Implementing your own generic unpacker

HITB Singapore 2015

Julien Lenoir - julien.lenoir@airbus.com

October 14, 2015

Outline

1. Introduction
2. Test driven design
3. Fine tune algorithm
4. Demo
5. Results
6. Conclusion

Outline

1. Introduction
2. Test driven design
3. Fine tune algorithm
4. Demo
5. Results
6. Conclusion

Context

Why did we do this?

- For malware classification purposes
- No opensource implementation matching our constraints

Constraints

- Work on bare metal as well as on any virtualization solution (VMware, VirtualBox, etc.)
- Rebuild a valid PE for static analysis. Runnable PE for dynamic analysis is even better
- Prevent malware from detecting unpacking process

Generic unpacking is not new

Existing tools

- Renovo (2007)
- Omniunpack (2007)
- Justin (2008)
- MutantX-S (2013)
- Packer Attacker (2015)

Our work

Own implementation of MutantX-S engine which is based on Justin

Targets simple packers

Our tool targets packers that fully unpack original code before executing it

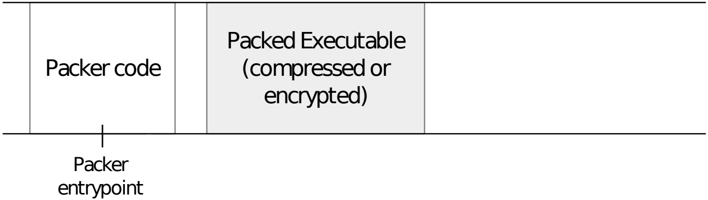
Works on

- Popular COTS packers (Aspack, Pecompact, etc.)
- Homemade packers

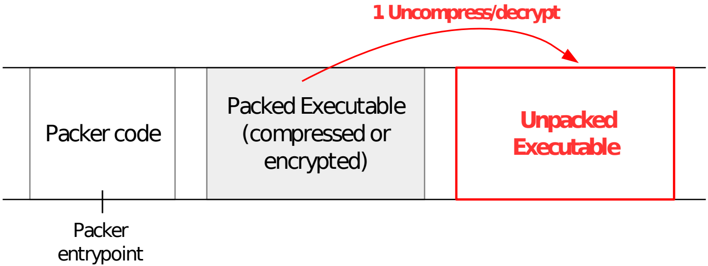
Does not work on

- Virtualizers (Armadillo, VMProtect)
- Packers that interleave unpacking layers and original code

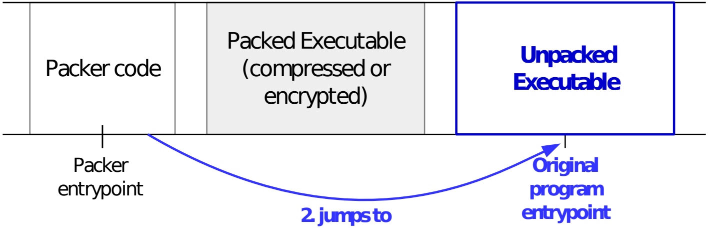
What is a simple packer?



What is a simple packer?



What is a simple packer?



Find the holy OEP

Goal

Find the original entry point (OEP)

General idea

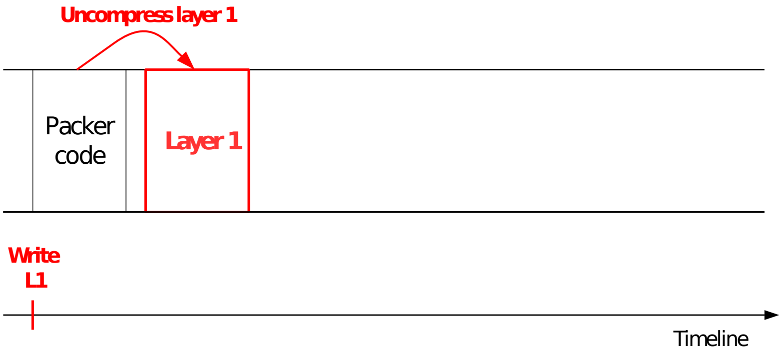
- Program is run in an instrumented Windows environment
- Dynamic code generation is monitored at page level

3 steps

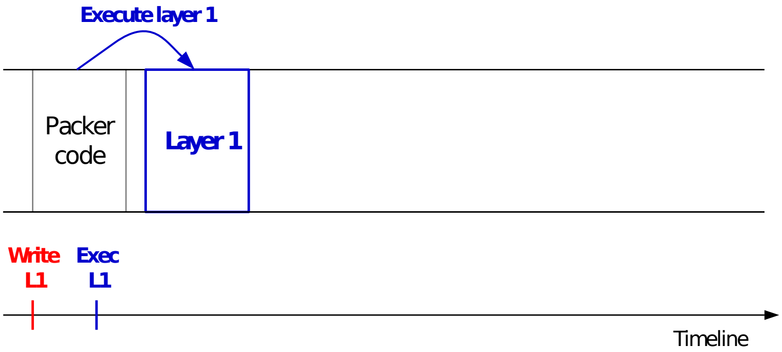
- Step 1: program is run once to trace both WRITE and EXECUTE on memory
- Step 2: apply an algorithm to this trace to determine OEP
- Step 3: program is run once again until OEP is reached, then dumped

