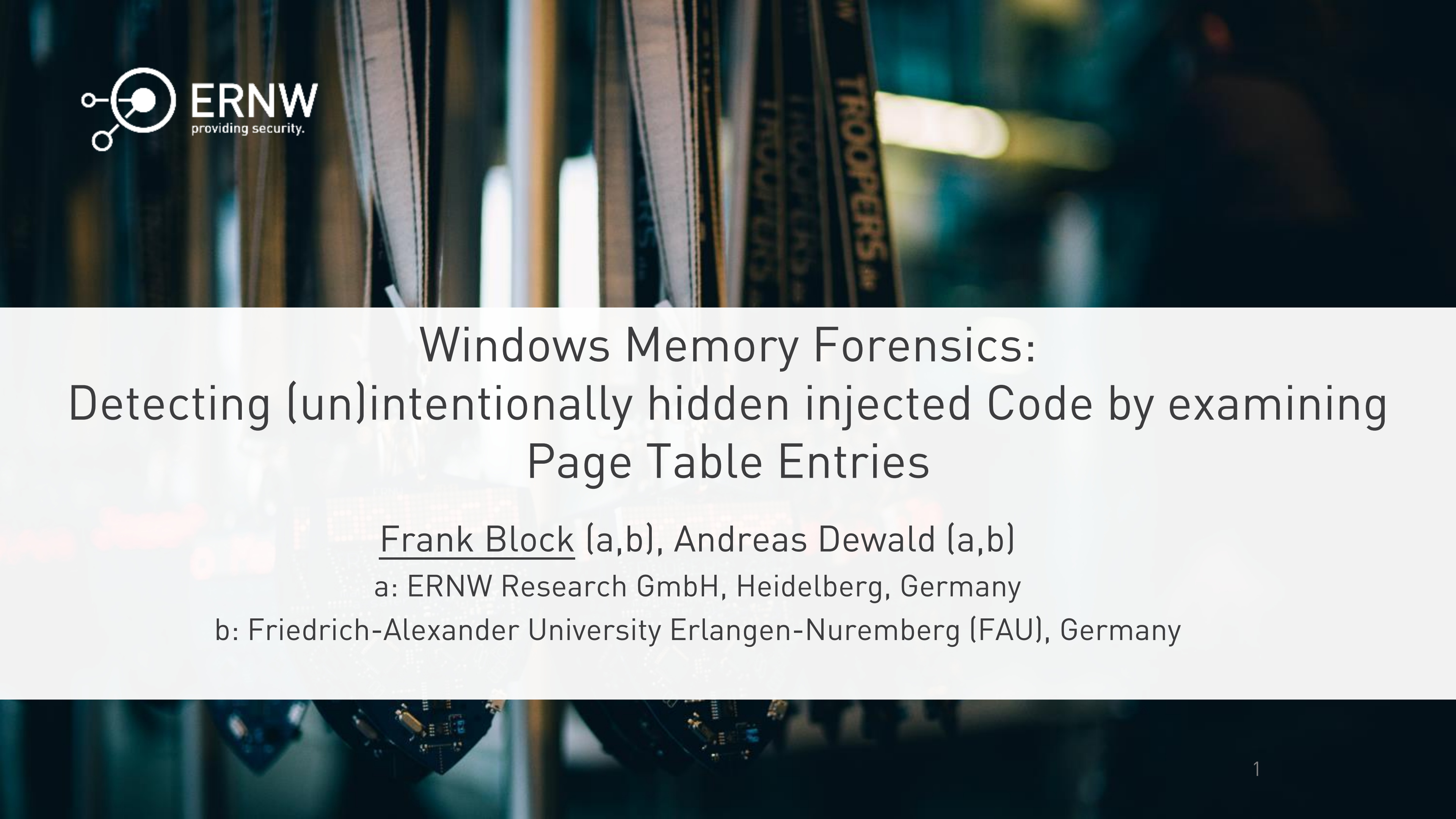
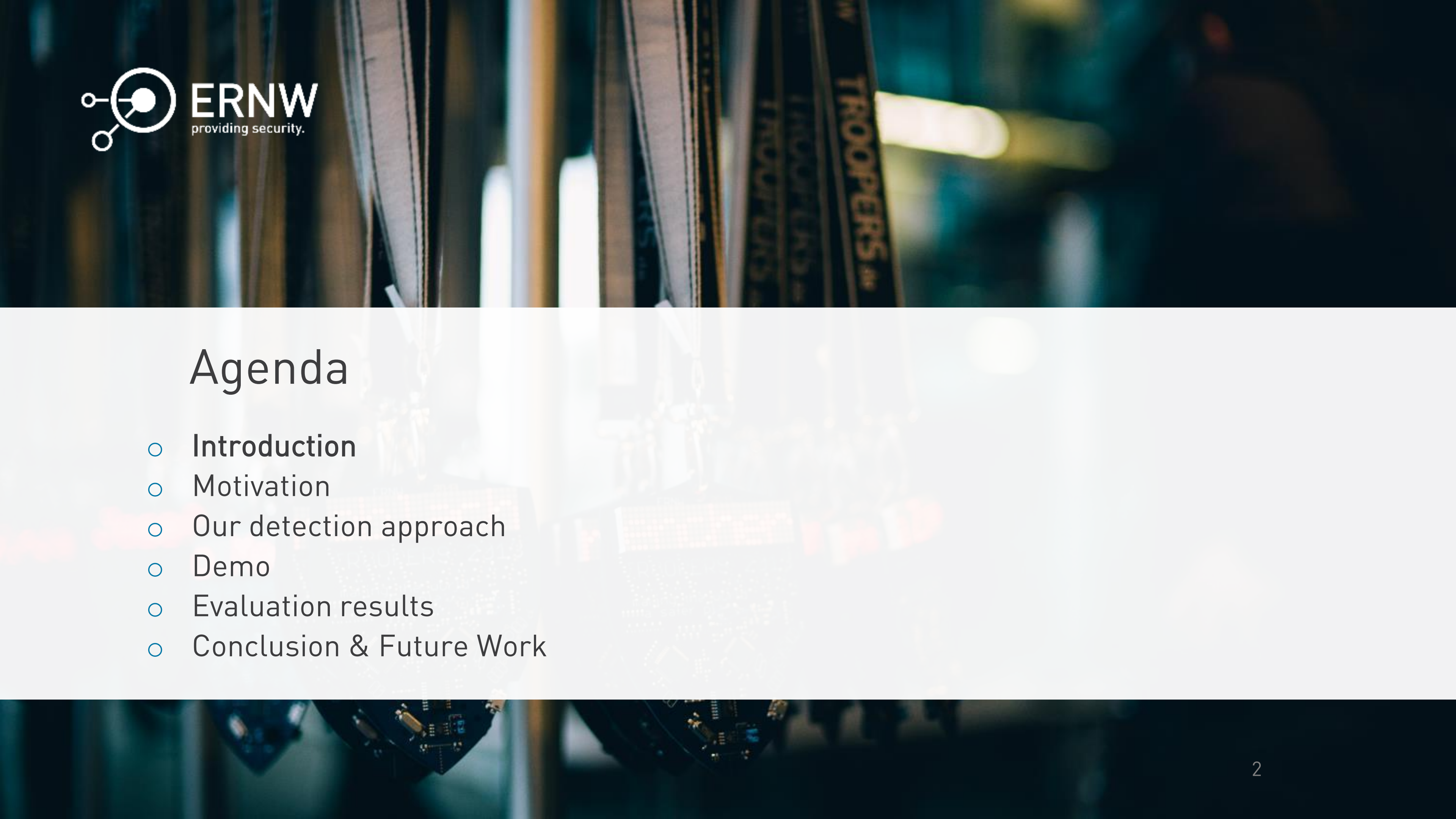


Windows Memory Forensics: Detecting (un)intentionally hidden injected Code by examining Page Table Entries

