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

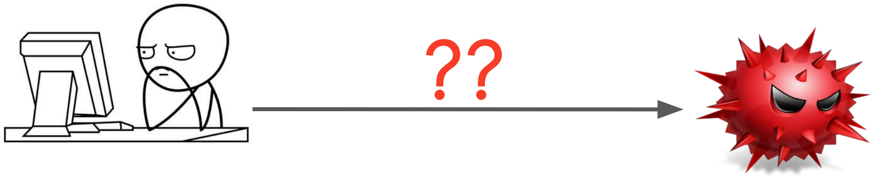
- Analysis of disassembled code
- Analysis of imported functions
- Analysis of strings

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



Solutions

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

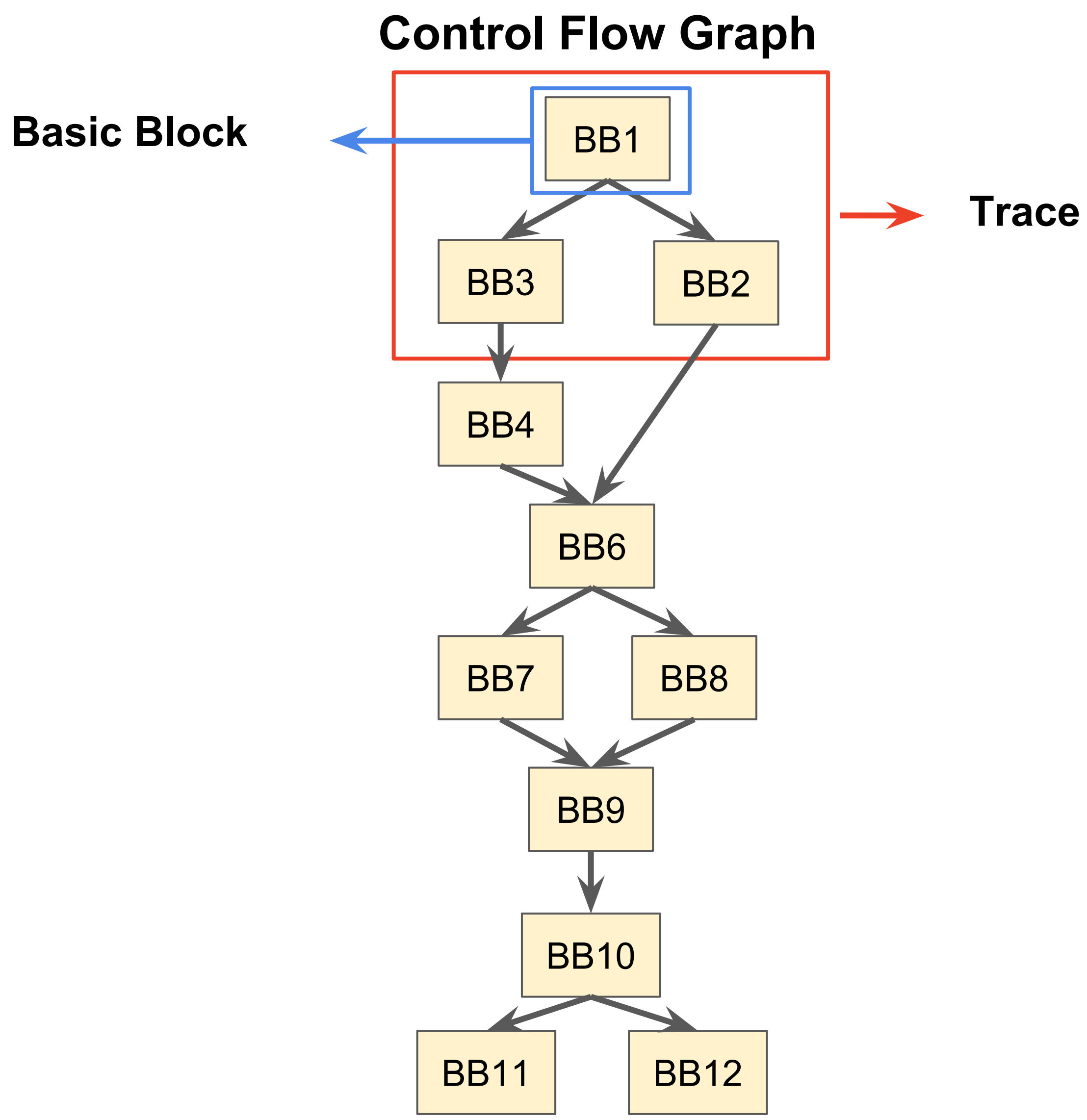
All hail

PinDemonium

Overview

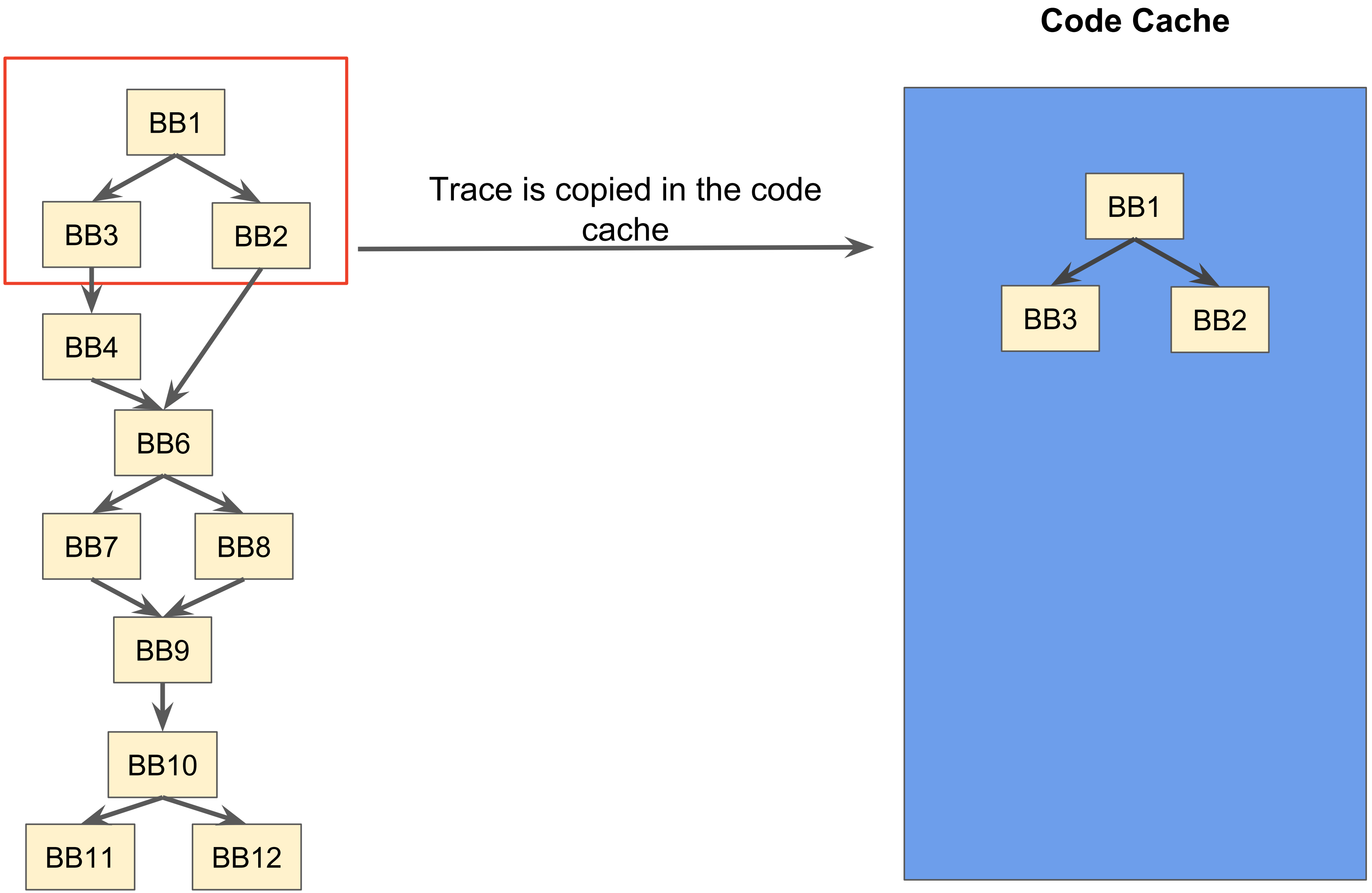
PinDemonium is a **generic unpacker** based on Intel PIN, a **dynamic binary instrumentation framework (DBI)**

What is a DBI?

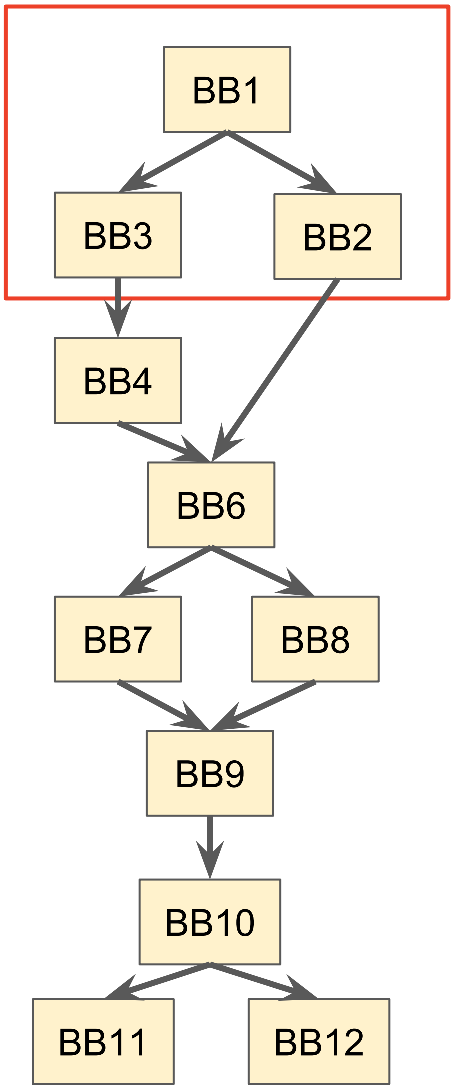


Control Flow Graph
Basic Block
Trace
BB1
BB3
BB2
BB4
BB6
BB7
BB8
BB9
BB10
BB11
BB12

What is a DBI?


Code Cache
BB1
BB3
BB2
BB4
BB6
BB7
BB8
BB9
BB10
BB11
BB12
Trace is copied in the code cache
BB1
BB3
BB2

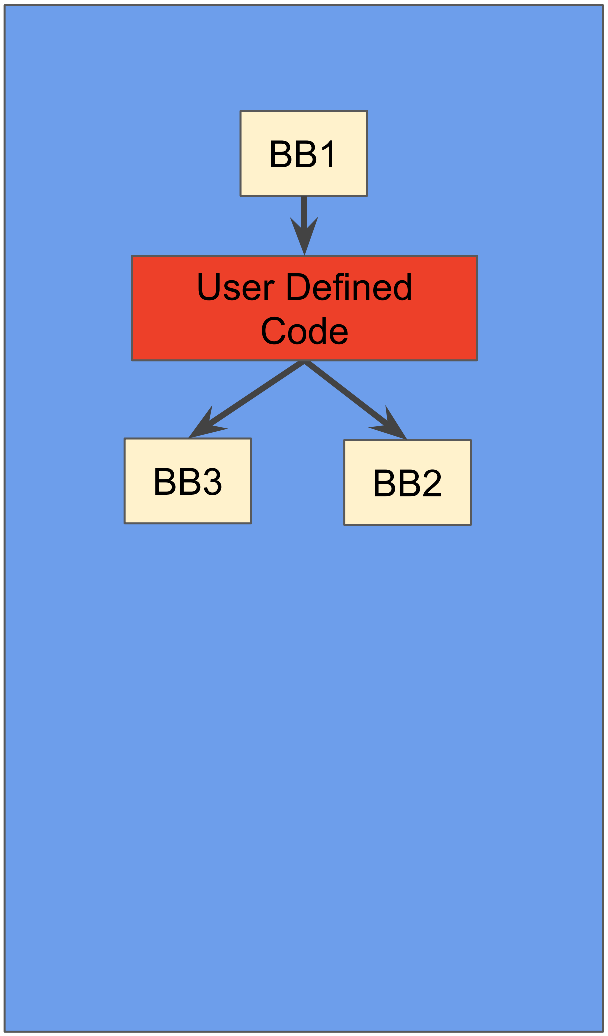
What is a DBI?