step 1: program execution

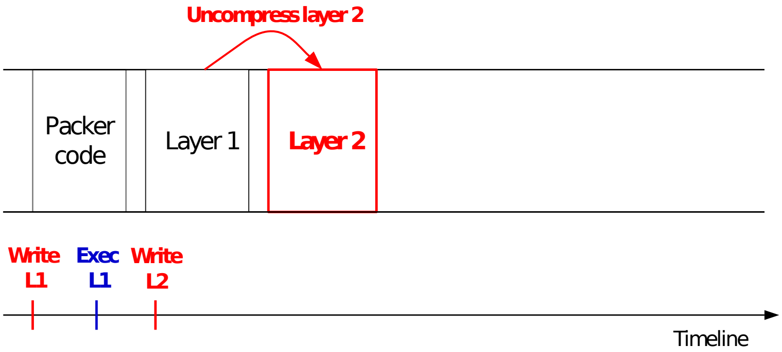
step 1: program execution



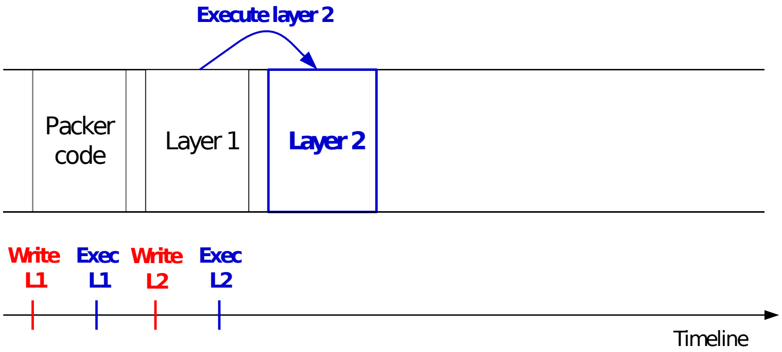
step 1: program execution



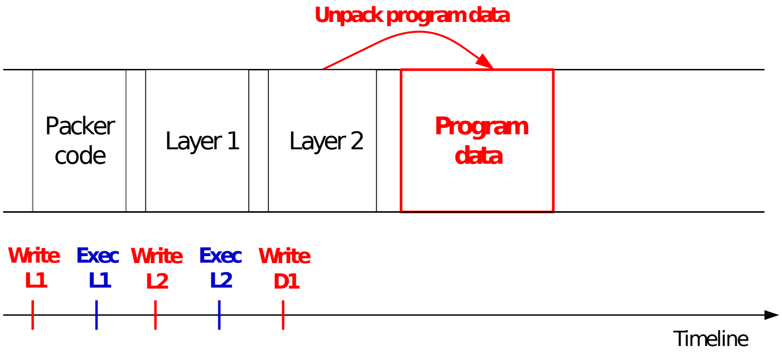
step 1: program execution



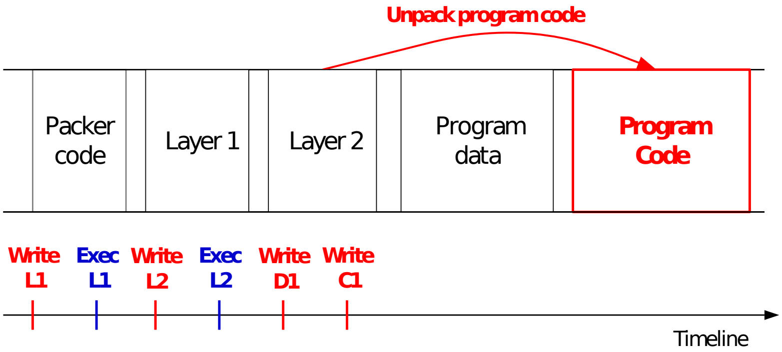
step 1: program execution



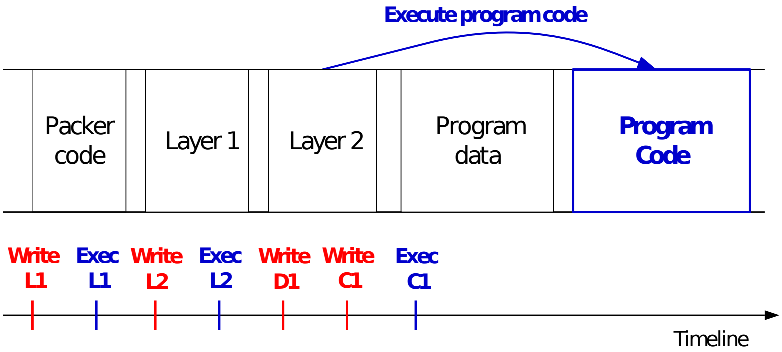
step 1: program execution



step 1: program execution

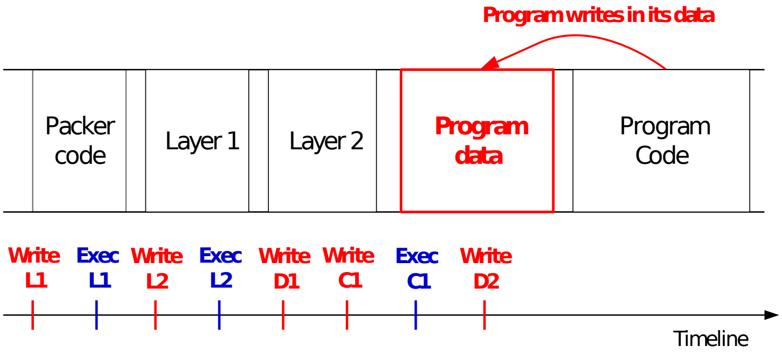


step 1: program execution

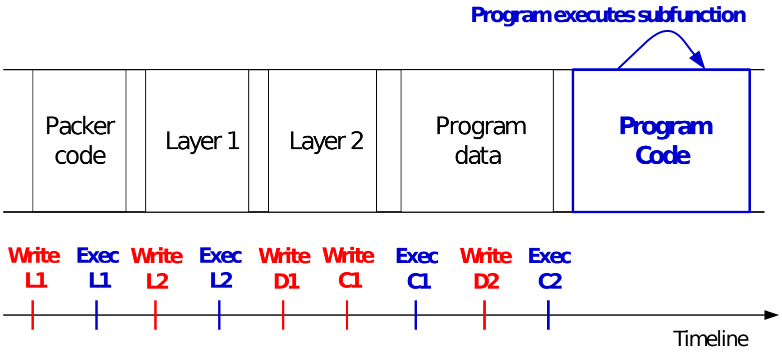


AIRBUS GROUP INNOVATIONS

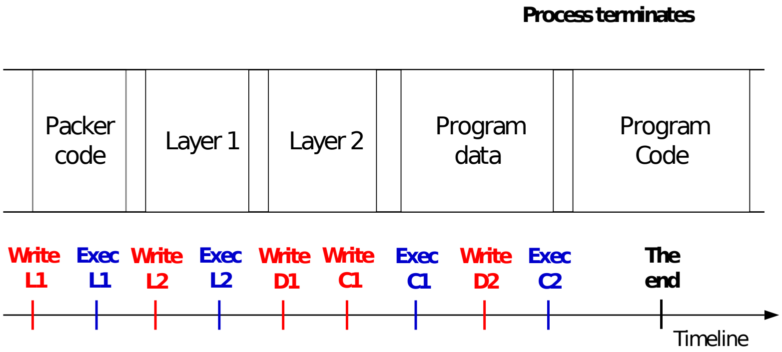
step 1: program execution



step 1: program execution

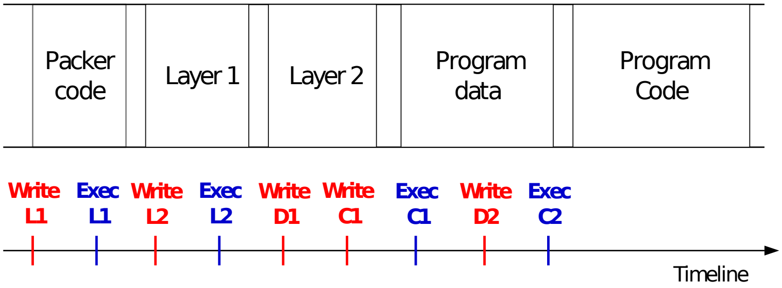


step 1: program execution



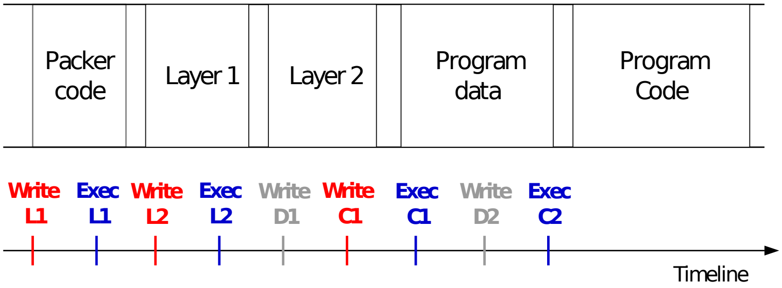
step 2: OEP identification

Apply algorithm on excution trace



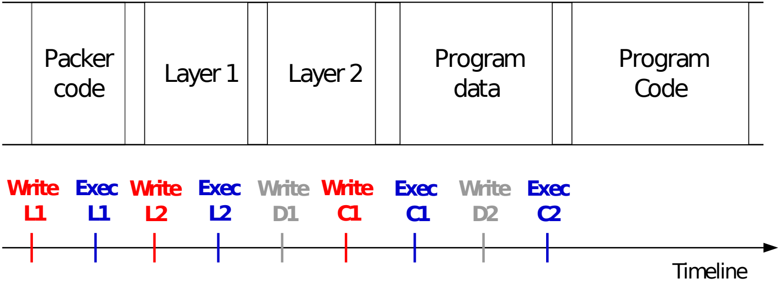
step 2: OEP identification

Filter out written only pages and executed only pages



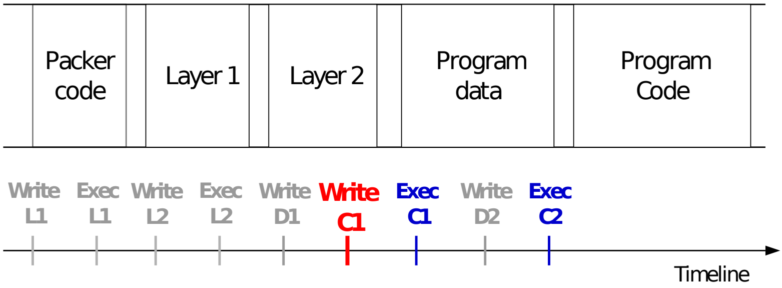
step 2: OEP identification

