Weird-Machine Motivated Practical Page Table Shellcode & Finding Out What's Running on Your System

Shane Macaulay
Director of Cloud Services
Killing the Rootkit! And how to find everything running on your system!!!

• Rootkit/APT technique for hiding processes
  – Unlink kernel structures “DKOM”

• New 64bit detection technique! DC22 exclusive 😊
  – System/Platform independent technique
  – Linux/BSD/Windows/ARM64/ADM64

• Works by analyzing physical memory & properties of MMU Virtual Memory system
The Long Road

• Barnaby Jack, forever in our hearts and minds.

“It’s about the journey not the destination.”
13 Years since ADMMutate (slide URL) http://1drv.ms/1rEBMjF

- ADMMutate (last DC talk was about polymorphic shellcode)
- The more things change
  - The more they stay the same
- Thought about PT shellcode with ADMMutate

- Attack is [hard/stress/] fun!!&$&%*:P;p;P
- Defense is hard/stress
Abusing x for fun & profit!

• It’s usually the QB that get’s the headlines, offensive bias in hacker scene!

• Defense is grind’s it out for little glory.
  – Let’s energize the “D” here, have some fun!!

• A Defensive exploit
  – Ultimately today were killing process hiding rootkits cross 64bit OS/Platforms TODAY! 😊
  – DKOM IS DEAD! Process hiding is DEAD!
Also 13 Years ago

• What else was going on back then?
  – **x86 assembler in Bash**

“cLleNUX”

“shasm is an assembler written in GNU Bash Version 2, which may work in other recent unix-style "shell" command interpreters.”
Ideals

• As best as possible, figure out all running code
  – Code/hacks/weird machine's included/considered
  – When have we done enough?

• We focus on establishing our understanding through real world targets: Hypervisor monitored guests.

• Combine protection pillars; structure analysis, physical memory traversal and integrity checking.
Practical concepts

- Attacks: WeIrD MaChinE
  - Lots of fun!
    - Much esoteric/eclectic More fantastical!!!
- Defense: Detecting * ← That means everything
  - Home field == USE THE “FORCE” A HYPERVERSOR!
    - Establishes verifiability of device state (i.e. not worried about platform attacks e.g. **BIOS/firmware/UEFI**)
    - Games in fault handler do not work on snapshot, even just extracting physical memory can be hard
    - Protection from **virtualized** (Dino Dai Zovi), that is serious/obvious impact to performance when nested.
Practical Page Table ShellCode

Motivations

• An attack devised to understand memory protection systems
  – Development necessitated comprehensive understanding of inner workings, system fault handling complexities and some of the lowest level (brain melting, see reference below) interaction of software and hardware on modern 64bit platforms.
  – Until Windows 7, page tables directly executable
    • NonExecutable is opt-in/non-default
  – The page-fault weird machine: lessons in instruction-less computation
    • Julian Bangert, Sergey Bratus, Rebecca Shapiro, Sean W. Smith from WOOT’13 Proceedings of the 7th USENIX conference on Offensive Technologies
### X64 Kernel Virtual Address Space

[http://www.codemachine.com/article_x64kvas.html](http://www.codemachine.com/article_x64kvas.html)

