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
Implementing your own generic unpacker

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?

<table>
<thead>
<tr>
<th>Packer code</th>
<th>Packed Executable (compressed or encrypted)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Packer entrypoint</td>
</tr>
</tbody>
</table>

October 14, 2015
What is a simple packer?

1. Uncompress/decrypt

Packer code

Packed Executable (compressed or encrypted)

Unpacked Executable

Packer entrypoint
What is a simple packer?

![Diagram showing the process of packer code, packed executable (compressed or encrypted), and unpacked executable. The diagram explains how packer code and packed executable are connected through the original program entry point and how unpacking occurs.](image)
Implementing your own generic unpacker

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
Implementing your own generic unpacker

step 1: program execution

Timeline

Packer code
Implementing your own generic unpacker

step 1: program execution

- Uncompress layer 1
- Packer code
- Layer 1
- Write L1

Timeline
Implementing your own generic unpacker

step 1: program execution

- Execute layer 1

Timeline

Packer code | Layer 1

Write L1  Exec L1
Implementing your own generic unpacker

step 1: program execution

Timeline

Uncompress layer 2

Packer code | Layer 1 | Layer 2
Write L1 | Exec L1 | Write L2
Implementing your own generic unpacker

step 1: program execution

Packer code
Layer 1
Layer 2

Execute layer 2

Timeline

Write L1
Exec L1
Write L2
Exec L2
Implementing your own generic unpacker

step 1: program execution

- **Packer code**
- **Layer 1**
- **Layer 2**

**Unpack program data**

- Write L1
- Execute L1
- Write L2
- Execute L2
- Write D1

Timeline
step 1: program execution

- Packer code
- Layer 1
- Layer 2
- Program data
- Program Code
- Write L1
- Exec L1
- Write L2
- Exec L2
- Write D1
- Write C1
Implementing your own generic unpacker

step 1: program execution

- Packer code
- Layer 1
- Layer 2
- Program data
- Program Code

Timeline:
- Write L1
- Exec L1
- Write L2
- Exec L2
- Write D1
- Write C1
- Exec C1

Execute program code
step 1: program execution

Program writes in its data

Timeline

Packer code | Layer 1 | Layer 2 | Program data | Program Code

Write L1 | Exec L1 | Write L2 | Exec L2 | Write D1 | Write C1 | Exec C1 | Write D2
step 1: program execution

Implementing your own generic unpacker

Program executes subfunction

<table>
<thead>
<tr>
<th>Packer code</th>
<th>Layer 1</th>
<th>Layer 2</th>
<th>Program data</th>
</tr>
</thead>
</table>

Write

L1
L1
L2
L2
D1
C1
C1
D2
C2

Timeline
Implementing your own generic unpacker

step 1: program execution

---

+ Packer code
  + Layer 1
  + Layer 2
  + Program data
  + Program Code

---

Process terminates

Write L1  Exec L1  Write L2  Exec L2  Write D1  Write C1  Exec C1  Write D2  Exec C2

Timeline

The end

October 14, 2015
Implementing your own generic unpacker

step 2: OEP identification

Apply algorithm on execution trace

<table>
<thead>
<tr>
<th>Packer code</th>
<th>Layer 1</th>
<th>Layer 2</th>
<th>Program data</th>
<th>Program Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write L1</td>
<td>Exec L1</td>
<td>Write L2</td>
<td>Exec L2</td>
<td>Write D1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Write C1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Exec C1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Write D2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Exec C2</td>
</tr>
</tbody>
</table>

Timeline
Implementing your own generic unpacker

step 2: OEP identification

Filter out written only pages and executed only pages

<table>
<thead>
<tr>
<th>Packer code</th>
<th>Layer 1</th>
<th>Layer 2</th>
<th>Program data</th>
<th>Program Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write L1</td>
<td>Exec L1</td>
<td>Write L2</td>
<td>Exec L2</td>
<td></td>
</tr>
<tr>
<td>Write D1</td>
<td>Write C1</td>
<td>Exec C1</td>
<td>Write D2</td>
<td>Exec C2</td>
</tr>
</tbody>
</table>