Keep pages that are executed **and** written



step 2: OEP identification

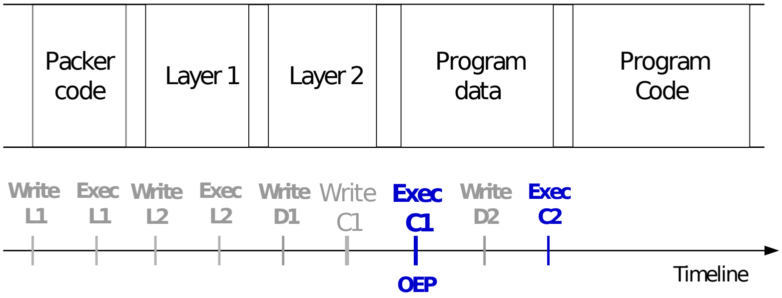
Find the last written page



AIRBUS GROUP INNOVATIONS

step 2: OEP identification

OEP is at first executed address after last write



AIRBUS GROUP

Tracking memory access

How?

- By changing memory access rights
- Write or execute access on memory page generates exceptions
- We catch those exceptions to monitor program behavior
- No page can be both executable **and** writable

In details

- Sets all pages to **executable** prior to execution
- Run the process
- On **write** attempt change page protection from **executable** to **writable**
- On **execute** attempt change page protection from **writable** to **executable**
- Do it until process terminates or a given time elapses

Outline

Main design choices

Our machinery runs inside the OS

Advantage

Compatible with any virtualization solution

Disadvantages

- A malware can detect virtualization: out of scope
- Targeted malware can detect our unpacker (driver name, etc.)

