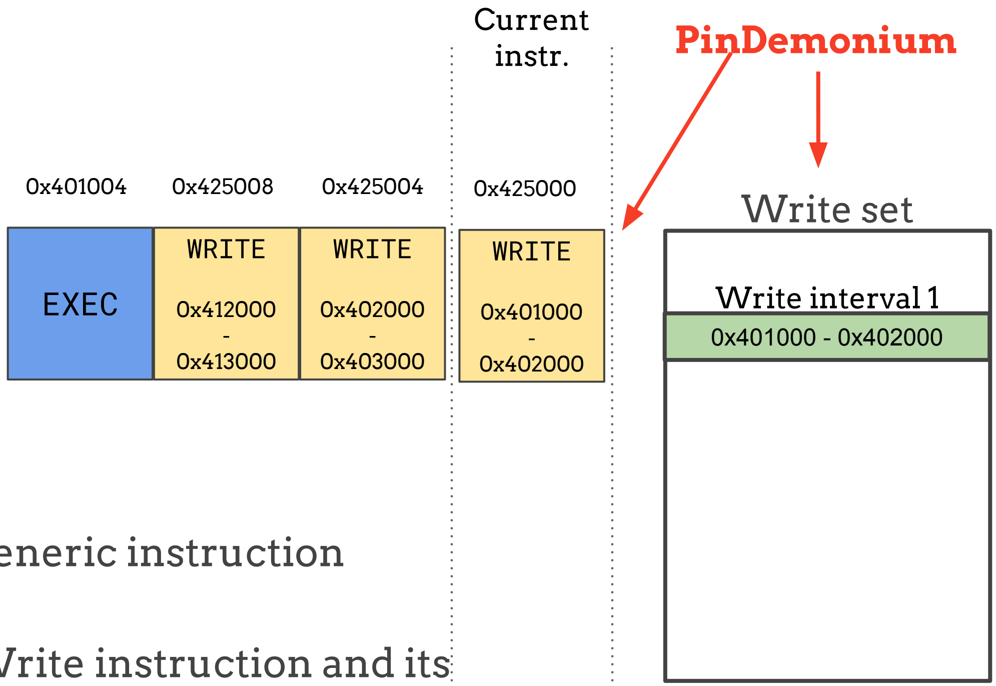
Write set

Legend:

EXEC	Generic instruction
WRITE Start addr. - End addr.	Write instruction and its ranges

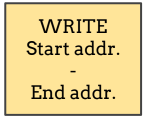
Steps:

Detect WxorX memory regions



Legend:

Generic instruction

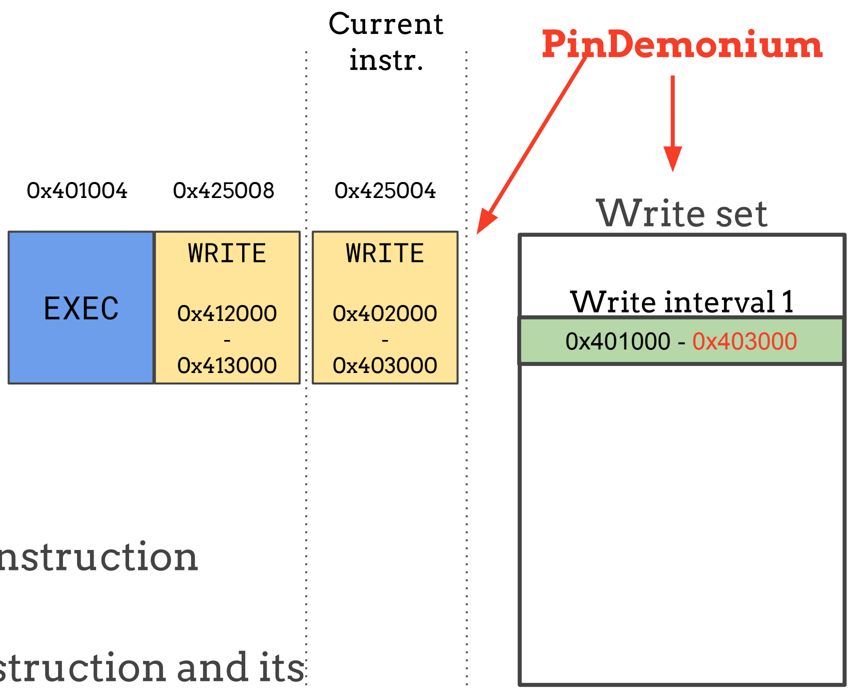


Write instruction and its ranges

Steps:

1. The current instruction is a write, no WI present, create the new WI

Detect WxorX memory regions



Legend:

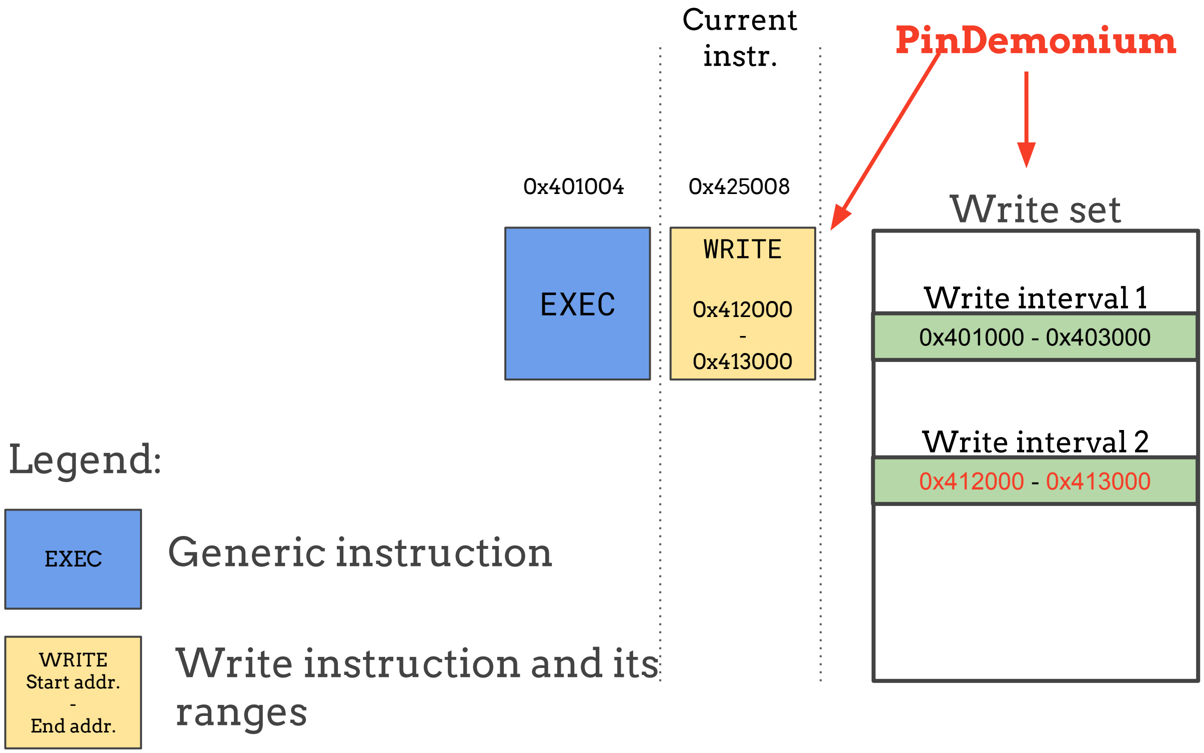
EXEC — Generic instruction

WRITE Start addr. - End addr. — Write instruction and its ranges

Steps:

2. The current instruction is a write, the ranges of the write overlaps an existing WI, update the matched WI

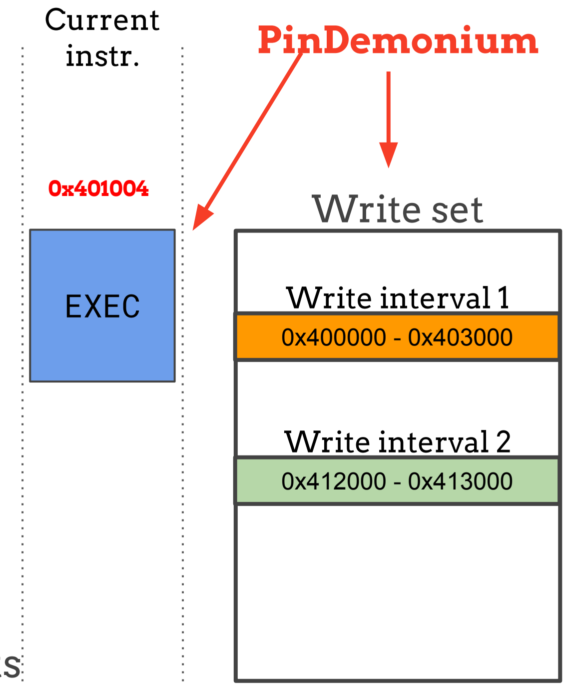
Detect WxorX memory regions



Steps:

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

Detect WxorX memory regions



Legend:

EXEC — Generic instruction

WRITE Start addr. - End addr. — Write instruction and its ranges

Steps:

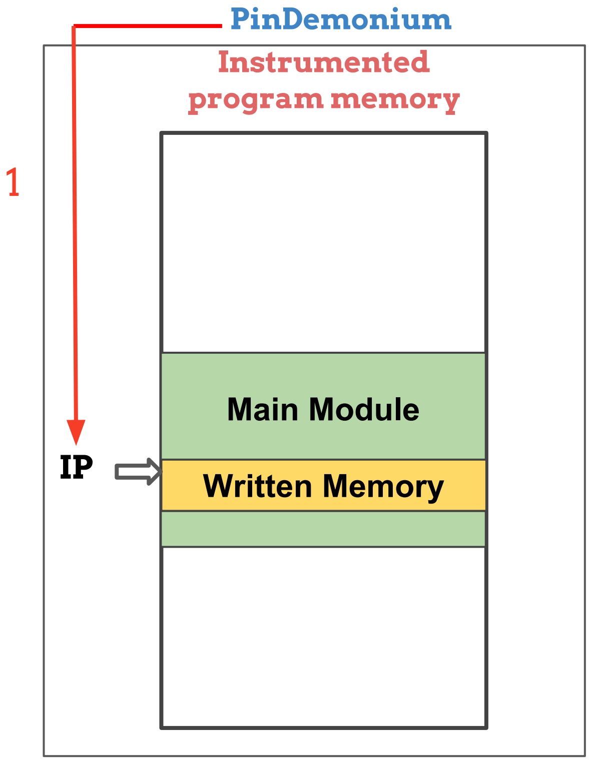
4. The EIP of the current instruction is inside a WI

WxorX RULE BROKEN

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…

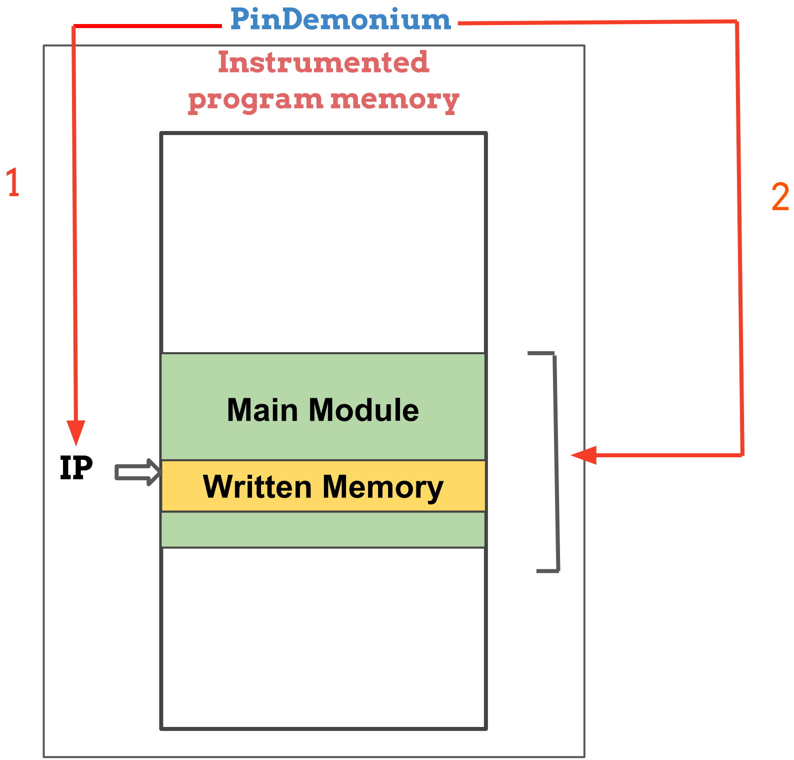
Dump the program correctly



Steps:

1. The execution of a written address is detected

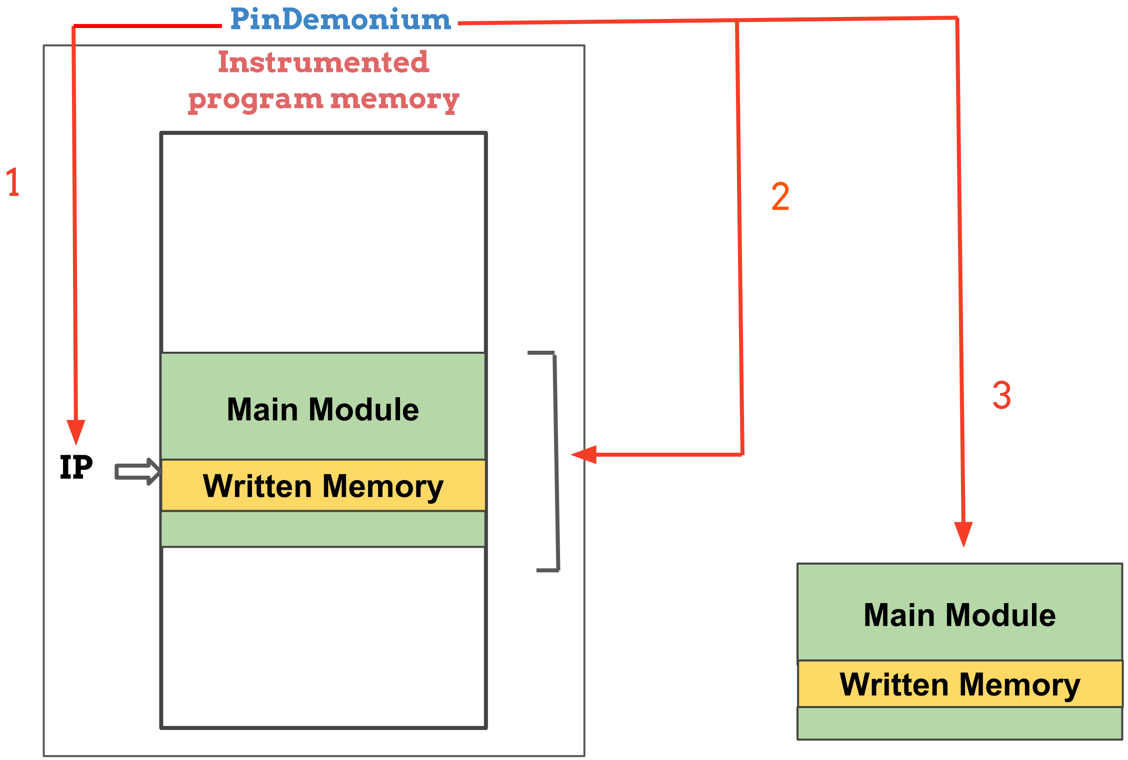
Dump the program correctly



Steps:

2. PinDemonium get the addresses of the main module

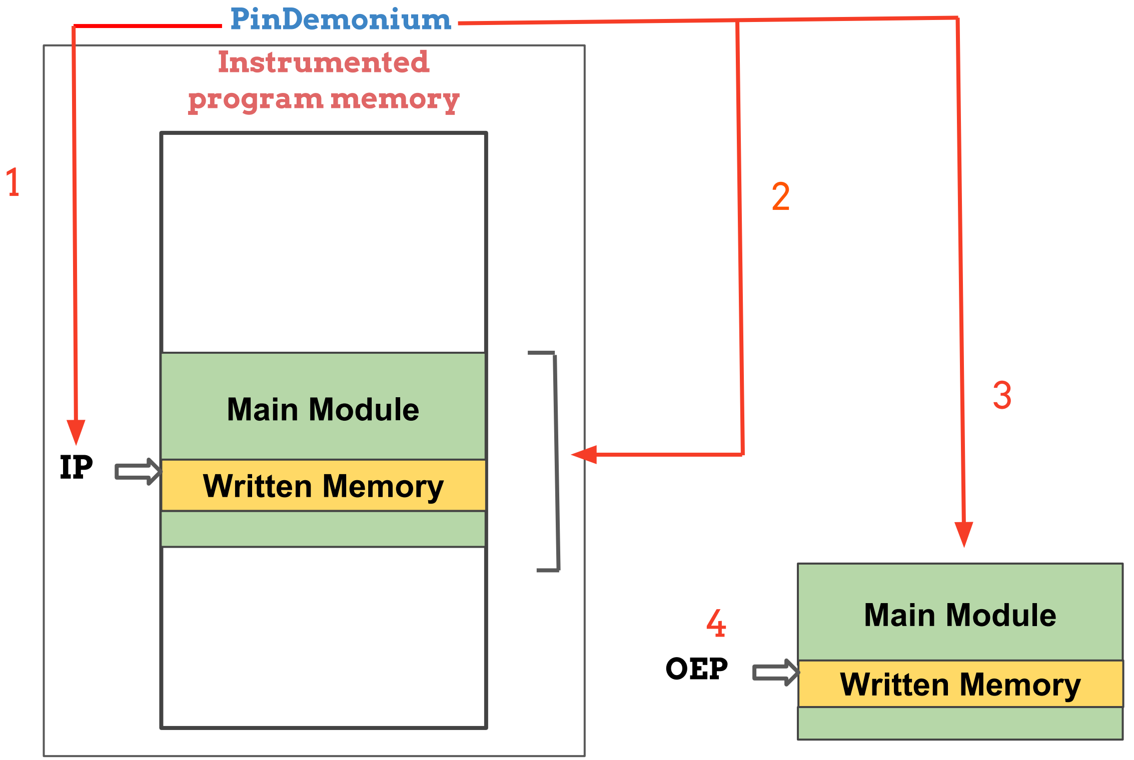
Dump the program correctly



Steps:

3. PinDemonium dumps these memory range

Dump the program correctly



Steps:

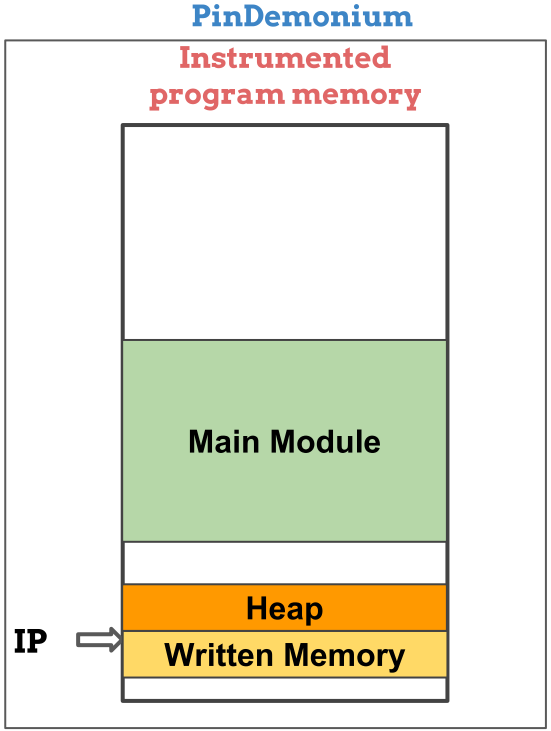
4. Scylla to reconstruct the PE and set the Original Entry Point

Have we already finished?