Timeline
Implementing your own generic unpacker

step 2: OEP identification

Keep pages that are executed and written

<table>
<thead>
<tr>
<th>Packer code</th>
<th>Layer 1</th>
<th>Layer 2</th>
<th>Program data</th>
<th>Program Code</th>
</tr>
</thead>
</table>

Write L1
Exec L1
Write L2
Exec L2
Write D1
Write C1
Exec C1
Write D2
Exec C2

Timeline
Implementing your own generic unpacker

step 2: OEP identification

Find the last written page

<table>
<thead>
<tr>
<th>Packer code</th>
<th>Layer 1</th>
<th>Layer 2</th>
<th>Program data</th>
<th>Program Code</th>
</tr>
</thead>
</table>

Timeline:
- Write L1
- Exec L1
- Write L2
- Exec L2
- Write D1
- Exec C1
- Write D2
- Exec C2
Implementing your own generic unpacker

step 2: OEP identification

OEP is at first executed address after last write

<table>
<thead>
<tr>
<th>Packer code</th>
<th>Layer 1</th>
<th>Layer 2</th>
<th>Program data</th>
<th>Program Code</th>
</tr>
</thead>
</table>

Write L1  Exec L1  Write L2  Exec L2  Write D1  Write C1  Exec C1  Write D2  Exec C2

Timeline
Implementing your own generic unpacker

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

1. Introduction
2. Test driven design
3. Fine tune algorithm
4. Demo
5. Results
6. Conclusion
Implementing your own generic unpacker

Main design choices

Our machinery runs inside the OS

<table>
<thead>
<tr>
<th>Advantage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compatible with any virtualization solution</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>A malware can detect virtualization: out of scope</td>
</tr>
<tr>
<td>Targeted malware can detect our unpacker (driver name, etc.)</td>
</tr>
</tbody>
</table>

Supported OS: Windows 7 32 bits in PAE mode

<table>
<thead>
<tr>
<th>Limitations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Old system but it is enough for userland programs</td>
</tr>
<tr>
<td>No support of 64 bit samples</td>
</tr>
</tbody>
</table>
Implementing your own generic unpacker

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
- ...

October 14, 2015
Implementing your own generic unpacker

Userland exception path

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

- KTrapOE
  - Page fault handler
  - Memory manager fault handler

- MmAccessFault
  - Memory management fault?
    - yes
    - no

- KiDispatchException
  - Handled by debugger?
    - no
    - yes
      - Userland Exception?
        - yes
          - Specific processing
        - no
          - vectored exception handlers
            - SEH handlers

- Resumed execution
Userland exception path

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

```
Implementing your own generic unpacker

Userland exception path

October 14, 2015

32
```
Userland exception path

Implementing your own generic unpacker

Userland exception path

4. Exception transferred to first registered handlers in userland process. Visible by all threads.
Userland exception path

5. Thread specific exception handlers (try / catch).
Implementing your own generic unpacker

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
Implementing your own generic unpacker

What happens

EIP at 40100E
EAX is 401234

```plaintext
PAGE 401000 EXECUTABLE

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

October 14, 2015
Implementing your own generic unpacker

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
```
Implementing your own generic unpacker

What happens

EIP at 40100E
EAX is 401234

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

<table>
<thead>
<tr>
<th>PAGE 401000 WRITABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>401007    NOP</td>
</tr>
<tr>
<td>401008    NOP</td>
</tr>
<tr>
<td>401009    MOV EAX,401234</td>
</tr>
<tr>
<td>&gt;40100E   XOR BYTE PTR DS:[EAX],42</td>
</tr>
<tr>
<td>401011    NOP</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>401234    db 0</td>
</tr>
</tbody>
</table>
Implementing your own generic unpacker

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>401007</td>
<td>NOP</td>
</tr>
<tr>
<td>401008</td>
<td>NOP</td>
</tr>
<tr>
<td>401009</td>
<td>MOV EAX,401234</td>
</tr>
<tr>
<td>&gt;40100E</td>
<td>XOR BYTE PTR DS:[EAX],42</td>
</tr>
<tr>
<td>401011</td>
<td>NOP</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>401234</td>
<td>db 0</td>
</tr>
</tbody>
</table>
Implementing your own generic unpacker