Frank Block (a,b), Andreas Dewald (a,b)

a: ERNW Research GmbH, Heidelberg, Germany

b: Friedrich-Alexander University Erlangen-Nuremberg (FAU), Germany

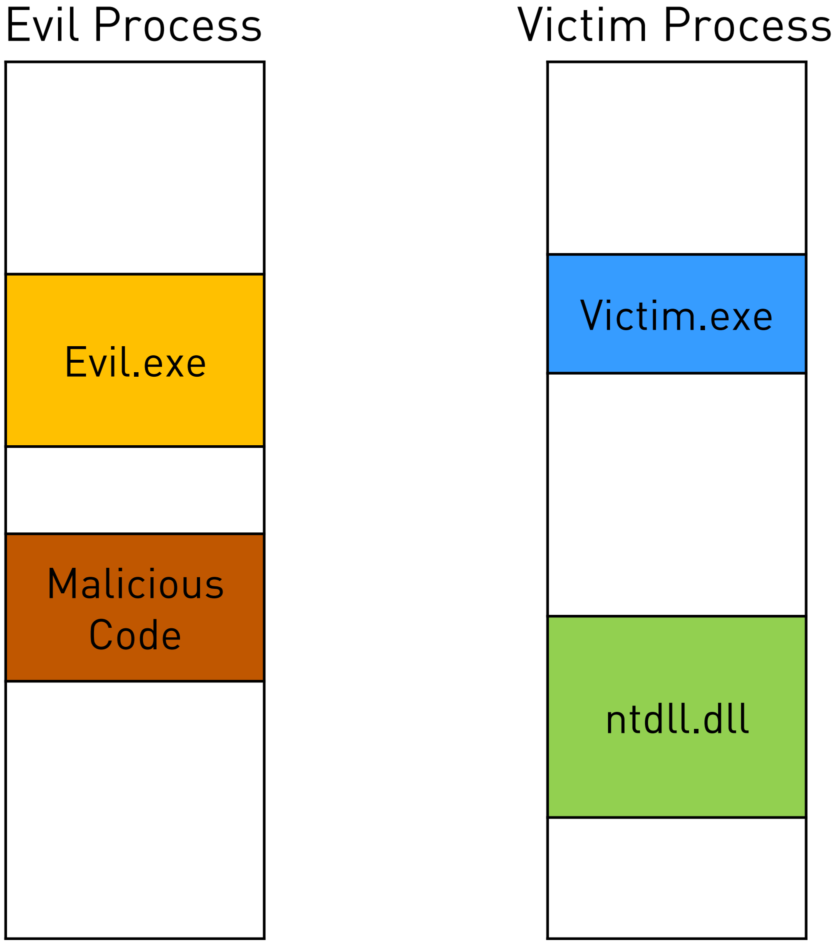


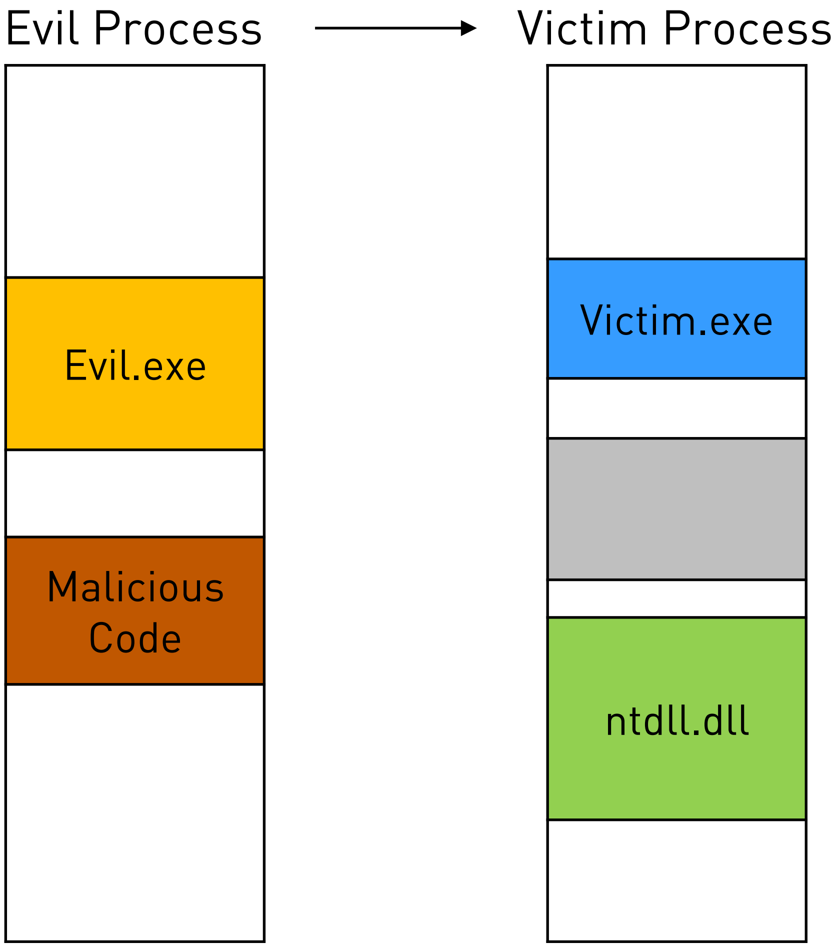
Agenda

- **Introduction**
- Motivation
- Our detection approach
- Demo
- Evaluation results
- Conclusion & Future Work

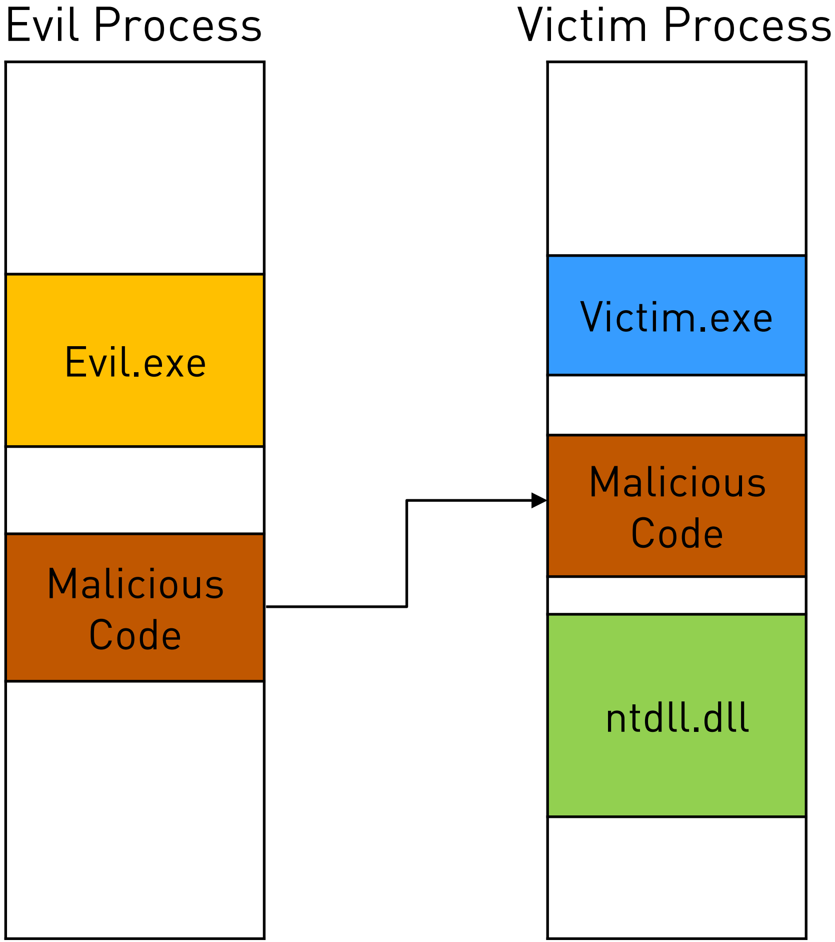
Code Injection: Why and How

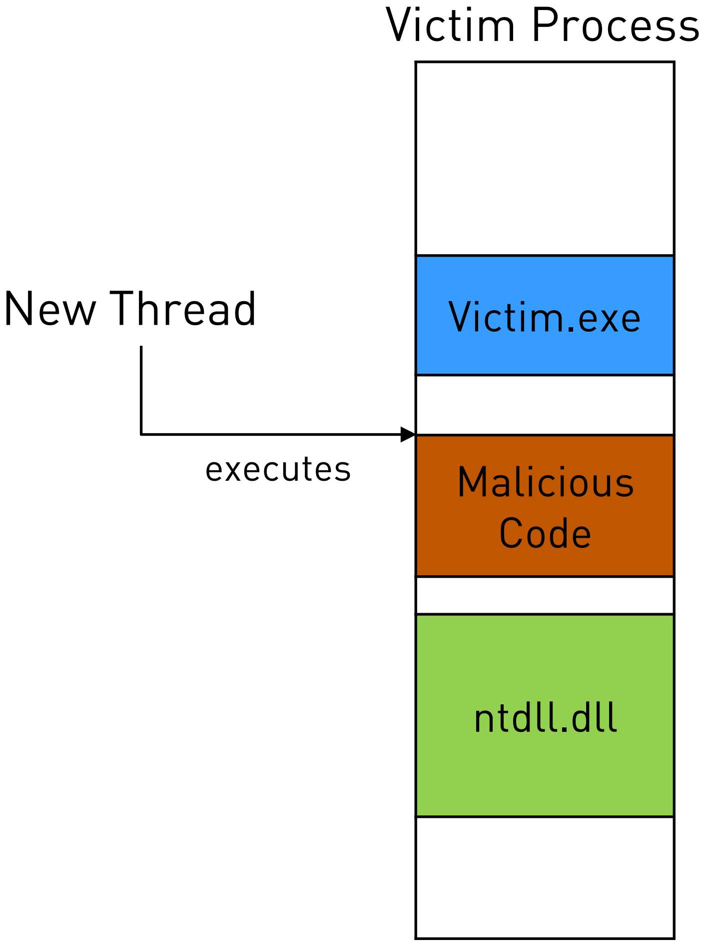
- Possible reasons:
  - The parent process might die after exploitation (e.g. heap spraying).
  - Malware does not want to be easily killed by a user (e.g. running ransomware).
  - Stealing/Manipulating data from the target process.
  - Hiding from the user/investigator.
  - ...
- A simple and common, 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

