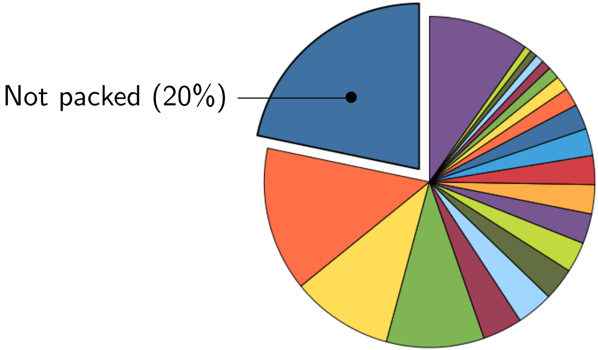
Malware Obfuscation Techniques: Packing

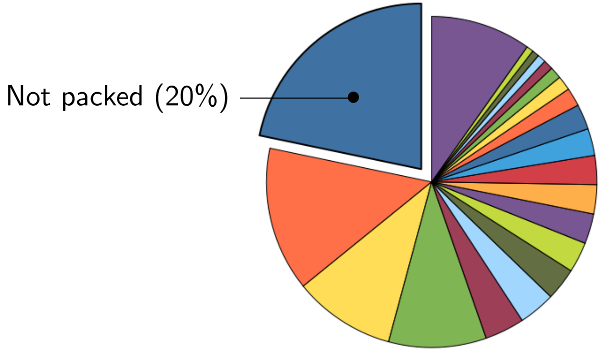
November 18, 2014

Malware and packing



80% of new malware are packed with various packers

Malware and packing



80% of new malware are packed with various packers

50% of new malware samples are simply repacked versions of existing malware

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

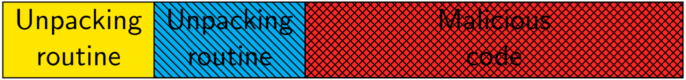
- A technique to hide the real code of a program through one or more layers of compression/encryption
- At run-time the unpacking routine restores the original code in memory and then executes it

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

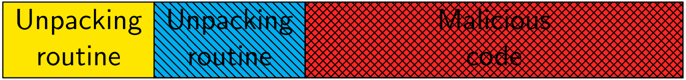
Code packing

- 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!

Traditional approaches to deal with packed code

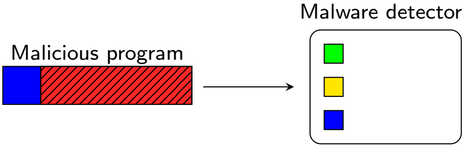
Algorithmic unpacking

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

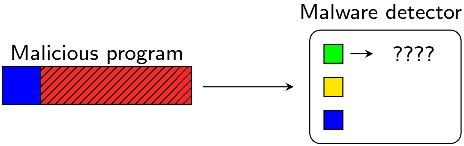
Use of specific unpacking routines to recover the original code (i.e., one routine per packing algorithm)



Traditional approaches to deal with packed code

Algorithmic unpacking

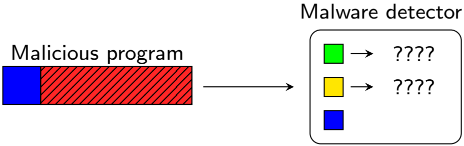
Use of specific unpacking routines to recover the original code (i.e., one routine per packing algorithm)



Traditional approaches to deal with packed code

Algorithmic unpacking

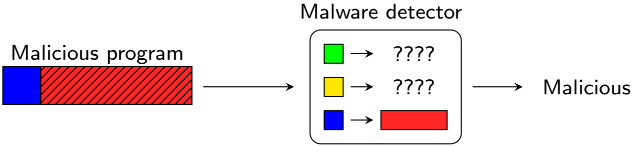
Use of specific unpacking routines to recover the original code (i.e., one routine per packing algorithm)



