
black hat
EUROPE 2019
DECEMBER 2-5, 2019
EXCEL LONDON, UK
Detecting (un)intentionally hidden injected Code by examining Page Table Entries
#BHEU @BLACK HAT EVENTS

Detecting (un)intentionally hidden injected Code by examining Page Table Entries

Frank Block

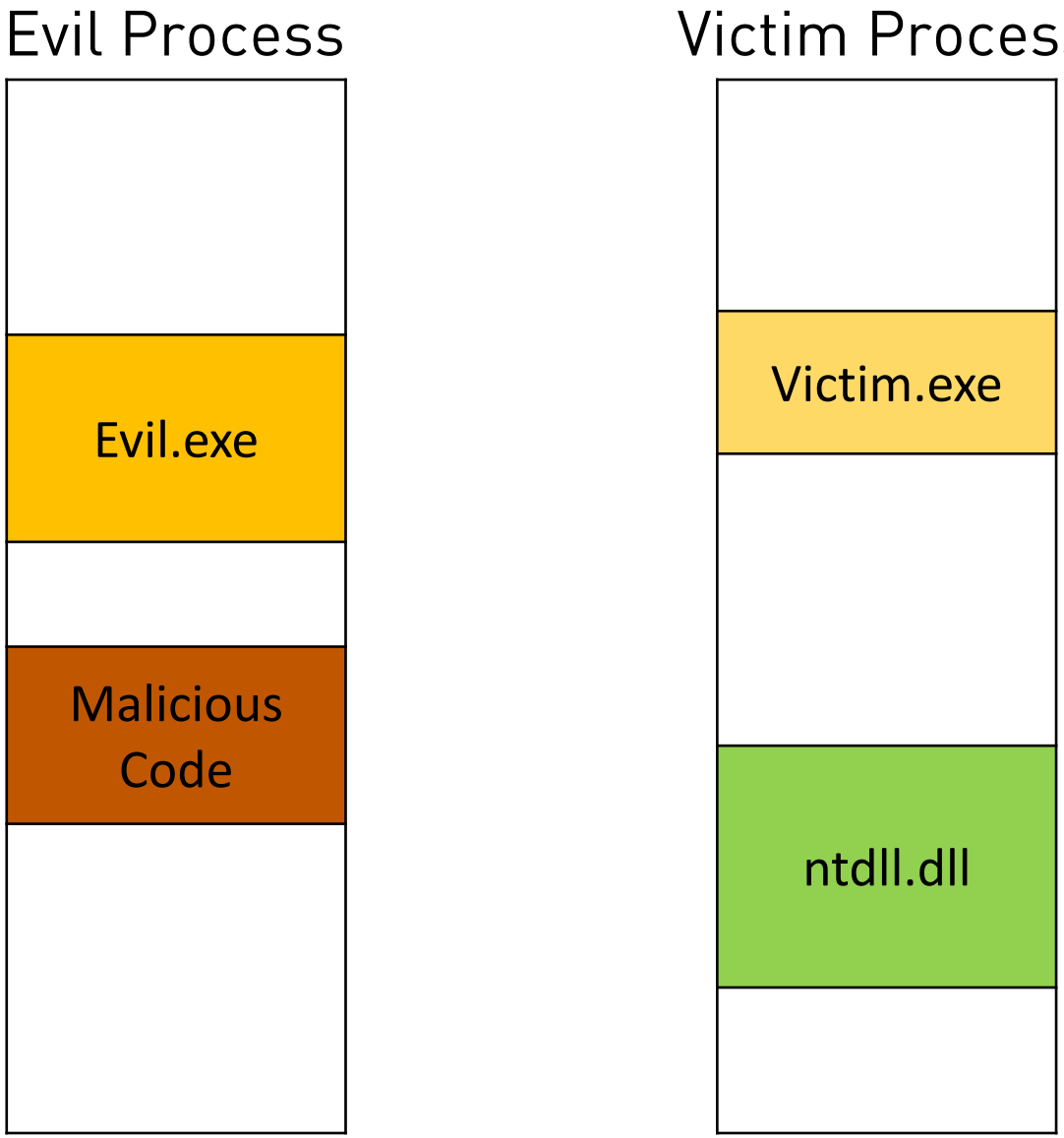
fblock@ernw.de

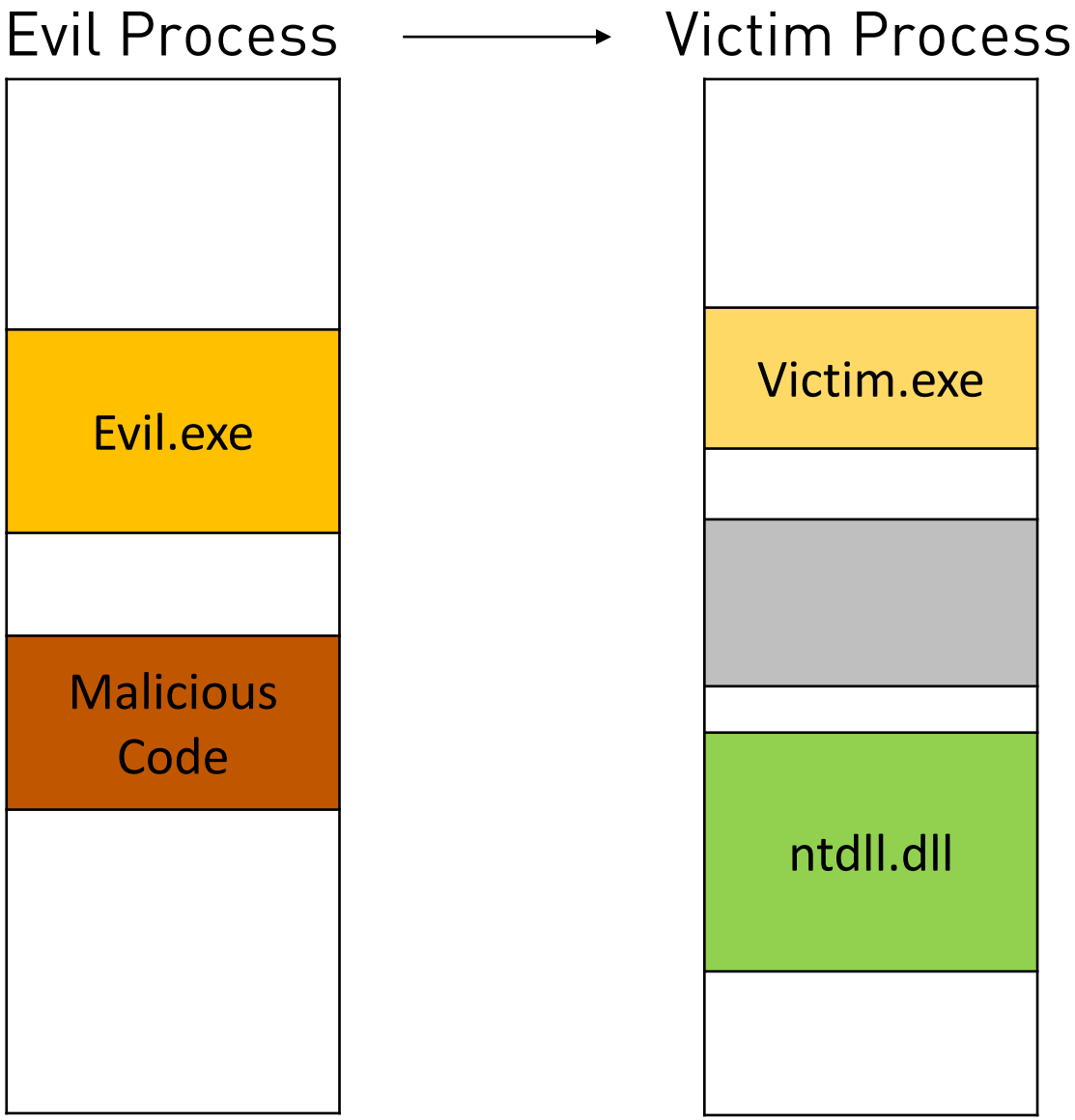
Agenda

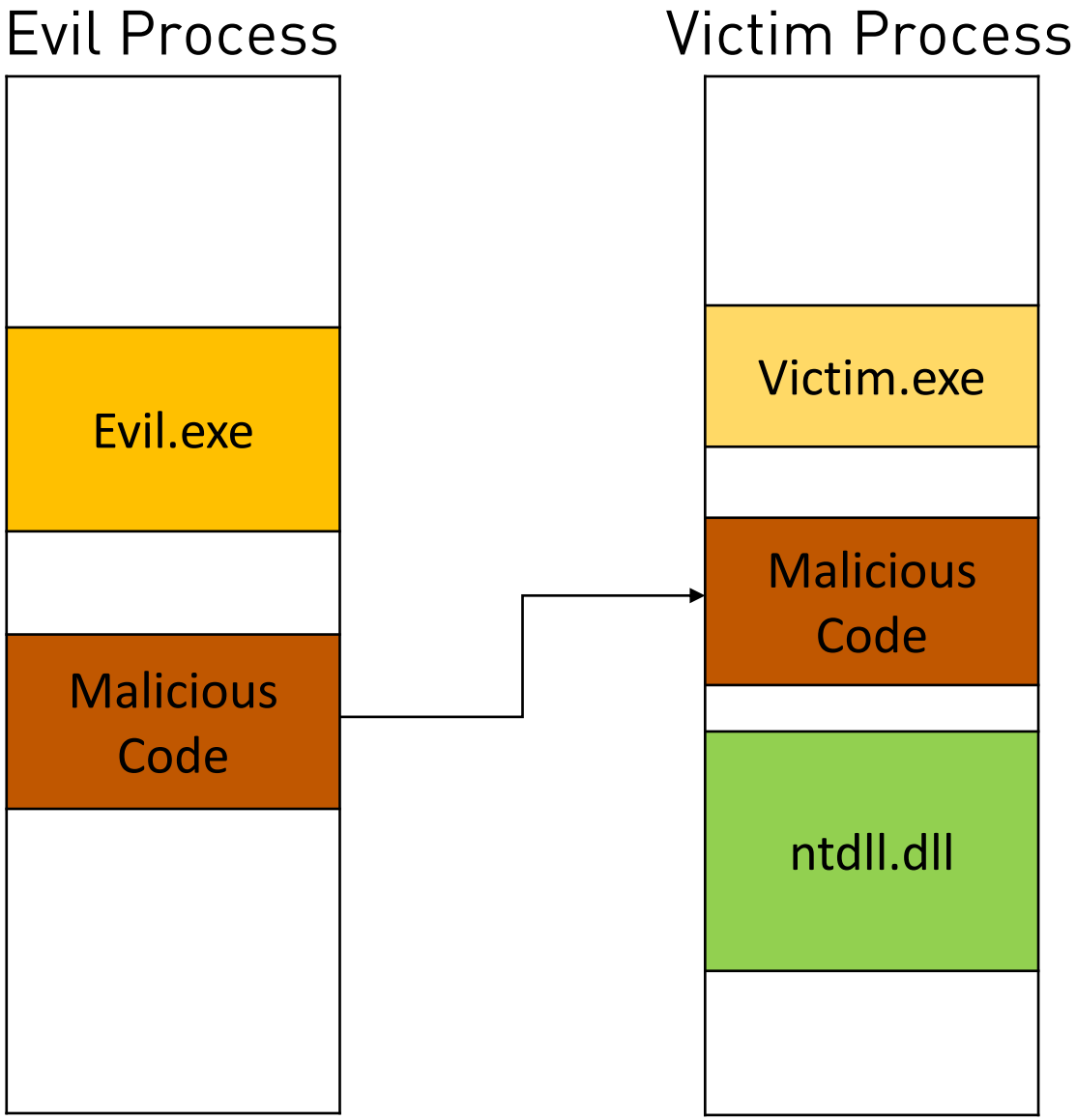
- Short introduction on Code injections
- Motivation
- How to hide
- How to detect
- Conclusion