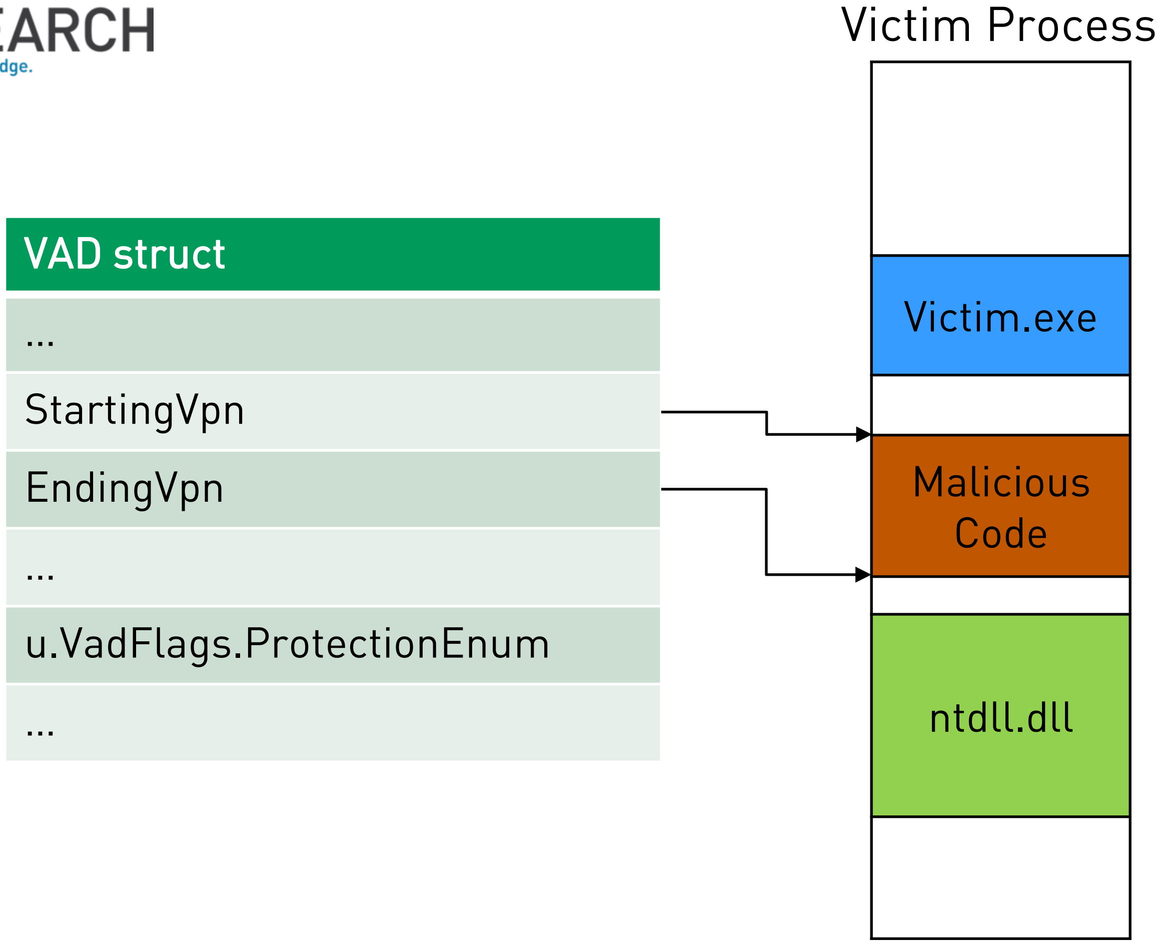

Evil Process
Victim Process
Evil.exe
Malicious Code
Victim.exe
Malicious Code
ntdll.dll


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

Example malfind output

```
Process: rs_target.exe Pid: 4748 Address: 0xc00000
Vad Tag: VadS Protection: EXECUTE_READWRITE
Flags: PrivateMemory: 1, Protection: 6

      0xc00000 b8 e0 20 a7 98 db d1 d9 74 24 f4 5a 29 c9 b1 42  ........t$.Z)..B
      0xc00010 31 42 12 83 c2 04 03 42 0e e2 f5 d9 eb 9b d9 74  1B.....B.......t
      0xc00020 24 f4 31 d2 b2 77 31 c9 64 8b 71 30 8b 76 0c 8b  $.1..w1.d.q0.v..
      0xc00030 76 1c 8b 46 08 8b 7e 20 8b 36 38 4f 18 75 f3 59  v..F..~..68O.u.Y
```



VAD struct

...

StartingVpn

EndingVpn

...

u.VadFlags.ProtectionEnum

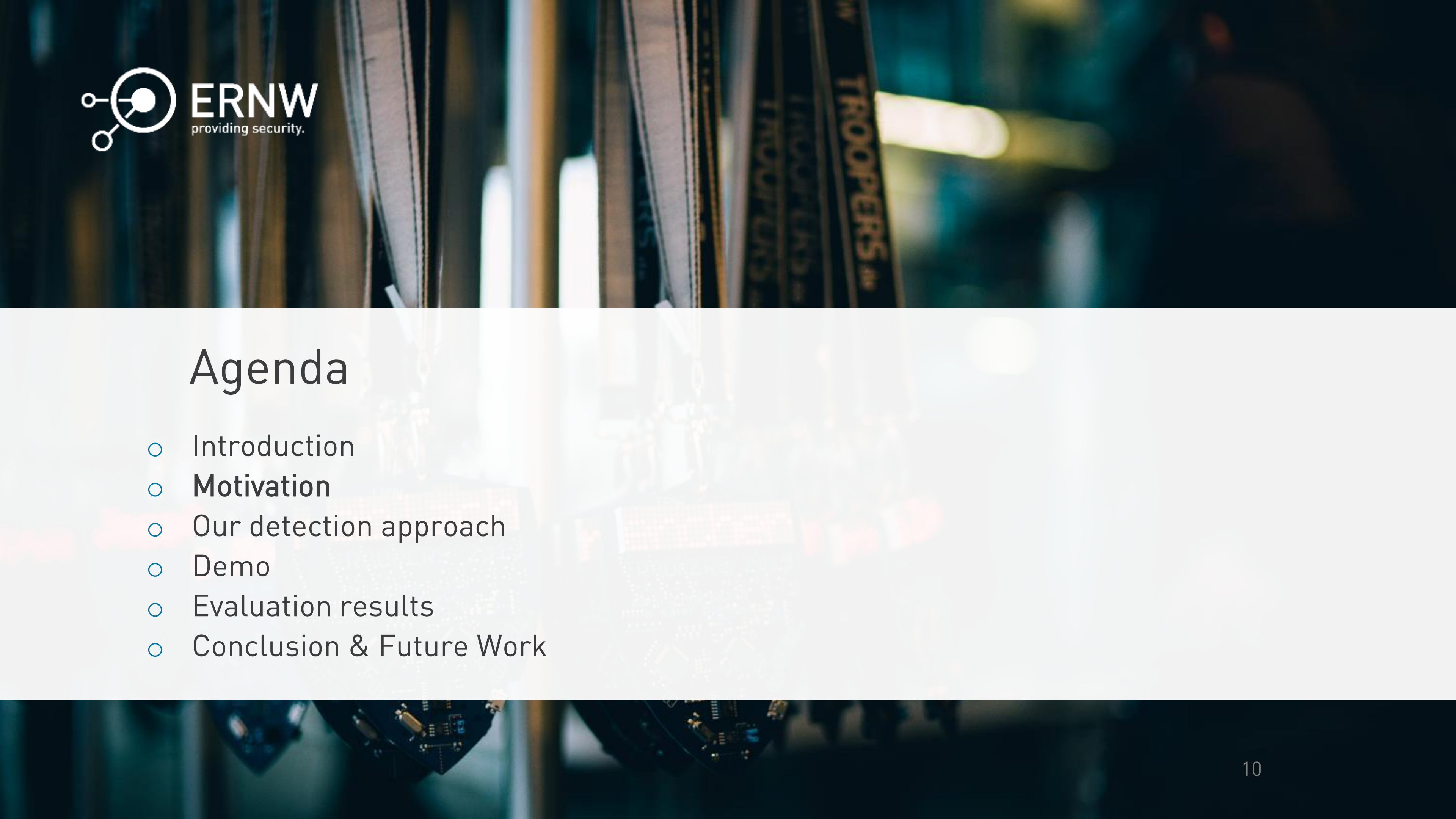
...