Traditional approaches to deal with packed code

Algorithmic unpacking

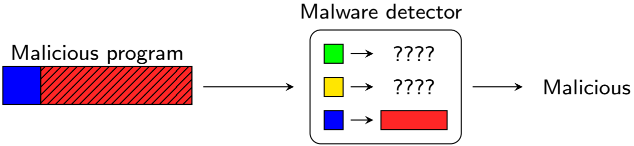
Use of specific unpacking routines to recover the original code (i.e., one routine per packing algorithm)



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)



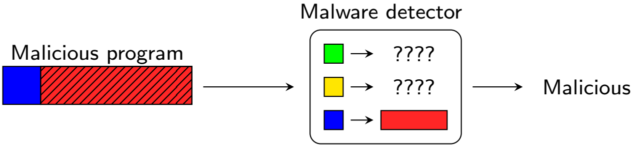
Generic unpacking

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)



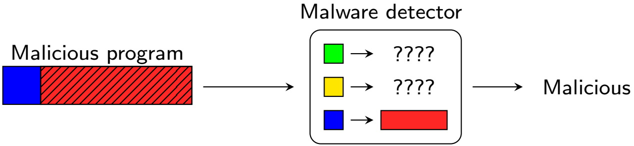
Generic unpacking

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)



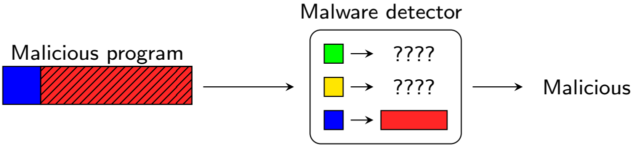
Generic unpacking

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)



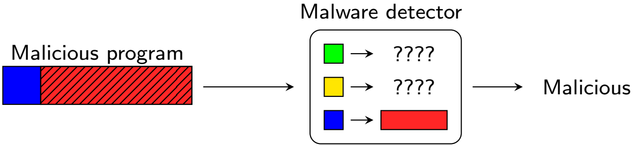
Generic unpacking

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

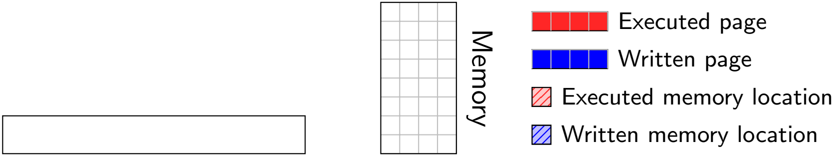
- Generic unpacking with low-overhead by using existing hardware mechanisms
- Precise unpacking by running the program on the native OS
- A new malware detection strategy, independent of packing, where the malware detector analyzes new pieces of code before they are executed.