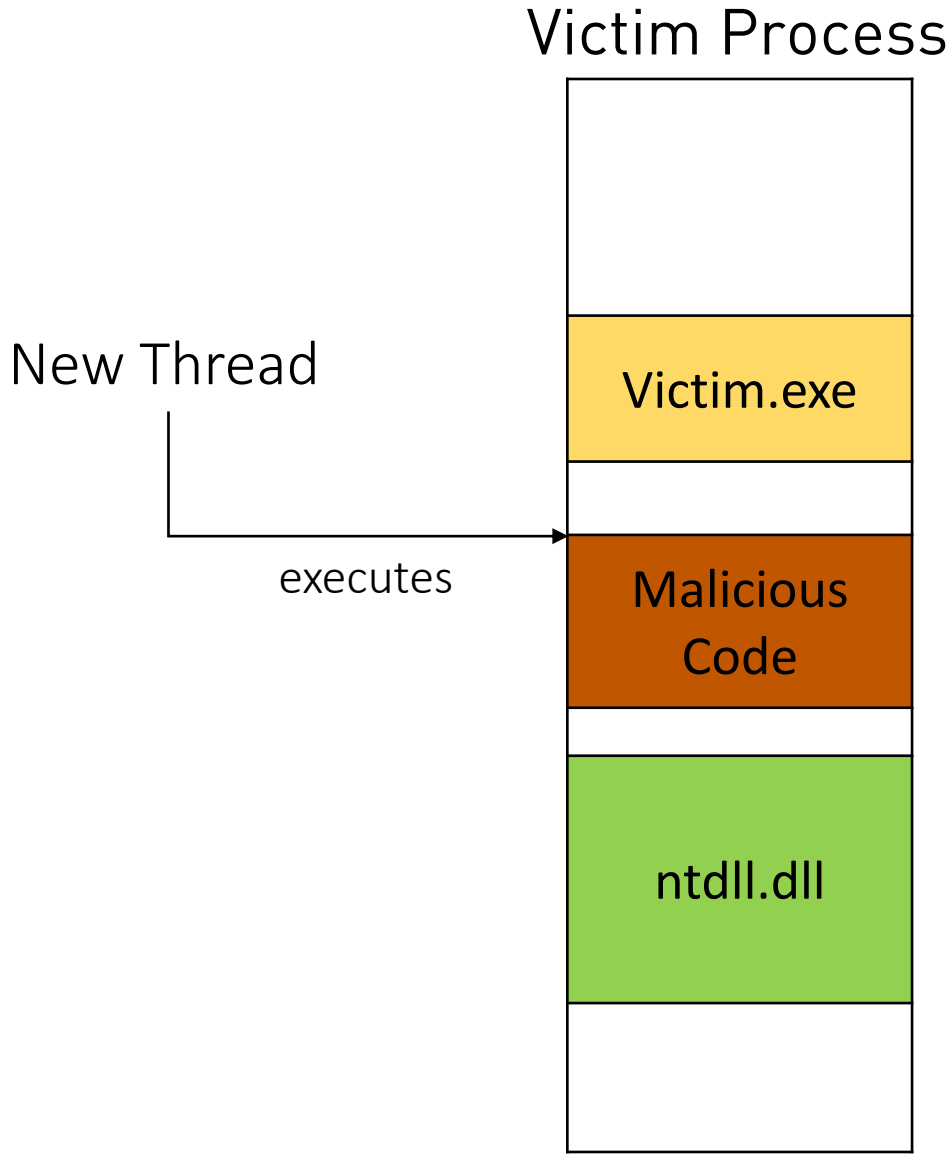
Code Injection: Why and How

- Possible reasons:
  - The parent process might die after exploitation (e.g. heap spraying).
  - Malware does not want to be easily killed by a user (e.g. running ransomware).
  - Stealing/Manipulating data from the target process.
  - Hiding from the user/investigator.
  - …
- A simple and common, but also noisy approach is this API sequence:
  - OpenProcess, VirtualAllocEx, WriteProcessMemory and CreateRemoteThread