Victim Process

Victim.exe

Malicious Code

ntdll.dll



Agenda

- Introduction
- **Motivation**
- Our detection approach
- Demo
- Evaluation results
- Conclusion & Future Work

The Starting Point for this Research

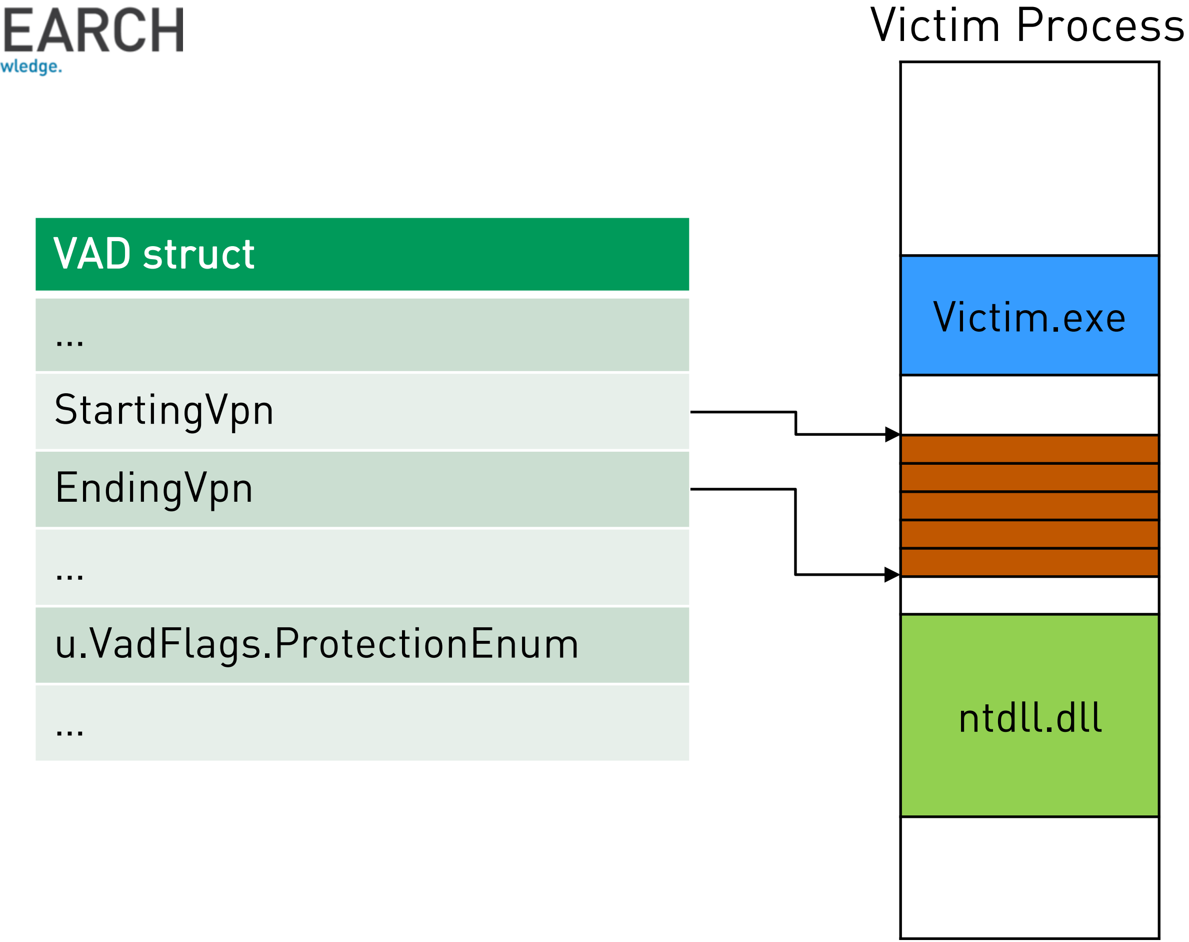
"One of the most misleading and poorly documented aspects of the **Protection** field from the **VAD** flags is that it's only the **initial protection** specified for all pages in the range when they were first reserved or committed.

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

On trusting VADs

- A VAD holds, for some of its meta data, only the initial state:
  - Protection
  - Mapped file (in regards to the content of its pages)
- But this state or the content of referenced memory might change over time.

- One example is the following simple trick:
  - VirtualAllocEx(..., READONLY, ...)
  - VirtualProtectEx(..., EXECUTE_READWRITE, ...)



Victim Process

VAD struct

...

StartingVpn

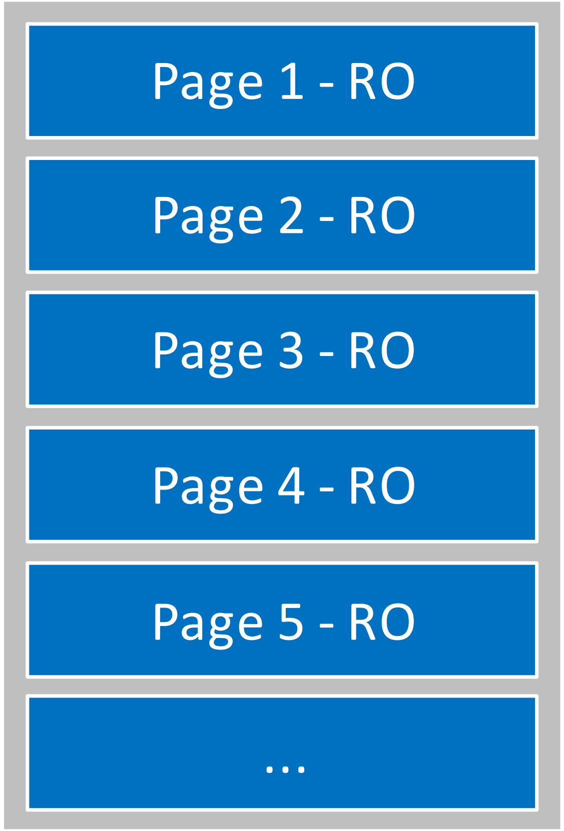
EndingVpn

...

u.VadFlags.ProtectionEnum

...

VAD - Initial Protection : ReadOnly



Page 1 - RO

Page 2 - RO

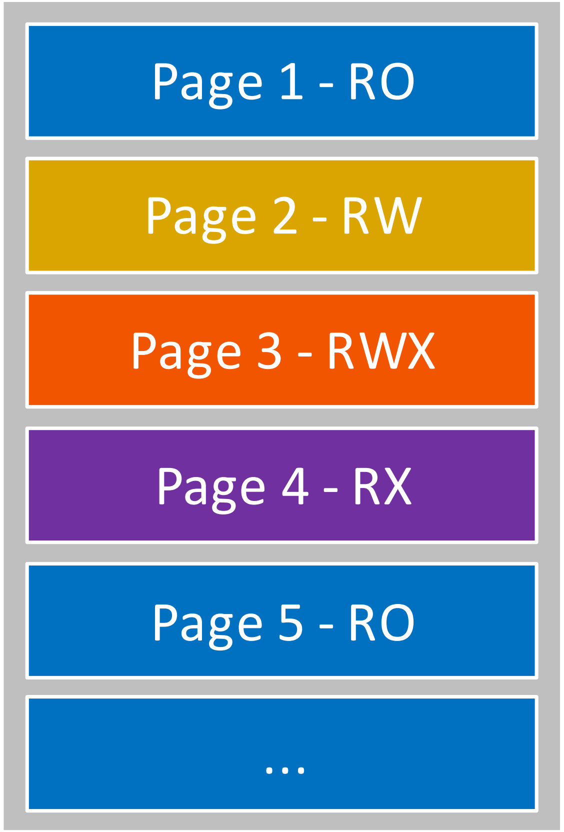
Page 3 - RO

Page 4 - RO

Page 5 - RO

...

VAD - Initial Protection : ReadOnly



Page 1 - RO

Page 2 - RW

Page 3 - RWX

Page 4 - RX

Page 5 - RO

...