<table>
<thead>
<tr>
<th>Start</th>
<th>End</th>
<th>Size</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFFF0800’00000000</td>
<td>FFFFFFF7F’FFFFFFFFF</td>
<td>238TB</td>
<td>Unused System Space</td>
<td>WIN9600 NOW USE &amp; CAN CONTAIN +X AREAS</td>
</tr>
<tr>
<td>FFFFF680’00000000</td>
<td>FFFFF6FF’FFFFFFFFF</td>
<td>512GB</td>
<td>PTE Space</td>
<td>-X used to be executable Win7</td>
</tr>
<tr>
<td>FFFFF700’00000000</td>
<td>FFFFF77F’FFFFFFFFF</td>
<td>512GB</td>
<td>HyperSpace</td>
<td>8.1 seems to have cleaned up here, 9200 had 1 +X page</td>
</tr>
<tr>
<td>FFFFF780’00000000</td>
<td>FFFFF780’00000000</td>
<td>4K</td>
<td>Shared System Page</td>
<td></td>
</tr>
<tr>
<td>FFFFF780’00001000</td>
<td>FFFFF7FF’FFFFFFFFF</td>
<td>512GB-4K</td>
<td>System Cache Working Set</td>
<td></td>
</tr>
<tr>
<td>FFFFF800’00000000</td>
<td>FFFFF87F’FFFFFFFFF</td>
<td>512GB</td>
<td>Initial Loader Mappings</td>
<td>Large Page (2MB) allocations</td>
</tr>
<tr>
<td>FFFFF880’00000000</td>
<td>FFFFF89F’FFFFFFFFF</td>
<td>128GB</td>
<td>Sys PTEs</td>
<td></td>
</tr>
<tr>
<td>FFFFF880’00000000</td>
<td>FFFFF8bF’FFFFFFFFF</td>
<td>128GB</td>
<td>Paged Pool Area</td>
<td></td>
</tr>
<tr>
<td>FFFFF900’00000000</td>
<td>FFFFF97F’FFFFFFFFF</td>
<td>512GB</td>
<td>Session Space</td>
<td></td>
</tr>
<tr>
<td>FFFFF980’00000000</td>
<td>FFFFFa70’FFFFFFFFF</td>
<td>1TB</td>
<td>Dynamic Kernel VA Space</td>
<td></td>
</tr>
<tr>
<td>FFFFFa80’00000000</td>
<td>*nt!MmNonPagedPoolStart-1</td>
<td>6TB Max</td>
<td>PFN Database</td>
<td>DEFAULT NO EXECUTE</td>
</tr>
<tr>
<td>*nt!MmNonPagedPoolStart</td>
<td>*nt!MmNonPagedPoolEnd</td>
<td>512GB Max</td>
<td>Non-Paged Pool</td>
<td></td>
</tr>
<tr>
<td>FFFFFFF’FFc00000</td>
<td>FFFFFFFF’FFFFFFFFF</td>
<td>4MB</td>
<td>HAL and Loader Mappings</td>
<td></td>
</tr>
</tbody>
</table>

**Notes:**
- Large Page (2MB) allocations
- WIN9600 NOW USE & CAN CONTAIN +X AREAS
- -X used to be executable Win7
- 8.1 seems to have cleaned up here, 9200 had 1 +X page
- DEFAULT NO EXECUTE
Page Table ShellCode weird-machine

- Win7- and earlier
  - Can we emit intended shellcode into PTE area?
    - Call VirtualAlloc() from user space results in executable memory in kernel
      - Just reserving memory causes a code-write operation into kernel space

<table>
<thead>
<tr>
<th>PXE at FFFFF6FB7DBEDF88 contains 000000000187063</th>
<th>PPE at FFFFF6FB7DBEDF88 contains 00000000134C04863</th>
<th>PDE at FFFFF6FB7DBF008 contains 0000000100512863</th>
<th>PTE at FFFFF6FB7E01EA0 contains 00000002D3B863</th>
</tr>
</thead>
<tbody>
<tr>
<td>pfns 187</td>
<td>pfns 134c04</td>
<td>pfns 100512</td>
<td>pfns 2dc3b</td>
</tr>
<tr>
<td>---DA--KW+EV</td>
<td>---DA--KW+EV</td>
<td>---DA--KW+EV</td>
<td>---DA--KW+EV</td>
</tr>
</tbody>
</table>
PT SC WM Died with Win8 (below)

- This works earlier than Win7, interesting to examine fault handling, but ultimately Win8 this is dead! 😊

<table>
<thead>
<tr>
<th>Child-SP</th>
<th>RetAddr</th>
<th>Call Site</th>
</tr>
</thead>
<tbody>
<tr>
<td>fffffd000<code>2b34ecf8 </code>fffff800`16066ee1</td>
<td>nt!LOCK_WORKING_SET</td>
<td></td>
</tr>
<tr>
<td>fffffd000<code>2b34ed00 </code>fffff800`1603f5ad</td>
<td>nt!MiSystemFault+0x911</td>
<td></td>
</tr>
<tr>
<td>fffffd000<code>2b34eda0 </code>fffff800`1615af2f</td>
<td>nt!MmAccessFault+0x7ed</td>
<td></td>
</tr>
<tr>
<td>fffffd000<code>2b34ee00 </code>fffff6fb`77fde37a</td>
<td>nt!KiPageFault+0x12f</td>
<td></td>
</tr>
<tr>
<td>fffffd000<code>2b34f078 </code>fffff800`01e423fe</td>
<td>0xffffffff6fb`77fde37a</td>
<td></td>
</tr>
<tr>
<td>fffffd000<code>2b34f080 </code>fffff800`163ae3e5</td>
<td>SIoctl!SioctlDeviceControl+0x27e</td>
<td></td>
</tr>
<tr>
<td>fffffd000<code>2b34f9b0 </code>fffff800`163aed7a</td>
<td>nt!IopXxxControlFile+0x845</td>
<td></td>
</tr>
<tr>
<td>fffffd000<code>2b34fb60 </code>fffff800`1615c4b3</td>
<td>nt!NtDeviceIoControlFile+0x56</td>
<td></td>
</tr>
<tr>
<td>fffffd000<code>2b34fbd0 00007ff9</code>c1b265ea</td>
<td>nt!KiSystemServiceCopyEnd+0x13</td>
<td></td>
</tr>
<tr>
<td>00000003a<code>ba9bf8f8 00007ff9</code>bef92c83</td>
<td>ntdll!NtDeviceIoControlFile+0xa</td>
<td></td>
</tr>
</tbody>
</table>
What about new tool (wanted ptshellcode thingy)?