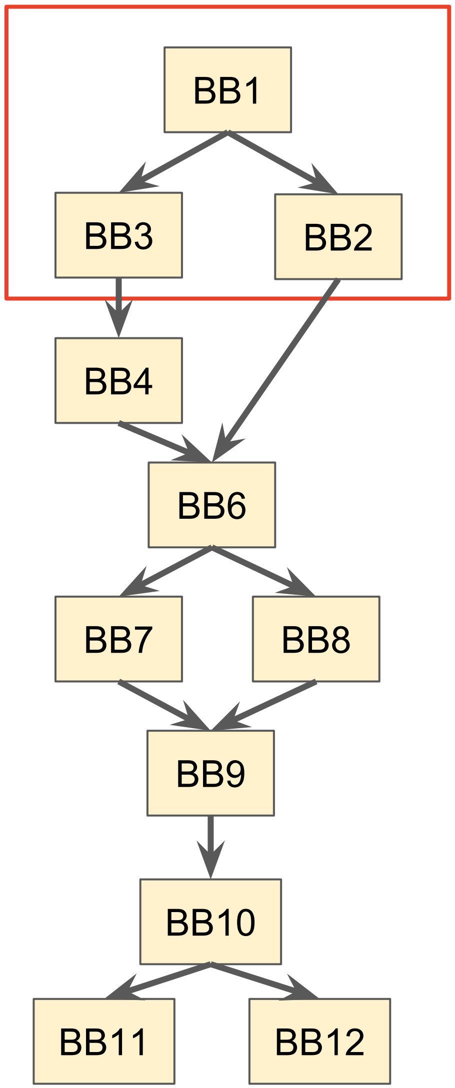
DBI provides the possibility to add user defined code after each:

- Instruction
- Basic Block
- Trace

Code Cache

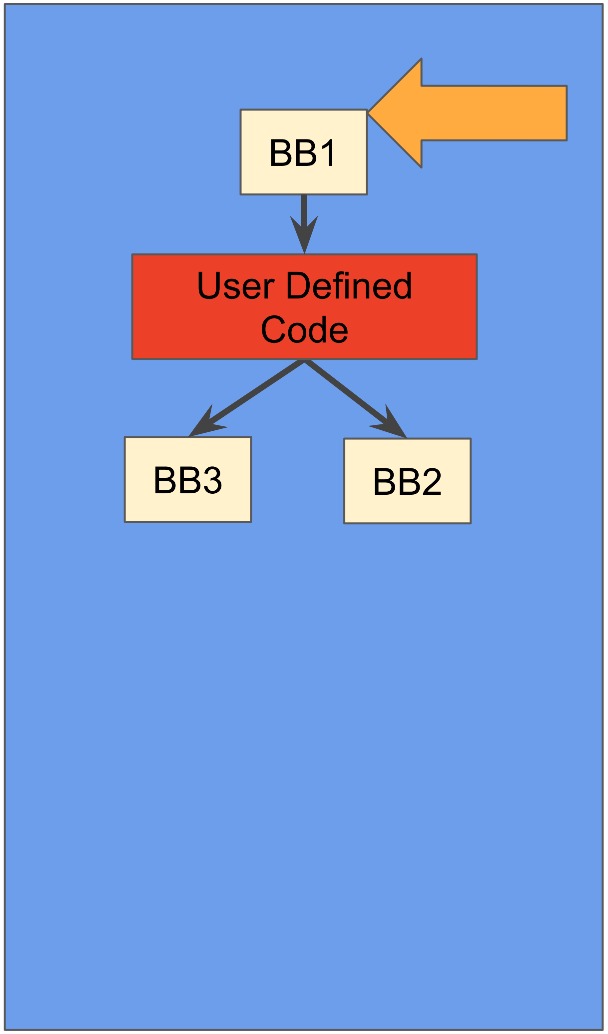


What is a DBI?



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

Original malware

Our journey begins

We begin to build the foundation of our system

Detect WxorX memory regions

black hat USA 2016

Let's exploit the key idea behind a generic unpacker implementing the WxorX handler module

Concepts:

- **Write Interval (WI)**: range of continuously written addresses
- **WxorX law broken**: instruction written by the program itself and then executed

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 present in the write set. If the condition is met then the WxorX law is broken.

Detect WxorX memory regions

Current instr.

PinDemonium

Write set

0x401000	0x412000 - 0x413000	0x402000 - 0x403000	0x401000 - 0x402000

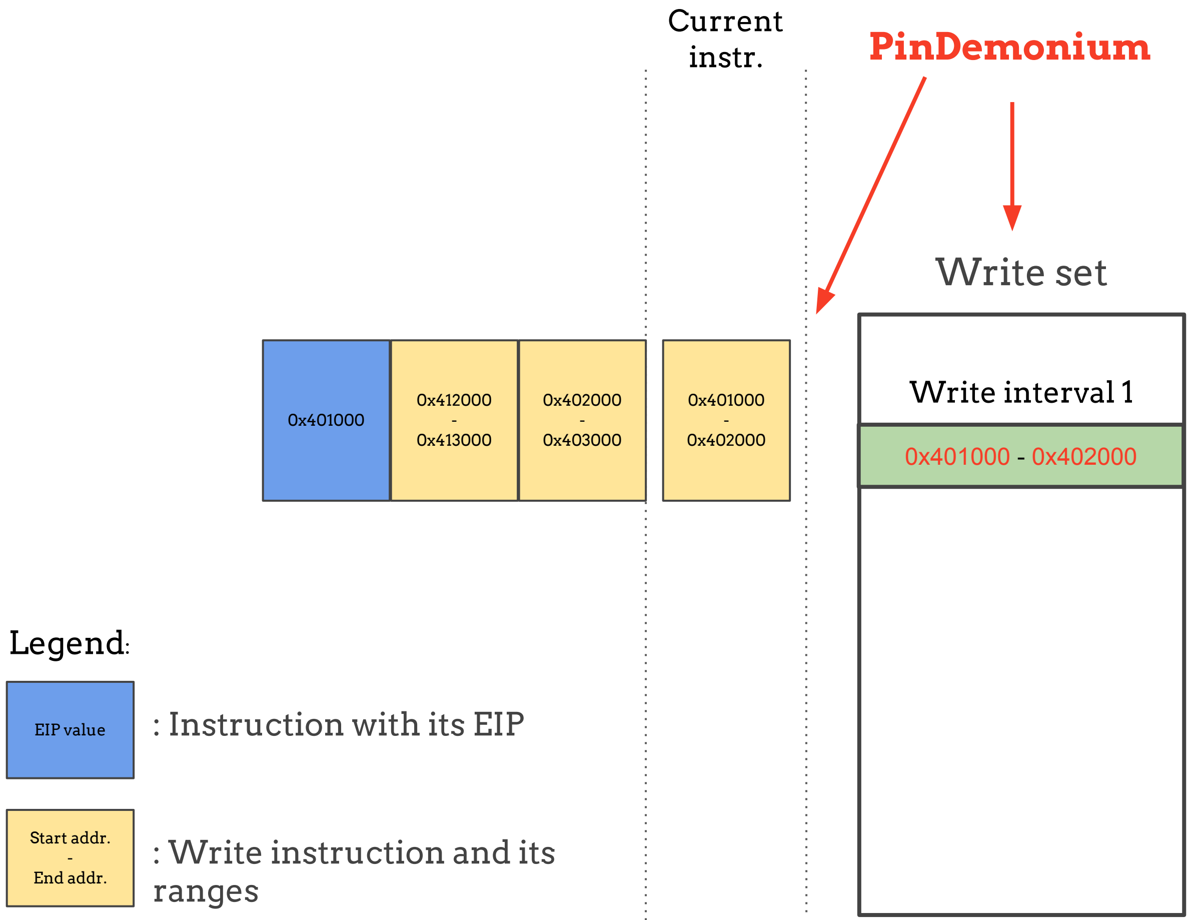
Legend:

EIP value : Instruction with its EIP

Start addr. - End addr. : Write instruction and its ranges

Steps:

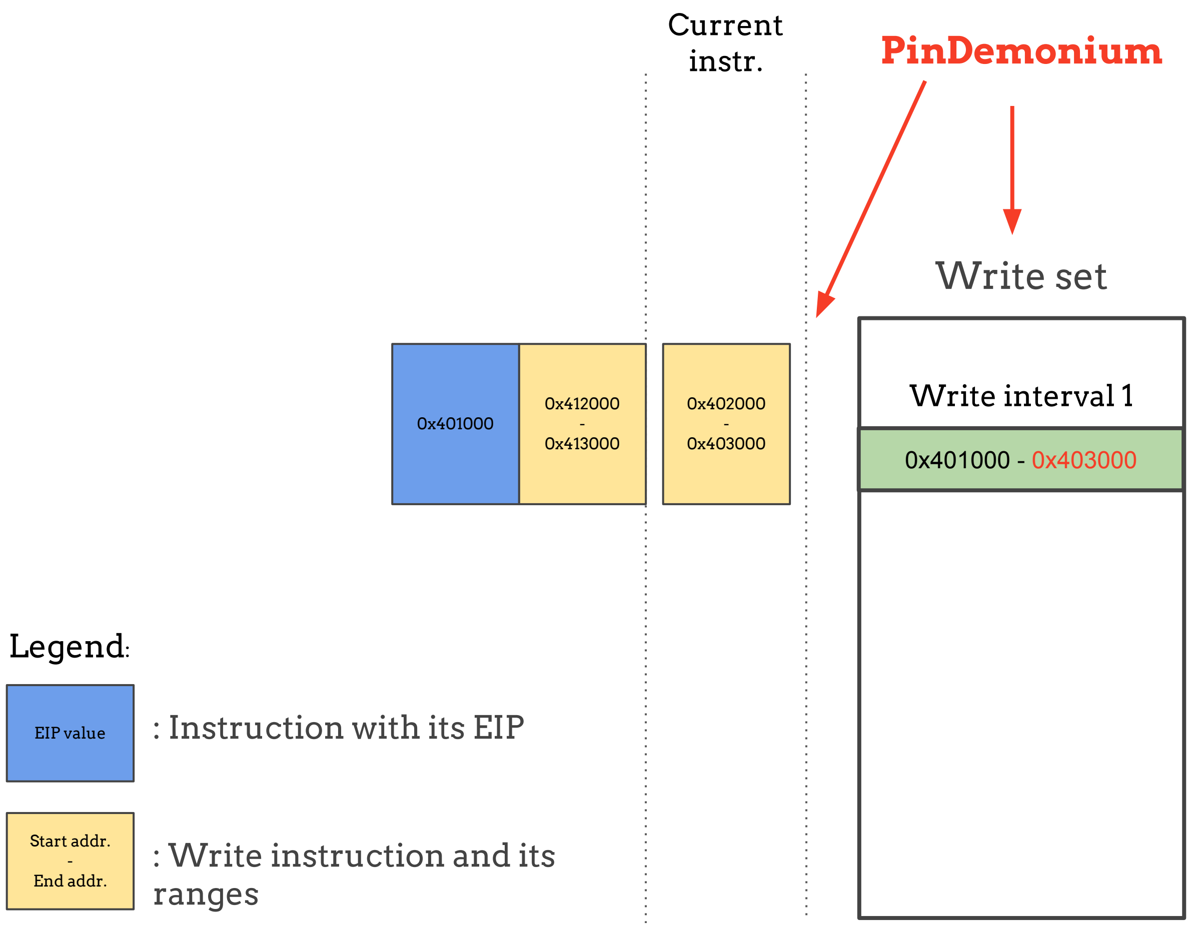
Detect WxorX memory regions



Steps:

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

Detect WxorX memory regions

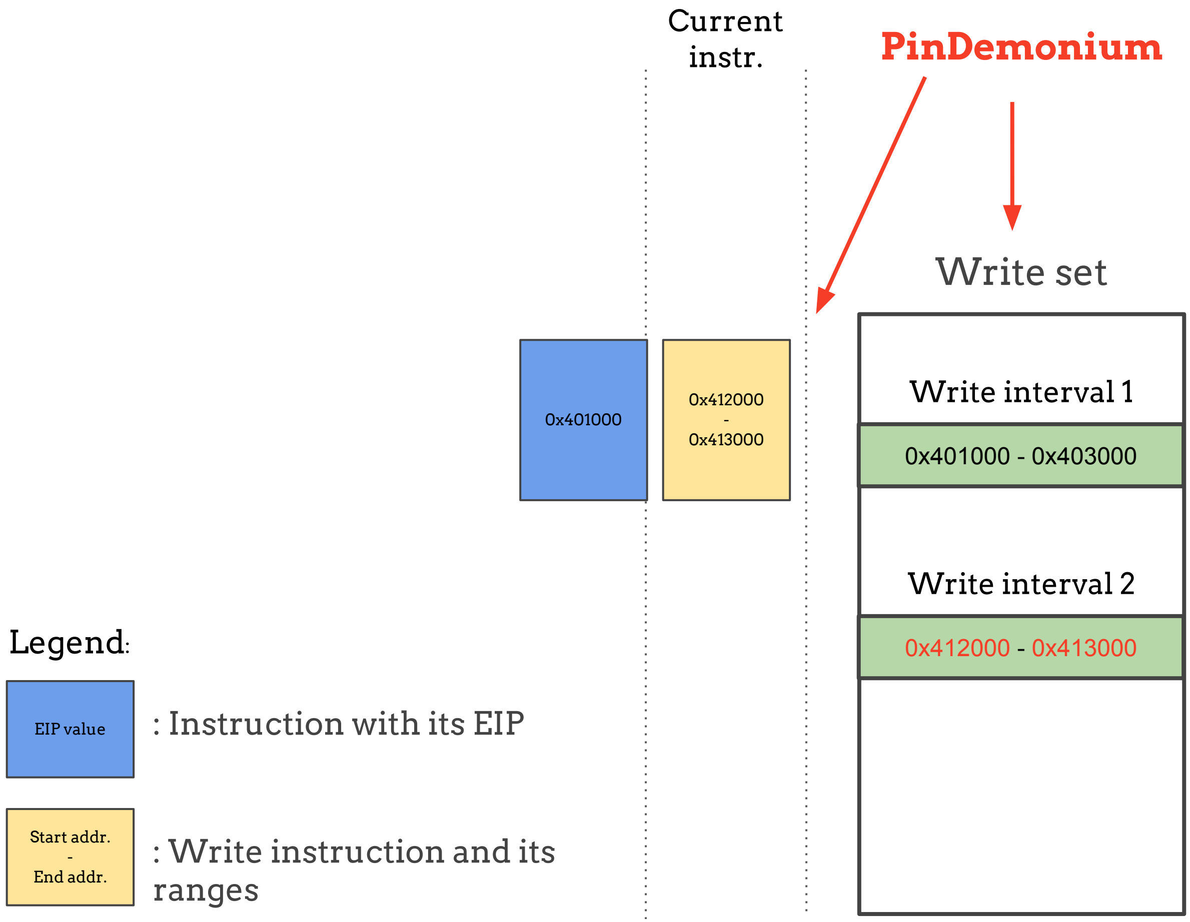


Steps:

1. The current instruction is a write, no WI present, create the new WI
2. The current instruction is a write, the ranges of the write overlaps an existing WI, update the matched WI

Detect WxorX memory regions

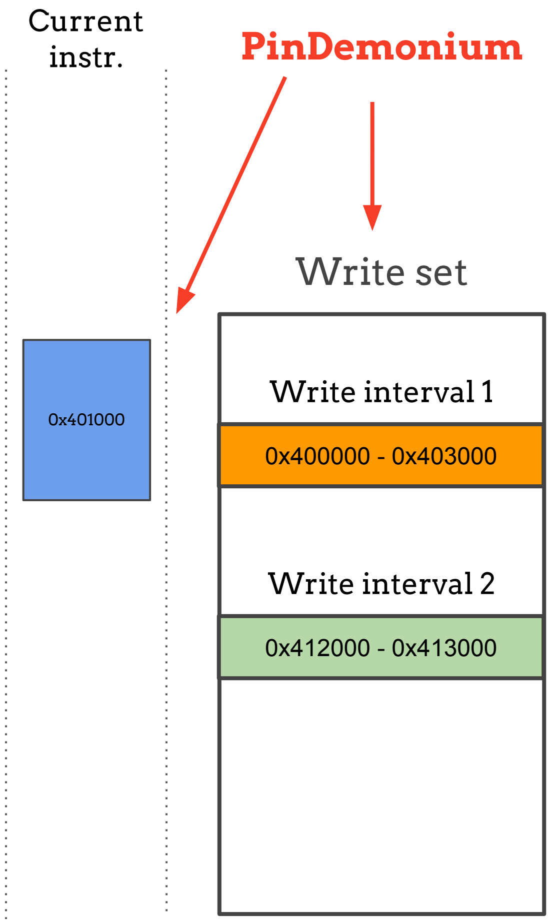
black hat USA 2016



Steps:

1. The current instruction is a write, no WI present, create the new WI
2. The current instruction is a write, the ranges of the write overlaps an existing WI, update the matched WI
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:

EIP value : Instruction with its EIP

Start addr. - End addr. : Write instruction and its ranges

Steps:

1. The current instruction is a write, no WI present, create the new WI
2. The current instruction is a write, the ranges of the write overlaps an existing WI, update the matched WI
3. The current instruction is a write, the ranges of the write don't overlap any WI, create a new WI
4. The EIP of the current instruction is inside a WI, WxorX law broken!