Evil Process
Evil.exe
Malicious Code
Victim Process
Victim.exe
ntdll.dll


Evil Process
Victim Process
Evil.exe
Malicious Code
Victim.exe
ntdll.dll


Evil Process
Victim Process
Evil.exe
Malicious Code
Victim.exe
Malicious Code
ntdll.dll

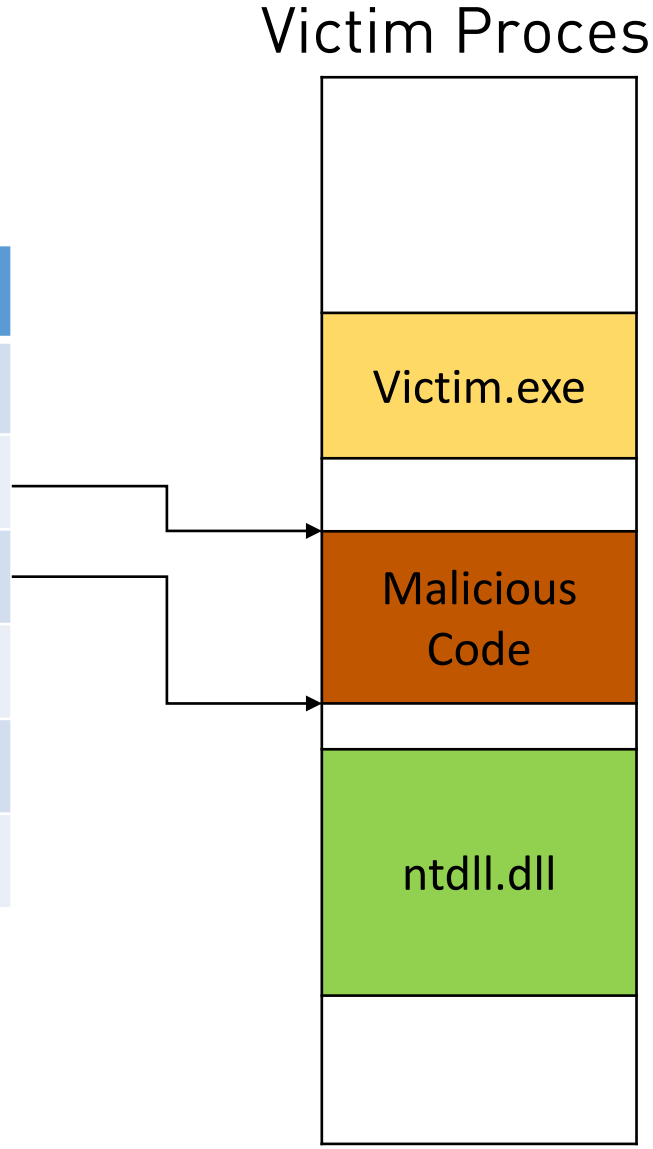

Victim Process
New Thread
executes
Victim.exe
Malicious Code
ntdll.dll

Example malfind Output for Reflective DLL Injection

```
Process: svchost.exe Pid: 3564 Address: 0x13f6cd60000
Vad Tag: VadS Protection: EXECUTE_READWRITE
Flags: PrivateMemory: 1, Protection: 6

 0x13f6cd60000 4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00  MZ..............
 0x13f6cd60010 b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  ........@.......
 0x13f6cd60020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
 0x13f6cd60030 00 00 00 00 00 00 00 00 00 00 00 00 e8 00 00 00  ................
```

VAD struct
…
StartingVpn
EndingVpn
…
u.VadFlags.**ProtectionEnum**
…



The Starting Point for this Research

"One of the most misleading and poorly documented aspects of the **Protection** field from the **VAD** flags is that it's only the **initial protection** specified for all pages in the range when they were first reserved or committed.

Thus, the **current** protection can be **drastically different**." Ligh et. al.[1]

f-block / **ReflectiveDLLInjection**
forked from stephenfewer/ReflectiveDLLInjection

Unwatch 1 | Star 0 | Fork 437

Code | Pull requests 0 | Actions | Projects 0 | Security | Insights | Settings

simple modification that hides injected code from malfind

Browse files

master

f-block committed on Jan 7 Verified 1 parent 24189a1 commit 3df1eca861c4b022d96df847c7e18554e348532b

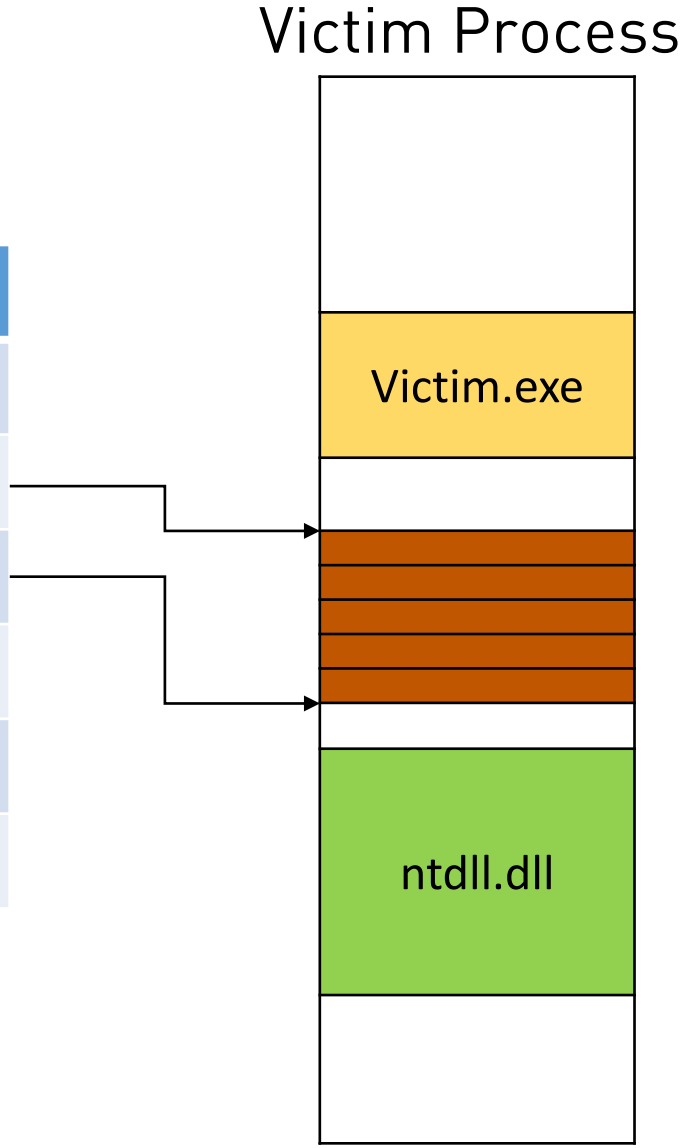
4 inject/src/LoadLibraryR.c

```
7,7 +207,9 @@ HANDLE WINAPI LoadRemoteLibraryR( HANDLE hProcess, LPVOID lpBuffer, DWORD dwLeng
-               lpRemoteLibraryBuffer = VirtualAllocEx( hProcess, NULL, dwLength, MEM_RESERVE|MEM_COMMIT, PAGE_EXECUTE_READWRITE );
+               lpRemoteLibraryBuffer = VirtualAllocEx(hProcess, NULL, dwLength, MEM_RESERVE | MEM_COMMIT, PAGE_READONLY);
+               DWORD oldPrection;
+               VirtualProtectEx(hProcess, lpRemoteLibraryBuffer, dwLength, PAGE_EXECUTE_READWRITE, &oldPrection);
```