• Was going to do a talk with an expansion of the PT shellcode concept
  – Was it going to be an ADMmutate update? .NET Compiler thingy some set of C macro’s or little script host RoP builder/engine/host?

• Application of technique is mostly dead, requires an info leak(maybe) and what about use bash to write it?
Some peace of mind – really!

• cross platform AMD64 process detection technique
  – obsoletes process hiding techniques used by all rootkits/malware!
    • Process hiding rootkits/malware technology being typical of APT

• Detection can be used as an attack (defensive attack pattern)
  – Defensive Exploit against ALL ROOTKITS!
The big picture ProcDetect

• Ultimately decided on a more advanced, and useful, tool for release today
  – Hear it for the D!

• ProcDetect should be with DefCon materials
  – Signed code example for AMD64 Windows
    • Other platform/OS to follow 😊
Attack v Defense

- Defensive Window of opportunity
  - Closing the door/window today!
- Defensive tactics can be new classes of defensive attack techniques
  - Offensive Forensics / Automation
  - Use the process detection here to post process and detect any/every hidden process ever spawned for all TIME! 😊
    - Keep interesting/known memory dumps around
  Right now; there are no possible attacks against this technique (“WE FOUND YOU!”)
In Memory Process Detection

- Dumping memory is a pain physically
- Scanning VS. List traversal
- Scanning
  - Can be very slow
  - Tends to be high assurance

- Link/Pointer Traversal
  - Easily confused
  - Super Fast!
What’s a Process?

- A Process is an **address space configuration**
  - A container for threads which are executed on a CPU.
  - Threads share address space.
  - Hard to know if you have them all.

- Can’t I inject a library/thread to an existing process?
  - Code overwrite or injection is an integrity issue
    - Hash Check
Process Detection

- Volatility to the rescue!
  https://code.google.com/p/volatility/wiki/CommandReference#psxview
  - It compares the following *logical* identifiers:
    - PsActiveProcessHead linked list
    - EPROCESS pool scanning
    - ETHREAD pool scanning (then it references the owning EPROCESS)
    - PspCidTable
    - Csrss.exe handle table
    - Csrss.exe internal linked list (unavailable Vista+)
<table>
<thead>
<tr>
<th>Tool</th>
<th>Virtual Address Translation in Kernel Space</th>
<th>Guessing OS version and Architecture</th>
<th>Getting Kernel Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Volatility Framework</td>
<td>2 factors: <code>_DISPATCHER_HEADER</code> and ImageFileName (PsIdleProcess)</td>
<td>1 factor: <code>_DBGKD_DEBUG_DATA_HEADER64</code></td>
<td>2 factors: <code>_DBGKD_DEBUG_DATA_HEADER64</code> and PsActiveProcessHead</td>
</tr>
<tr>
<td>Mandiant Memoryze</td>
<td>4 factors: <code>_DISPATCHER_HEADER</code>, PoolTag, Flags and ImageFileName (PsInitialState Process)</td>
<td>2 factors: <code>_DISPATCHER_HEADER</code> and offset value of ImageFileName (PsInitialState Process)</td>
<td>None</td>
</tr>
<tr>
<td>HBGary Responder</td>
<td>None</td>
<td>1 factor: OperatingSystem Version of kernel header</td>
<td>1 factor: ImageFileName (PsInitialState Process)</td>
</tr>
</tbody>
</table>

**Takahiro Haruyama** -- April 2014, discuss his BH Europe 2012 talk with respect to **Abort Factors**.
64bit Process Detection

• Earlier presentation for kernel code
  – E.g. CSW14 Diff CPU Page table & Logical kernel objects (to detect hidden kernel modules, “rootkit revealer”)

• Also uses page tables “Locating x86 paging structures in memory images”
  – Karla Saur, Julian B. Grizzard