Nope…

Unpacking on the heap

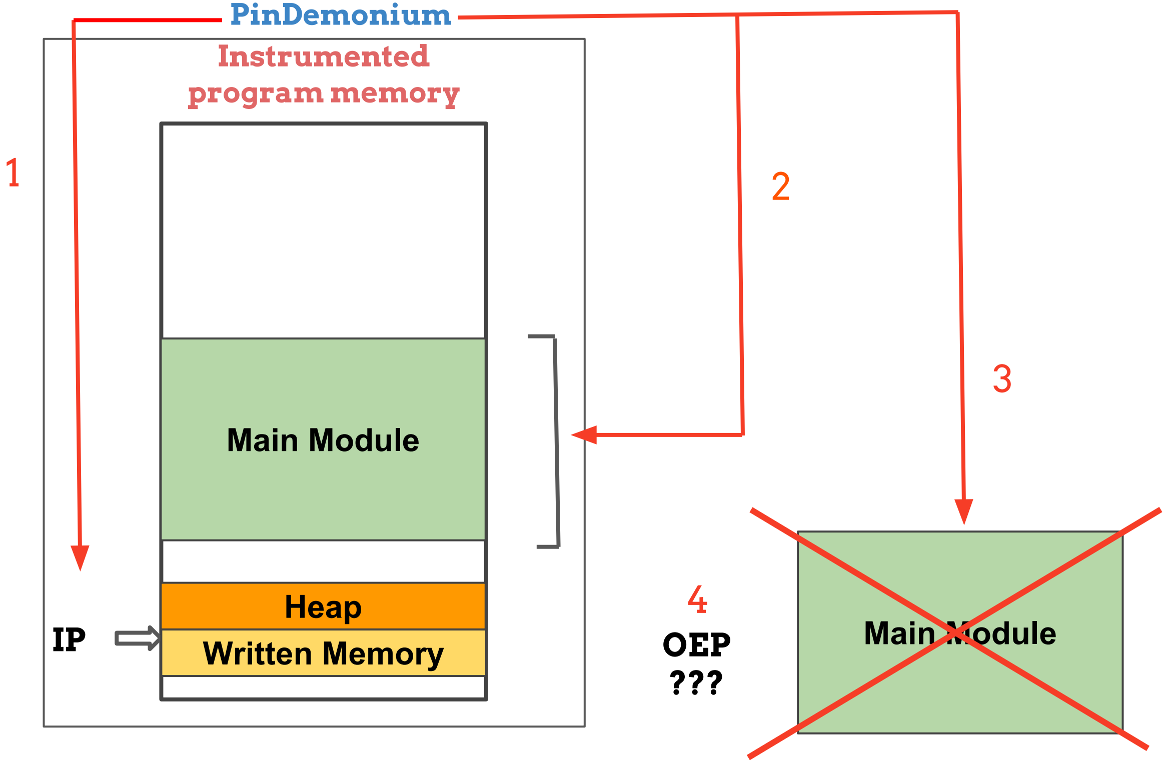
What if the original code is written on the heap?



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!

Magic	000000F8	Word	010B	PE32
MajorLinkerVersion	000000FA	Byte	0A	
MinorLinkerVersion	000000FB	Byte	00	
SizeOfCode	000000FC	Dword	00003A00	
SizeOfInitializedData	00000100	Dword	00003600	
SizeOfUninitializedD...	00000104	Dword	00000000	
AddressOfEntryPoint	00000108	Dword	01E90000	Invalid

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!

Magic	000000F8	Word	010B	PE32
MajorLinkerVersion	000000FA	Byte	0A	
MinorLinkerVersion	000000FB	Byte	00	
SizeOfCode	000000FC	Dword	00003A00	
SizeOfInitializedData	00000100	Dword	00003600	
SizeOfUninitializedD...	00000104	Dword	00000000	
AddressOfEntryPoint	00000108	Dword	0001A000	.heap

Unpacking on the heap

black hat USA 2016

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

```
.heap:0041A000                 assume es:nothing, ss:nothing, ds:_dat
.heap:0041A000
.heap:0041A000                 public start
.heap:0041A000 start:                                   ; DATA XREF: I
.heap:0041A000                 add     eax, 1
.heap:0041A003                 add     eax, 2
.heap:0041A006                 mov     eax, ds:22B0000h
.heap:0041A00B                 mov     eax, 22C0000h
.heap:0041A010                 call    eax
.heap:0041A010 ; ---------------------------------------------------------
```

