Getting Physical
Extreme abuse of Intel based Paging Systems

(extended version)
Nicolas A. Economou
Enrique E. Nissim
About us

- Enrique Elias Nissim
  - Information System Engineer
  - Previously worked at Core Security as an Information Security Consultant
  - Now joining Intel Corp at Mexico to work in Graphics Security
  - Infosec Enthusiast (Exploit Writing, Reversing, Pentest, Programming)
  - Discovered some 0Days in Kernel components
  - @kiqueNissim

- Nicolas Alejandro Economou
  - Exploit Writer specialized in Windows kernel exploitation at Core Security Technologies for +10 years.
  - Infosec Enthusiast (Exploit Writing, Reversing, Patch Diffing and Programming)
  - Several defensive/offensive research, presentations and security tools as turbodiff, Sentinel and Agafi
  - @NicoEconomou
- Arbitrary Write: Explanation
- Reviewing Modern Kernel Protections
- Current ways of abusing kernel arbitrary writes
- Intel Paging Mechanism
- Windows
  - Implementation
  - Attacks
  - Live Demo
- Linux
  - Implementation
  - Attacks
  - Live Demo
- Conclusions
What is an arbitrary write?

- **Arbitrary Write:**
  - This is the result of exploiting a binary bug.
  - You can write a crafted value (or not) *where you want* (write-what-where) -> **MOV [EAX], EBX**

- **As a result:** If you write in the correct place, you can get primitives to **read/write memory** or you can control **EIP/RIP**

- **Examples:**
  - Heap overflows – overwrite pointers that point to specific structs
  - Memory Corruptions – idem above
  - Use after free – nt/win32k – Decrementing one unit ("**DEC [EAX]**")
## Reviewing Modern Protections

<table>
<thead>
<tr>
<th>Protection</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DEP/NX</strong></td>
<td>is a security feature included in modern operating systems. It marks areas of memory as either &quot;executable&quot; or &quot;nonexecutable&quot;.</td>
</tr>
<tr>
<td><strong>KASLR</strong></td>
<td>Address-space layout randomization (ASLR) is a well-known technique to make exploits harder by placing various objects at random, rather than fixed, memory addresses.</td>
</tr>
<tr>
<td><strong>Integrity Levels</strong></td>
<td>call restrictions for applications running in low integrity level – since Windows 8.1</td>
</tr>
</tbody>
</table>
Reviewing Modern Protections

- **SMEP**: Supervisor Mode Execution Prevention allows pages to be protected from supervisor-mode instruction fetches. If SMEP = 1, software operating in supervisor mode cannot fetch instructions from linear addresses that are accessible in user mode.

- **SMAP**: allows pages to be protected from supervisor-mode data accesses. If SMAP = 1, software operating in supervisor mode cannot access data at linear addresses that are accessible in user mode.
Current techniques
Current techniques

- Low Integrity Level in “Windows 8.1” suppressed all the kernel addresses returned by “NtQuerySystemInformation”

- The most affected exploits are “Local Privilege Escalation” launched from sandboxes like IE, Chrome, etc.
Call Restrictions

- **Running in Medium Integrity Level**
  - You know where the kernel base is, process tokens, some kernel structs, etc
  - Exploitation tends to be “trivial”

- **Running in Low Integrity Level**
  - You *DON’T* know where the kernel base is, process tokens, some kernel structs, etc
  - You need a *memory leak* (*second vulnerability*) to get some predictable kernel address
  - Without memory leaks exploitation tends to be much harder.
What can be done?

If you are running in **Low/Medium Integrity Level** and you have:

- Full arbitrary write (DWORD/QWORD):
  - You can overwrite GDI objects
    - Kernel GDI objects addresses are in USER SPACE – “Keen Team” technique.
    - This technique consists of linking one GDI object to another one

- Partial arbitrary write (WORD):
  - You can overwrite GDI objects
    - It depends on the low part of the object address what you want to overwrite, sometimes it is not possible.
What about...

- Partial writes?
- Single BIT controlled?
- Decrement a controlled position?
- You don’t have control over the value?