• New process detection technique is faster - single pass
  – Similar to “pmodump”, enhanced with 64bit & additional checks (64bit scan has much more verifiability)
64bit Process Detection Integrity

• Not easily attacked
  – Many modifications result in BSOD
  – Able to extract candidate memory for integrity checking of memory pages to fully qualify
    • Can make “non-abortable” if willing to do slower check
    • Current check is really good

  – Always room to grow with respect to countermeasures and performance
Physical/Virtual Memory

Page Frames
4k physical blocks
A Page Frame Number (PFN) is the physical memory “address”

Virtual Address/Pages
Page protection is applied to virtual pages/address ranges
A quick indirection

- Slides 37-39 from Dave Probert (Windows Kernel Architect, Microsoft)
  - Windows Kernel Architecture Internals

- Next slide show’s a big hint, can you guess? It’s an example of process page table layout/configuration.
  - You have to love all of those arrow’s 😊
Self-mapping page tables

Virtual Access to PTE for va 0xe4321000

GetPteAddress:
0xe4321000
=> 0xc0390c84
Self Map trick in Linux

• **Virtual Memory in the IA-64 Linux Kernel**
  – Stephane Eranian and David Mosberger
    • 4.3.2 Virtually-mapped linear page tables

  “linear page tables are not very practical when implemented in physical memory”
  “The trick that makes this possible is to place a self-mapping entry in the global directory.”
Self Map process detection Windows AMD64

- Self Map exists for each process (not only kernel:)
- Examining a page table - `!process 0 0 → dirbase/cr3` (e.g. `7820e000`)

```
!dq 7820e000
#7820e000 00800000`60917867
!dq 7820e000+0xf68
#7820ef68 80000000`7820e863
```

^-^- current PFN found --^- (PFN FTW)
PFN FTW Trick! (or Defensive exploit!!)

#\texttt{7820ef68 80000000`7820e863}^\texttt{^\_\_\_\_\_\_\_\_}\texttt{^}

64Bit is a more powerful check

Valid PFN will be bounded by system physical memory constraints
typedef struct _HARDWARE_PTE {
    ULONG Long Valid : 1;          // Indicates hardware or software handling (mode 1&2)
    ULONG Long Write : 1;
    ULONG Long Owner : 1;
    ULONG Long WriteThrough : 1;
    ULONG Long CacheDisable : 1;
    ULONG Long Accessed : 1;
    ULONG Long Dirty : 1;
    ULONG Long LargePage : 1;      // Mode 2
    ULONG Long Global : 1;
    ULONG Long CopyOnWrite : 1;
    ULONG Long Prototype : 1;      // Mode 2
    ULONG Long reserved : 1;
    ULONG Long PageFrameNumber : 36; // PFN, always incrementing (mode 1&2)
    ULONG Long reserved1 : 4;
    ULONG Long SoftwareWsIndex : 11;
    ULONG Long NoExecute : 1;
} HARDWARE_PTE, *PHARDWARE_PTE;
These are the OFFSETS your looking for.

- 512 way Table (512 * 8 = 0x1000, a page)
  - PFN Offset 0 configured and valid bit
  - PFN Offset 0x1ed Point’s to self and valid bit
    - This allows us to identify *current position

- Mode2 has more checks for typical page table
- Mode1 is for heightened assurance
  - Both work together to extract PFN & MEMORY_RUN gaps
Self Map Detection Attacks

- Attacks against performance
  - If we de-tune performance we can validate spoof entries and various malformed cases
  - Windows zero’s memory quickly (no exiting processes, so far:)
- !ed [physical] can be done to assess evasive techniques
  - Simply destroying self map results in BSOD!! 😊
  - *Looking for feedback testing to identify better more comprehensive PTE flag checks (edge cases, missed tables or extra checks)*
Implementation (basically in 1 line)

```c
// scan every page from lpMapping to lpMapping+MAP_SIZE
for(unsigned long long i=0; i < WinLimit; i+=512)
{
    // first entry of table should not be null and end in 0x867
    // lower bits 0x867 configured
    if(lpMapping[i] != 0 && (lpMapping[i] & 0xffff) == 0x867)
    {
        // self map should be at index 0xf68/8 == 0x1ed
        ULONGLONG selfMap = lpMapping[i+0x1ED];

        // if we can find a possible self map, extract current PFN
        ULONGLONG low12Bits = selfMap & 0xffff;
        if(selfMap != 0 && (low12Bits == 0x863 || low12Bits == 0x063))
        {
            ULONGLONG offset = CurrWindowBase+(i*8);
            MMPTE_64 selfPTE;
            selfPTE.u.Long.QuadPart = selfMap;

            ULONGLONG shift = (selfPTE.u.Hard.PageFrameNumber << PAGE_SHIFT);
            ULONGLONG diff = offset > shift ? offset - shift : shift - offset;
            printf("Possible Directory Base Register Value = [%llx] File Off", offset);
        }
    }
}
```
Example execution (.vmem starts @0 offset), .DMP (0x2000+)
or other autodetect header offset 😊
Detected Memory Runs