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


```
.heap:0041A000 start:                          ; DATA XREF: HEADER:004002D4↑o
.heap:0041A000                 add     eax, 1
.heap:0041A003                 add     eax, 2
.heap:0041A006                 mov     eax, dword ptr ds:aAaaa_0 ; "AAAA"
.heap:0041A00B                 mov     eax, 22C0000h
.heap:0041A010                 call    eax
.heap:0041A010 ; -----------------------------------
.heap:0041A012                 dw 0
.heap:0041A014                 align 200h
.heap:0041A200                 dd 380h dup(?)
.heap:0041A200 _heap           ends
.heap:0041A200
seg010:020D0000 ; ==================================
seg010:020D0000
seg010:020D0000 ; Segment type: Regular
seg010:020D0000 ; Segment alignment '' can not be rep
seg010:020D0000 seg010         segment para private
```

```
; ===========================================================================

; Segment type: Regular
; Segment alignment '' can not be represented in assembly
seg021          segment para private '' use32
                assume cs:seg021
                ;org 22C0000h
                assume es:nothing, ss:nothing, ds: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

black hat USA 2016

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

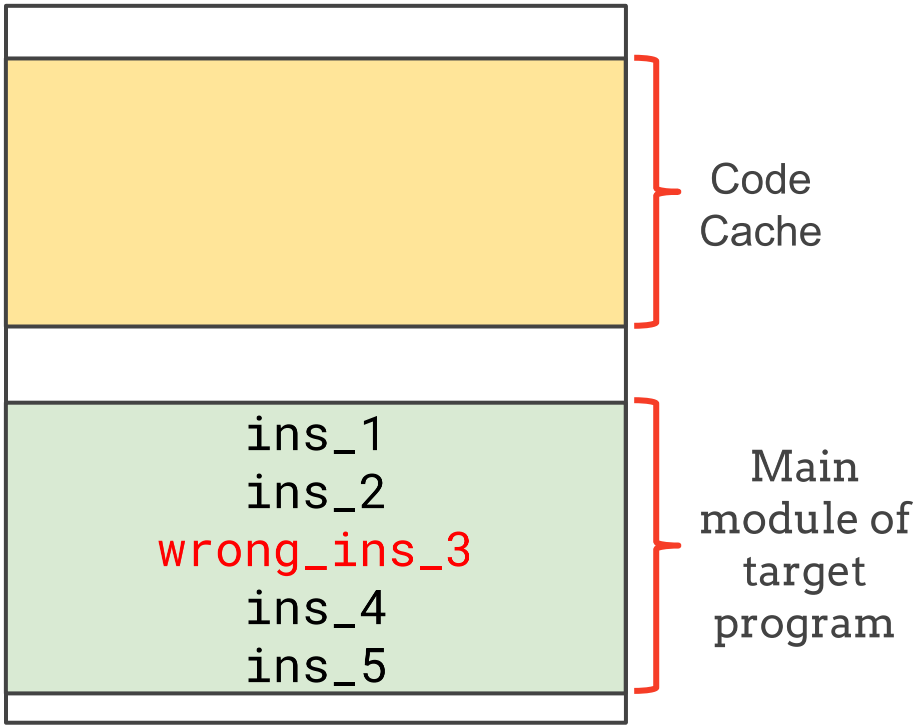
Advanced Problems

You either die a hero or you live long enough to see yourself become the villain

Exploit PIN functioning to break PIN

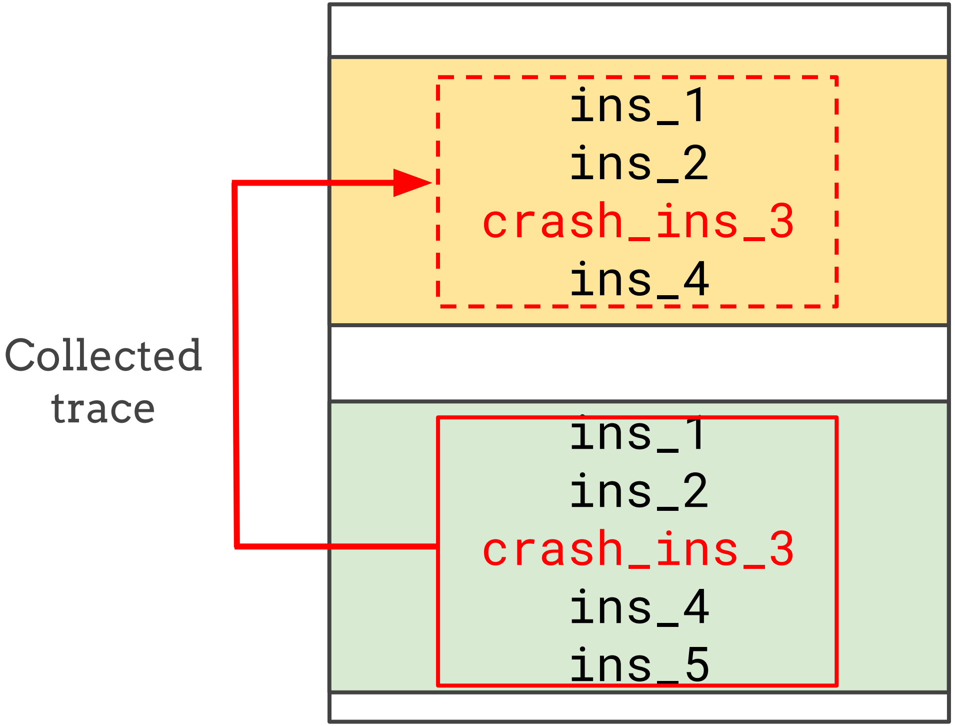
A.k.a. Self modifying trace

Self modifying trace



Steps:

Self modifying trace



Steps:

1. The trace is collected in the code cache

Self modifying trace

Execution starts

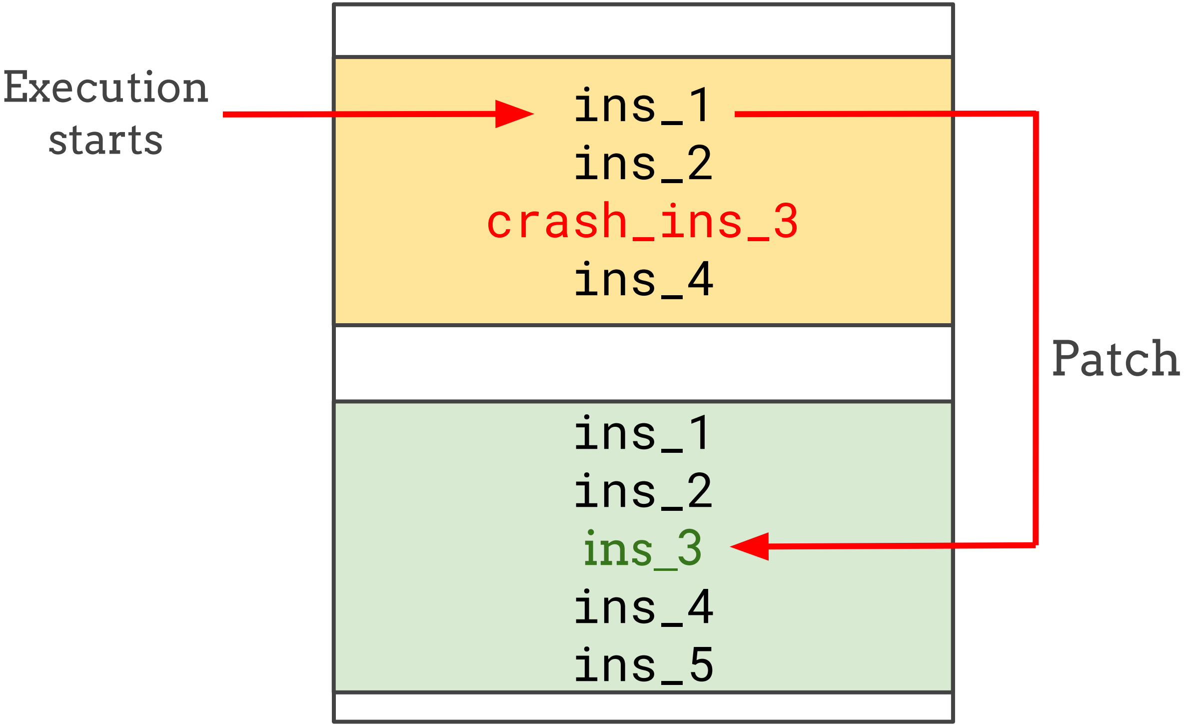
```
ins_1
ins_2
crash_ins_3
ins_4
```