Supported OS: Windows 7 32 bits in PAE mode

Limitations

- Old system but it is enough for userland programs
- No support of 64 bit samples

Keep track of unpacking

We don’t want the packer to

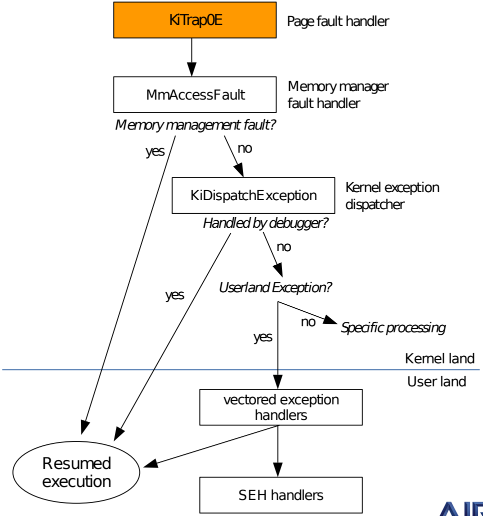
- Allocate memory both writable and executable
- Change its memory protection
- Generate dynamically code without our knowledge

Hooking memory system calls

- NtAllocateVirtualMemory
- NtProtectVirtualMemory
- ...

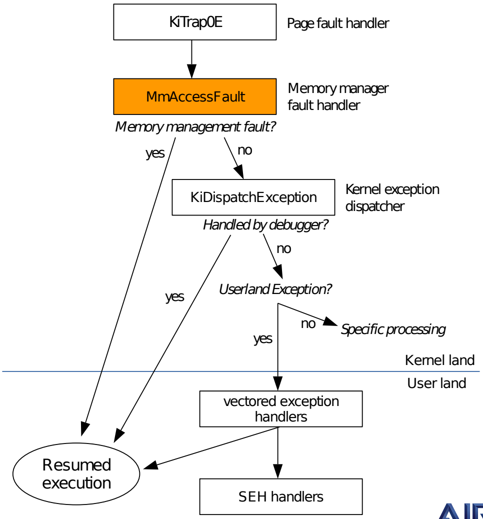
Userland exception path

1. Processor transfers execution to the kernel #PF handler.



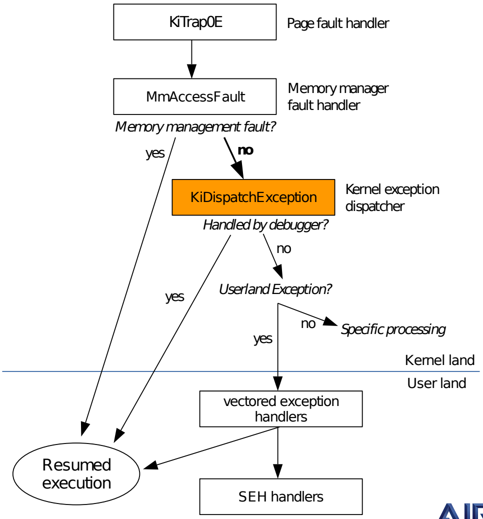
Userland exception path

2. Handles memory management faults. Like physical page in page file (swap).



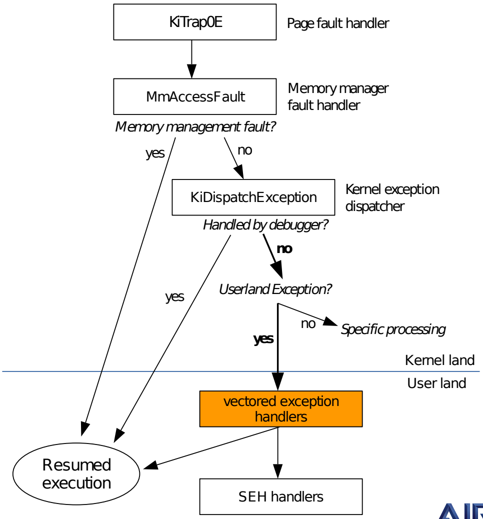
Userland exception path

3. Sort userland and kernel land exceptions. Forward exceptions to debuggers.



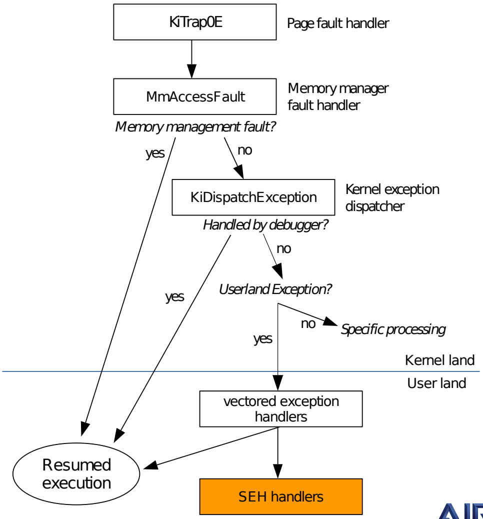
Userland exception path

4. Exception transfered to first registered handlers in userland process. Visible by all threads.



Userland exception path

5. Thread specific exception handlers (try / catch).



Architecture: first attempt

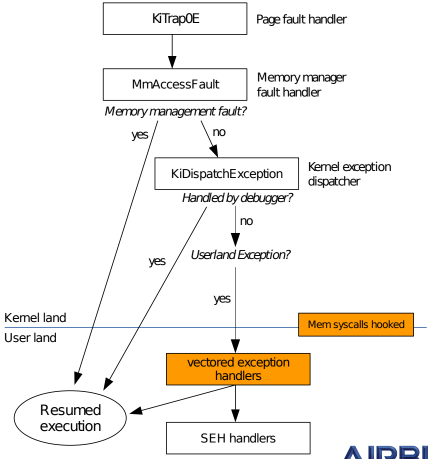
Catching exceptions at userland level

Advantage

Easy to implement

Disadvantage

Need to have code inside target process



Problem: self modifying page

Case

Encountered in **mpress** packed executables

What happens:

- Some memory pages are meant to be RWX
- Those pages are self modifying
- We enter an infinite loop

What happens

EIP at 401009
EAX is 0

```
PAGE 401000 EXECUTABLE

 401007    NOP
 401008    NOP
>401009    MOV EAX,401234
 40100E    XOR BYTE PTR DS:[EAX],42
 401011    NOP
 ...
 401234    db     0
```

What happens

EIP at 40100E
EAX is 401234

```
PAGE 401000 EXECUTABLE

 401007    NOP
 401008    NOP
 401009    MOV EAX,401234
>40100E    XOR BYTE PTR DS:[EAX],42
 401011    NOP
  ...
 401234    db     0
```

What happens

EIP at 40100E
EAX is 401234

Exception (type 1 write)
Invalid write access on address 401234

```
PAGE 401000 EXECUTABLE

 401007    NOP
 401008    NOP
 401009    MOV EAX,401234
>40100E    XOR BYTE PTR DS:[EAX],42
 401011    NOP
 ...
 401234    db    0
```

What happens

EIP at 40100E
EAX is 401234

Exception (type 1 write)
Invalid write access on address 401234
Swap page protection

```
PAGE 401000 WRITABLE

 401007    NOP
 401008    NOP
 401009    MOV EAX,401234
>40100E    XOR BYTE PTR DS:[EAX],42
 401011    NOP
 ...
 401234    db    0
```

What happens

EIP at 40100E
EAX is 401234

Exception (type 1 write)
Invalid write access on address 401234
Swap page protection
Resume process execution at 40100E

```
PAGE 401000 WRITABLE

 401007    NOP
 401008    NOP
 401009    MOV EAX,401234
>40100E    XOR BYTE PTR DS:[EAX],42
 401011    NOP
 ...
 401234    db     0
```

What happens

EIP at 40100E
EAX is 401234

Exception (type 1 write)
Invalid write access on address 401234
Swap page protection
Resume process execution at 40100E
Exception (type 8 execute)
Invalid execute access on address 40100E

```
PAGE 401000 WRITABLE

 401007    NOP
 401008    NOP
 401009    MOV EAX,401234
>40100E    XOR BYTE PTR DS:[EAX],42
 401011    NOP
  ...
 401234    db     0
```

What happens

EIP at 40100E
EAX is 401234

Exception (type 1 write)
Invalid write access on address 401234
Swap page protection
Resume process execution at 40100E
Exception (type 8 execute)
Invalid execute access on address 40100E
Swap page protection

```
PAGE 401000 EXECUTABLE

 401007     NOP
 401008     NOP
 401009     MOV EAX,401234
>40100E     XOR BYTE PTR DS:[EAX],42
 401011     NOP
  ...
 401234     db     0
```

What happens

EIP at 40100E
EAX is 401234

Exception (type 1 write)
Invalid write access on address 401234
Swap page protection
Resume process execution at 40100E
Exception (type 8 execute)
Invalid execute access on address 40100E
Swap page protection
Resume process execution at 40100E
...
Infinite loop

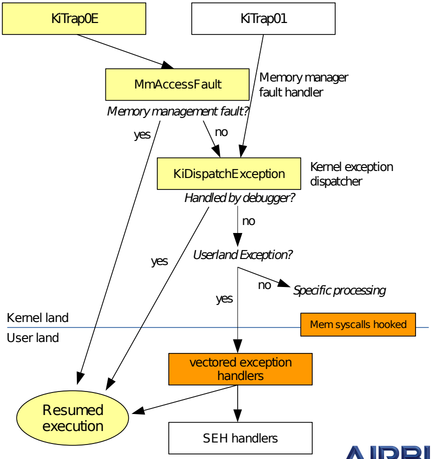
```
PAGE 401000 EXECUTABLE

 401007    NOP
 401008    NOP
 401009    MOV EAX,401234
>40100E    XOR BYTE PTR DS:[EAX],42
 401011    NOP
 ...
 401234    db    0
```

Architecture update: catch single-step exceptions

In two steps

1. Access violation :
   - Set page **writable** and **executable**
   - Activate single-step
   - Resume process execution

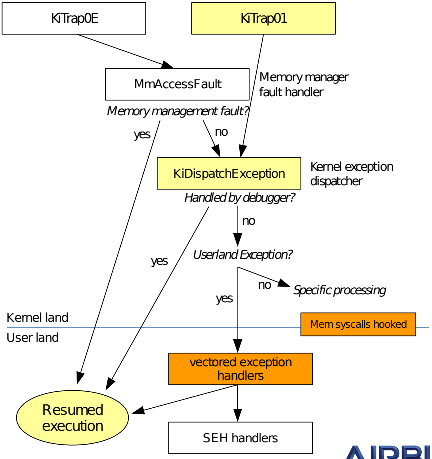


Architecture update: catch single-step exceptions

In two steps

1. Access violation :
   - Set page **writable** and **executable**
   - Activate single-step
   - Resume process execution

2. Int01 Trap (single-step) :
   - Restore page protection to **executable**
   - Remove single-step
   - Resume process execution



Problem: syscall sanitization

Case

Encountered in a binary packed with **NSPack 2.4**

What happens:

- packer calls *NtProtectVirtualMemory* during its unpacking process
- This syscall has output arguments
- Argument address is **executable** but not **writable**
- Syscall fails and so does unpacking

What happens

System call input sanitization is exception based:

```
NTSTATUS NtProtectVirtualMemory(..., int * pOldAccess)
{
  try
  {
    ProbeForWrite(pOldAccess, sizeof(int));

    MiProtectVirutalMemory(... ,pOldAccess);
  }
  except
  {
    return ERROR_NO_ACCESS;
  }
}
```

