Malware Obfuscation Techniques: Packing

November 18, 2014
80% of new malware are packed with various packers
80% of new malware are packed with various packers

50% of new malware samples are simply repacked versions of existing malware
A technique to hide the real code of a program through one or more layers of compression/encryption

At run-time the unpacking routine restores the original code in memory and then executes it
Code packing

- A technique to hide the real code of a program through one or more layers of compression/encryption
- At run-time the unpacking routine restores the original code in memory and then executes it

Malicious code
Code packing

- A technique to hide the real code of a program through one or more layers of compression/encryption
- At run-time the unpacking routine restores the original code in memory and then executes it
Code packing

- A technique to hide the real code of a program through one or more layers of compression/encryption
- At run-time the unpacking routine restores the original code in memory and then executes it
Code packing

- A technique to hide the real code of a program through one or more layers of compression/encryption
- At run-time the unpacking routine restores the original code in memory and then executes it

The effectiveness of malware detectors depends on the ability to recover the “real” malicious code, but recovery often fails!
Algorithmic unpacking
Use of specific unpacking routines to recover the original code (i.e., one routine per packing algorithm)
Traditional approaches to deal with packed code

Algorithmic unpacking

Use of specific unpacking routines to recover the original code (i.e., one routine per packing algorithm)
Traditional approaches to deal with packed code

Algorithmic unpacking

Use of specific unpacking routines to recover the original code (i.e., one routine per packing algorithm)
Traditional approaches to deal with packed code

Algorithmic unpacking

Use of specific unpacking routines to recover the original code (i.e., one routine per packing algorithm)
Algorithmic unpacking

Use of specific unpacking routines to recover the original code (i.e., one routine per packing algorithm)
Traditional approaches to deal with packed code

Algorithmic unpacking
Use of specific unpacking routines to recover the original code (i.e., one routine per packing algorithm)

Generic unpacking
Emulation/tracing of the execution until the unpacking routine terminates (e.g., PolyUnpack [ACSAC 06] and Renovo [WORM 07])
Traditional approaches to deal with packed code

Algorithmic unpacking
Use of specific unpacking routines to recover the original code (i.e., one routine per packing algorithm)

Generic unpacking
Emulation/tracing of the execution until the unpacking routine terminates (e.g., PolyUnpack [ACSAC 06] and Renovo [WORM 07])
Traditional approaches to deal with packed code

**Algorithmic unpacking**
Use of specific unpacking routines to recover the original code (i.e., one routine per packing algorithm)

**Generic unpacking**
Emulation/tracing of the execution until the unpacking routine terminates (e.g., PolyUnpack [ACSAC 06] and Renovo [WORM 07])
Traditional approaches to deal with packed code

Algorithmic unpacking
Use of specific unpacking routines to recover the original code (i.e., one routine per packing algorithm)

Generic unpacking
Emulation/tracing of the execution until the unpacking routine terminates (e.g., PolyUnpack [ACSAC 06] and Renovo [WORM 07])
Traditional approaches to deal with packed code

**Algorithmic unpacking**
Use of specific unpacking routines to recover the original code (i.e., one routine per packing algorithm)

**Generic unpacking**
Emulation/tracing of the execution until the unpacking routine terminates (e.g., PolyUnpack [ACSAC 06] and Renovo [WORM 07])
A simple generic unpacker

- Track all memory writes and the program counter
- The execution of a previously written memory location denotes the end of an unpacking stage
- All written-then-executed memory locations should then be analyzed by a malware detector
A simple generic unpacker

- Track all memory writes and the program counter
- The execution of a previously written memory location denotes the end of an unpacking stage
- All written-then-executed memory locations should then be analyzed by a malware detector

Extend this idea to design an iterative unpacking algorithm that achieves low overhead yet does not compromise the security of the system
Goals of Real-Time Unpackers

- Generic unpacking with low-overhead by using existing hardware mechanisms
- Precise unpacking by running the program on the native OS
- A new malware detection strategy, independent of packing, where the malware detector analyzes new pieces of code before they are executed.
Efficient tracking of memory accesses

Coarse-grained memory access tracking (at page level), through the use of hardware mechanisms
Efficient tracking of memory accesses

Coarse-grained memory access tracking (at page level), through the use of hardware mechanisms

Unfortunately...