- Let’s see...
Intel Paging Mechanism
Paging 101

- Paging is a functionality provided by the MMU and used by the processor to implement virtual memory.
- A virtual address is the one used in processor instructions; this must be translated into a physical address to actually refer a memory location.
PAE Paging
x64 Paging

Diagram explaining the x64 paging structure.
Interesting fields to know for our purposes:

- **R/W**: readonly/readwrite
- **U/S**: if set, the range mapped by the entry is accessible at CPL3. Otherwise it is only accessible at CPL0.
- **PS**: if set, the entry describes a LARGE_PAGE.
- **XD**: if set, instruction fetching is not allowed for the region mapped by the entry.
Paging Implications

- All memory accesses and instruction fetching done by the processor will use virtual addresses.

- Given that the OS needs to manipulate the table entries not only for memory allocation but also for page level protection, all the paging structures of the current process are mapped to virtual memory.

- In order to comply with performance and memory savings requirements, a common approach taken by operating systems is to make use of self-reference table entry or a fixed location where all the paging structures will reside.
Windows Paging Implementation
Windows Implementation

- Each process has its own set of paging tables

- All paging structures virtual addresses can be calculated

- 512GB of virtual range is assigned for Paging Structures (x64)
Windows Implementation

- Only one PML4 entry is used for Paging management (0x1ED)

- Entry 0x1ED is self-referential (physical address points to PML4 physical address)

- Virtual range described:
  - 0xFFFFF680’00000000 – 0xFFFFF6FF’FFFFFFFF
Quick Formula

```c
_int64 get_pxe_address(_int64 address)
{
    _int64 result = address >> 9;
    result = result | 0xFFFFF68000000000;
    result = result & 0xFFFFF6FFFFFFFF8;
    return result;
}
```
```c
int get_pxe_32(int address)
{
    int result = address >> 9;
    result = result | 0xC0000000;
    result = result & 0xC07FFFF8;
    return result;
}
```
Strengths and Weaknesses

- **Strengths:**
  - Paging structures reside in random physical addresses

- **Weaknesses:**
  - Paging tables are in fixed virtual addresses
  - Paging tables are writables
Windows Paging Attacks

some clarifications
Techniques Overview

- We are going to show 2 different ways of abusing write-what-where conditions. (3 ways in the extended version)

- They do not require memory leaks.

- The 3 ways work from Low Integrity Level included.

- All Windows versions are affected. Specially Win 8, 8.1 and Win 10.
Windows Paging Attacks

“HAL’s heap”
HAL’s Heap

- Same virtual address for all Windows versions: 0xffffffff'ffd00000

- Same physical address by OS version

- Some juicy kernel function pointers located there
# HAL’s Heap

- **HAL’s heap x64 – physical address list**

<table>
<thead>
<tr>
<th>OS Version</th>
<th>Virtual Address</th>
<th>Physical Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows 7/2008 R2</td>
<td>0xffffffff’ffd00000</td>
<td>0x100000 (1mb)</td>
</tr>
<tr>
<td>Windows 8/2012</td>
<td>0xffffffff’ffd00000</td>
<td>0x100000 (1mb)</td>
</tr>
<tr>
<td>Windows 8.1/2012 R2</td>
<td>0xffffffff’ffd00000</td>
<td>0x1000 (4kb)</td>
</tr>
<tr>
<td>Windows 10/10 TH2</td>
<td>0xffffffff’ffd00000</td>
<td>0x1000 (4kb)</td>
</tr>
</tbody>
</table>
HAL’s Heap

- **HAL’s heap x64** – ‘HalpInterruptController’ pointer list:
  
  +20: hal!HalpApicInitializeLocalUnit
  +28: hal!HalpApicInitializeIoUnit
  +30: hal!HalpApicSetPriority
  +38: hal!HalpApicGetLocalUnitError
  +40: hal!HalpApicClearLocalUnitError
  +48: NULL
  +50: hal!HalpApicSetLogicalId
  +58: NULL
  +60: hal!HalpApicWriteEndOfInterrupt
  +68: hal!HalpApic1EndOfInterrupt
  +70: hal!HalpApicSetLineState
  +78: hal!HalpApicRequestInterrupt
HAL’s Heap

- We know the physical address of the HAL’s heap
- We know where our Page Table Entries are
- And we are able to allocate memory in USER SPACE (VirtualAlloc)

- It means that
  - We can use an arbitrary write to modify a PTE of our allocated virtual memory
  - We can point this PTE to the HAL’s heap physical address
HAL’s Heap

PTE mapping VA 0xFFFFFFFF’FFD00000

Physical address 0x1000

kernel space read-write

PTE mapping VA 0x401000

Physical address 0xNNNNNNNNNN

user space read-write!