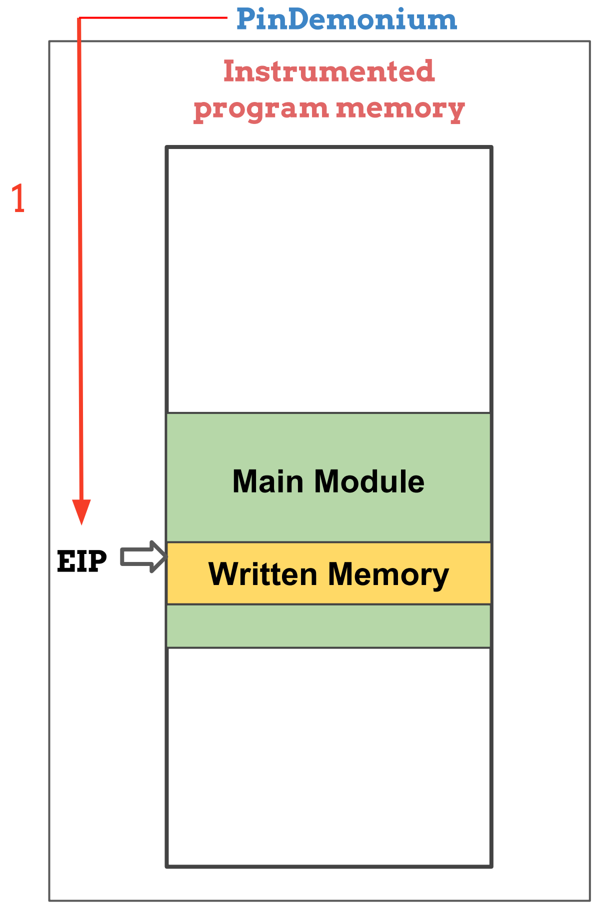
DUMP THE MEMORY!

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

In order to dump the program we have exploited the capabilities of our dumping module and Scylla

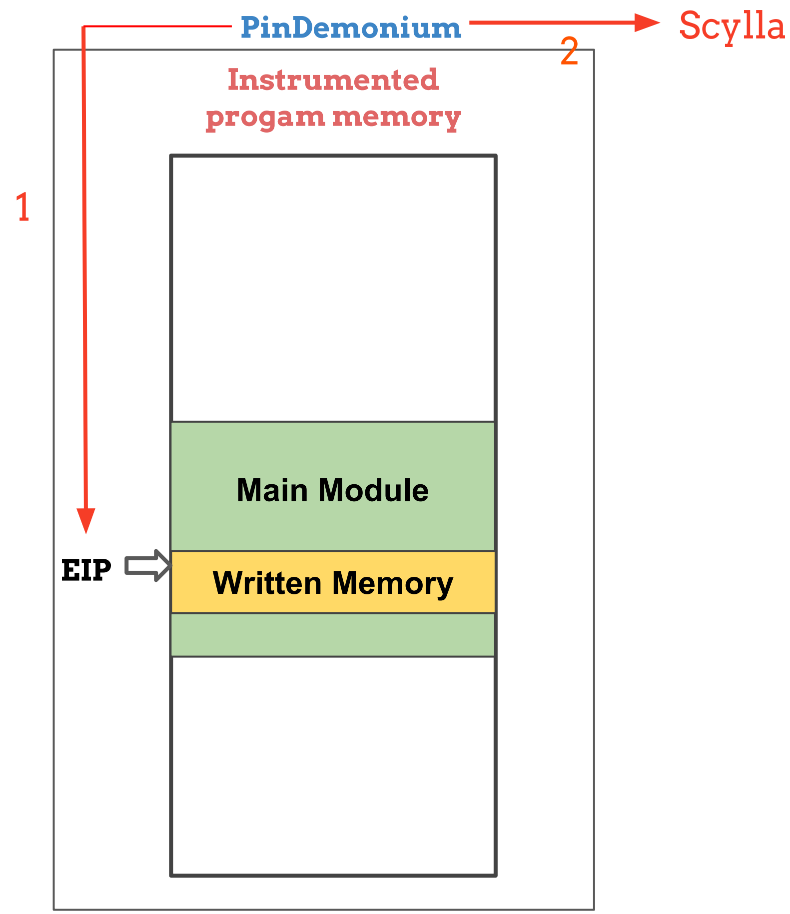


Steps:

1. The execution of a written address is detected

Dump the program correctly

In order to dump the program we have exploited the capabilities of our dumping module and Scylla

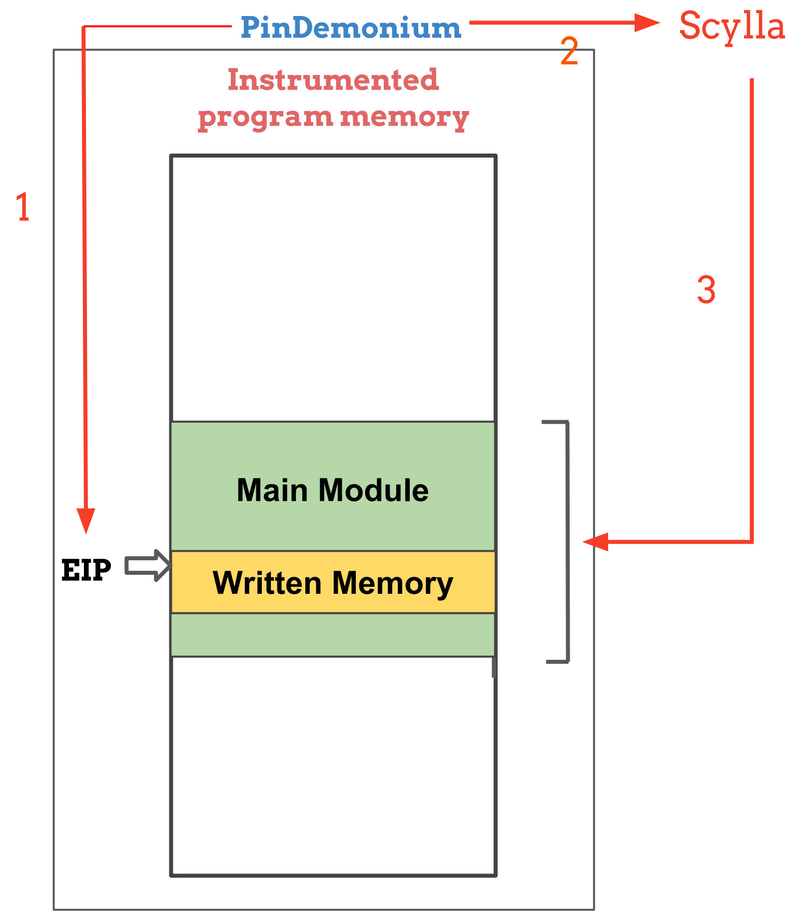


Steps:

1. The execution of a written address is detected
2. PinDemonium calls Scylla

Dump the program correctly

In order to dump the program we have exploited the capabilities of our dumping module and Scylla

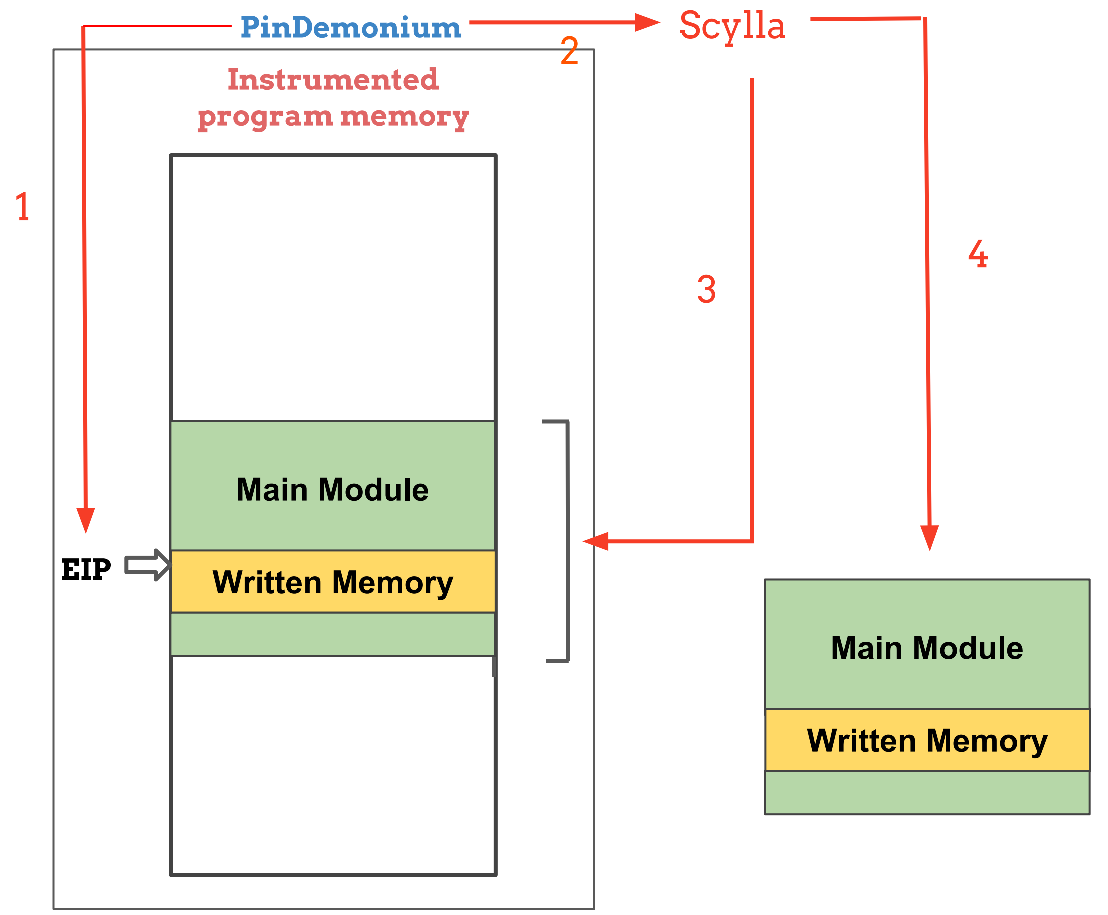


Steps:

1. The execution of a written address is detected
2. PinDemonium calls Scylla
3. Scylla gets the addresses of the main module

Dump the program correctly

In order to dump the program we have exploited the capabilities of our dumping module and Scylla



Steps:

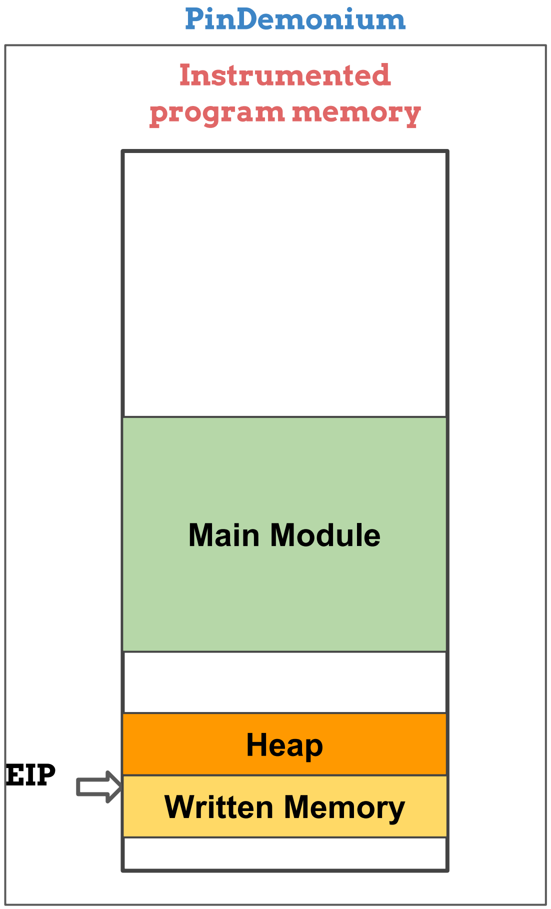
1. The execution of a written address is detected
2. PinDemonium calls Scylla
3. Scylla gets the addresses of the main module
4. Scylla dumps the main module along with the written addresses on a file

Have we already finished?