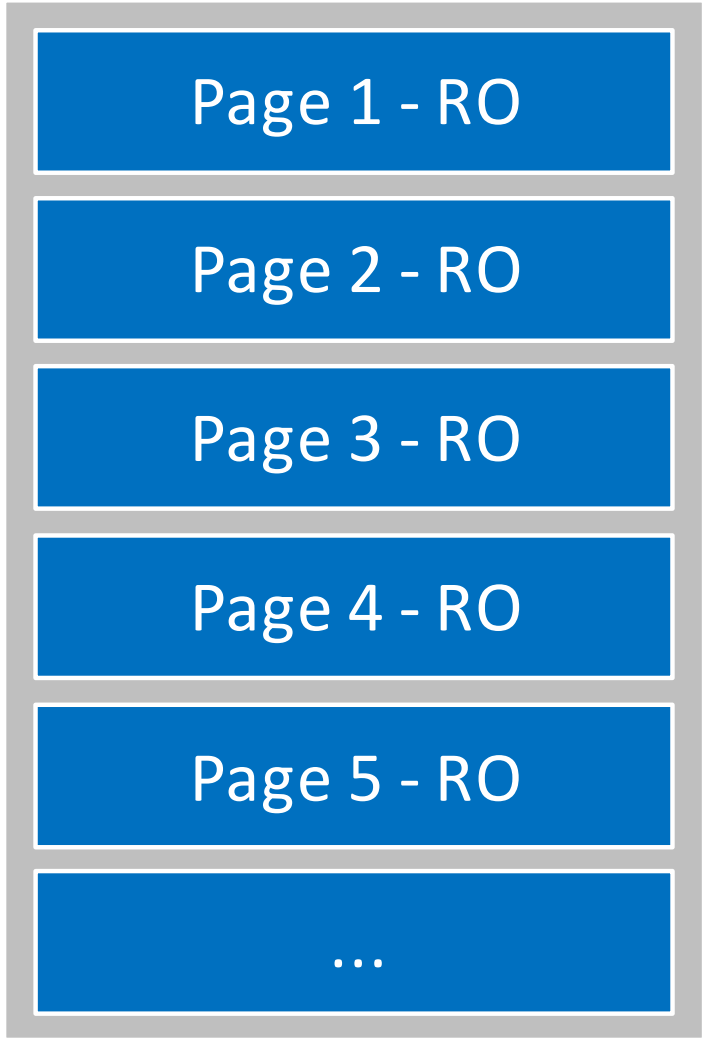
malfind output for modified Reflective DLL Injection

Nothing to see here

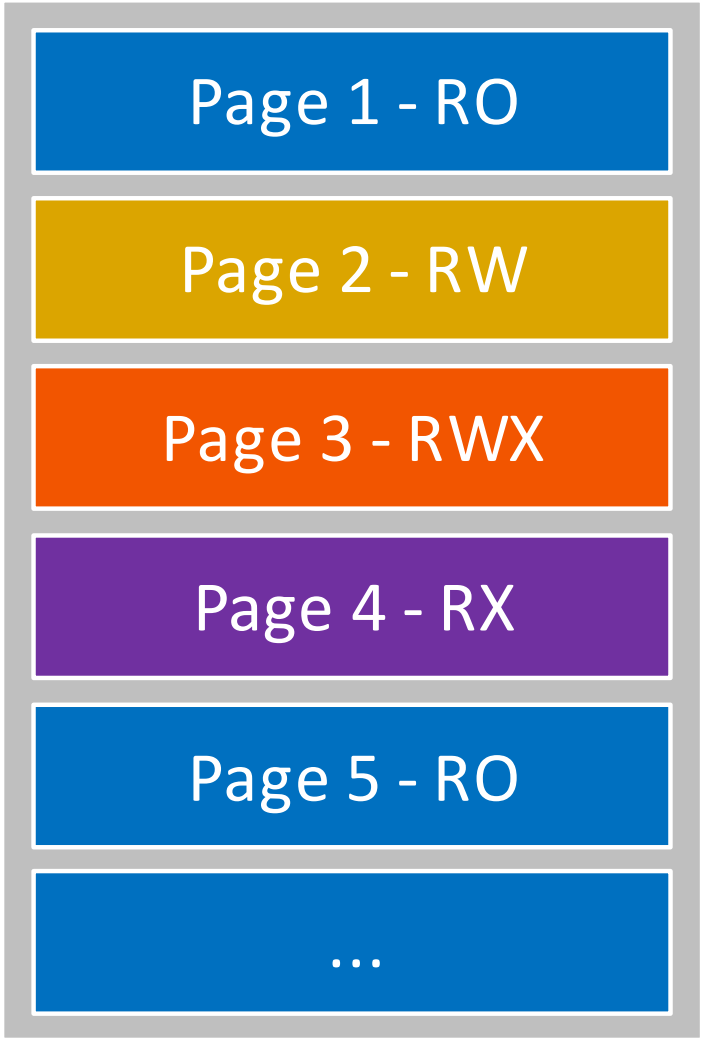
VAD struct
…
StartingVpn
EndingVpn
…
u.VadFlags.ProtectionEnum
…



VAD - Initial Protection : ReadOnly



VAD - Initial Protection : ReadOnly



On trusting VADs contd.

- On the other hand we have the modification of mapped image files.
- Malware can use pages of mapped files for code too: *EXECUTE_WRITECOPY*
- Prior detection techniques at most compared the information from VADs and the PEB (Process Hollowing) or were looking for hooks.
  - One exception: White et. al.[2]

Further Hiding Techniques

- Shared memory with *EXECUTE_WRITECOPY* protection
- Mapped data files
- Paged out pages: (un)intentional hiding

Current detection plugins

- Detection mainly based on VADs/memory
  - malfind
  - hashtest
- Detection mainly based on other criteria (e.g. threads)
  - threadmap
  - malthfind
  - hollowfind
  - malfofind
  - Psinfo
  - gargoyle

Hiding results

- With the VirtualAllocEx/VirtualProtectEx trick we've successfully hidden injected code from *malfind*, *hashtest* and *Psinfo*.
- With paged out pages we've successfully hidden injected code from *malfind*, *hashtest*, *Psinfo* and *malthfind*.
- With shared memory/mapped data files with *EXECUTE_WRITECOPY* protection, we've successfully hidden injected code from *malfind*.
- Only *hollowfind*, *malfofind* and *Psinfo* were unimpressed by the hiding techniques in regards to Process Hollowing.