HAL’s heap

User allocation
HAL’s Heap

- As a result:
  - We get read/write access from USER SPACE to the HAL’s heap
  - We get access to some HAL’s heap function pointers
  - We use this information to get the “HAL.DLL” base address
  - We overwrite “hal!HalpApicRequestInterrupt” pointer
  - We disable SMEP by ROPing (Ekoparty 2015: “Windows SMEP bypass: U=S”)
  - And finally, we get system privileges ...
Some days before Cansec...
Improving the Technique

- We can improve the approach considerably by using a LARGE_PAGE.

- If we write a single byte into an EMPTY PDE, we map 2MB starting from PFN 0 (this will include the physical address of the HAL’s Heap) with R/W access from user mode.

- E.g:

  00 00 00 00 00 00 00 00 00 -> E7 00 00 00 00 00 00 00 00
Let’s say we have a simple DEC [RAX] (Win32k UAF)

We can use it to decrement an empty PDE in a shifted way.

\[
00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 \\
00 FF FF FF FF FF FF FF FF - FF 00 00 00 00 00 00 00
\]

We effectively mapped a User R/W LARGE_PAGE starting at PFN 0 by enabling all the bits!
Windows Paging Attacks

“Heap Spraying Page Directories”
Heap spraying Page Directories

- PDPTs are in fixed virtual addresses (we can calculate this)

- PDPT entries point to Page Directories

- A Page Directory maps up to 1GB (if all entries used)
Heap spraying Page Directories

- Page Directories points to Page Tables or LARGE PAGES

- A Large Page maps 2MB of physical memory (bit PS=1)

- PDPT entries can be overwritten (via a partial arb.write – value controlled or not)
Heap spraying Page Directories

- We can heap spray our PROCESS MEMORY with fake Page Directories with all entries used (by using “VirtualAlloc” + “memcpy”)

- The idea is to produce a physical memory exhaustion

- If we choose a valid random physical address, we will probably find our data in high physical addresses!
- So, if we **overwrite a PDPTE** that maps memory in our process (E.g PDPTE that maps VA 1GB ~ 2GB)

- And we point this entry to an "**arbitrary physical address**" used by our heap spray

- It means that we had just **mapped** 1GB of memory as read-write
Heap spraying Page Directories

**PDpte** mapping VA 1GB-2GB

User space read-write

addr. chosen by Windows

Kernel allocation (Real PD)

addr. chosen by us

User Allocation (Fake PD)
Depending on the chosen physical address, we can:

- Map the HAL’s heap

- Find a valid Page Table and dump the rest of the target memory

- Find and modify the kernel code, structures, etc, without any restriction.
- As a result
  - We can insert ring-0 shellcode by replacing kernel code

- We don’t need to bypass SMEP

- And finally, we get system privileges...
Windows Paging Attacks

”Self-ref of death”
Self-ref of Death

- We know that PML4 entry 0x1ED is used for Memory Paging management

- We know that this entry is referencing itself

- And we know that it’s at 0xffffffff6fb`7dbedf68

<table>
<thead>
<tr>
<th>Sign Extension</th>
<th>PML4</th>
<th>PDPT</th>
<th>PD</th>
<th>PT</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xffffffff</td>
<td>0x1ED</td>
<td>0x1ED</td>
<td>0x1ED</td>
<td>0x1ED</td>
<td>0xF68</td>
</tr>
</tbody>
</table>
Self-ref of Death

If we replace SUPERVISOR by USER?
Self-ref of Death

We get access from USER SPACE to all USER tables !!!
Self-ref of Death

- So, if we have a “bit/byte/word/dword” arbitrary write, we can get access from USER MODE to all User Tables including the PML4!