```
ins_1
ins_2
crash_ins_3
ins_4
ins_5
```

Steps:

2. Execute the analysis routine before the write

Self modifying trace



Steps:

3. The wrong instruction is patched in the main module

Self modifying trace

Execute here

ins_1
ins_2
crash_ins_3
ins_4

ins_1
ins_2
ins_3
ins_4
ins_5

Steps:

4. The wrong_ins_3 is executed

CRASH!

Solution

Self modifying trace

ins_1(write)
ins_2
crash_ins_3
ins_4

ins_1
ins_2
crash_ins_3
ins_4
ins_5

List of written addresses

Steps:

Self modifying trace

1

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

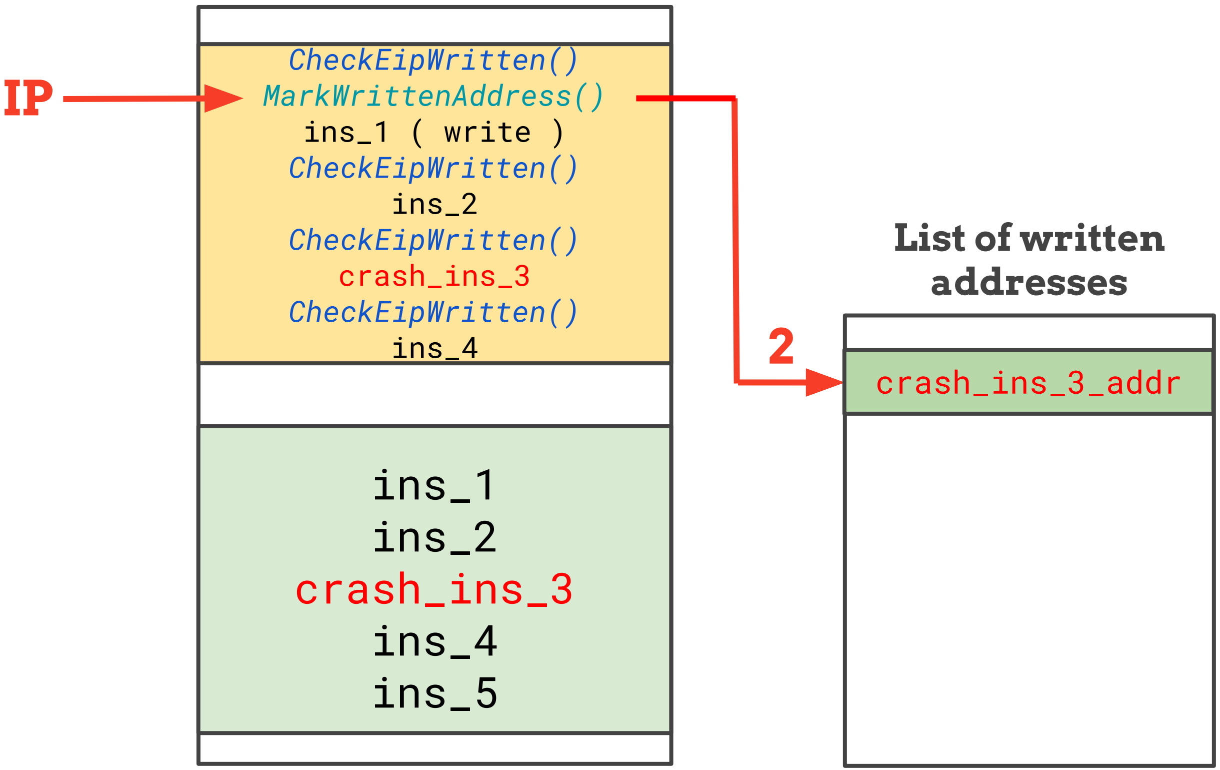
```
ins_1
ins_2
crash_ins_3
ins_4
ins_5
```

List of written addresses

Steps:

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

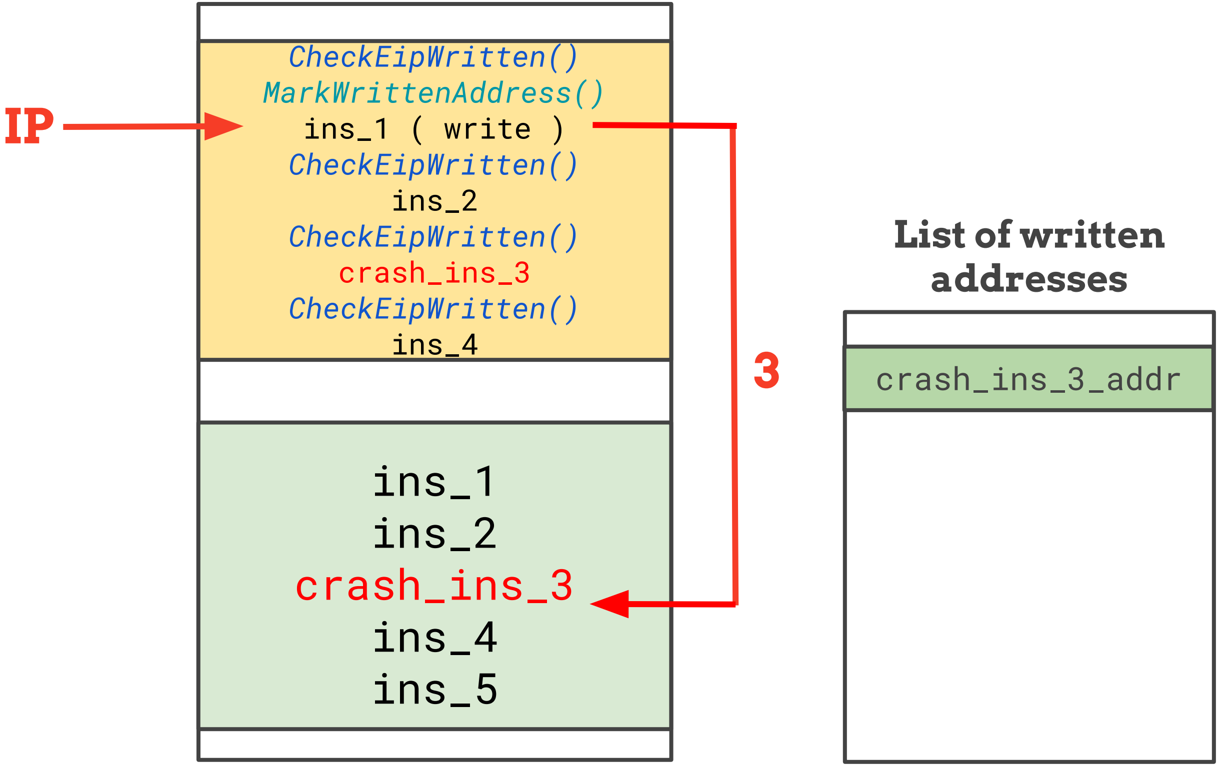
Self modifying trace



Steps:

2. Execute the analysis routine before the write

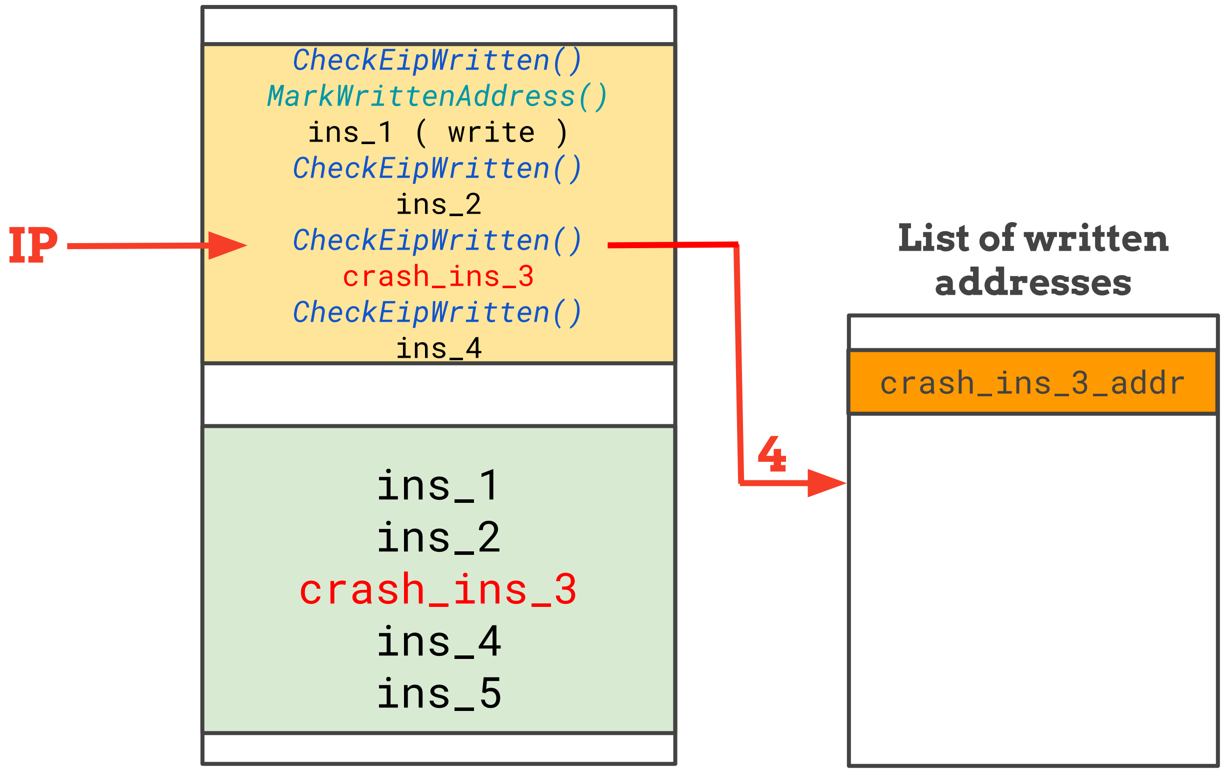
Self modifying trace



Steps:

3. The crash_ins_3 is patched in the main module

Self modifying trace

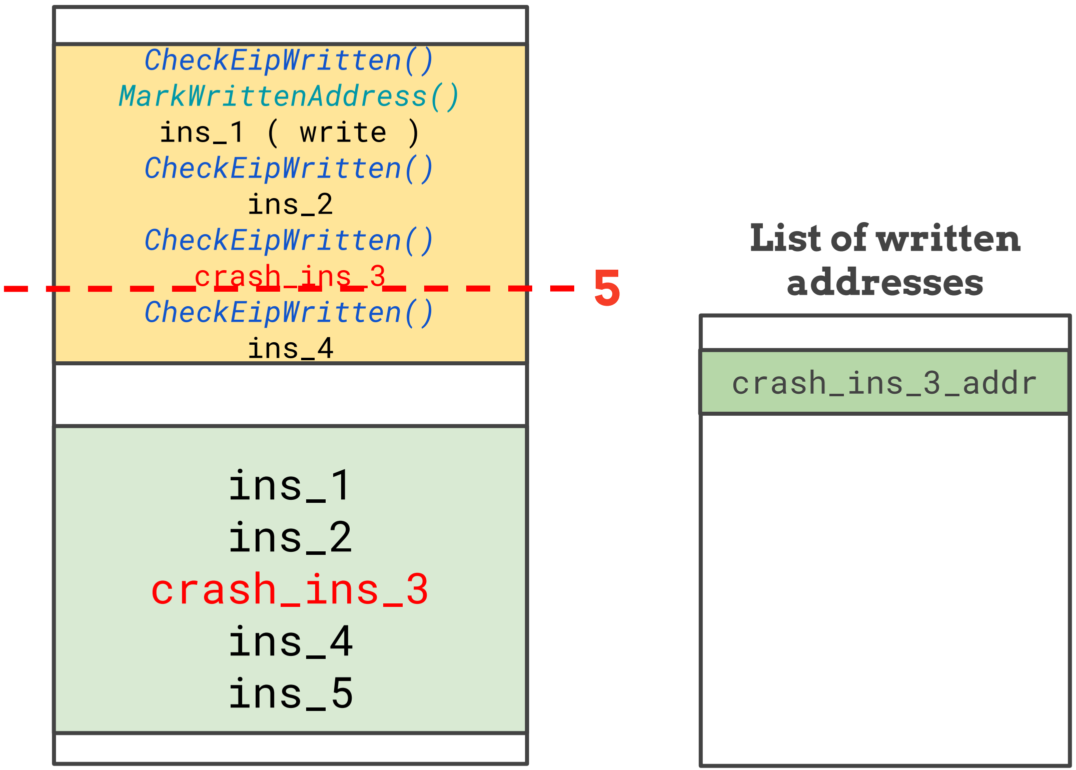


Steps:

4. Check if crash_ins_3 address is inside the list

YES!

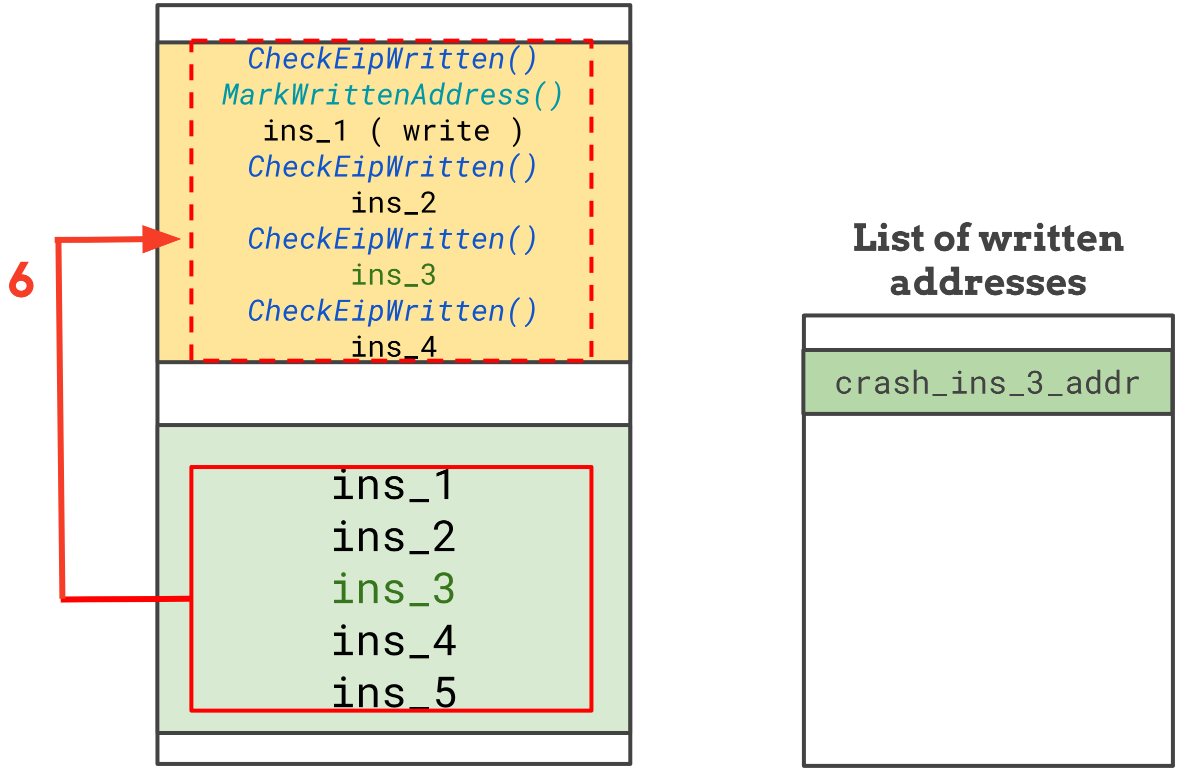
Self modifying trace



Steps:

5. Stop the execution

Self modifying trace



Steps:

6. Recollect the new trace

Are there other ways to break the WxorX rule?

Process Injection

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

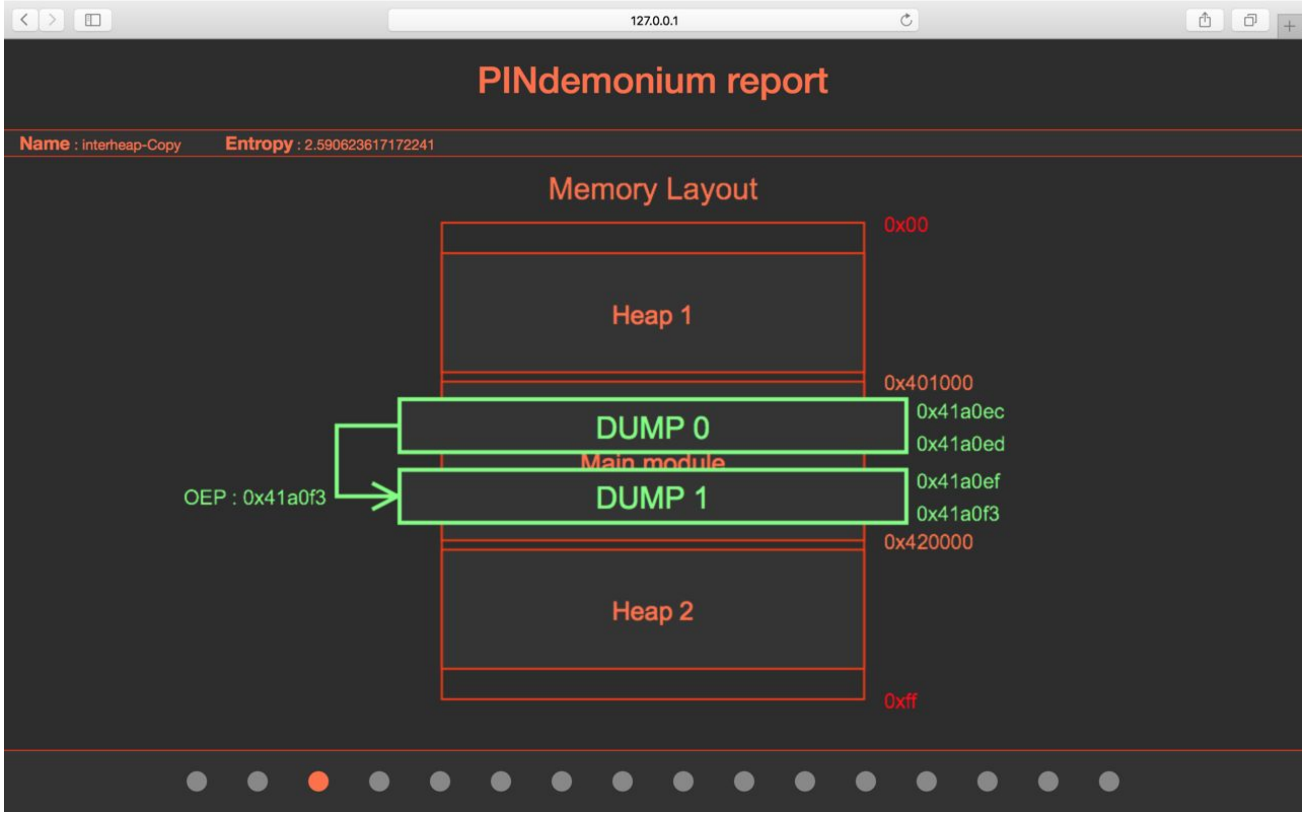
Identify remote writes to other processes by hooking system calls:

- NtWriteVirutalMemory
- 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

black hat USA 2016

	Upx	FSG	Mew	mpress	PeCompact	Obsidium	ExePacker	ezip
MessageBox	✓	✓	✓	✓	✓	✗	✓	✓
WinRAR	✓	✓	✓	✓	✓	✗	✓	✓

	Xcomp	PElock	ASProtect	ASPack	eXpressor	exe32packer	beropacker	Hyperion	PeSpin
MessageBox	✓	!	✓	✓	!	✓	✓	✓	✓
WinRAR	✓	!	✓	✓	!	✓	✓	✓	✓

! → Original code dumped but Import directory not reconstructed

Experiment 2 : wild samples

Number of packed (checked manually) samples

1066

	N°	% of all
Unpacked and working	519	49
Unpacked but Different behaviour	150	14
Unpacked but not working	139	13
Not unpacked	258	24

Experiment 2 : wild samples

black hat USA 2016

Number of packed (checked manually) samples

1066

	N°	% of all
Unpacked and working	519	49
Unpacked but Different behaviour	150	14
Unpacked but not working	139	13
Not unpacked	258	24

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

Thank you!