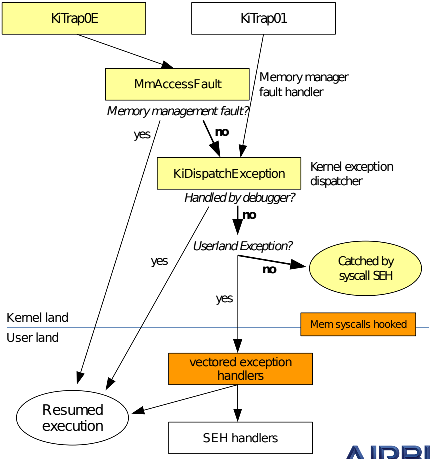
- *ProbeForWrite* **actually** writes the whole buffer to ensure it is writable
- If not writable, exception is generated and caught by the system call

What happens

Exception goes through

- Page Fault Hander
- Memory management fault handler
- Kernel exception dispatcher
- System call registered SEH

It never reaches userland, we cannot handle it!



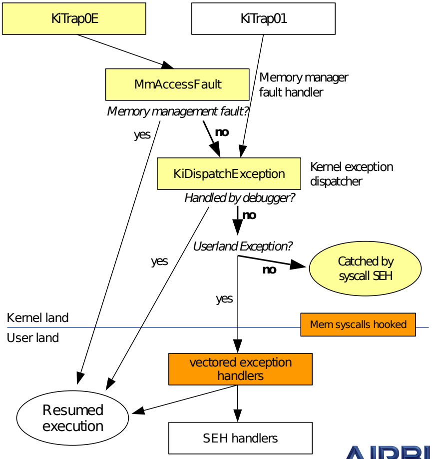
What happens

Exception goes through

- Page Fault Hander
- Memory management fault handler
- Kernel exception dispatcher
- System call registered SEH

It never reaches userland, we cannot handle it!

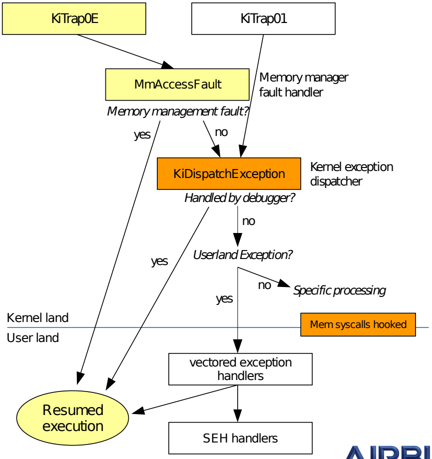
Catching exceptions in userland is not a good idea



Architecture update: catch exceptions in kernel

In two steps

1. Access violation :
   - Temporary set the page as **writable**
   - Activate single step
   - Resume kernel execution

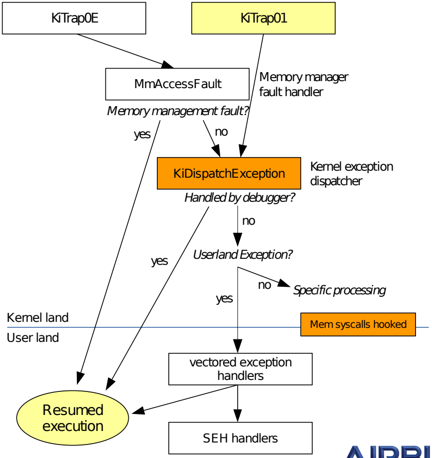


Architecture update: catch exceptions in kernel

In two steps

1. Access violation :
   - Temporary set the page as **writable**
   - Activate single step
   - Resume kernel execution

2. Int01 Trap (single-step) :
   - Restore page protection to **executable**
   - Remove single-step
   - Resume kernel execution



Another tricky case

```
VirtualProtect(memory_address, RWX);

VirtualQuery(address,&PageProtection);
if (PageProtection == RWX)
{
  goto continue_unpacking;
}
else
{
  goto error;
}
```

- Hooking of memory system calls is not sufficient
- We need to maintain a *packer view* of the process memory

Another tricky case

```
VirtualProtect(memory_address, RWX);

VirtualQuery(address,&PageProtection);
if (PageProtection == RWX)
{
  goto continue_unpacking;
}
else
{
  goto error;
}
```

- Hooking of memory system calls is not sufficient
- We need to maintain a "packer view" of the process memory
- Were does the OS store information related to memory?

In physical memory

63	62 61 60 59 58 57 56 55 54 53 52 51 … M[1]	M-1 … 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12	11 10 9	8	7	6	5	4	3	2	1	0	
XD	Reserved	Address of 4KB page frame	Ign.	G	PAT	D	A	PCD	PWT	U/S	R/W	1	**PTE: 4KB page**
Ignored												0	**PTE: not present**

64 bits PTE entry in PAE mode

Present PTE :

- 1 bit for present
- 2 bits for memory protection: combination of R,W,E
- 3 ignored (free) bits

Non present PTE :

- 1 bit for present
- 63 ignored (free) bits

In physical memory

Windows memory manager stores information in both invalid and valid PTEs

Examples of invalid PTEs

- Demand zero: demand paging
- Page File: physical page is in paging file
- Prototype PTE: shared memory

In valid PTEs

Information related to copy-on-write mechanisms

In kernel virtual memory

Two memory structures involved:

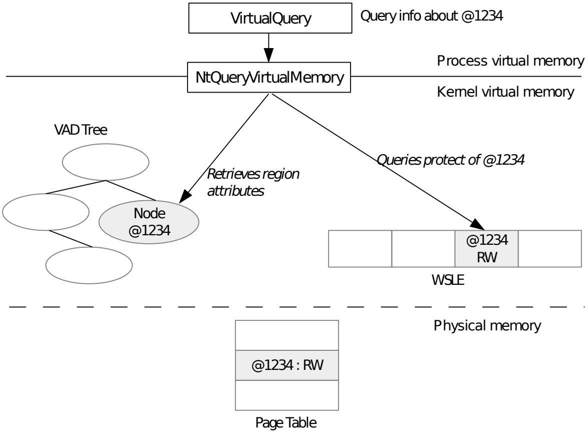
Virtual Address Descriptors

- The view of the process memory virtual address space
- Binary tree where every node is a memory region
- Information related to memory regions