- There is a weakness in the self-referential technique, only one entry is set a SUPERVISOR, the rest is USER

- To be clear, after our arbitrary write, if we read from user space at 0xffffffff6fb`7dbed000, we see our PML4!
Self-ref of Death

- It means that we can add/modify/delete entries in the four paging levels

- So, we can do the same as seen before
  - Point one PTE to the HAL’s heap
  - or dump the complete physical memory
  - or modify kernel parts
Windows Live Demo
Windows Demo

- **Target:**
  - “Windows 10” 64 bits

- **Scenario:**
  - Running in **Low Integrity Level**

- **Objective:**
  - Dump physical memory and get SYSTEM privileges by using “Self-ref of Death”
Linux Paging Implementation
Linux Implementation

- Only one PML4 entry is used for Paging management (0x110)

- Entry 0x110 is NOT self-referential like Windows

- Virtual range described:
  - 0xFFFF8800’00000000 – 0xFFFF887F’FFFFFFFF
- Each process has its own **PML4** table in a unique virtual address (opposite to Windows)

- Physical addresses can be read as virtual addresses by adding a base.

- This base is called **__PAGE_OFFSET**:
  - For 32 bits: 0xc0000000
  - For 64 bits: 0xffff8800’00000000
Linux Implementation

- Most of page table entries reside in random virtual and physical addresses.

- **But...** there are some PDPTs, PDs and PTs in fixed physical addresses.
Linux Implementation

- PDPT physical address list (pointed by entry 0x110)

<table>
<thead>
<tr>
<th>OS version</th>
<th>Virtual Address</th>
<th>Physical Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debian 8.3 3.16.0-4-amd64</td>
<td>0xFFFFF8800’01AF4000</td>
<td>0x01AF4000 (~26mb)</td>
</tr>
<tr>
<td>Xubuntu 14.04 3.19.0-25-gen</td>
<td>0xFFFFF8800’01FD4000</td>
<td>0x01FD4000 (~31mb)</td>
</tr>
<tr>
<td>Ubuntu 15.10 4.2.0-16-gen</td>
<td>0xFFFFF8800’01FF0000</td>
<td>0x01FF0000 (~32mb)</td>
</tr>
<tr>
<td>Ubuntu 14.04.3 LTS 3.19.0-25-generic</td>
<td>0xC1B51000</td>
<td>0x01B51000 (~27mb)</td>
</tr>
</tbody>
</table>
Strengths and Weaknesses

- **Strengths:**
  - None

- **Weaknesses:**
  - Some PDPTs, PDs and PTs are in fixed virtual addresses
  - Paging structures are writable
Linux Paging Attacks

“Setting the vDSO as rwx”
What is the vDSO?

- vDSO (virtual dynamic shared object)
- Small shared library mapped into all user processes
- It was created to reduce the context-switch overhead
vDSO Page Entry

- It’s set as “r-x” in user space

- The vDSO virtual address changes per process

- The PDE that describes this user space area is RANDOM
Setting the vDSO as “rwx”

- But ... the physical address is **fixed**
  - E.g: “Debian 8.3” 64 bits: **0x1805000**

- So, we can calculate the kernel virtual address
  - E.g: “Debian 8.3” 64 bits: **0xffff880001805000**

- For “Debian 8.3”, the **PDE** (large page) which maps the vDSO physical address is **fixed** and **writable**!
Setting the vDSO as “rwx”

- Even worse, the PML4 and the PDPT entries are set with the **USER** bit!!!

- So, What if we set the PDE as **USER** by using an arb.write?
Setting the vDSO as “rwx”
Setting the vDSO as “rwx”

Now it can be read from USER!
Setting the vDSO as “rwx”

- As a result:
  - We get read-write access to the vDSO from USER SPACE

  - We can modify/hook functions located there like “gettimeofday”

  - When a UID 0 process invokes this function, our shellcode will be called and will spawn a new root shell
Linux Paging Attacks

“Creating self-ref entries”
Creating self-ref entries

- There are several entries that always use the same fixed physical addresses:
  - PML4E 0x110
  - PML4E 0x192
  - PLM4E 0x1FE
  - PLM4E 0x1FF
  - ... To be continued...