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

<table>
<thead>
<tr>
<th>PAGE 401000 WRITABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>401007    NOP</td>
</tr>
<tr>
<td>401008    NOP</td>
</tr>
<tr>
<td>401009    MOV EAX,401234</td>
</tr>
<tr>
<td>&gt;40100E   XOR BYTE PTR DS:[EAX],42</td>
</tr>
<tr>
<td>401011    NOP</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>401234    db 0</td>
</tr>
</tbody>
</table>
Implementing your own generic unpacker

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>401007</td>
<td>NOP</td>
</tr>
<tr>
<td>401008</td>
<td>NOP</td>
</tr>
<tr>
<td>401009</td>
<td>MOV EAX,401234</td>
</tr>
<tr>
<td>&gt;40100E</td>
<td>XOR BYTE PTR DS:[EAX],42</td>
</tr>
<tr>
<td>401011</td>
<td>NOP</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>401234</td>
<td>db 0</td>
</tr>
</tbody>
</table>
Implementing your own generic unpacker

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
Implementing your own generic unpacker

Architecture update: catch single-step exceptions

In two steps

1. Access violation:
   - Set page **writable** and **executable**
   - Activate single-step
   - Resume process execution
Implementing your own generic unpacker

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
Implementing your own generic unpacker

Problem: syscall sanitization

Case
Encountered in a binary packed with NSPack 2.4

What happens:
- packer calls \textit{NtProtectVirtualMemory} during its unpacking process
- This syscall has output arguments
- Argument address is \textbf{executable} but not \textbf{writable}
- Syscall fails and so does unpacking
What happens

System call input sanitization is exception based:

```c
NTSTATUS NtProtectVirtualMemory( ... , int * pOldAccess) {
    try {
        ProbeForWrite(pOldAccess, sizeof(int));
        MiProtectVirtualMemory( ... ,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
Implementing your own generic unpacker

What happens

Exception goes through

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

It never reaches userland, we cannot handle it!
Implementing your own generic unpacker

What happens

Exception goes through
- Page Fault Handler
- 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
Implementing your own generic unpacker

Architecture update: catch exceptions in kernel

In two steps

1. Access violation:
   - Temporary set the page as **writable**
   - Activate single step
   - Resume kernel execution
Implementing your own generic unpacker

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

Implementing your own generic unpacker

```c
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
Implementing your own generic unpacker