Nope...

Unpacking on the heap

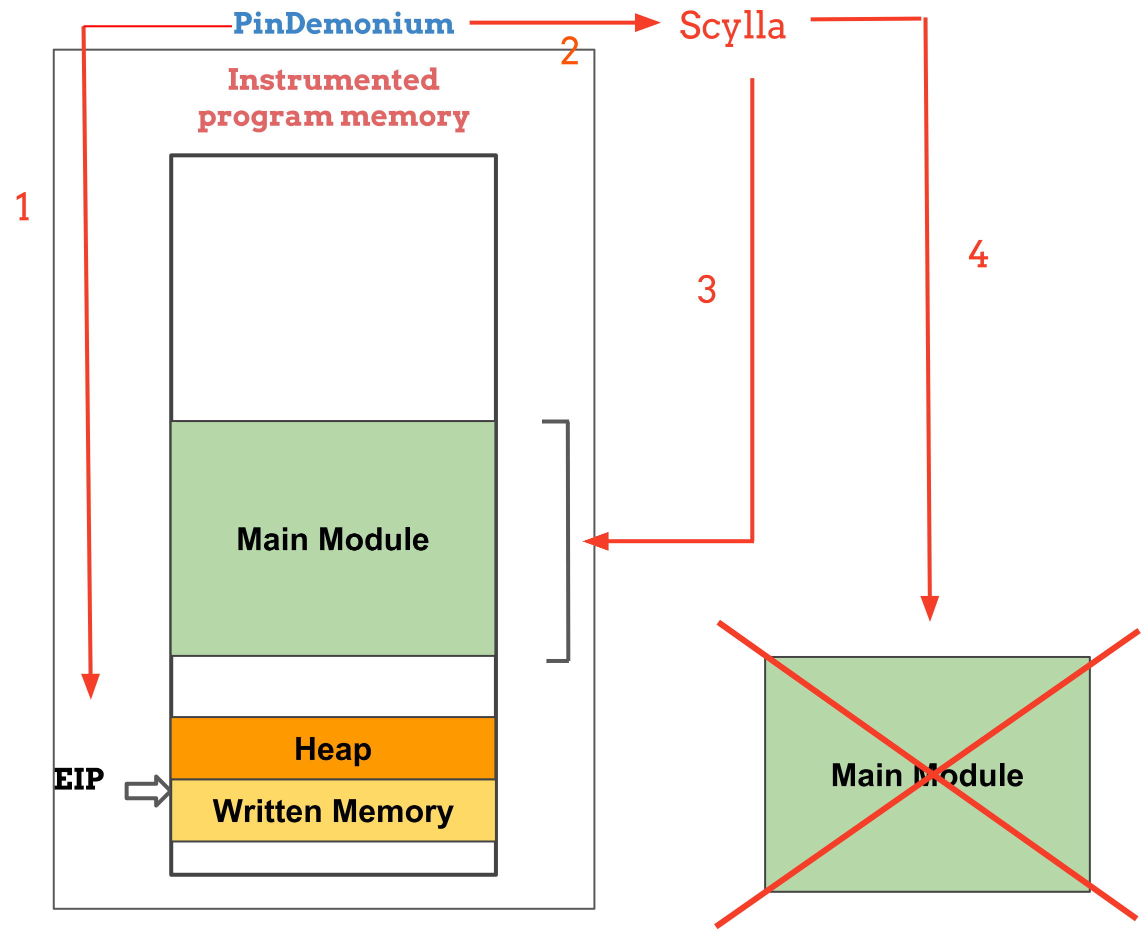
What if the original code is written on the heap?



Steps:

Unpacking on the heap

What if the original code is written on the heap?



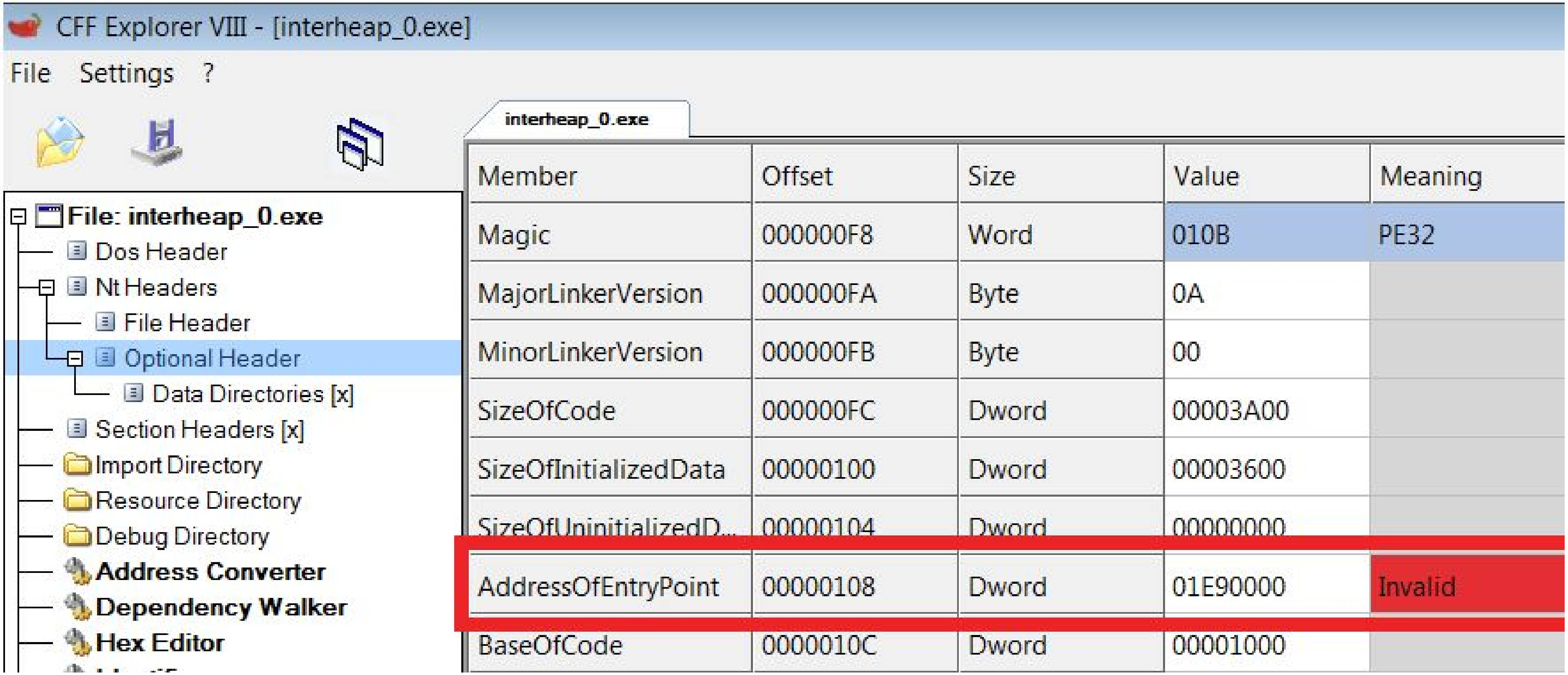
Steps:

1. The execution of a written address is detected
2. PinDemonium calls Scylla
3. Scylla gets the addresses of the main module
4. Scylla dumps the main module

WRONG!

Unpacking on the heap

The OEP doesn’t make sense!



Member	Offset	Size	Value	Meaning
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
BaseOfCode	0000010C	Dword	00001000	

Unpacking on the heap

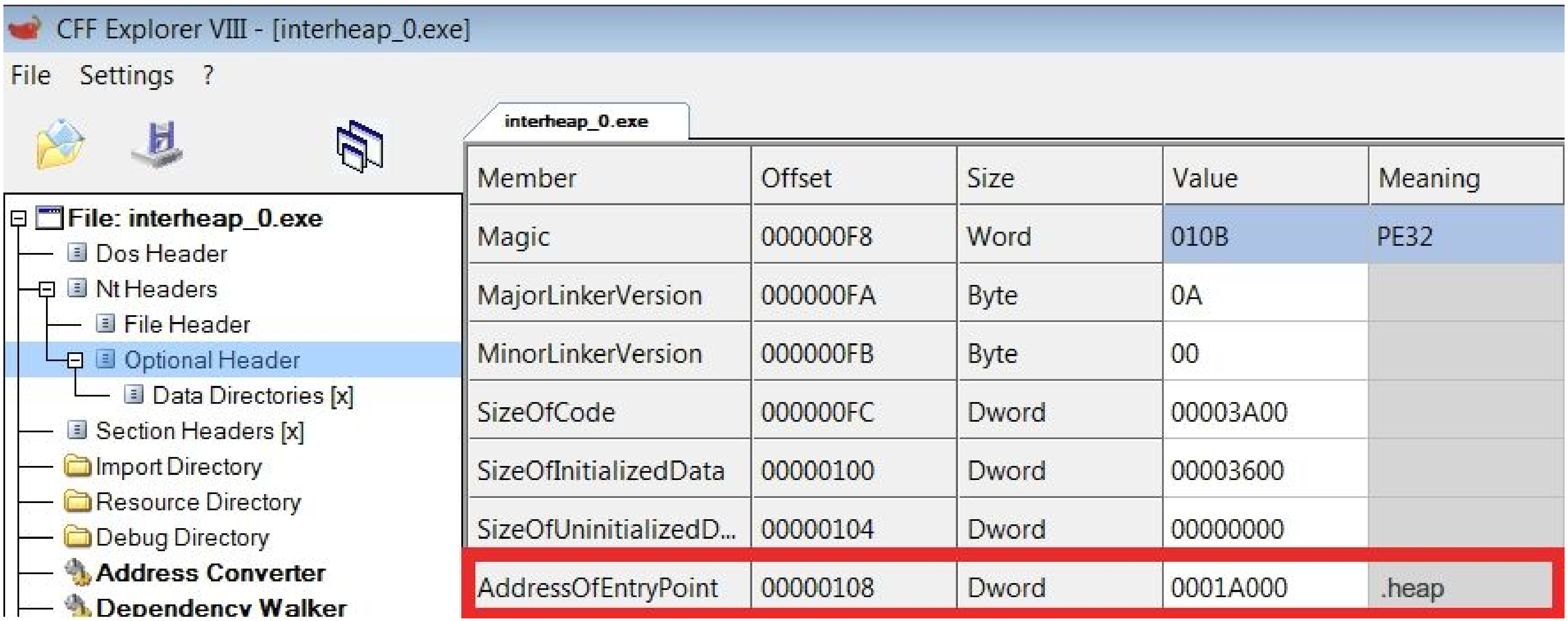
Solution

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

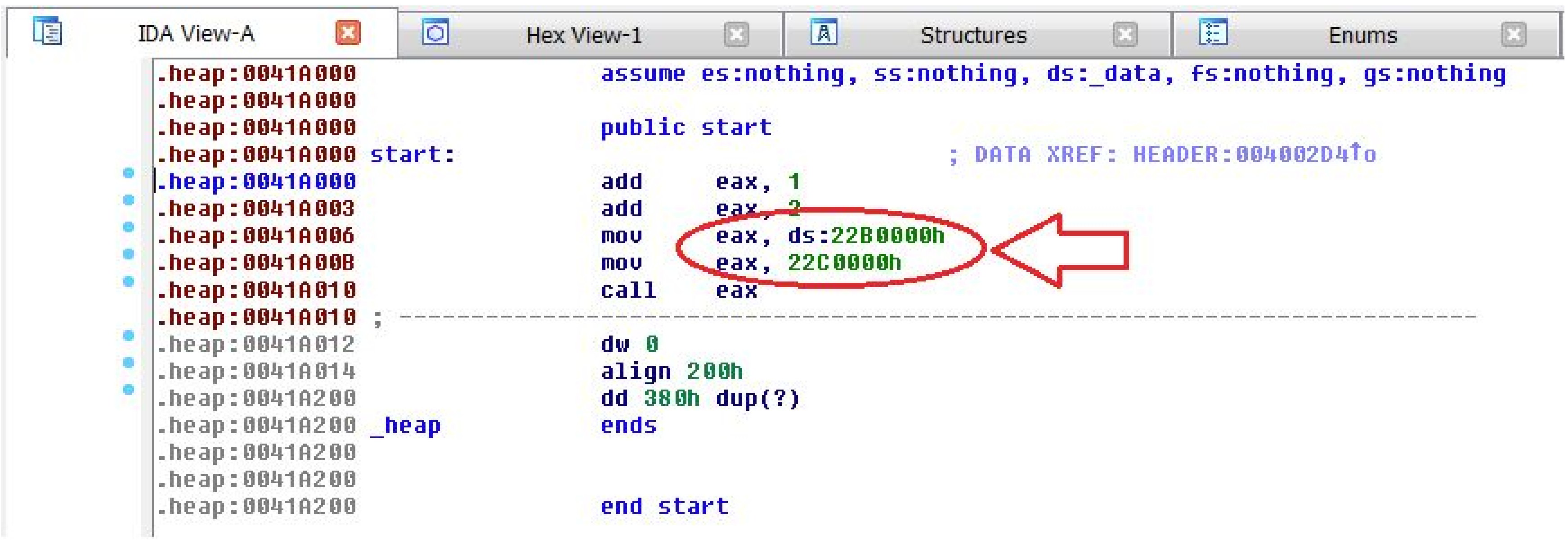
Unpacking on the heap

The OEP is correct!