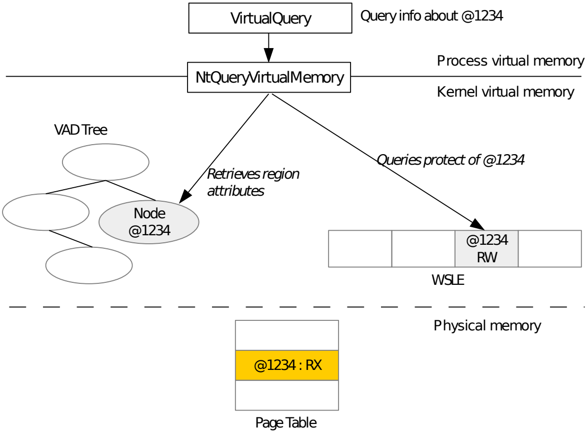
Working set list entries

Global array containing protection of every memory page

Example of VirtualQuery



Unsynchronizing memory structures



Unsynchronizing memory structures

Good points

- No need for a *packer view* any more
- No need to mess with complex kernel memory structures

Beware of resynchronization

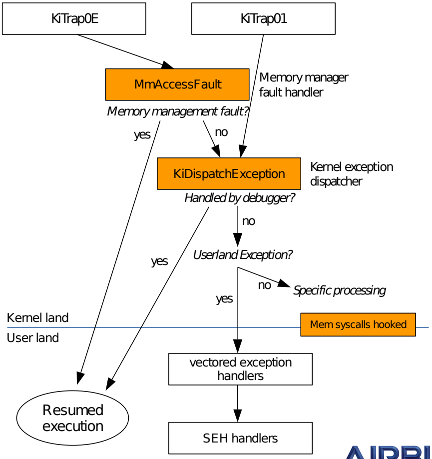
- Happens on memory system calls
- When memory manager handles page faults (demand paging, etc.)

Architecture: final

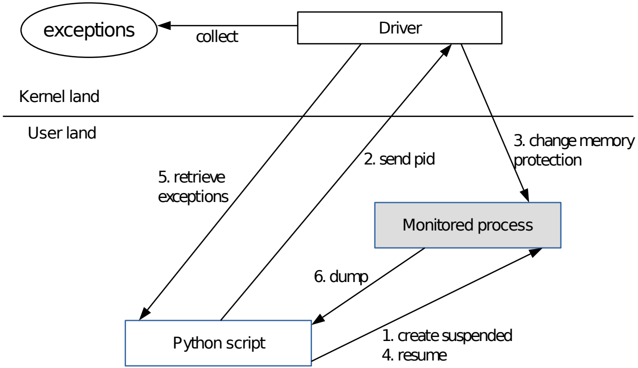
Hook in two places:

Memory manager fault handler for page faults

Kernel exceptions dispatcher for single-step exceptions



Global architecture



[0] Dump and IAT rebuild is done with Scylla library

Outline

Loader issue

Issue

Unpacking algorithm can be disturbed by the unpacked process startup

By the DLL loader if

- The process loads libraries dynamically on startup (after OEP)
- Those libraries are rebased

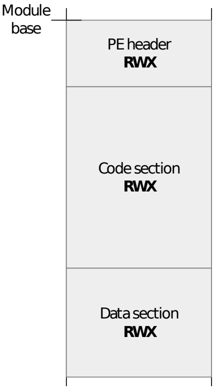
Userland library loader

All DLLs have a standard entrypoint *Dllmain* called during library loading

Loader does

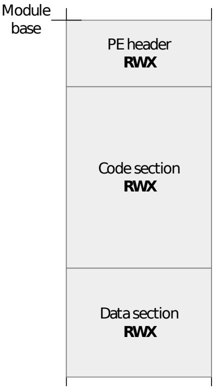
- Ensure the DLL is not already loaded
- Map the DLL in memory, possibly rebased at randomized address
- Patch relocations if DLL is rebased
- Set appropriate protection on PE sections
- Executes DLL entrypoint (*DllMain*)