- Round value by offset to find gap size, adjust to automate memory run detection
  - Takahiro Haruyama blog post on related issue (large memory) and also memory run detection issues from logical sources

- *previous slide, detecting gap, when offset changes;
  - ROUND_UP(0xb4b56000, 0x40000000) = first run end 0xc00000
  - ROUND_DOWN(0x1181f1000, 0x40000000)
Detect processes of guests from host dump

- A host memory dump will include page tables for every guest VM process as well as host process entries
  - Lots of room to grow here, deep integration with HyperVisor page mapping data may be straightforward
    - E.g. parsing of MMInternal.h / MMPAGESUBPOOL in VirtualBox

- Issues
  - Hypervisor may not wipe when moving an instance or after it’s been suspended (ghost processes)
    - I’d rather detect ghosts than fail 😊

- Nested paging not a problem
Skew is evident for guest instances. An typical kernel PFN is observed (scream 187 to a mo…) as the first (large jump 0x2..->0x4…) in a range of skewed diff values (another layer of decoding to adjust, similar to what happens when snapshot is requested and disk memory is serialized)

Final host processes identifiable by Diff realignment
Future Weird Machine overload 😞

• Microsoft Research
  – Tracking Rootkit Footprints with a Practical Memory Analysis System -- Weidong Cui, Marcus Peinado, Zhilei Xu, Ellick Chan
  – “The goal of MAS is to identify all memory changes a rootkit makes…. MAS does so in three steps: static analysis, memory traversal and integrity checking”

• Seems really hard problem (source code used in MAS), how can we verify this level of state?
Public symbols to the rescue’ish 😊

- Public symbols, RTTI or other type inference technique to find/root(tree/linked) all pointers
  - Thread stack return into verifiable code
    - Anti RoP Attack
  - Advanced methods kernel pool (does not require source) verification
    - Integrity Checking of Function Pointers in Kernel Pools via Virtual Machine Introspection
  - At least kernel alerts, logs and various tracing can be trusted if we have code integrity, process/thread detection.
  - Future is not too bad for Defense!
Summary

• Attacks: WeIrD MaChInE
  – Worst case scenario most weird machine activity can hopefully be detected through simple tracing, logging and monitoring tools
  • What about the next GPU/UEFI backdoor? → use a hypervisor guest to establish device/low layer trust capability

• Defenses: Detecting hidden 64bit processes
  • Deep future holds deep verifiability for more devices 😊 (get free The Memory Cruncher™ TMC & BlockWatch™)
    – Active Protection System (APS)

• FINALLY DEFENSIVE FUN & PROFIT! With the D!
Summary

• Always use a VM
  – At least simplify memory dumping

• Use ProcDetect
  – Have fun detecting!
  – Process hiding rootkit is dead
  – 64bits helps peace of mind

• We can detect a process anywhere (host, guest, nested, on the network (probably😉))!
Issues, Considerations Caveats

• Use a hypervisor – secure the guest/host (very hardened host)
  – Hypervisor escape == you’re a high value to risk nice exploit
    • Probably NOT YOU!
    • BluePill type attacks, hopeful still to consider (but perf hit of nesting should be obvious)

• SefMap Detection relies on page table.
  – Maybe “no paging process”— (same as x86 paging paper)
  – TSS considerations, monitor other tables with stacks?
  – Remote DMA?
    • Please no! 😞
Attention Wikipedia editors **DKOM 😊**

“Not only is this very difficult to..”

We have a high assurance capability, applicable cross 64bit platforms (linux/freebsd also arm64, etc…) , for process detection.

Even though threads are distinct execution contexts, the property of shared MMU configuration establishes a verification capability that OS kernel object manipulation cannot effect.
Thank you & Questions

• I hope I referenced earlier works sufficiently, this topic is broad and expansive, thanks to the many security professionals who analyze memory, reverse-engineered, dove deep and discussed their understanding.
• References, follow embedded links and their links