There are various flavors of Code Injections

- APC Injections
- Process Hollowing
- AtomBombing
- Reflective DLL Injection
- …

- All have one aspect in common: They result in new/modified code/data in the target process's domain.

What are we looking for?

- Rootkit Paradox (Kornblum[3])
  - In Essence: While the rootkit tries to hide its existence, in order to do nasty stuff, its code must (at least once) be **locatable** and **executable**.

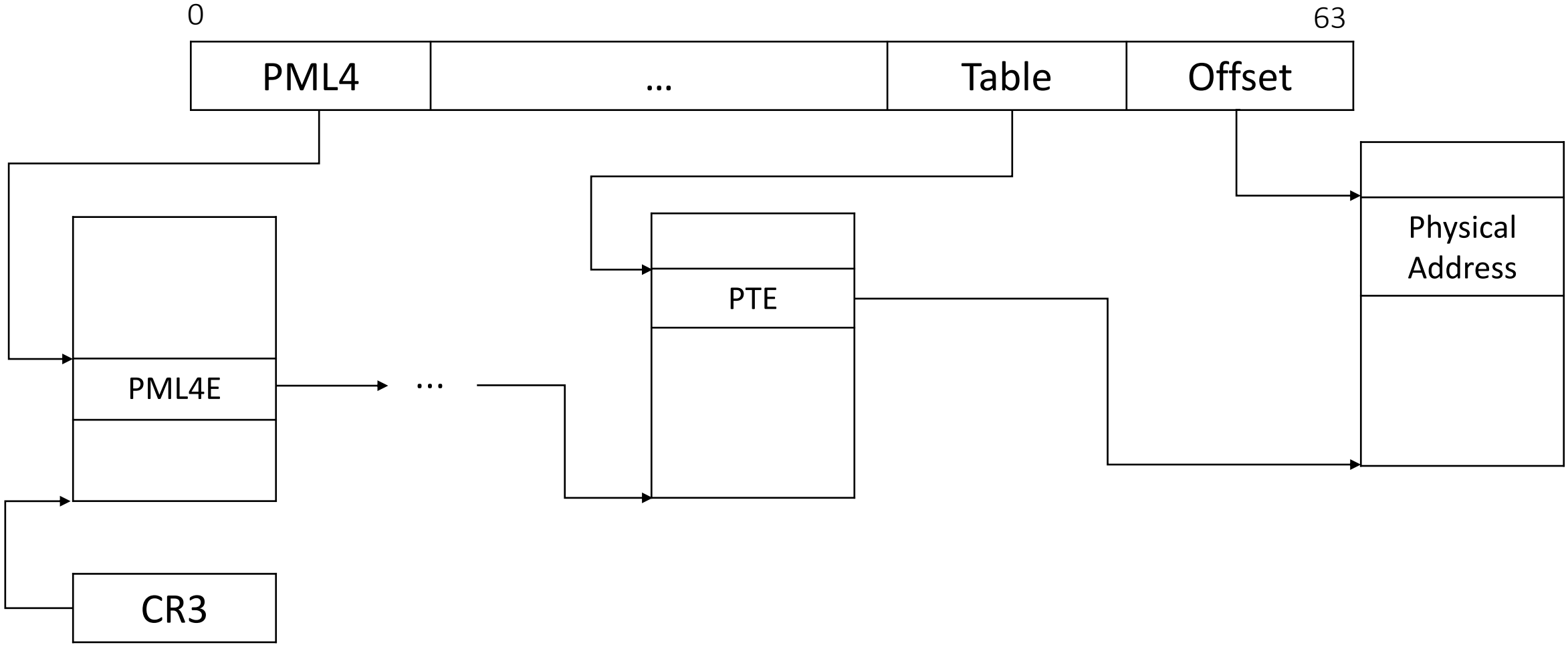
- So, the goal is to identify any executable data in user space.

PTEs and the PFN Database

- PTE (Page Table Entry)
  - 64bit (x64/x86-PAE) sized “struct”, defining a physical page (if valid).
  - “The final truth”, as the CPU’s decision on reading/writing/executing data from a given address is dependent on the bits in its PTE.

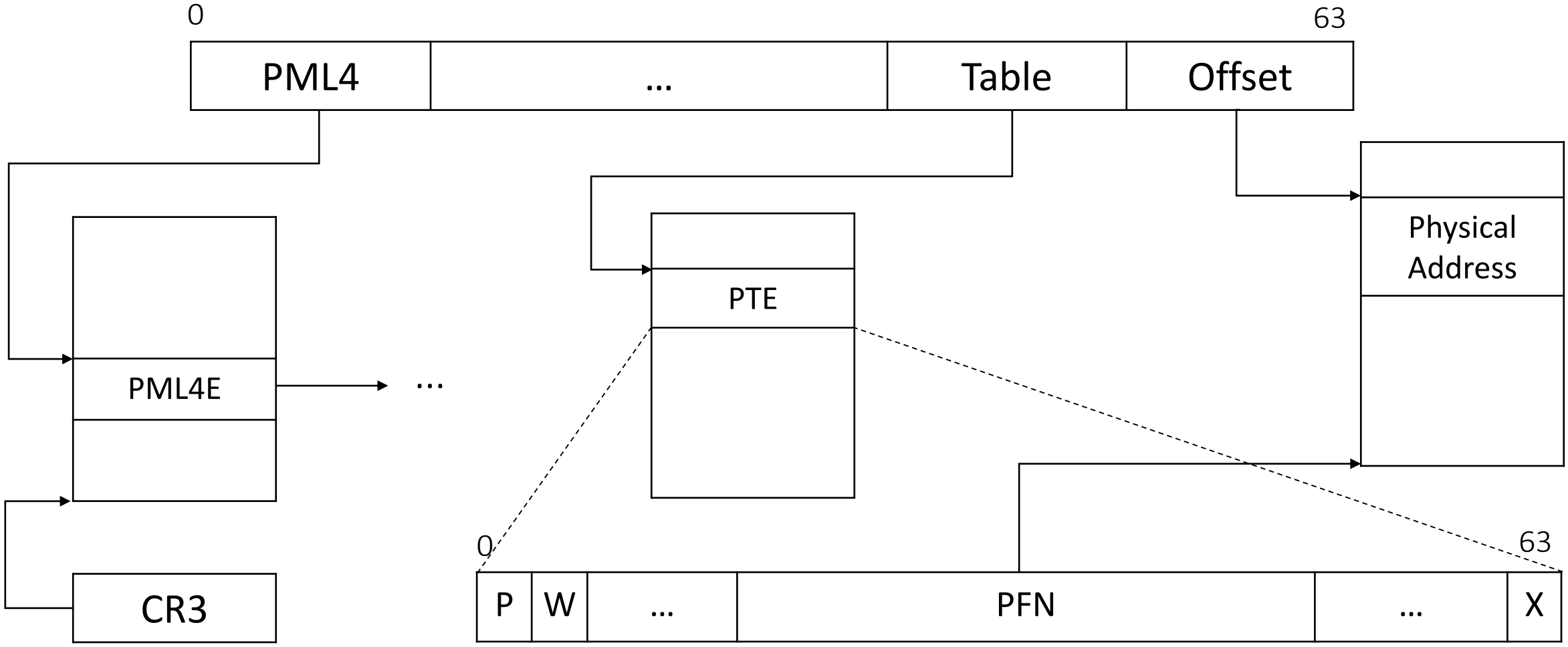
- PFN Database is the physical point of view on the available pages.
  - In our case mainly used to answer one question: Has this page been modified?