Another tricky case

```c
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
- Where does the OS store information related to memory?
Implementing your own generic unpacker

In physical memory

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
Implementing your own generic unpacker

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:

<table>
<thead>
<tr>
<th>Virtual Address Descriptors</th>
</tr>
</thead>
<tbody>
<tr>
<td>The view of the process memory virtual address space</td>
</tr>
<tr>
<td>Binary tree where every node is a memory region</td>
</tr>
<tr>
<td>Information related to memory regions</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Working set list entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global array containing protection of every memory page</td>
</tr>
</tbody>
</table>
Implementing your own generic unpacker

Example of VirtualQuery

- VirtualQuery
  - Query info about @1234
- NtQueryVirtualMemory
  - Process virtual memory
  - Kernel virtual memory
  - Retrieves region attributes
  - Queries protect of @1234
- Node @1234
- VAD Tree
- Page Table
  - @1234: RW
- Physical memory
  - @1234: RW

October 14, 2015
Implementing your own generic unpacker

Unsynchronizing memory structures

VirtualQuery

NtQueryVirtualMemory

- Queries protect of @1234
- Retrieves region attributes
- Query info about @1234

Process virtual memory
Kernel virtual memory

VAD Tree

Node @1234

@1234 RW
WSLE
Physical memory

@1234: RX
Page Table
## 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.)
Implementing your own generic unpacker

Architecture: final

Hook in two places:

Memory manager fault handler for page faults

Kernel exceptions dispatcher for single-step exceptions
Implementing your own generic unpacker

Global architecture

1. create suspended
2. send pid
3. change memory protection
4. resume
5. retrieve exceptions

Python script

Driver

exceptions

Monitored process

Kernel land

User land

Dump and IAT rebuild is done with Scylla library
Implementing your own generic unpacker

Outline

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

<table>
<thead>
<tr>
<th>Issue</th>
<th>Unpacking algorithm can be disturbed by the unpacked process startup</th>
</tr>
</thead>
</table>
| 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*)
Implementing your own generic unpacker

Loader at work

1. Protects sections
Implementing your own generic unpacker

Loader at work

1. Protects sections
2. Patch relocations
Implementing your own generic unpacker

Loader at work

1. Protects sections
2. Patch relocations
3. Protects sections
Implementing your own generic unpacker

Loader at work

1. Protects sections

2. Patch relocations

3. Protects sections

4. Calls Dllmain
Implementing your own generic unpacker

Loader artifact

Unpacked program loads a library dynamically
Loader artifact

Unpacked program loads a library dynamically
Implementing your own generic unpacker

Loader artifact

Invalid OEP computation

<table>
<thead>
<tr>
<th>Packer code</th>
<th>Program Code</th>
<th>Program Code</th>
</tr>
</thead>
</table>

- Real OEP
- Write reloc
- Exec dllmain
- Wrong OEP

Timeline
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
Implementing your own generic unpacker

## 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!
Implementing your own generic unpacker

Outline

1. Introduction
2. Test driven design
3. Fine tune algorithm
4. Demo
5. Results
6. Conclusion
Implementing your own generic unpacker

Demo time!
Implementing your own generic unpacker

Outline

1. Introduction
2. Test driven design
3. Fine tune algorithm
4. Demo
5. Results
6. Conclusion
Implementing your own generic unpacker

No that easy to test

<table>
<thead>
<tr>
<th>Packers</th>
</tr>
</thead>
<tbody>
<tr>
<td>● Many different packers</td>
</tr>
<tr>
<td>● Not always easy to get</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Packed samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>● What is exactly the version of packer used?</td>
</tr>
<tr>
<td>● What are the options enables when packing sample</td>
</tr>
</tbody>
</table>
## During design

**Methodology:**
- Using packers (default options)
- Using sorted packed samples (Tutz4you)

<table>
<thead>
<tr>
<th>Packer</th>
<th>Dump with valid OEP</th>
<th>Working PE</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPX (3.91)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>MPRESS (2.19)</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>PeCompact (2.X)</td>
<td>Yes</td>
<td>Yes/No</td>
</tr>
<tr>
<td>NsPack (2.4 to 3.7)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Aspack (2.2)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Asprotect</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Armadillo</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>VMProtect</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
Implementing your own generic unpacker

**On random virustotal samples**

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

<table>
<thead>
<tr>
<th>Packer</th>
<th>Valid PE</th>
<th>Valid OEP found</th>
<th>Unpacked PE runs</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPX</td>
<td>13</td>
<td>12 (~90%)</td>
<td>2 (~15%)</td>
</tr>
<tr>
<td>Aspack</td>
<td>12</td>
<td>9 (~75%)</td>
<td>3 (~25%)</td>
</tr>
<tr>
<td>NSpack</td>
<td>15</td>
<td>9 (~60%)</td>
<td>5 (~30%)</td>
</tr>
<tr>
<td>PeCompact</td>
<td>14</td>
<td>10 (~91%)</td>
<td>4 (~29%)</td>
</tr>
<tr>
<td>Upack</td>
<td>15</td>
<td>13 (~86%)</td>
<td>4 (~26%)</td>
</tr>
<tr>
<td>fsg</td>
<td>10</td>
<td>7 (~70%)</td>
<td>2 (~20%)</td>
</tr>
<tr>
<td>exe32pack</td>
<td>6</td>
<td>4 (~66%)</td>
<td>0 (~0%)</td>
</tr>
</tbody>
</table>
Implementing your own generic unpacker

Outline

1. Introduction
2. Test driven design
3. Fine tune algorithm
4. Demo
5. Results
6. Conclusion
Implementing your own generic unpacker

**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!
Implementing your own generic unpacker

Thank you for listening!

Any questions?