Unpacking on the heap

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



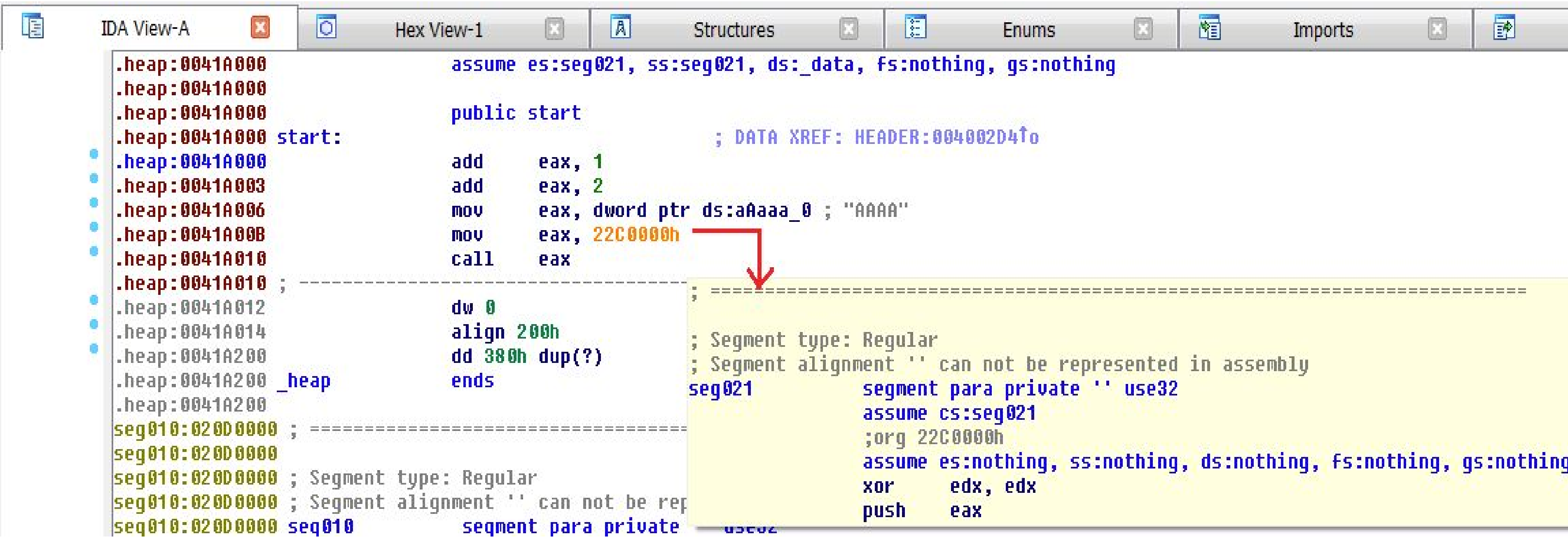
Unpacking on the heap

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. Identify the new allocated or modified ones by comparing the MD5 of their previous content
3. Dump these heap-zones
4. Create new segments inside the .idb for each of them
5. Copy the heap-zones content inside these new segments!

Unpacking on the heap



Two down, two still standing!

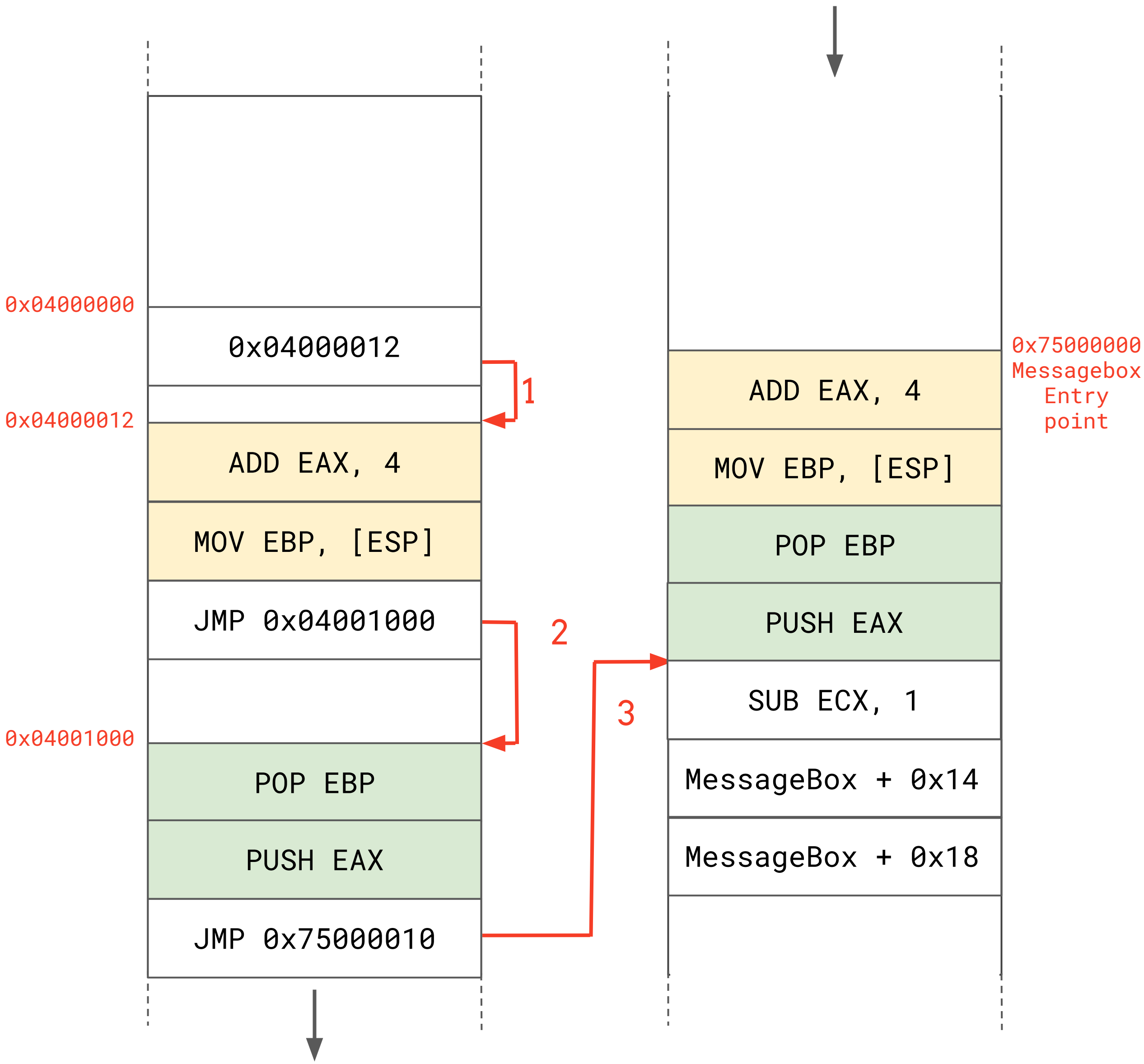
Reverser we are coming for you! Let's **deobfuscate some imported functions...**

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

Deobfuscate the IAT



Steps:

1. Is the address 0x04000012 inside the DLL memory region? No, continue until next jump...
   `ins_delta = 0`
2. Is the address 0x04001000 inside the DLL memory region? No, continue until next jump...
   `ins_delta = 8`
3. Is the address 0x75000010 inside the DLL memory region? YES! Let's patch the IAT entry
   `ins_delta = 16`

[0x4000012] = (0x75000010 - 16) = 0x75000000

CORRECT!

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

Detect patterns of evasive code which may have prevented the complete unpacking of the malware like Anti-VM and Anti-Debug techniques

Identifying malware family

When a known malware family rule is matched after multiple unpacking layers probably this is the correct dump

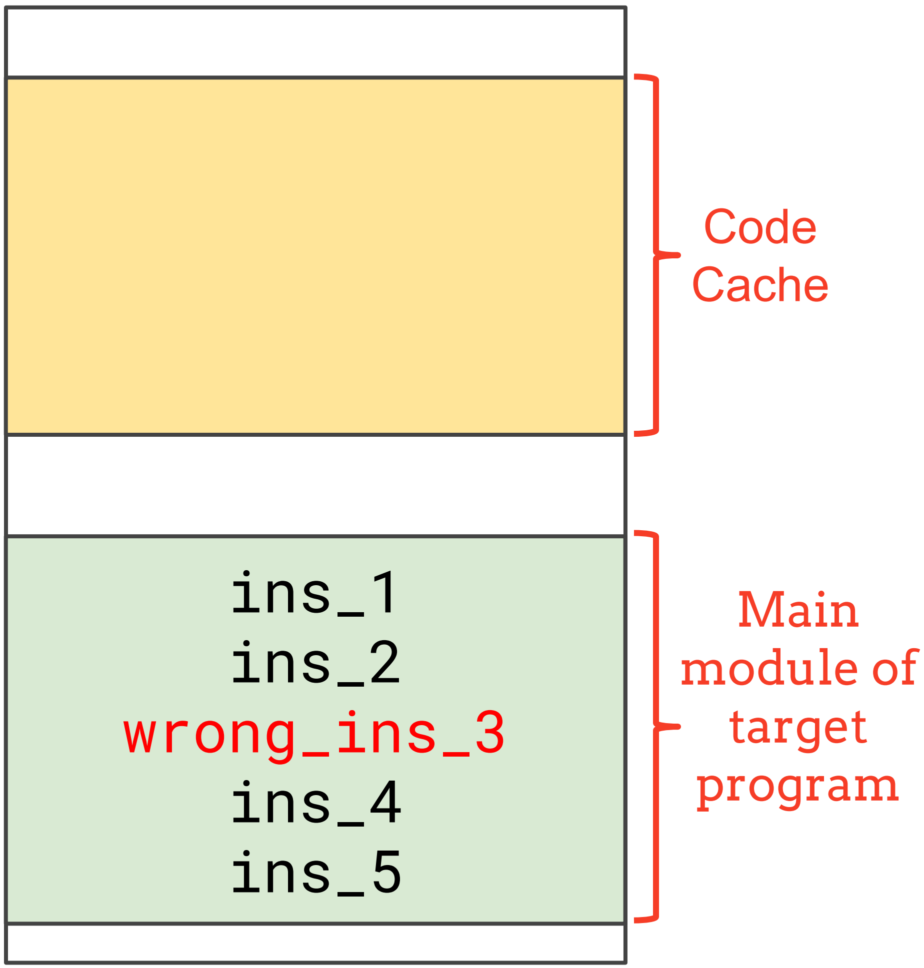
Advanced Problems

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

Exploit PIN functionality to break PIN

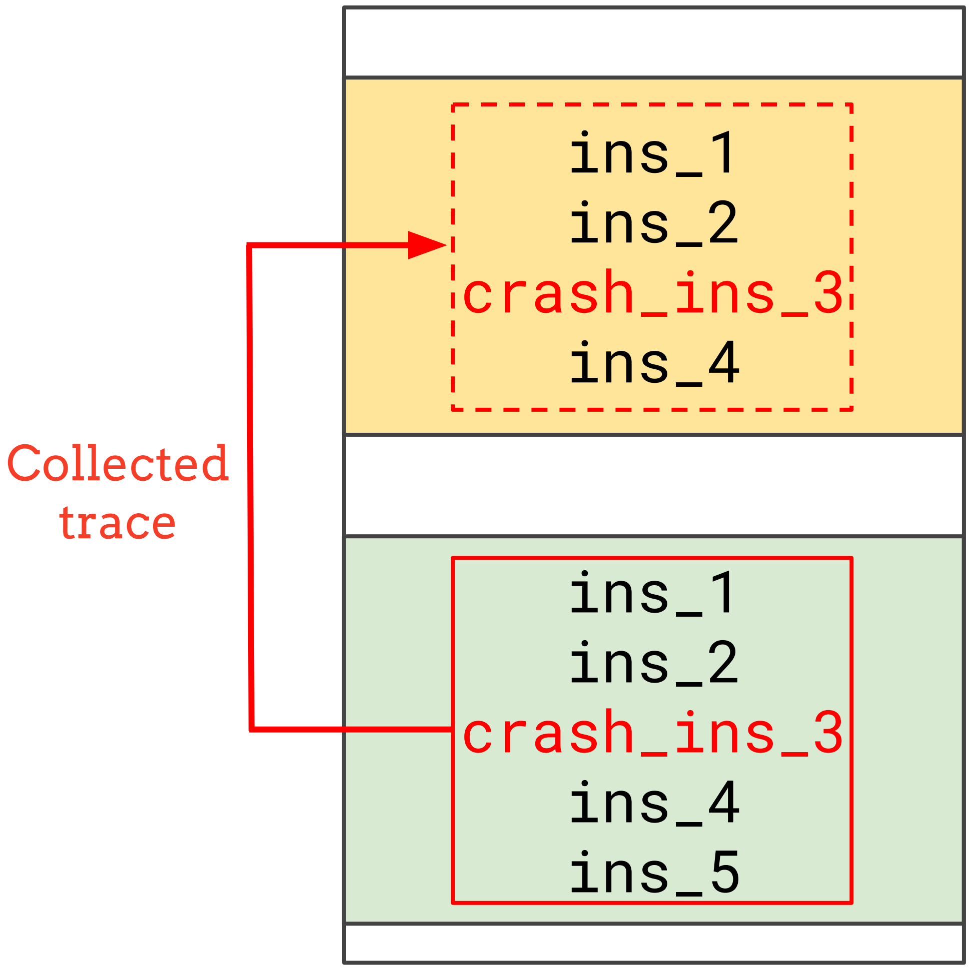
A.k.a. Self modifying code

Self modifying code



Steps:

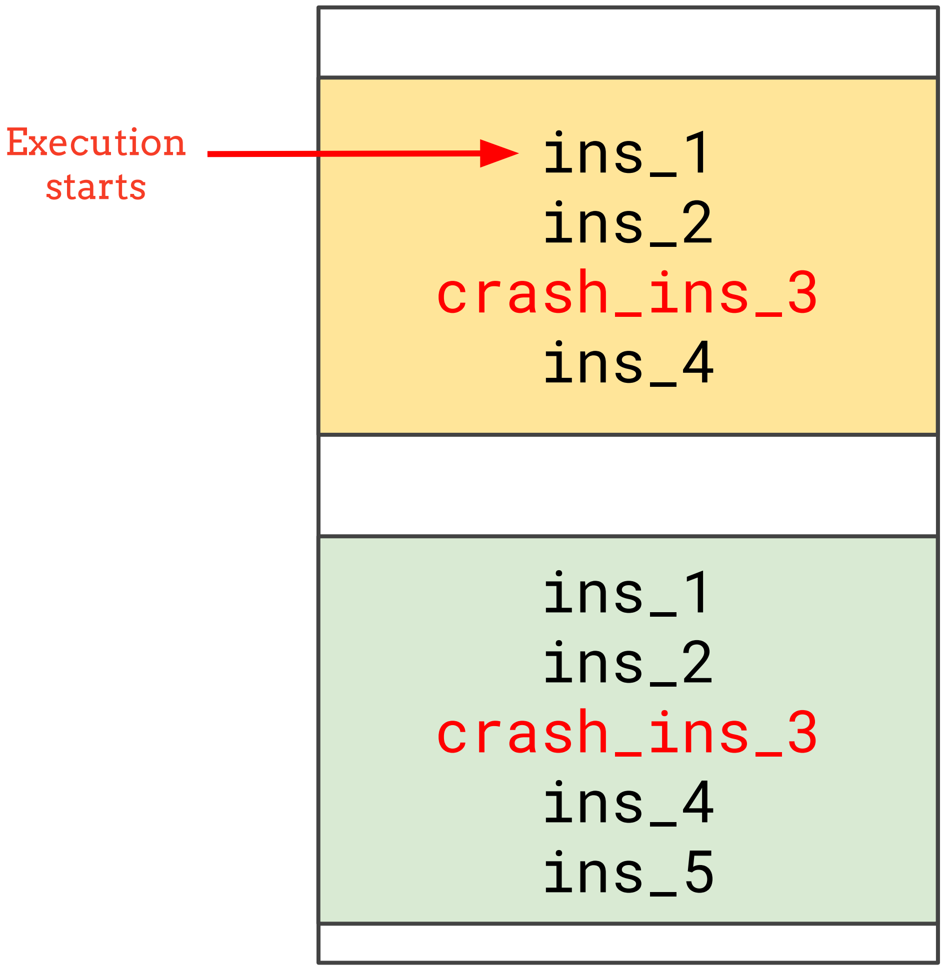
Self modifying code



Steps:

1. The trace is collected in the code cache

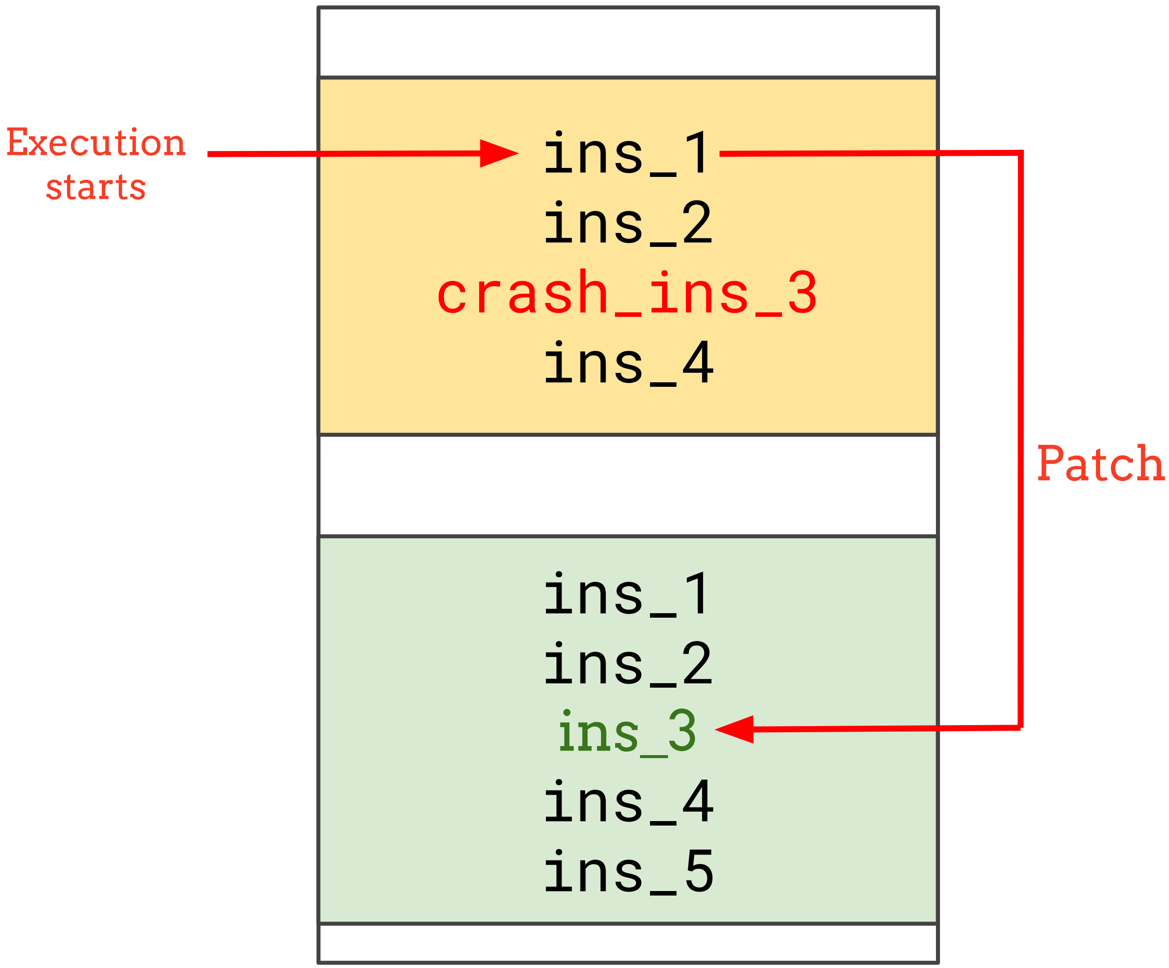
Self modifying code



Steps:

1. The trace is collected in the code cache
2. The execution starts in the code cache

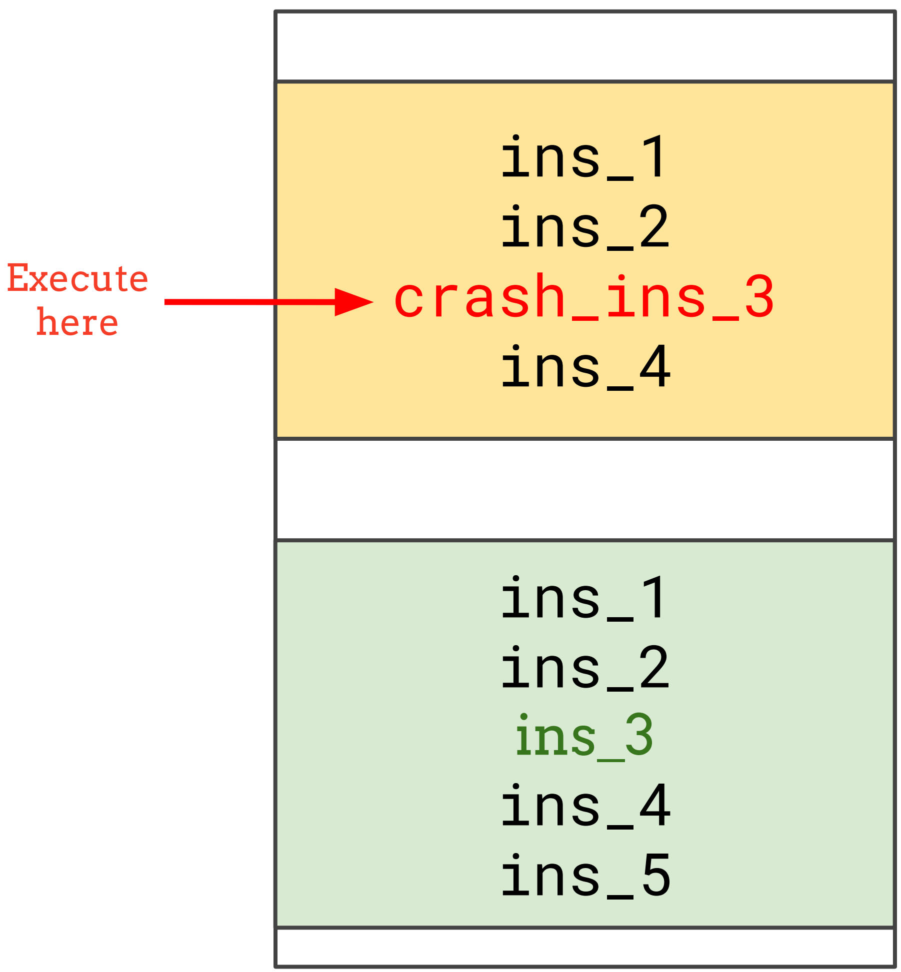
Self modifying code



Steps:

1. The trace is collected in the code cache
2. The execution starts in the code cache
3. The wrong instruction is patched in the main module

Self modifying code



Steps:

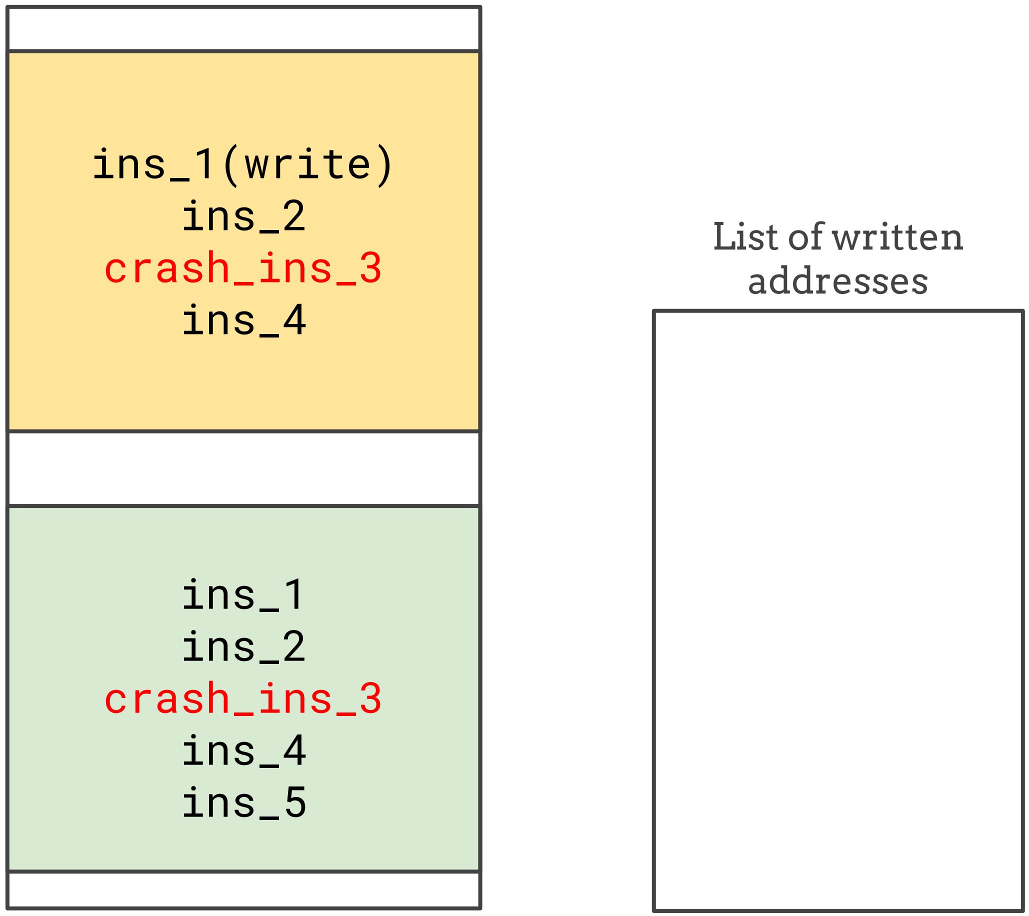
1. The trace is collected in the code cache
2. The execution starts in the code cache
3. The wrong instruction is patched in the main module
4. The `wrong_ins_3` is executed

CRASH!

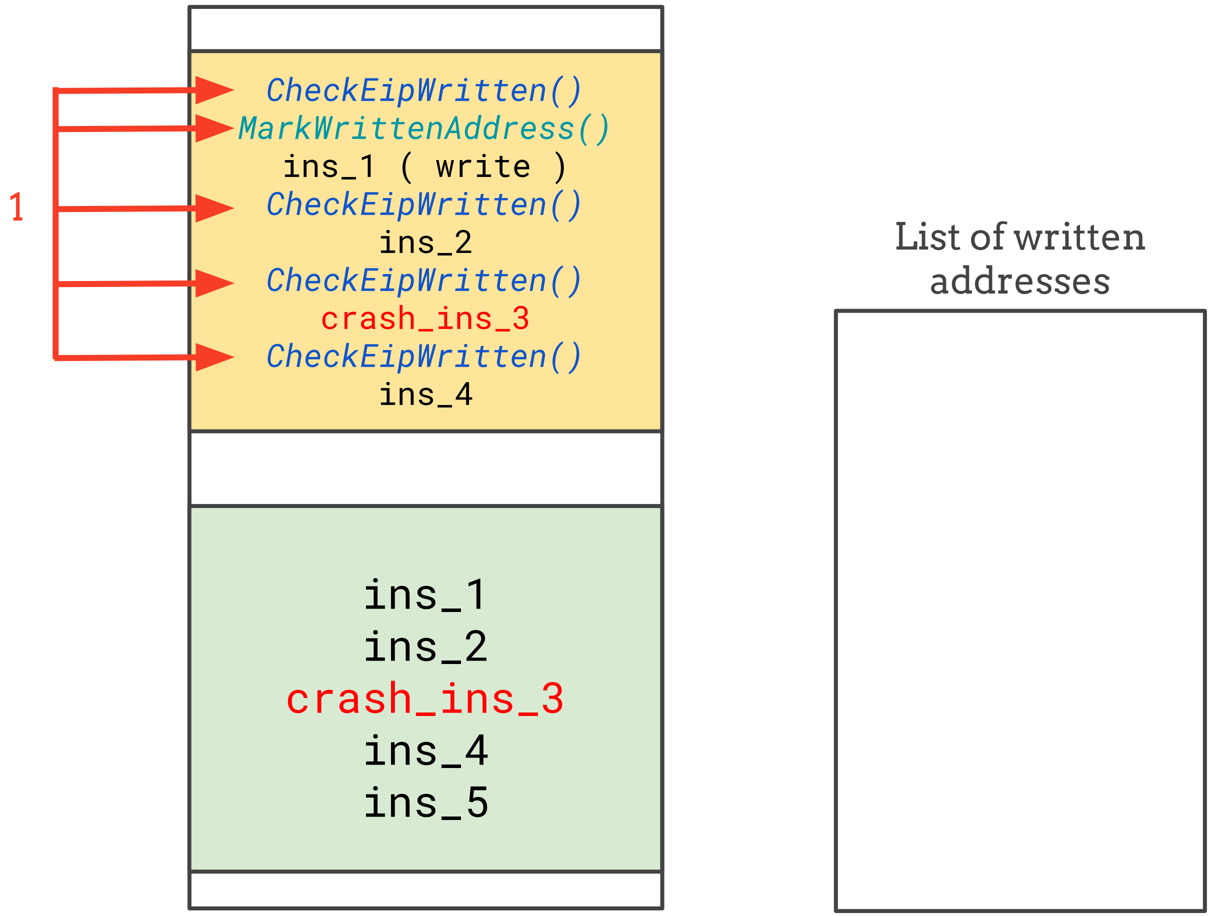
Solution

Self modifying code

Steps:



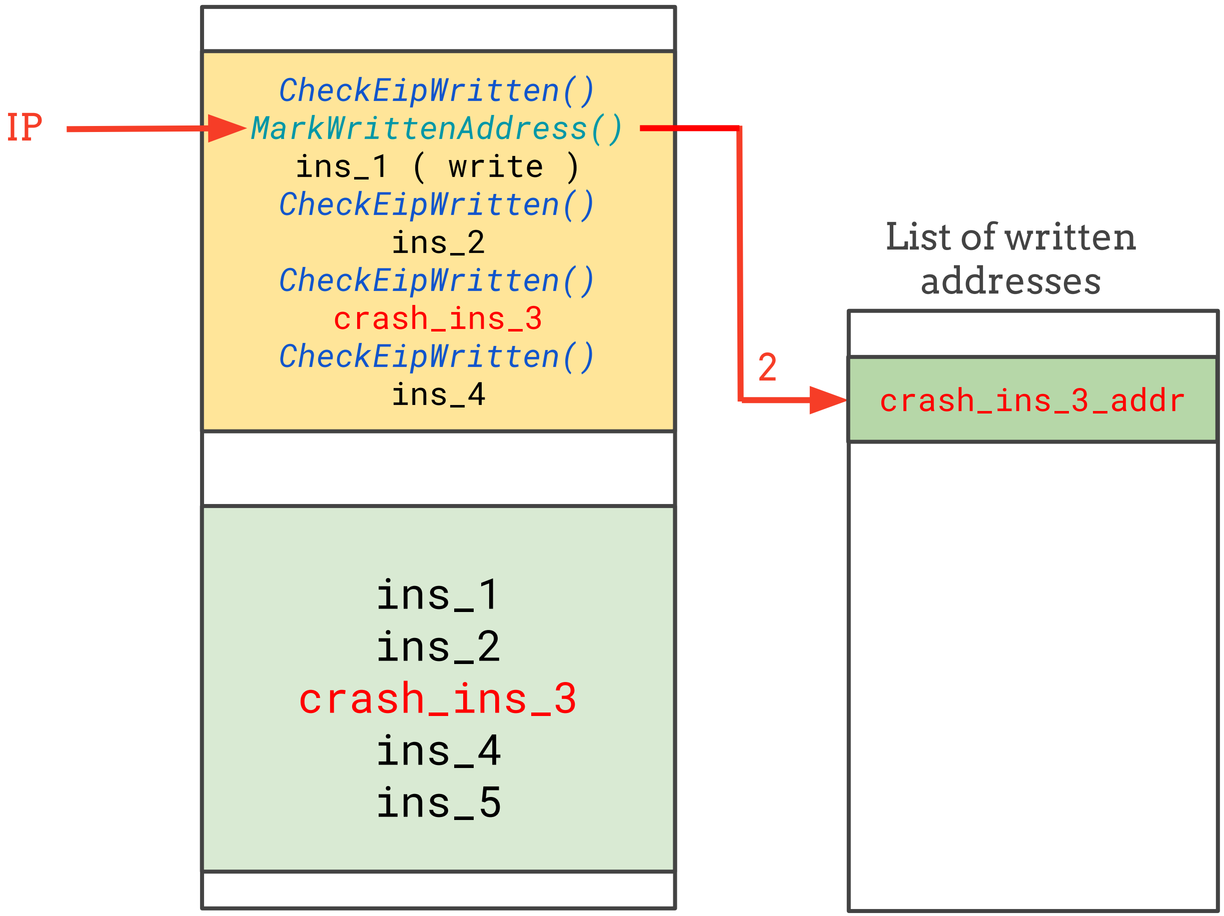
Self modifying code



Steps:

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

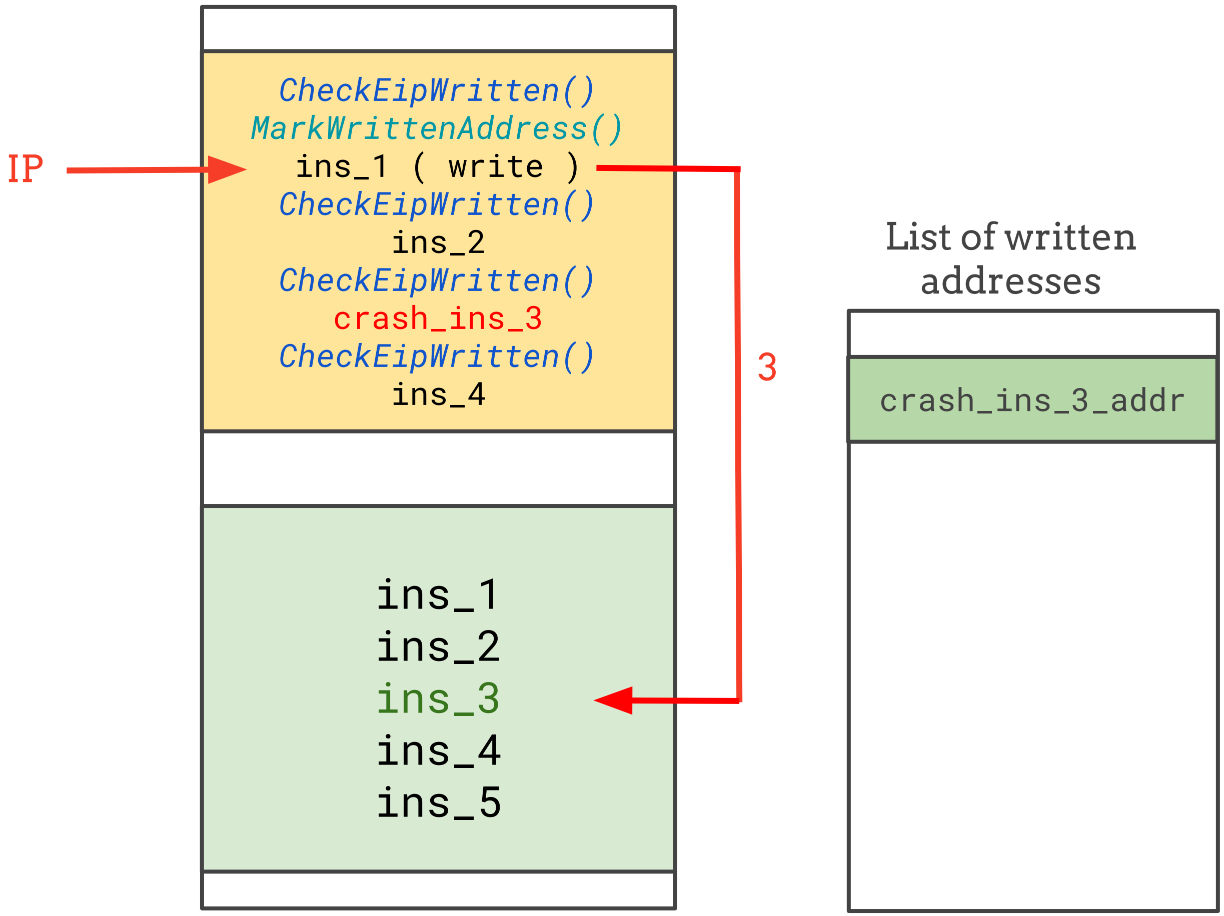
Self modifying code



Steps:

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

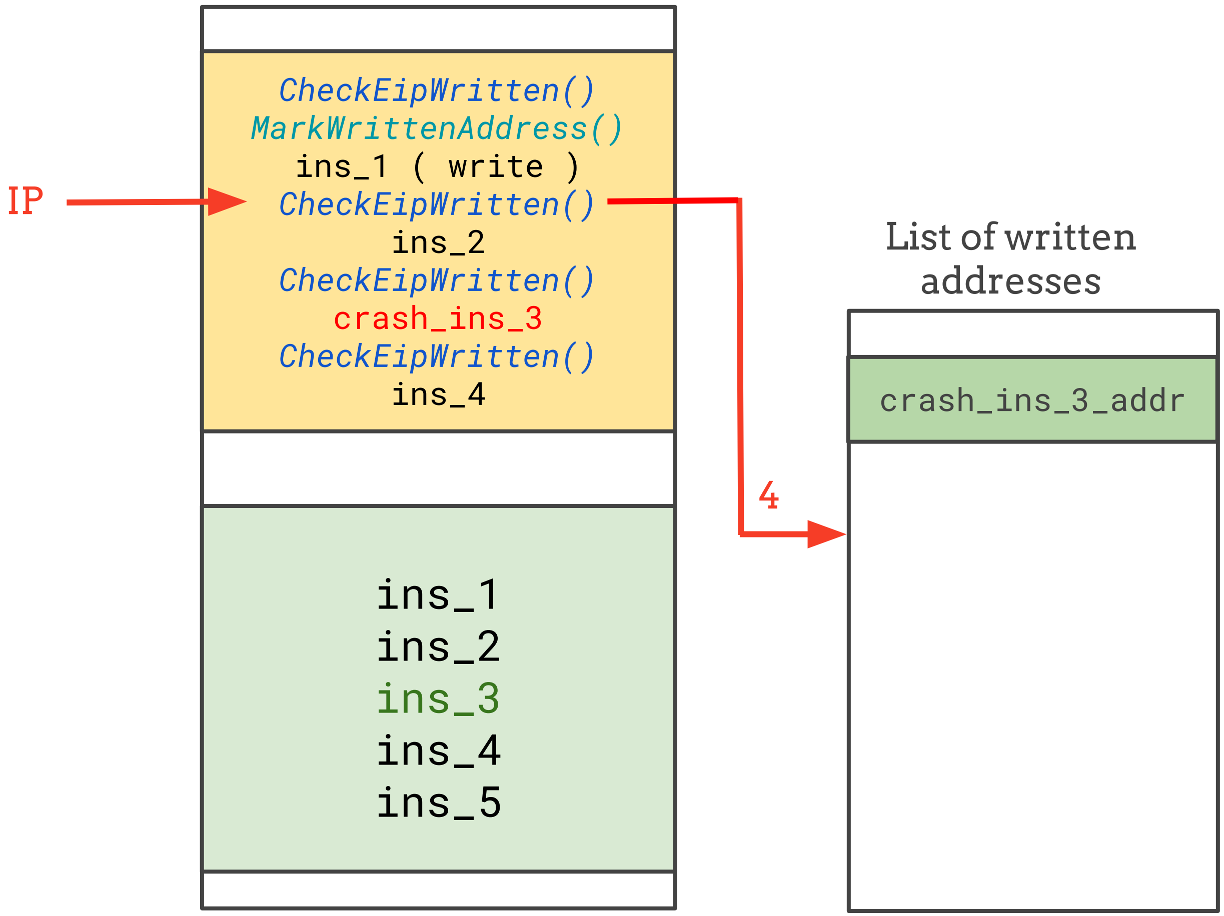
Self modifying code



Steps:

1. Insert one analysis routine before each instruction and another one if the instruction is a write
2. Execute the analysis routine before the write
3. The crash_ins_3 is patched in the main module

Self modifying code

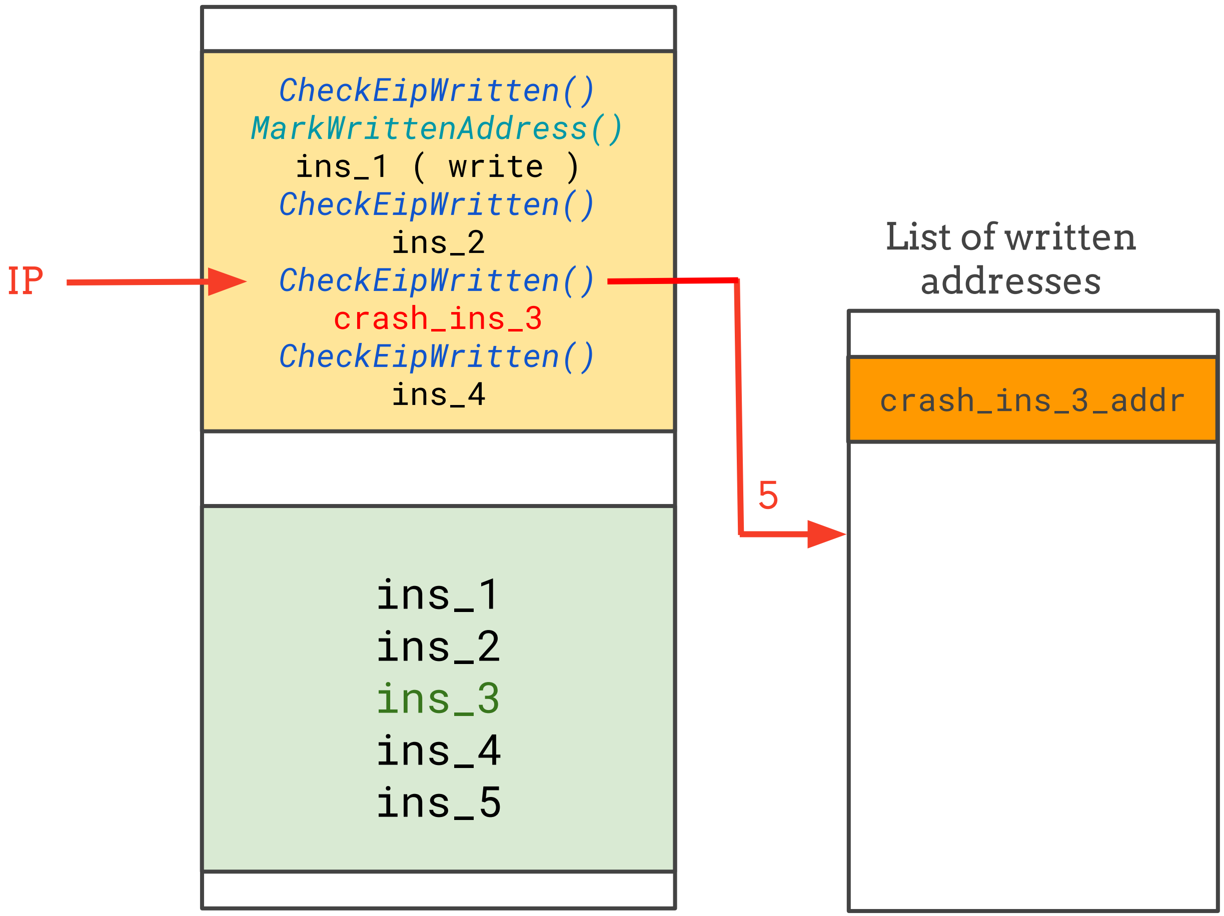


Steps:

1. Insert one analysis routine before each instruction and another one if the instruction is a write
2. Execute the analysis routine before the write
3. The crash_ins_3 is patched in the main module
4. Check if ins_2 address is inside the list

NOPE...