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

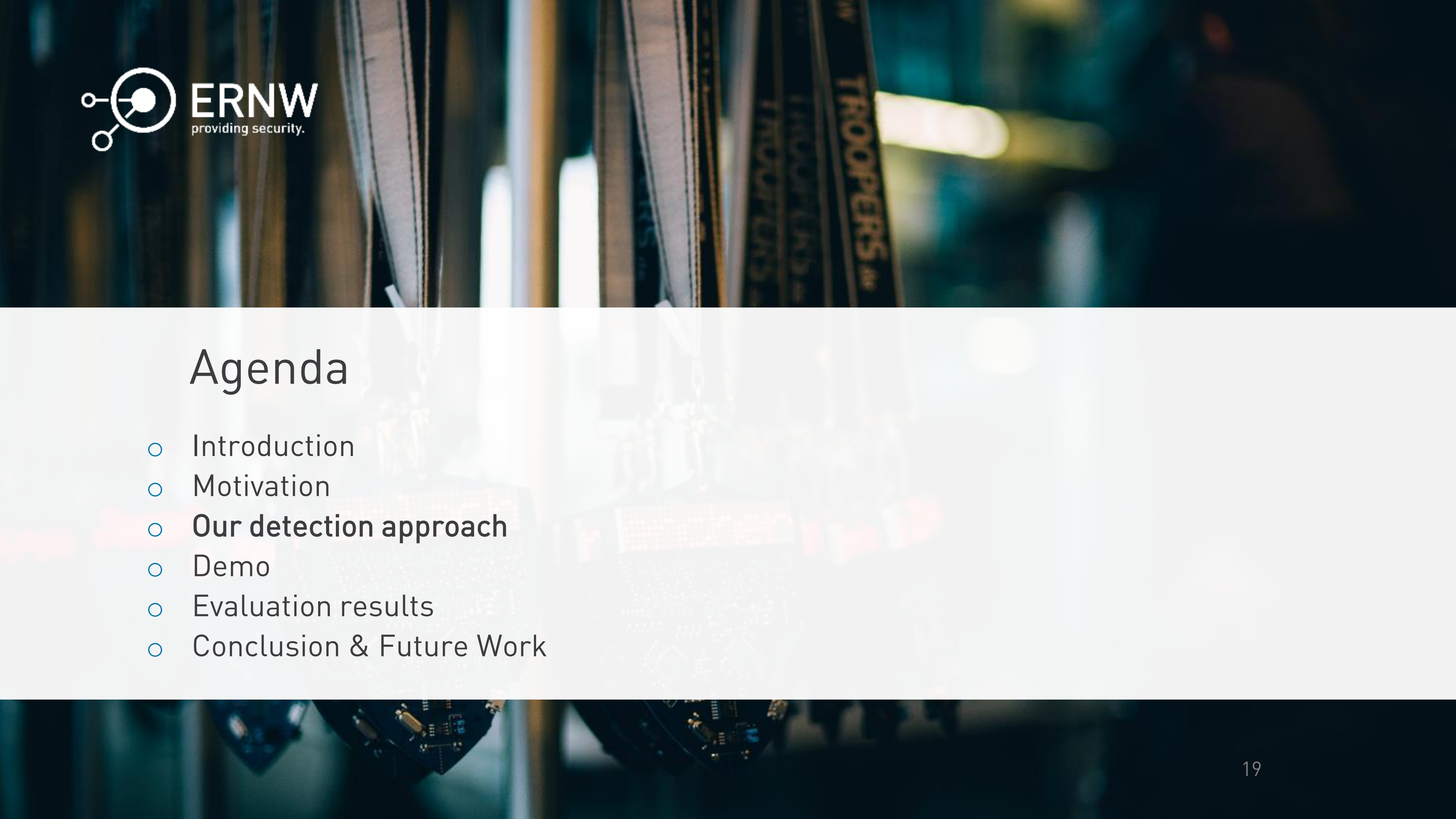
Mapped Image Files

- Another example is the modification of mapped image files e.g. through relocations, self decoding loops or code injections.
- When looking at memory regions belonging to mapped files (such as the executable), prior detection techniques at most compared the information from VADs and the PEB (Process Hollowing).
  - One exception: White et. al.[2]

- But malware can use pages of mapped files for code too.
  - EXECUTE_WRITECOPY

Further Hiding Techniques

- Mapped data files
- Shared memory with Copy-on-write protection
- Paged out pages: (un)intentional hiding.



Agenda

- Introduction
- Motivation
- **Our detection approach**
- Demo
- Evaluation results
- Conclusion & Future Work

State of the Art Code Injections

- APC Injections
- Process Hollowing
- AtomBombing
- (Gargoyle)
- …

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

What are we looking for?

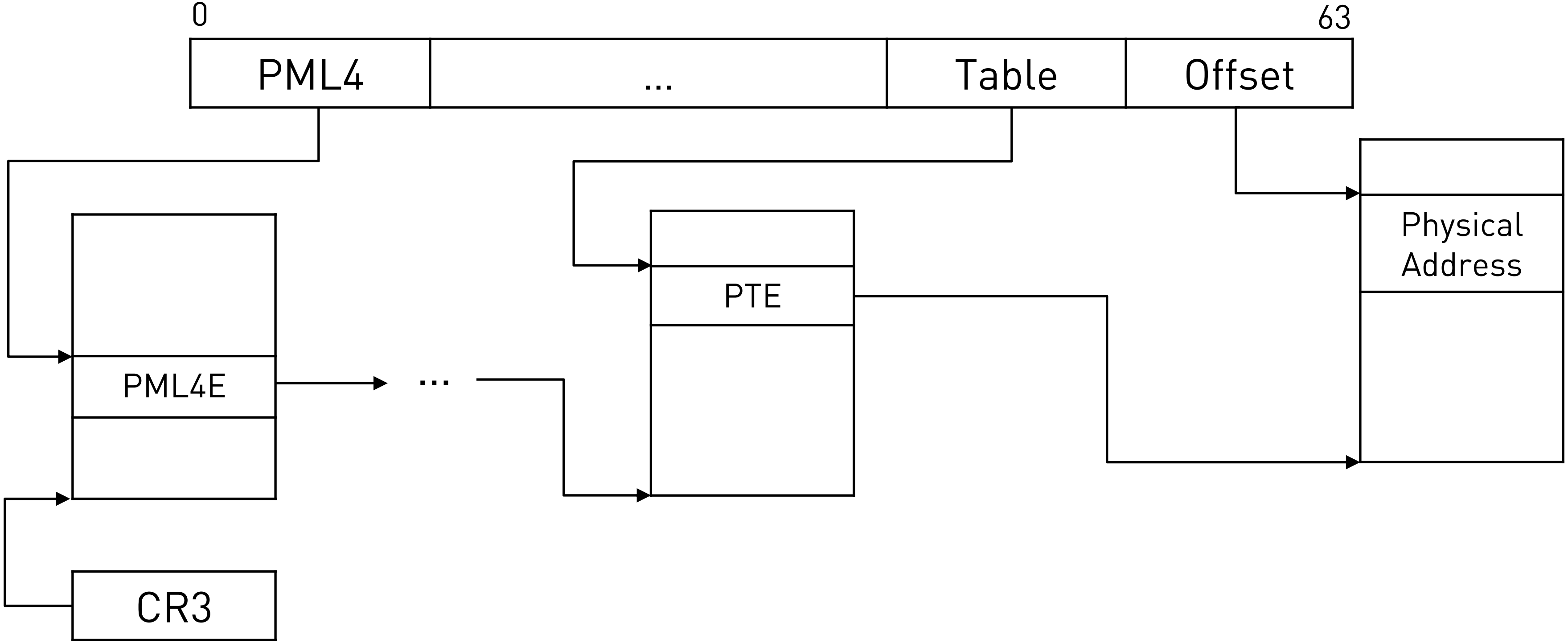
- Rootkit Paradox (Kornblum[3])
  - In Essence: While the rootkit tries to hide its existence, in order to do nasty stuff, its code must (at least once) be **locatable** and **executable**.
- So, the goal is to identify any executable data in user space.

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



0

63

PML4

...

Table

Offset

PML4E

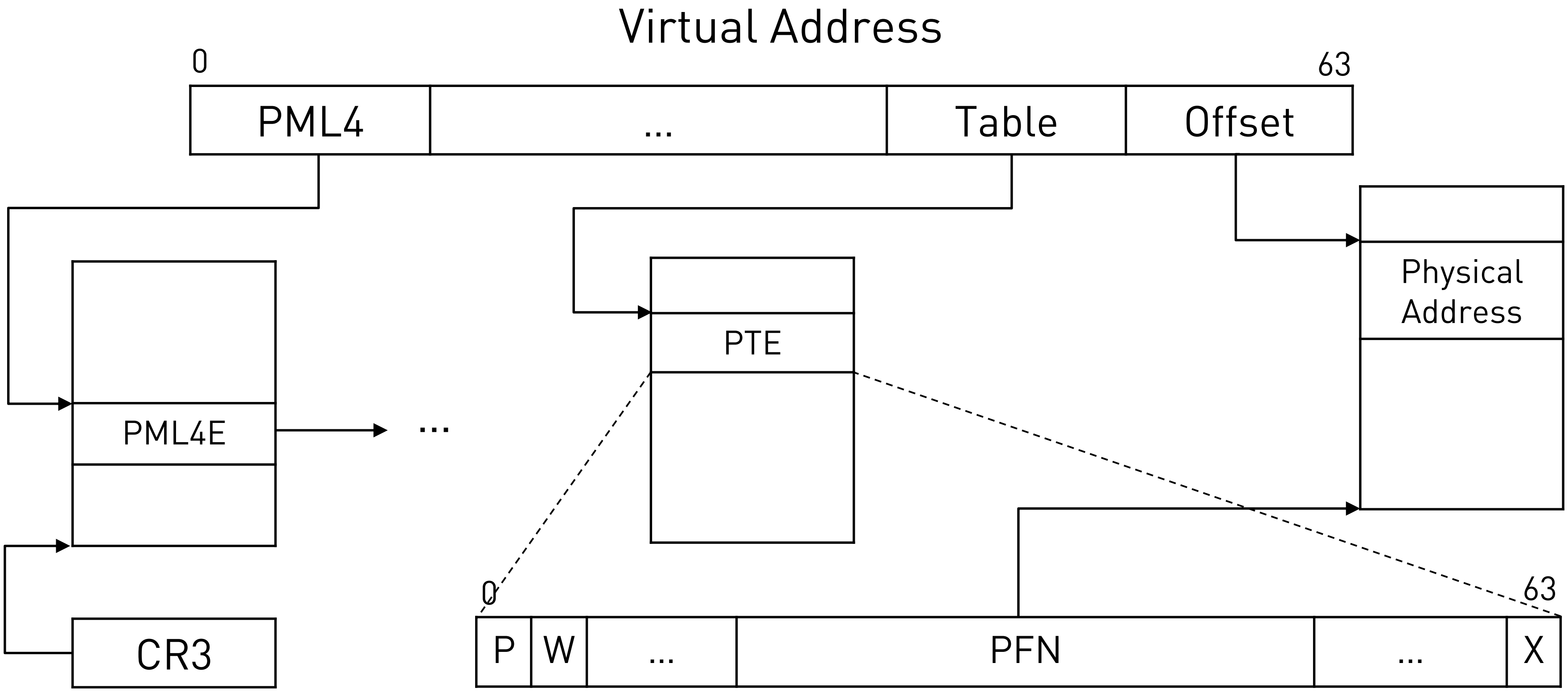
...