▶ Written-then-executed locations are indicative of unpacking but not indicative of the end of unpacking
▶ Coarse-grained memory access tracking further increases the chances to detect spurious unpacking stages (up to hundreds of thousands stages)
Efficient tracking of memory accesses

Coarse-grained memory access tracking (at page level), through the use of hardware mechanisms

- **Executed page**
- **Written page**
- **Executed memory location**
- **Written memory location**

Unfortunately...
- Written-then-executed locations are indicative of unpacking but not indicative of the end of unpacking
- Coarse-grained memory access tracking further increases the chances to detect spurious unpacking stages (up to hundreds of thousands stages)
Efficient tracking of memory accesses

Coarse-grained memory access tracking (at page level), through the use of hardware mechanisms

Unfortunately...

- Written-then-executed locations are indicative of unpacking but not indicative of the end of unpacking
- Coarse-grained memory access tracking further increases the chances to detect spurious unpacking stages (up to hundreds of thousands stages)
Coarse-grained memory access tracking (at page level), through the use of hardware mechanisms

- Executed page
- Written page
- Executed memory location
- Written memory location

Malicious code

Unfortunately...

▶ Written-then-executed locations are indicative of unpacking but not indicative of the end of unpacking
▶ Coarse-grained memory accesses tracking further increases the chances to detect spurious unpacking stages (up to hundreds of thousands stages)
Efficient tracking of memory accesses

Coarse-grained memory access tracking (at page level), through the use of hardware mechanisms

Unfortunately...

- Written-then-executed locations are indicative of unpacking but not indicative of the end of unpacking
- Coarse-grained memory accesses tracking further increases the chances to detect spurious unpacking stages (up to hundreds of thousands stages)
Efficient tracking of memory accesses

Coarse-grained memory access tracking (at page level), through the use of hardware mechanisms

Unfortunately...

- Written-then-executed locations are indicative of unpacking but not indicative of the end of unpacking
- Coarse-grained memory accesses tracking further increases the chances to detect spurious unpacking stages (up to hundreds of thousands stages)
Efficient tracking of memory accesses

Coarse-grained memory access tracking (at page level), through the use of hardware mechanisms

Unfortunately...

- Written-then-executed locations are indicative of unpacking but not indicative of the end of unpacking
- Coarse-grained memory accesses tracking further increases the chances to detect spurious unpacking stages (up to hundreds of thousands stages)

The overhead introduced by invoking the malware detector every time a written page is executed is prohibitive!
Better approximating the end of an unpacking stage

Ideally:

![Diagram showing the sequence of events with 'Start', 'Scan', and 'Halt' stages]
Better approximating the end of an unpacking stage

Ideally:

With coarse-grained memory access tracking:
Better approximating the end of an unpacking stage

Ideally:

With coarse-grained memory access tracking:

Mitigate the imprecision of the coarse-grained memory accesses tracking by considering an unpacking stage concluded when the execution of a previously written page is followed by a dangerous system call.
Dangerous system calls

To achieve its malicious goals, the malware has to interact with the system (through system calls)
Dangerous system calls

To achieve its malicious goals, the malware has to interact with the system (through system calls)

Only few system calls are dangerous
A system call is dangerous if its execution can leave the system in an unsafe state
Dangerous system calls

To achieve its malicious goals, the malware has to interact with the system (through system calls)

Only few system calls are dangerous
A system call is dangerous if its execution can leave the system in an unsafe state

NtOpenFile NtOpenKey NtDeleteFile
Unpacker algorithm

**Input:** an execution trace $\langle e_0, e_1, \ldots \rangle$

where a trace event can be:

- $w(p)$ write access to a memory page $p$
- $x(p)$ instruction execution from a memory page $p$
- $s$ invocation of the system call $s$
Unpacker algorithm

Execution trace
\[ \langle x(0), w(2), s_0, w(1), x(1), s_1, x(2), s_2, \ldots \rangle \]

Memory pages status

<table>
<thead>
<tr>
<th>Page</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>W</td>
</tr>
<tr>
<td>0</td>
<td>WX</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Unpacker algorithm

Execution trace
\[ \langle x(0), w(2), s_0, w(1), x(1), s_1, x(2), s_2, \ldots \rangle \]

Memory pages status

<table>
<thead>
<tr>
<th>Page #</th>
<th>Access</th>
<th>W</th>
<th>WX</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>W</td>
<td>WX</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The memory page 0 is executed
Unpacker algorithm