Virtual Address

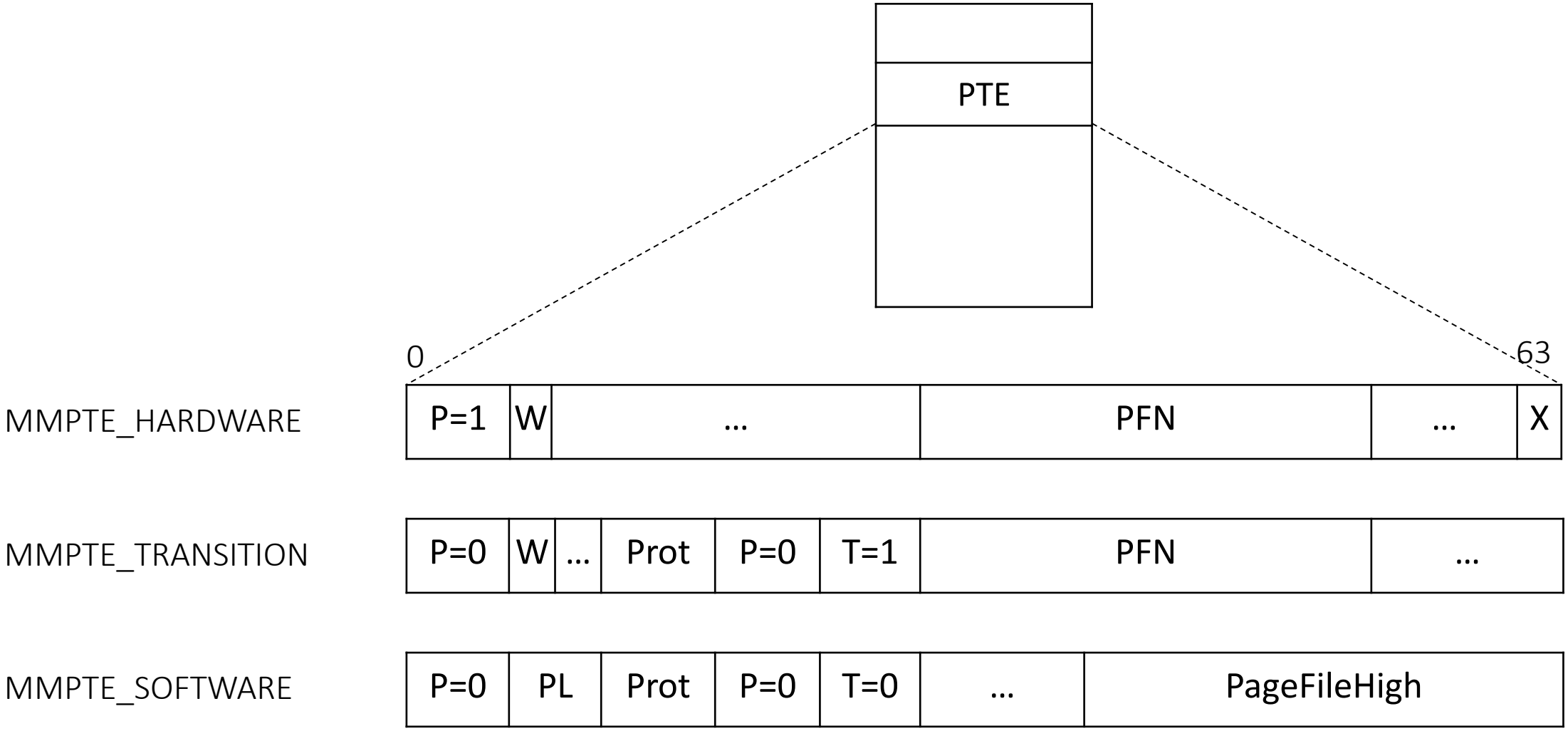


Source: Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A

Virtual Address



Source: Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A

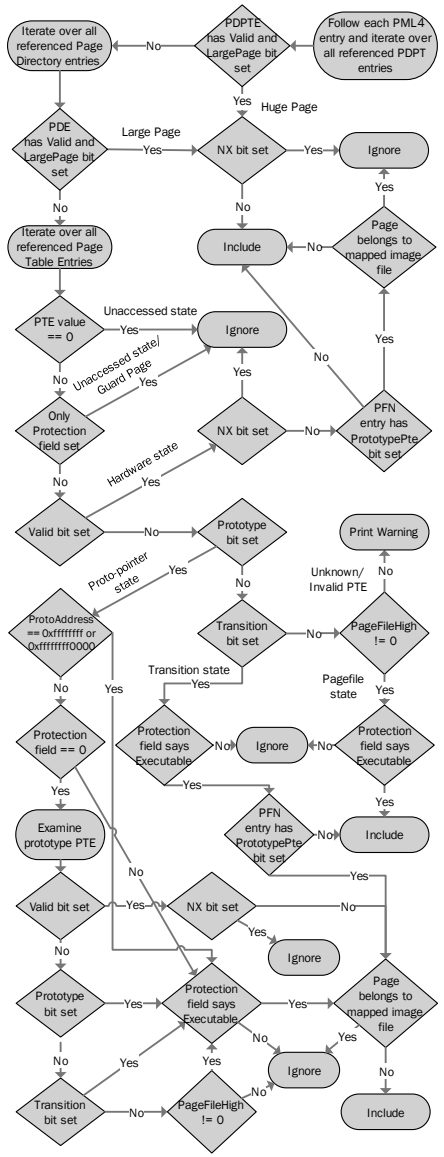

PTE
0
63
MMPTE_HARDWARE
P=1
W
…
PFN
…
X
MMPTE_TRANSITION
P=0
W
…
Prot
P=0
T=1
PFN
…
MMPTE_SOFTWARE
P=0
PL
Prot
P=0
T=0
…
PageFileHigh