PTE

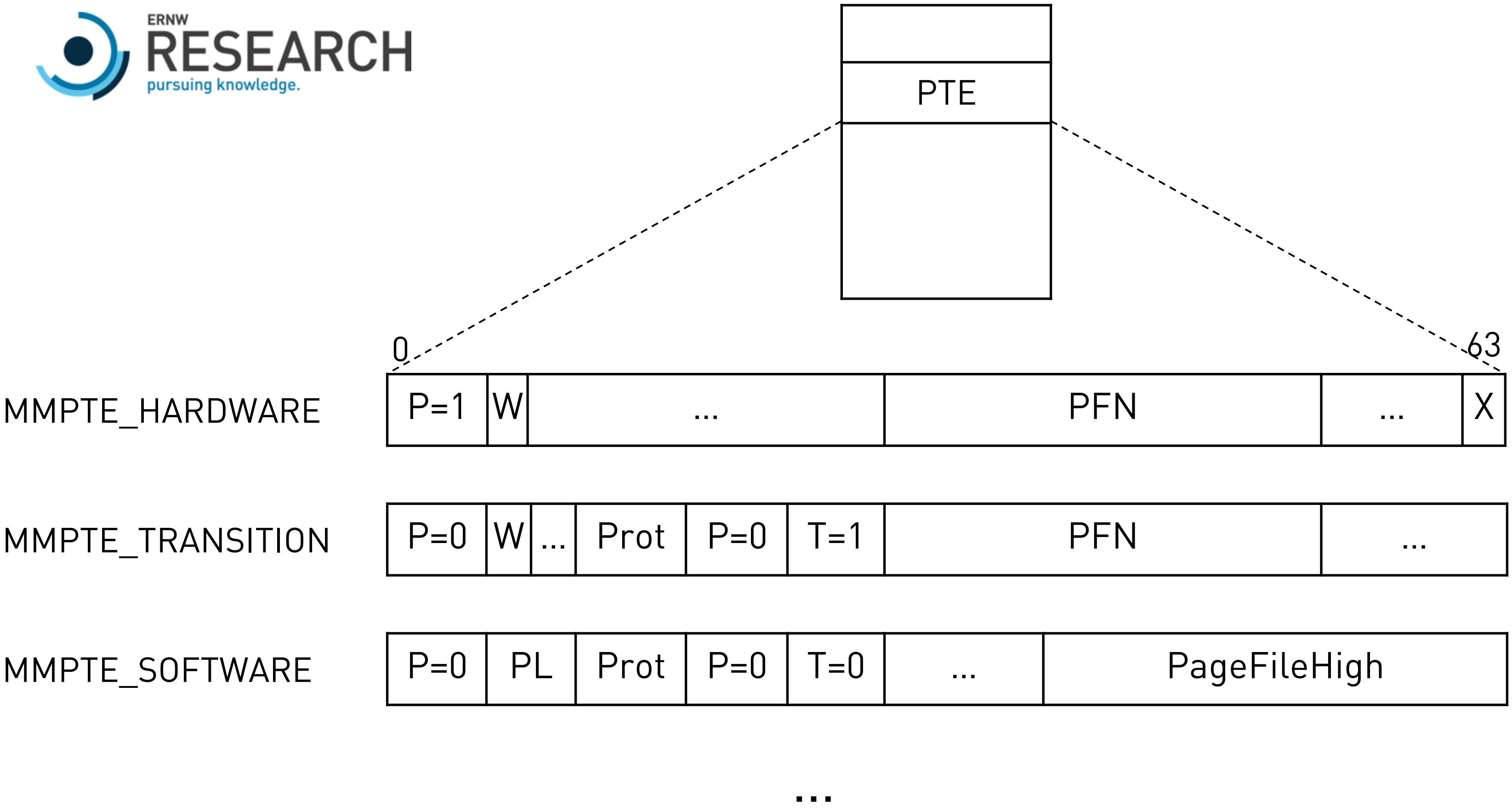
Physical Address

CR3

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



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



PTE

0

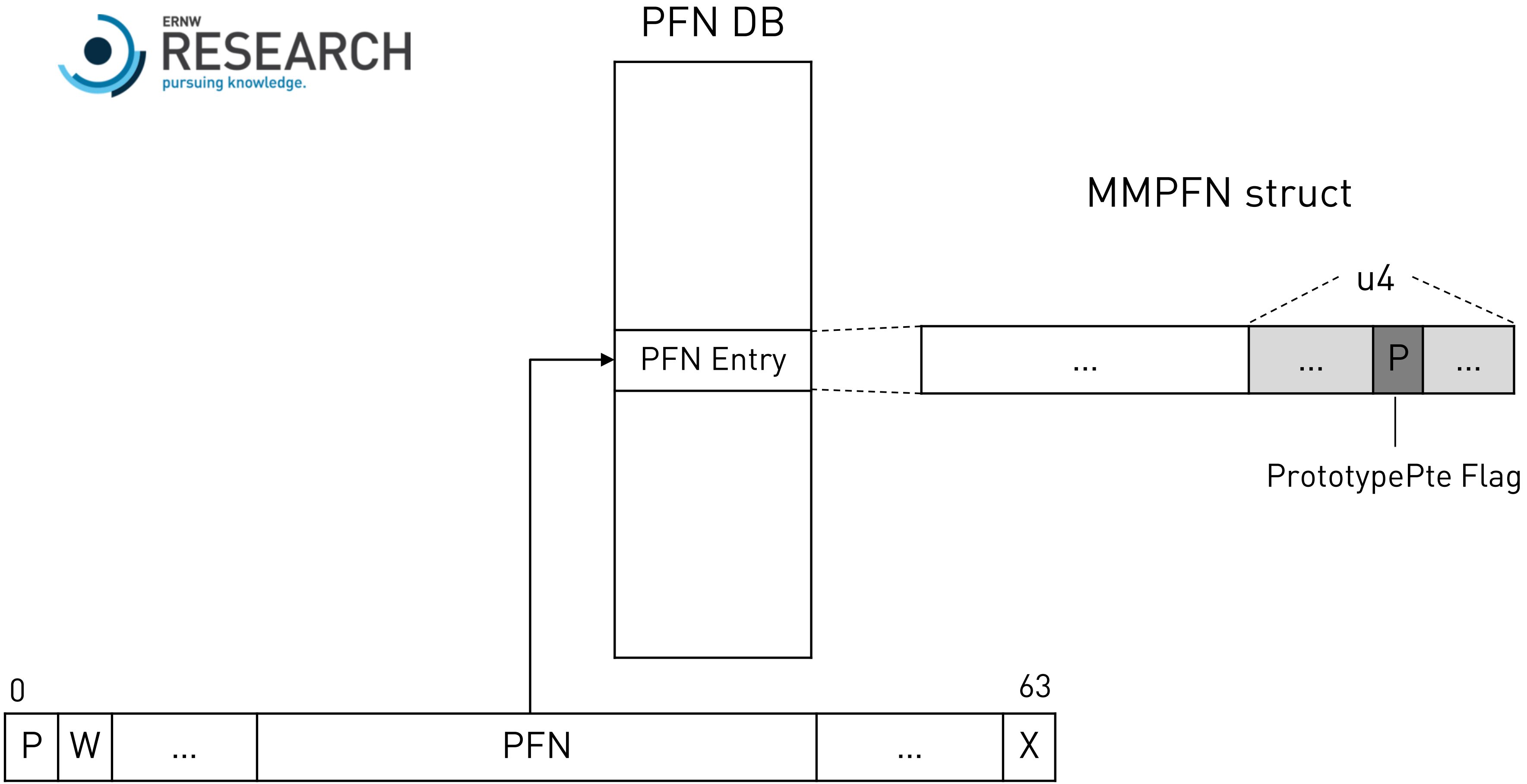
63

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

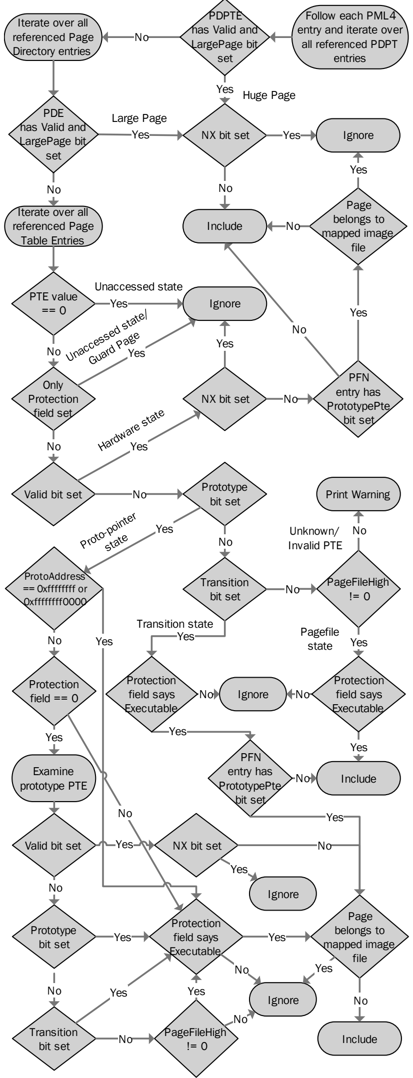
...

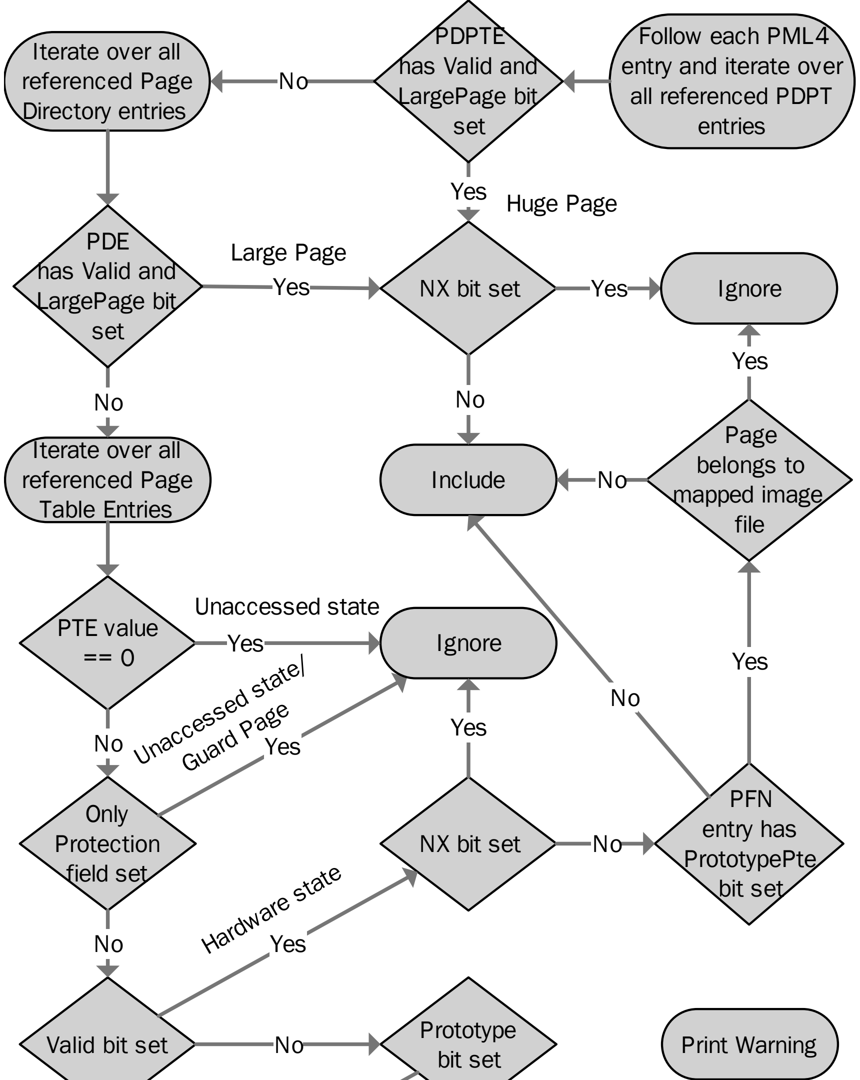

PFN DB
PFN Entry
MMPFN struct
u4
...
...
P
...
PrototypePte Flag
0
63
P
W
...
PFN
...
X

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?


Follow each PML4 entry and iterate over all referenced PDPT entries
PDPTE has Valid and LargePage bit set
No
Iterate over all referenced Page Directory entries
Yes
Huge Page
PDE has Valid and LargePage bit set
Large Page
Yes
NX bit set
Yes
Ignore
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
Yes
Only Protection field set
NX bit set
No
PFN entry has PrototypePte bit set
No