Efficient tracking of memory accesses

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

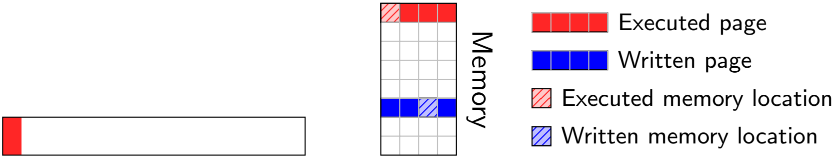
Efficient tracking of memory accesses

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



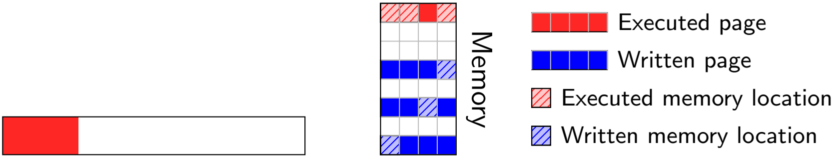
Efficient tracking of memory accesses

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



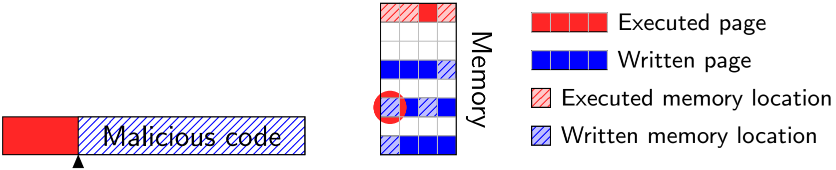
Efficient tracking of memory accesses

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



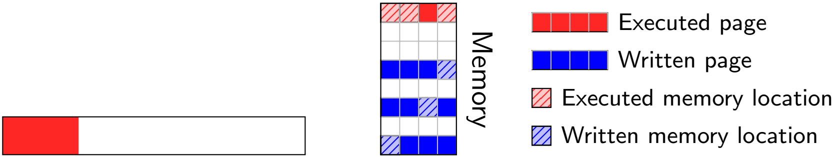
Efficient tracking of memory accesses

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

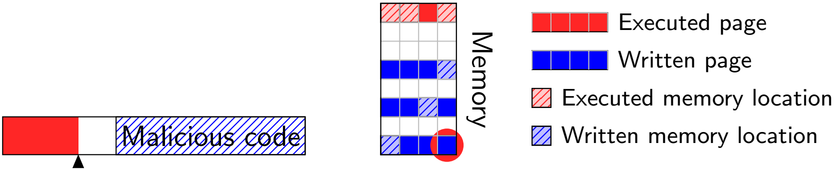


Unfortunately...

- Written-then-executed locations are indicative of unpacking but not indicative of the end of unpacking
- Coarse-grained memory 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

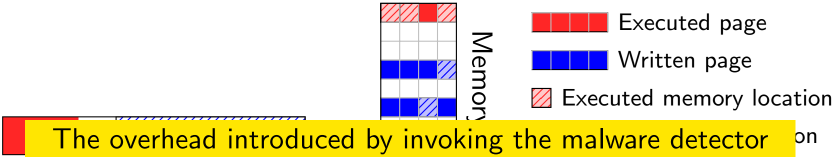


Unfortunately...

- 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



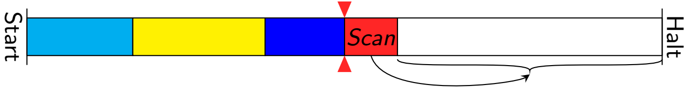
The overhead introduced by invoking the malware detector every time a written page is executed is prohibitive!

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)

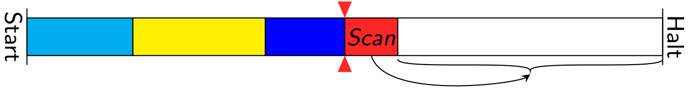
Better approximating the end of an unpacking stage

Ideally:

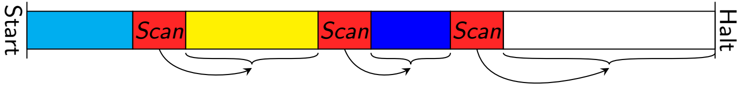


Better approximating the end of an unpacking stage

Ideally:

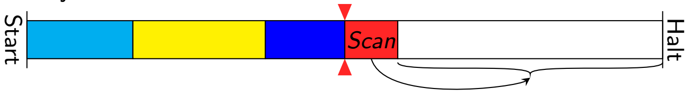


With coarse-grained memory access tracking:

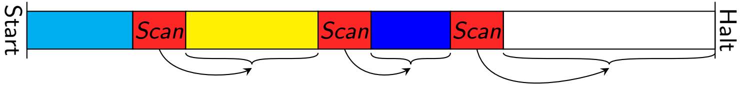


Better approximating the end of an unpacking stage

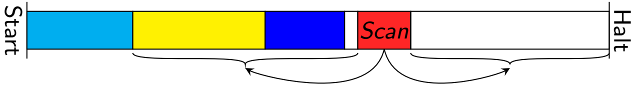
Ideally:



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

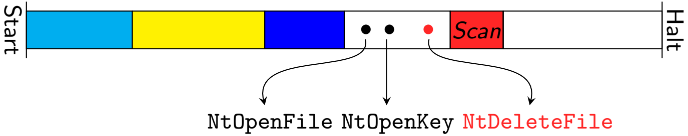
A system call is dangerous if its execution can leave the system in an unsafe state

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



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

Page #	**Access**	
	W	**WX**
0		
1		
2		
...		