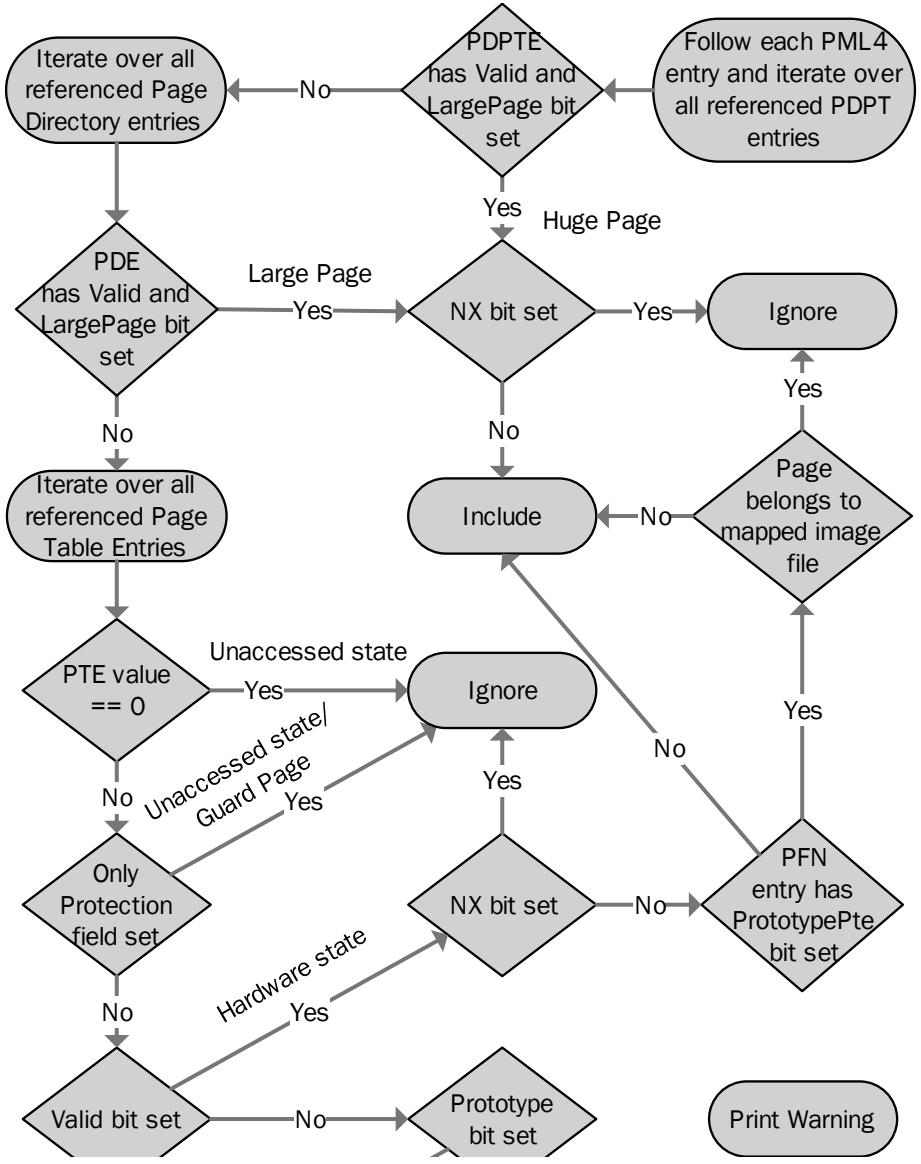
and more

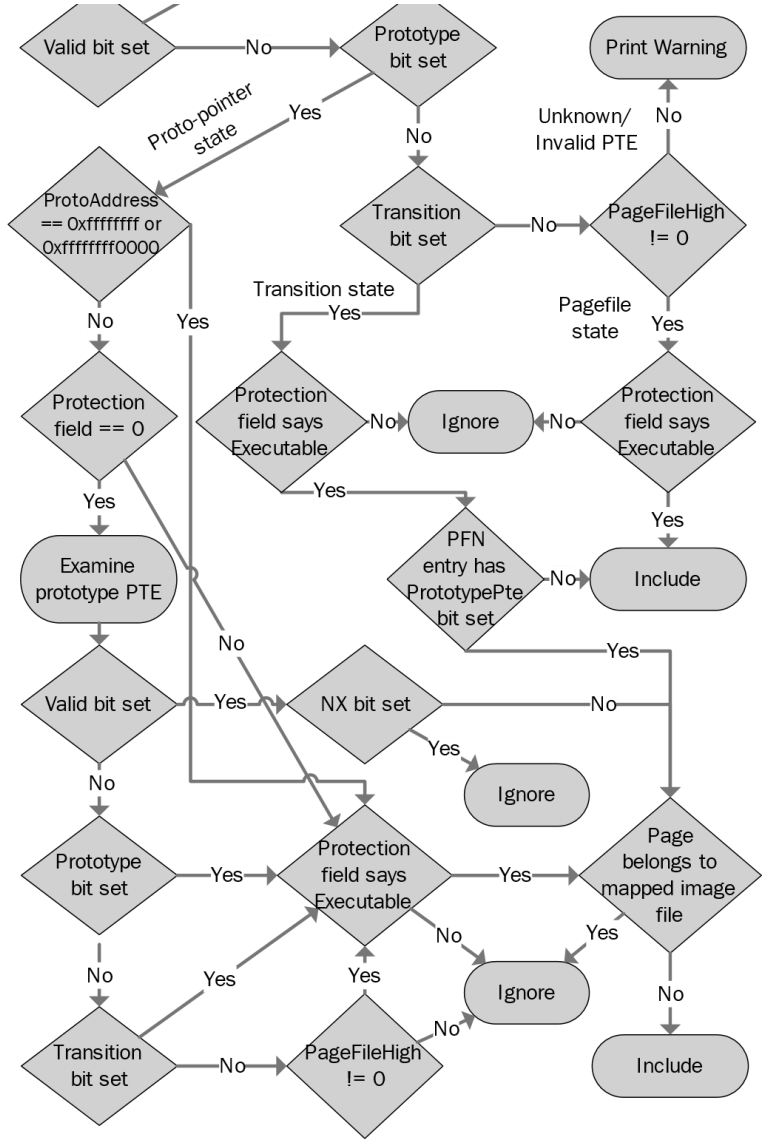
PTEs and the PFN Database

- So what can we detect with those?
  - Executable pages in general, no matter where they are (in mapped files, not related to any file, swapped out, ...).
    - E.g. executed code on the stack in a DEP disabled process.
  - Executable **and** Modified pages for mapped image files.

- And how?


Iterate over all referenced Page Directory entries
No
PDPTE has Valid and LargePage bit set
Follow each PML4 entry and iterate over all referenced PDPT entries
Yes
Huge Page
PDE has Valid and LargePage bit set
Large Page
Yes
NX bit set
Yes
Ignore
Yes
No
No
Iterate over all referenced Page Table Entries
Include
No
Page belongs to mapped image file
Yes
PTE value == 0
Unaccessed state
Yes
Ignore
No
Unaccessed state/ Guard Page
Yes
Yes
No
Only Protection field set
NX bit set
No
PFN entry has PrototypePte bit set
Yes
No
Hardware state
Yes
Valid bit set
No
Prototype bit set
Print Warning
Proto-pointer state
Yes
Unknown/ Invalid PTE
No
No
ProtoAddress == 0xffffffff or 0xfffffff0000
Transition bit set
No
PageFileHigh != 0
Transition state
Yes
Pagefile state
Yes
No
Yes
Protection field == 0
Protection field says Executable
No
Ignore
No
Protection field says Executable
Yes
Yes
Yes
PFN entry has PrototypePte bit set
No
Include
Examine prototype PTE
No
Yes
Valid bit set
Yes
NX bit set
No
Yes
Yes
Ignore
No
Prototype bit set
Yes
Protection field says Executable
Yes
Page belongs to mapped image file
No
Yes
Yes
Yes
Ignore
No
Transition bit set
No
PageFileHigh != 0
No
Include