Execution trace
\[ \langle x(0), w(2), s_0, w(1), x(1), s_1, x(2), s_2, \ldots \rangle \]

Memory pages status

<table>
<thead>
<tr>
<th>Page #</th>
<th>Access</th>
<th>W</th>
<th>WX</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>●</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The memory page 2 is written
The page is recorded in the set \( W \) of written pages
Unpacker algorithm

Execution trace
\( \langle x(0), w(2), s_0, w(1), x(1), s_1, x(2), s_2, \ldots \rangle \)

\( s_0 \) is \texttt{NtOpenFile}

Memory pages status

<table>
<thead>
<tr>
<th>Page</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>W</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>\bullet</td>
</tr>
<tr>
<td>\ldots</td>
<td></td>
</tr>
</tbody>
</table>

The system call \( s_0 \) is executed (not dangerous and \( WX \) is empty)
Unpacker algorithm

Execution trace
\[ \langle x(0), w(2), s_0, w(1), x(1), s_1, x(2), s_2, \ldots \rangle \]

Memory pages status

<table>
<thead>
<tr>
<th>Page #</th>
<th>Access</th>
<th>W</th>
<th>WX</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The memory page 1 is written
The page is recorded in the set \( W \) of written pages
Unpacker algorithm

Execution trace
\[ \langle x(0), w(2), s_0, w(1), x(1), s_1, x(2), s_2, \ldots \rangle \]

Memory pages status

<table>
<thead>
<tr>
<th>Page #</th>
<th>Access</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>2</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The memory page 1 is executed
The page is recorded in the set \( WX \) of written-then-executed pages
Unpacker algorithm

Execution trace
\[\langle x(0), w(2), s_0, w(1), x(1), s_1, x(2), s_2, \ldots \rangle\]

\(s_1\) is \texttt{NtOpenKey}

Memory pages status

<table>
<thead>
<tr>
<th>Page #</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>W</td>
</tr>
<tr>
<td>0</td>
<td>•</td>
</tr>
<tr>
<td>1</td>
<td>•</td>
</tr>
<tr>
<td>2</td>
<td>•</td>
</tr>
<tr>
<td>...</td>
<td>•</td>
</tr>
</tbody>
</table>

The system call \(s_1\) is executed (not dangerous)
Unpacker algorithm

Execution trace
\[\langle x(0), w(2), s_0, w(1), x(1), s_1, x(2), s_2, \ldots \rangle\]

Memory pages status

<table>
<thead>
<tr>
<th>Page #</th>
<th>Access</th>
<th>W</th>
<th>WX</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The memory page 2 is executed
The page is recorded in the set WX of written-then-executed pages
Unpacker algorithm

Execution trace

\[
\langle x(0), w(2), s_0, w(1), x(1), s_1, x(2), s_2, \ldots \rangle
\]

\(s_2\) is \texttt{NtDeleteFile}

Memory pages status

<table>
<thead>
<tr>
<th>Page #</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>W</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>●</td>
</tr>
<tr>
<td>2</td>
<td>●</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

The system call \(s_2\) is executed (dangerous)

The malware detector is invoked to scan all the memory pages in \(W\)
Unpacker algorithm

Execution trace
\[ \langle x(0), w(2), s_0, w(1), x(1), s_1, x(2), s_2, \ldots \rangle \]

Memory pages status

<table>
<thead>
<tr>
<th>Page #</th>
<th>Access</th>
<th>W</th>
<th>WX</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>\ldots</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If the program is not malicious the sets \( W \) and \( WX \) are emptied and the execution is resumed.
OmniUnpack for Microsoft Windows XP

Suspicious program

System-call monitor

Memory access monitor

OmniUnpack kernel driver

ClamAV Malware detector

User

Kernel
The $W \oplus X$ policy is enforced on the memory pages of the suspicious program.

Page-fault exceptions are trapped by OmniUnpack.

Non executable pages can be emulated via software.
Any malware detection strategy can be used to scan the code generated during the previous stage.
Food for Thoughts & Exercises

- Try to find out a method in order to evade Omninpack system
- **Following the parasite developed for the last homework** try to patch the got table on-the-fly and wrap some function and logs the parameters.
- **Add a layer of protection to the parasite against the static analysis** the parasite should be able to unpack yourself during the execution of the binary.
Thank You!
Q&A?