Loader at work

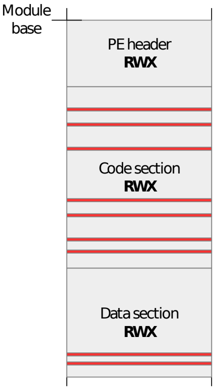


1. Protects sections

Loader at work

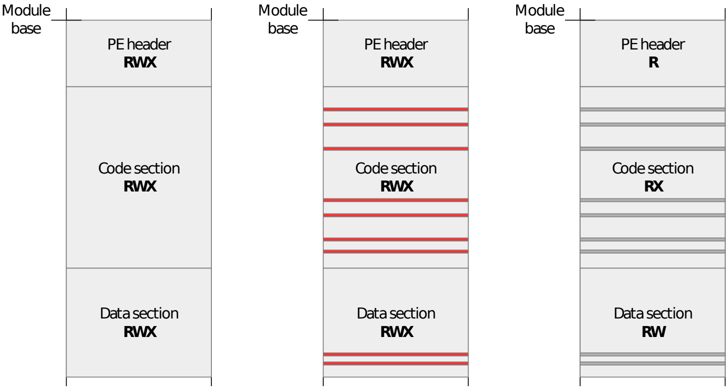


1. Protects sections



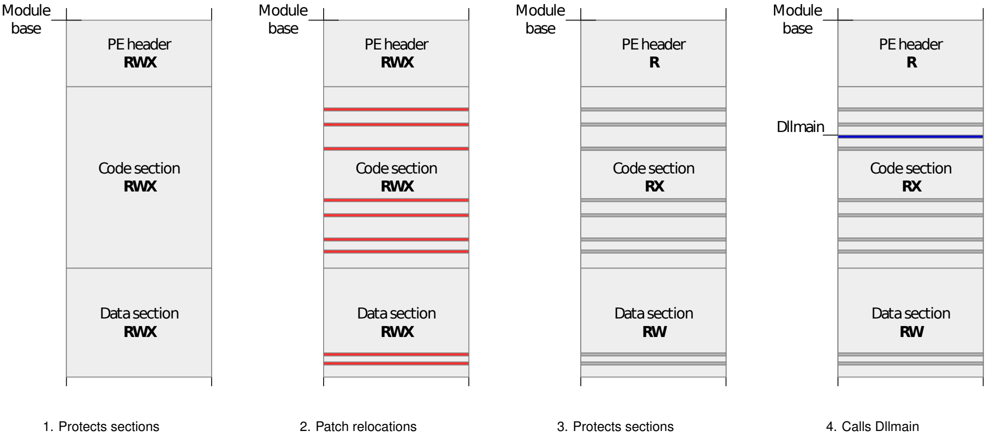
2. Patch relocations

Loader at work



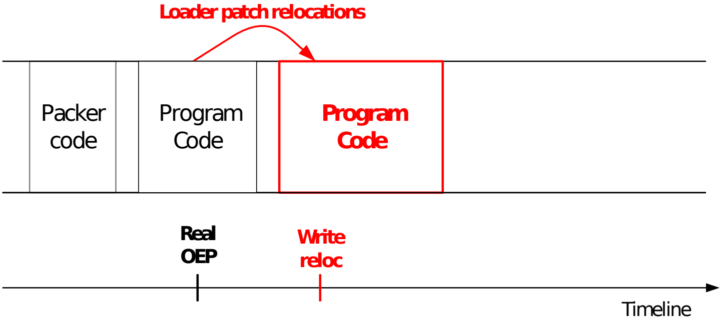
1. Protects sections
2. Patch relocations
3. Protects sections

Loader at work



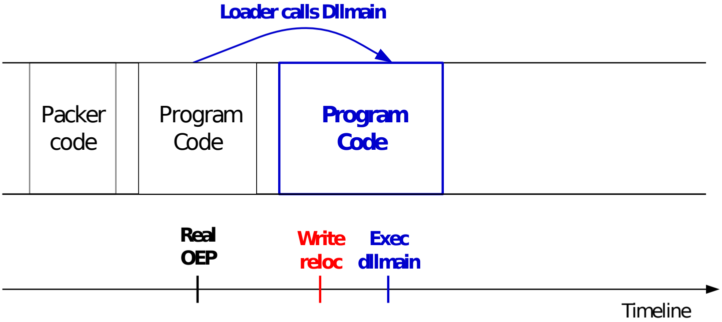
Loader artifact

Unpacked program loads a library dynamically



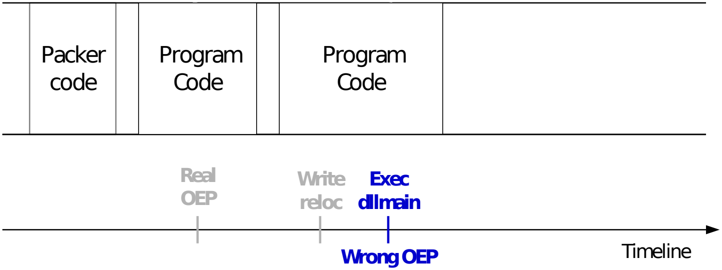
Loader artifact

Unpacked program loads a library dynamically



Loader artifact

Invalid OEP computation



Tune algorithm

Unpacking executable

Filter out exceptions induced by the loader during loading

Loader information

- Is loader at work
- Which DLL is being loaded
- Which thread of the process is loading the DLL

Tune algorithm

Unpacking DLLs

Keep **only** exceptions induced by the loader during loading process

Packed DLLs

- Packer code execute in *Dllmain*
- Packer jumps to DLL OEP: real *Dllmain*

We can determine DLL OEP and dump the unpacked DLLs !

Outline

Demo time!

Outline

No that easy to test

Packers

- Many different packers
- Not always easy to get

Packed samples

- What is exactly the version of packer used ?
- What are the options enables when packing sample

During design

Methodology :

- Using packers (default options)
- Using sorted packed samples (Tutz4you)

Packer	Dump with valid OEP	Working PE
UPX (3.91)	Yes	Yes
MPRESS (2.19)	Yes	No
PeCompact (2.X)	Yes	Yes/No
NsPack (2.4 to 3.7)	Yes	Yes
Aspack (2.2)	Yes	Yes
Asprotect	Yes	No
Armadillo	No	No
VMProtect	No	No

On random virustotal samples

Methodology :

- Request many packed samples from virus total
- Keep 20 for each packer samples randomly
- Manual anlysis to ensure OEP is valid

Packer	Valid PE	Valid OEP found	Unpacked PE runs
UPX	13	12 (~90%)	2(~15%)
Aspack	12	9 (~75%)	3(~25%)
NSpack	15	9 (~60%)	5(~30%)
PeCompact	14	10 (~91%)	4(~29%)
Upack	15	13 (~86%)	4 (~26%)
fsg	10	7 (~70%)	2(~20%)
exe32pack	6	4 (~66%)	0(~0%)

Outline

Good point

Easy and automatable unpacking of simple packers

What should we improve?

- Add heuristics to improve end of unpacking detection
- Support of Windows 7 64 bits?
- Support of Windows 10?

Code available at

https://bitbucket.org/iwseclabs/gunpack.git

Maybe you can

Make **your own** generic unpacker!

Thank you for listening !

Any questions ?