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

Page #	**Access W**	**Access WX**
0		
1		
2		
...		

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

Page #	Access W	Access WX
0		
1		
2	•	
...		

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 `NtOpenFile`

Memory pages status

Page #	Access W	Access WX
0		
1		
2	•	
...		

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

Page #	Access W	Access WX
0		
1	•	
2	•	
...		

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

Page #	Access W	Access WX
0		
1	•	•
2	•	
...		

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 `NtOpenKey`

Memory pages status

Page #	**Access W**	**Access WX**
0		
1	•	•
2	•	
...		

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

Page #	**Access** **W**	**WX**
0		
1	•	•
2	•	•
...		

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 `NtDeleteFile`

Memory pages status

Page #	Access W	Access WX
0		
1	•	•
2	•	•
...		

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

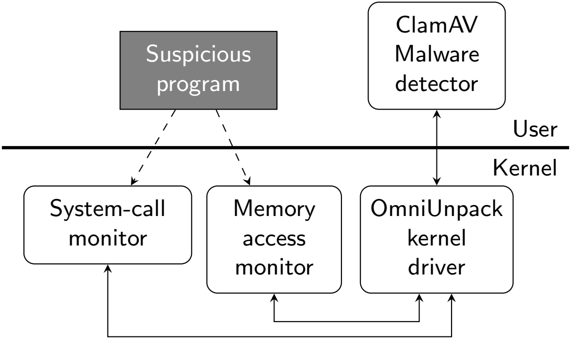
$\langle x(0),\ w(2),\ s_0,\ w(1),\ x(1),\ s_1,\ x(2),\ s_2,\ \ldots\rangle$

Memory pages status

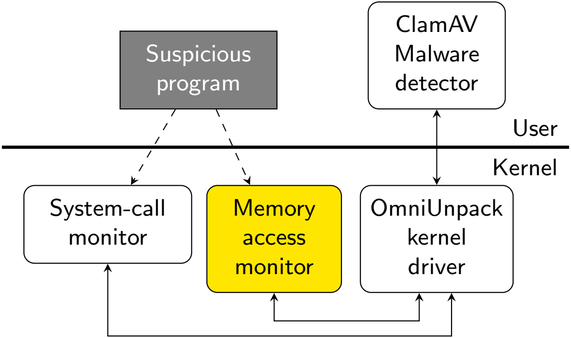
Page	**Access**	
#	**W**	**WX**
0		
1		
2		
...		

If the program is not malicious the sets W and WX are emptied and the execution is resumed

OmniUnpack for Microsoft Windows XP

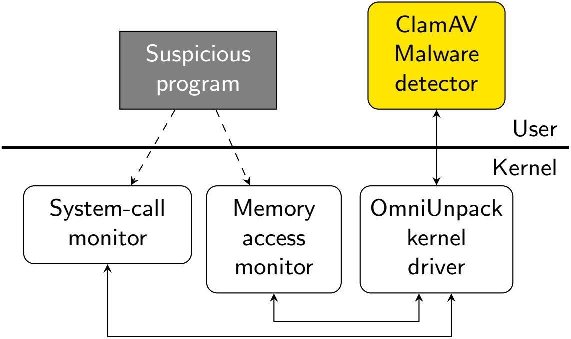


OmniUnpack for Microsoft Windows XP



- 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

OmniUnpack for Microsoft Windows XP



- 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 parassite 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 parassite against the static analysis** the parassite should be able to unpack yourself during the execution of the binary.

Q&A

Thank You!
Q&A?