Hardware state
Yes
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
No
Yes
Transition state
Yes
Pagefile state
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
No
Yes
No
Yes
Yes
Ignore
No
Transition bit set
No
PageFileHigh != 0
No
Include



Follow each PML4 entry and iterate over all referenced PDPT entries

PDPTE has Valid and LargePage bit set

No

Iterate over all referenced Page Directory entries

Yes

Huge Page

PDE has Valid and LargePage bit set

Large Page

Yes

NX bit set

Yes

Ignore

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

Yes

Only Protection field set

NX bit set

No

PFN entry has PrototypePte bit set

No

Hardware state

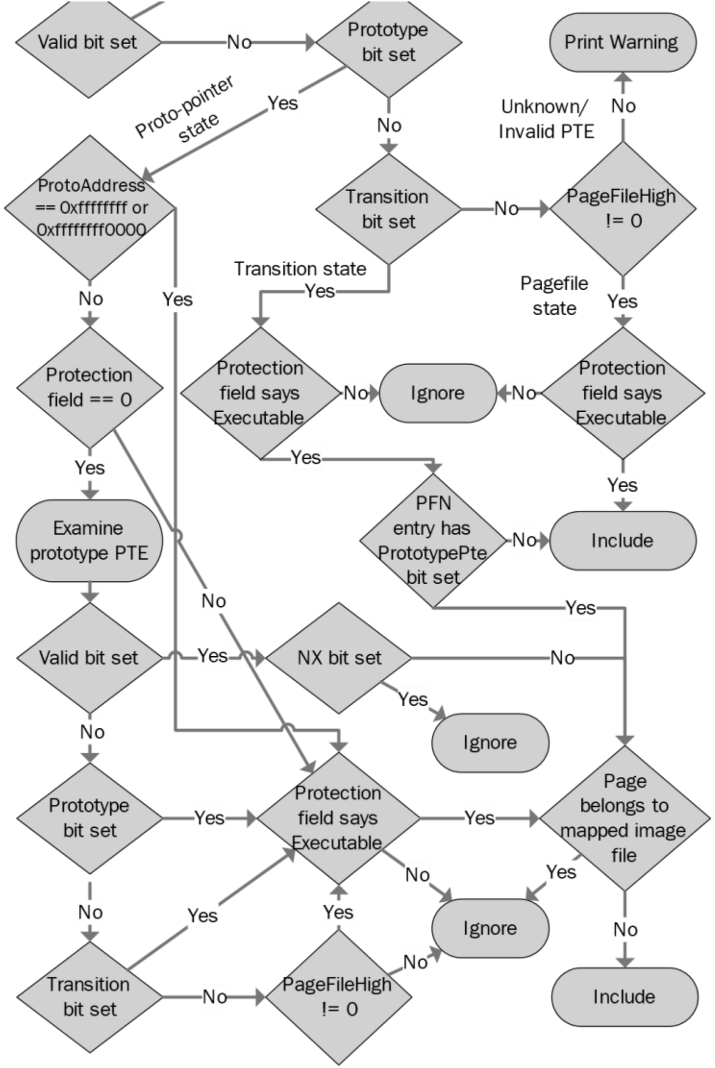
Yes

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
No
Yes
No
Yes
Yes
Ignore
No
No
Transition bit set
No
PageFileHigh != 0
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.

Stack

...