- There are fixed entries for ALL levels of the paging hierarchy (PML4, PDPTs, PDs, PTs)
Creating self-ref entries

- PML4 entry 0x110 points to a fixed PDPT
- This PML entry is set as USER (0x67)
- We know the virtual and physical address pointed by this entry
Creating self-ref entries

PML4

0x110 U

PDPT

we’ll add a self-ref entry
Creating self-ref entries

- PML4
  - 0x110 U
- PDPT
  - 0x6 U
Creating self-ref entries
Creating self-ref entries

- Real example - “Debian 8.3 x64”
  - We add a PDPT entry at 0xFFFFF8800’01AF4010 (entry 0x2)
  - The written value is “67 04 AF 01 00 00 00 00”
  - This entry is self-referential

- Calculating the mapped virtual address by this entry:
  \[ \text{va} = 0xFFFFF8800’00000000 \]
  \[ \text{va} += 512\text{gb} * 0x110 \text{ PML entries} \]
  \[ \text{va} += (1\text{gb} + 2\text{mb} + 4\text{kb}) * 0x2 \text{ PDPT entries} \]
  \[ \text{va} = 0xFFFFF8880’80402000 \]
Creating self-ref entries

- So, we are able to add/modify/delete PDPT entries

- We then add another entry in this PDPT and it’s used as PTE.

- This **SPURIOUS PTE** allows us to read and write the complete target’s physical memory!
Linux Live Demo
- **Target:**
  - Debian 8.3 64 bits - 3.16.0-4-amd64

- **Scenario:**
  - Running as normal-unprivileged user

- **Objective:**
  - Getting root privileges by modifying the **vDSO**
Linux Paging
Attacks (bonus track):
”PaX/Grsec notes”
Pax/Grsec notes

- **PaX** is a *patch* for the *Linux kernel* that implements least privilege protections for memory pages.

- **Grsecurity** is a *set of patches* for the *Linux kernel* which emphasizes security enhancements.

- **Grsec + PaX** change the rules of what we saw previously
Pax/Grsec notes

- **PaX/Grsec** implements SMEP/SMAP by software

- It uses two different PML tables, one for **USER MODE** and one for **KERNEL MODE**

- When a syscall is invoked, the kernel changes **CR3** by pointing to the **KERNEL MODE PML table**
Pax/Grsec notes

- The same physical address is used to map the PML for all current processes

- In our Debian 8.3 compiled/focused to server mode
  - CR3 for kernel mode points to 0x15f0000
  - CR3 for user mode points to 0x15f1000

- Each process has a PGD (Page Global Directory)
Pax/Grsec notes

- These PGDs are “mirrored” in CR3 by Pax/Grsec

- For **KERNEL SPACE** entries: The first three level page tables are in **fixed** “virtual/physical” addresses

- For **USER SPACE** entries: PDPTs, PDs and PTs are **RANDOM**
A small detail, all not RANDOM page tables are set as READ-ONLY ...

So, it's not possible to overwrite a fixed page directory/table entry 😞

We will find another way to bypass it ... ;-)}
Conclusions
Windows conclusions

- Paging tables shouldn’t be in fixed VA addresses
  - It can be abused by LOCAL and REMOTE kernel exploits

- The PML entry (0x1ed) should be RANDOMIZED
  - 256 entries are available for the OS kernel
  - Only ~20 entries are used by Windows

- All fixed paging structures should be read-only
Linux conclusions

- Paging tables shouldn’t be in **fixed addresses**
  - It can be abused by LOCAL and REMOTE kernel exploits

- All fixed paging structures should be **read-only**

- Some advice, compile the kernel with **Grsec ;)**
Questions?

Thanks

Enrique Nissim
@kiqueNissim
n3k1990@gmail.com

Nicolas Economou
@NicoEconomou
neconomou@coresecurity.com