Self modifying code



Steps:

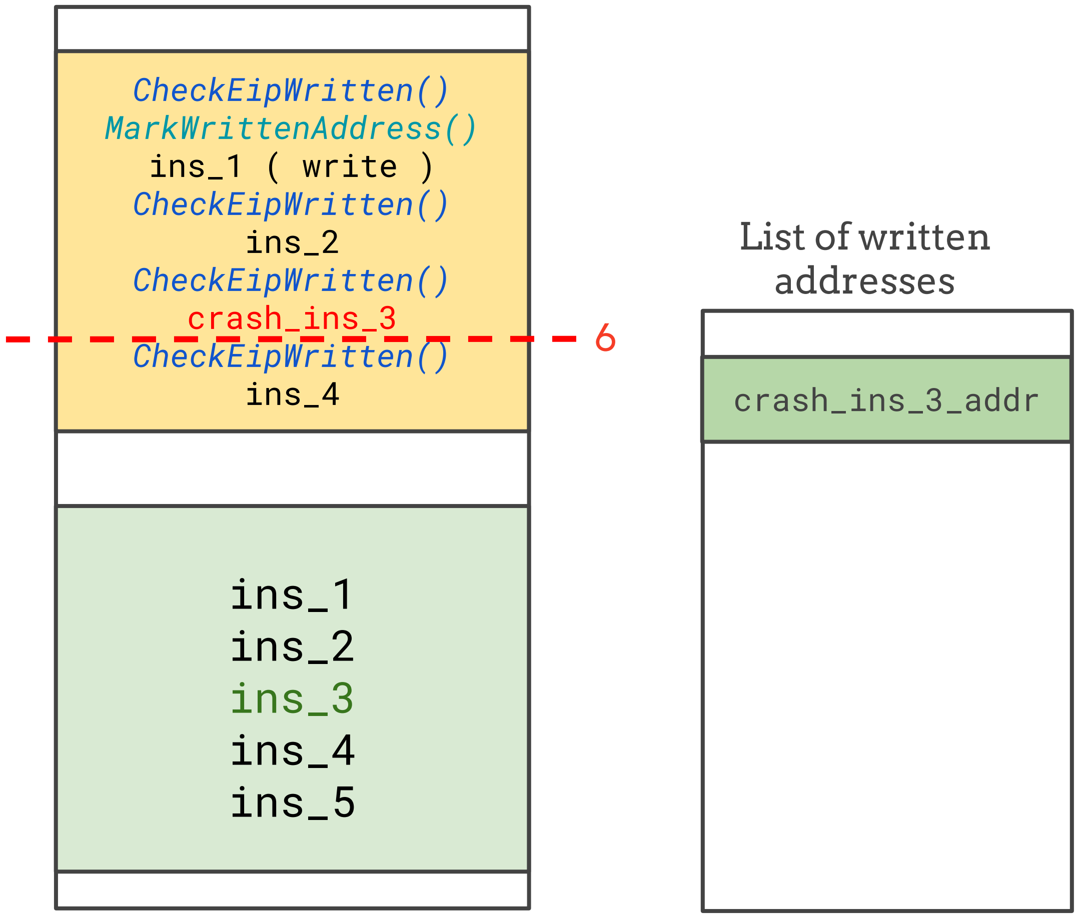
1. Insert one analysis routine before each instruction and another one if the instruction is a write
2. Execute the analysis routine before the write
3. The crash_ins_3 is patched in the main module
4. Check if ins_2 address is inside the list

   NOPE...

5. Check if crash_ins_3 address is inside the list

   YES!

Self modifying code



Steps:

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

2. Execute the analysis routine before the write

3. The crash_ins_3 is patched in the main module

4. Check if ins_2 address is inside the list

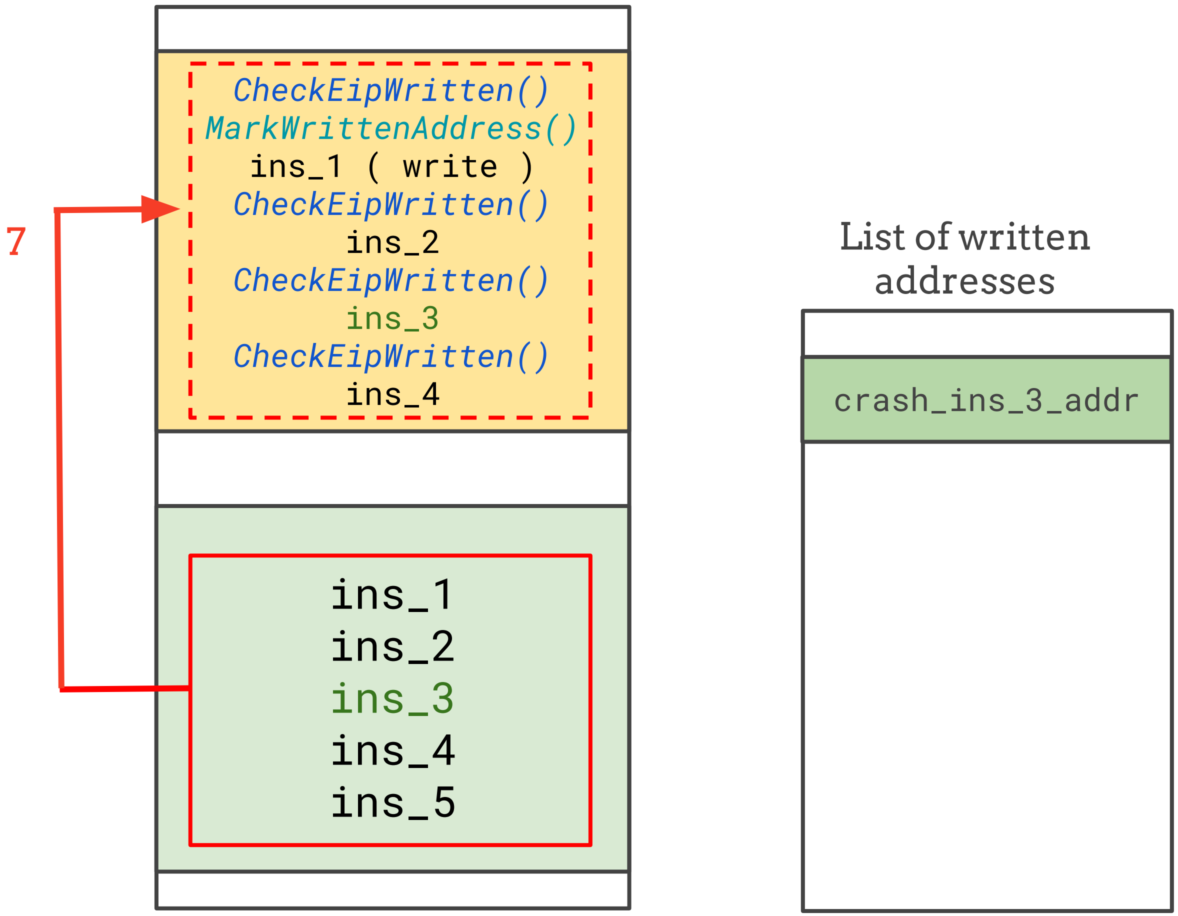
   NOPE...

5. Check if crash_ins_3 address is inside the list

   YES!

6. Stop the execution

Self modifying code



Steps:

1. Insert one analysis routine before each instruction and another one if the instruction is a write
2. Execute the analysis routine before the write
3. The crash_ins_3 is patched in the main module
4. Check if ins_2 address is inside the list

   NOPE...

5. Check if crash_ins_3 address is inside the list

   YES!

6. Stop the execution
7. Recollect the new trace

   CORRECT!

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**

OUR WxorX TRACKER IS NO MORE SUFFICIENT!

Solution

Process Injection

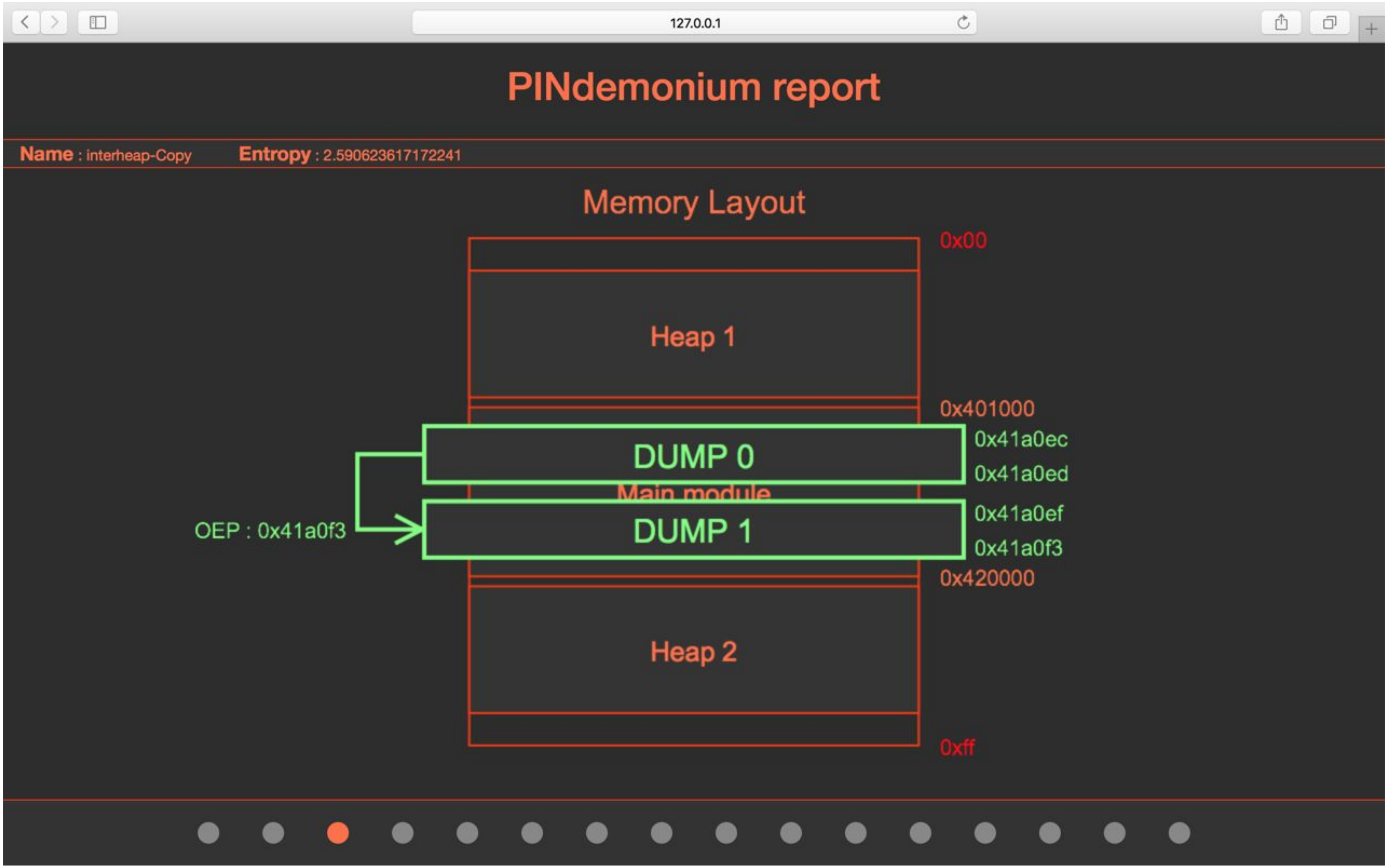
Identify remote writes to other processes by hooking system calls:

- 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

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

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

! → Original code dumped but Import directory not reconstructed

Experiment 2 : wild samples

Number of packed (checked manually) samples

1066

	N°	%
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

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%

DEMO

Limitations

- More advanced IAT obfuscation techniques are not handled
- 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
- 17 common packers defeated
- 63% of random samples correctly unpacked (known and custom packers employed)

The source code is available at

https://github.com/Seba0691/PINdemonium

Thank you!