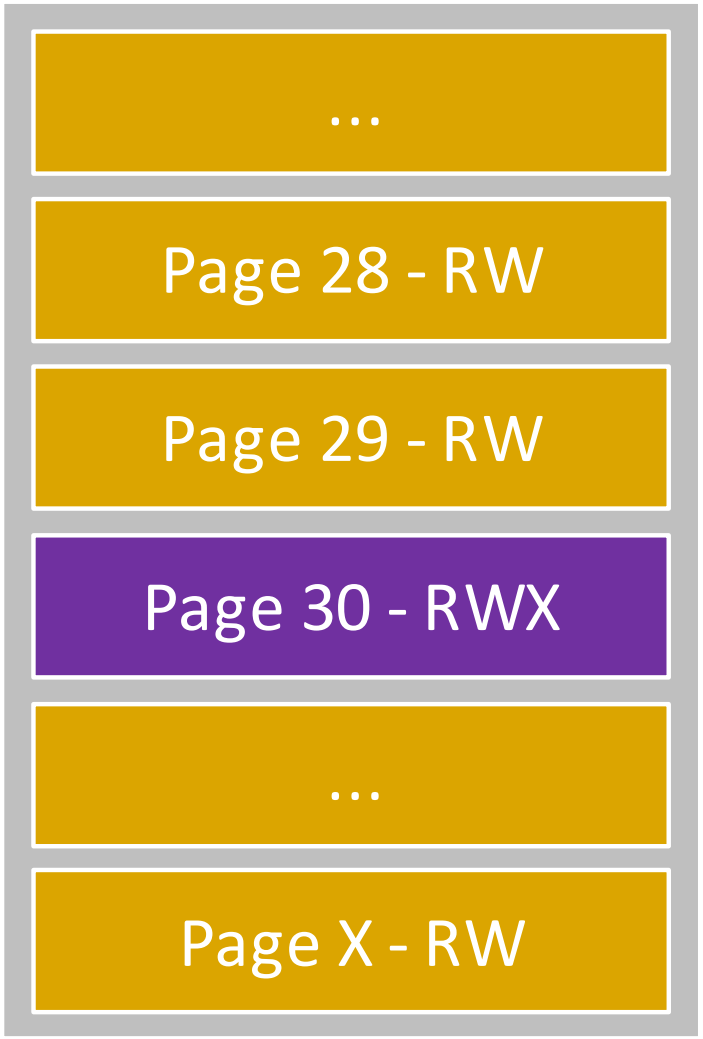

Iterate over all referenced Page Directory entries
No
PDPTE has Valid and LargePage bit set
Follow each PML4 entry and iterate over all referenced PDPT entries
Yes
Huge Page
PDE has Valid and LargePage bit set
Large Page
Yes
NX bit set
Yes
Ignore
Yes
No
No
Iterate over all referenced Page Table Entries
Include
No
Page belongs to mapped image file
Unaccessed state
PTE value == 0
Yes
Ignore
Yes
No
Unaccessed state/ Guard Page
Yes
Yes
No
Only Protection field set
NX bit set
No
PFN entry has PrototypePte bit set
Hardware state
No
Yes
Valid bit set
No
Prototype bit set
Print Warning


Valid bit set
No
Prototype bit set
Print Warning
Proto-pointer state
Yes
No
Unknown/ Invalid PTE
No
ProtoAddress == 0xffffffff or 0xffffffff0000
Transition bit set
No
PageFileHigh != 0
Transition state
Yes
Pagefile state
Yes
No
Yes
Protection field == 0
Protection field says Executable
No
Ignore
No
Protection field says Executable
Yes
Yes
Yes
Examine prototype PTE
PFN entry has PrototypePte bit set
No
Include
No
Yes
Valid bit set
Yes
NX bit set
No
Yes
Ignore
No
Prototype bit set
Yes
Protection field says Executable
Yes
Page belongs to mapped image file
Yes
No
Yes
Yes
Ignore
No
No
Transition bit set
No
PageFileHigh != 0
No
Include

Case study DEP

- When DEP is not active for a running process, code can get executed from pages with e.g. READWRITE protection.
- But per default, all non-executable pages have still the NX bit set.
- If instructions should be fetched from such a page, an access violation occurs and the OS takes over.
- Windows will then unset the NX bit for that particular page and the CPU can fetch instructions from it.
- This makes it easy with our approach to identify those, as they stand out.

Stack



ptenum Output for executed Code on Stack

```
Process: messagebox.exe Pid: 2508 Address: 0x460000
Vad Tag: VadS Protection: READWRITE
Flags: PrivateMemory: 1, Protection: 4
The Vadtype is: Private
1 non empty page(s) with a total size of 0x1000 bytes in this VAD were executable
(and for mapped image files also modified).
Skipping the first 0x1ff000 bytes, as they are either not modified (only applies
for mapped image files), empty or not executable.

    0x65f000 00 00 00 00 90 f1 65 00 c7 15 6c 76 01 00 00 00  ......e...lv....
    0x65f010 00 00 00 00 96 14 6c 76 57 0c 01 43 09 00 80 01  ......lvW..C....
    0x65f020 a9 14 6c 76 78 f0 65 00 00 00 00 00 00 00 00 00  ..lvx.e.........
    0x65f030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

ptenum Output for Reflective DLL Injection

```
Process: notepad.exe Pid: 4284 Address: 0x1400dea0000
Vad Tag: VadS Protection: READONLY
Flags: PrivateMemory: 1, Protection: 1
The Vadtype is: Private
22 non empty page(s) with a total size of 0x16000 bytes in this VAD were executable
(and for mapped image files also modified).
Skipping the first 0x1000 bytes, as they are either not modified (only applies for
mapped image files), empty or not executable.

 0x1400dea1000 4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00  MZ..............
 0x1400dea1010 b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  ........@.......
 0x1400dea1020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
 0x1400dea1030 00 00 00 00 00 00 00 00 00 00 00 00 10 01 00 00  ................
```

ptenum Output for Reflective DLL Injection with paged out Pages

```
Process: notepad.exe Pid: 4284 Address: 0x1400dea0000

Vad Tag: VadS Protection: READONLY

PrivateMemory: 1, Protection: 1

The Vadtype is: Private

22 non empty page(s) with a total size of 0x16000 bytes in this VAD were
executable (and for mapped image files also modified).

Seems like all executable pages from this VAD are not available from the
memory dump (e.g. because they have been paged out). So there is nothing to
dump/disassemble here.
```

Limitations & Future Work

- This approach does not detect injected code/data in non executable pages – e.g. gargoyle.
- Does not work with paged out Paging Structures and no pagefile given (could do a fallback to *malfind* like approach – is however again prone to attacks).
- The amount of data to examine can be huge, mainly because of modified pages of mapped image files.

- Plugin is however suitable as:
  - Improved malfind (with the *--ignore_image_files* option).
  - Before/After comparison.
- Usage in existing code injection plugins to improve their results.
- Volatility version will be coming soon.

Black Hat Sound Bytes

- It is possible to hide from current code injection plugins with simple tricks.
  - -> Don’t blindly trust VADs.

- To find executable memory reliably, Page Table Entries must be examined.

- There is a Rekall plugin now (*ptenum*) that does that for you.

Online Resources

- Links to the most current version of the *ptenum* plugin and all tools/data to reproduce the research results can be found here: https://github.com/f-block/BlackHat-EU-2019

- The research paper by me and Andreas Dewald, with more details, can be found here: https://www.sciencedirect.com/science/article/pii/S1742287619301574

I wanted to thank

- The people behind Rekall for their amazing work.
- The authors of Volatility and “The Art Of Memory Forensics”.
- The authors of all mentioned detection plugins.
- Last but not least Enrico Martignetti for his great book on Windows memory management: “What Makes It Page? The Windows 7 (X64) Virtual Memory Manager”

Thank you for your Attention

Questions/Criticism/Remarks/Suggestions?

Sources

- [1] Ligh, M.H., Case, A., Levy, J., Walters, A., 2014. The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory
- [2] Andrew White, Bradley Schatz, Ernest Foo, 2013. Integrity Verification of User Space Code, https://dfrws.org/file/206/download?token=jDpt_E9p
- [3] Jesse Kornblum, 2006. https://pdfs.semanticscholar.org/dd79/86995b903a9c1ba16e228f6debfc3cf539cc.pdf