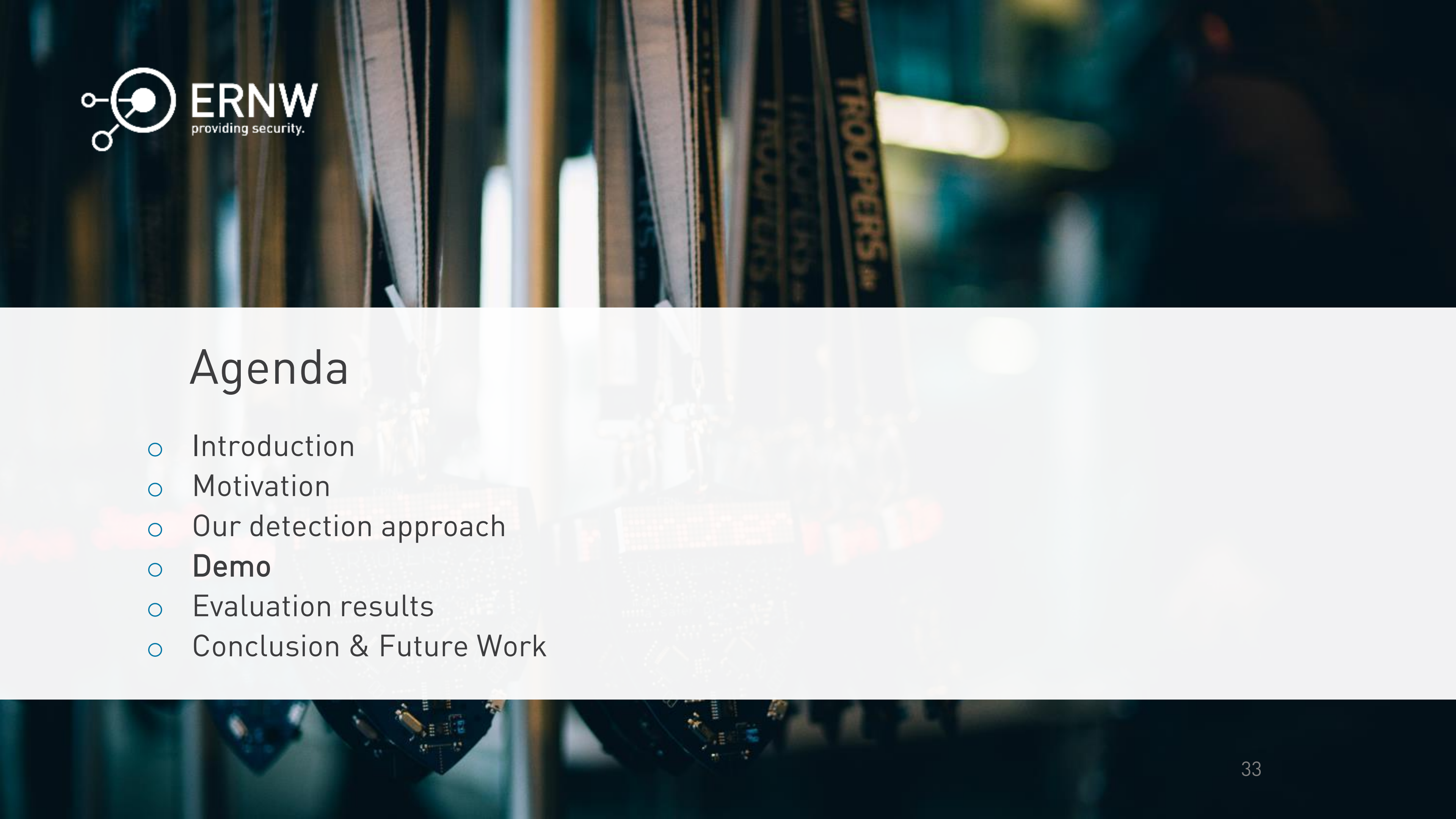
Page 28 - RW

Page 29 - RW

Page 30 - RWX

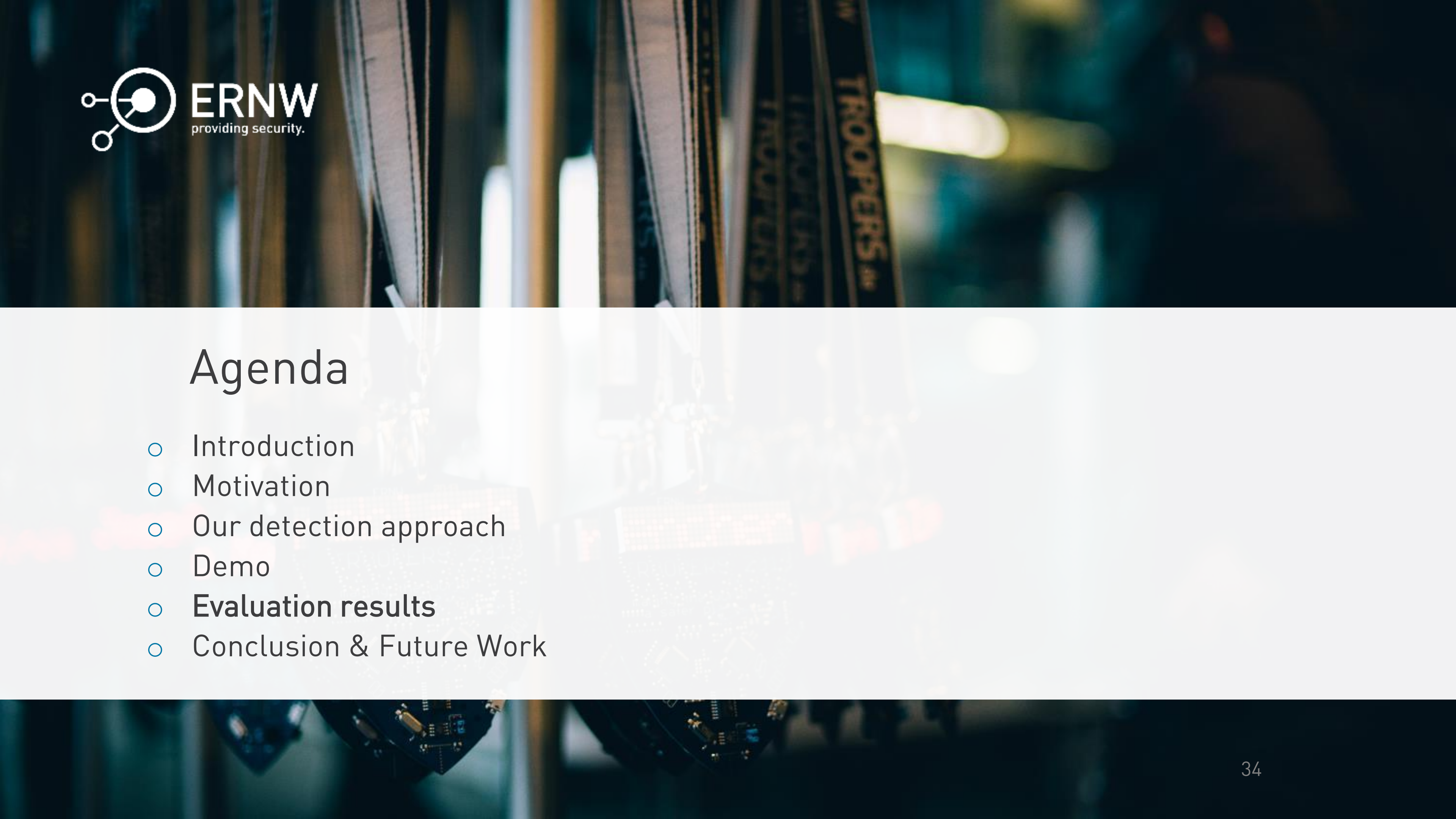
...

Page X - RW



Agenda

- Introduction
- Motivation
- Our detection approach
- **Demo**
- Evaluation results
- Conclusion & Future Work



Agenda

- Introduction
- Motivation
- Our detection approach
- Demo
- **Evaluation results**
- Conclusion & Future Work

Evaluation results

- No plugin detected all memory regions containing injected code.
- Also our failed for Gargoyle and the paged out DEP scenario.
  - Expected result: not executable.
- 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*.
- *hollowfind*, *malfofind* and *Psinfo* were unimpressed by the hiding techniques in regards to ProcessHollowing.

Agenda

- Introduction
- Motivation
- Our detection approach
- Demo
- Evaluation results
- **Conclusion & Future Work**

Conclusion & Future Work

- It is possible to hide from current code injection plugins.
- Our approach detects injected code in executable pages despite the described (un)intentional hiding techniques.

- Does not detect injected code/data in non executable pages.
- Does not work with paged out Paging Structures and no pagefile (could do a fallback to malfind like approach – is however again prone to attacks).

Conclusion & Future Work

- The amount of data to examine can be huge, mainly because of modified pages of mapped image files.
- Approach is suitable as:
  - Improved malfind.
  - Before/After comparison.

- Usage in existing code injection plugins to improve their results.

ERNW
RESEARCH
pursuing knowledge.

Thank you for your Attention

Questions/Criticism/Remarks/Suggestions?

The online repository can be found at:
https://github.com/f-block/DFRWS-USA-2019

✉ fblock@ernw.de

www.ernw.de

www.insinuator.net

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