Pool Tag Quick Scanning for Windows Memory Analysis

Joe T. Sylve, **Vico Marziale**, and Golden G. Richard III

DFRWS EU 2016
Who are any of us really?

• Senior Research Developer @BlackBag Tech
  – find cool artifacts, figure out how to parse them
  – develop new techniques
  – get them into our tools

• PhD from UNO in CS (2009)
  – research focus on efficient digital forensics

• Also done: DF practice, PenTesting, Malware Analysis

• FOSS dev: Scalpel, Registry Decoder, Spotlight Inspector, DAMM (built on top of Volatility)

• Organizer BSidesNOLA April 16 in New Orleans
  – come on over and I’ll buy the beers
The Problem

• Memory forensics coming into wider use
• Applications for DF
  – crypto, cached data, volatile system state
• And IR
  – malware, intrusion detection
• Just like disk, memory sizes are increasing rapidly
  – newer Windows systems max out at ~4TB
• Some memory analysis relies on scanning
  – like file carving but for in-memory structures
• I (and likely you) want everything to be faster
  – without loss of ... anything
  – especially in IR Land
The (Basic) Solution

• Generally, the most important things we scan for are kernel structures
  – e.g., _EPROCESS process descriptors
• These things exist in kernel memory
• Kernel memory divided into a set of pools
• Many of the things we care about are only allocated from specific pools
  – a much smaller scanning space
Memory Pools

• Dynamically sized (heaps)
• Kernel allocations in system address range
  – kernel address space
  – mapped into every process
• Paged pool: can be paged out to disk
• Non-paged pool: cannot be paged out to disk
  – so guaranteed to be in a memory image
  – kernel structures (processes, network stuff)
  – drivers
  – observed as small as 64MiB (allocated)
• Begin with a _POOL_HEADER structure
<table>
<thead>
<tr>
<th>Offset</th>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0x000</td>
<td>PreviousSize</td>
<td>Bitfield Pos 0, 8 Bits</td>
</tr>
<tr>
<td>+0x000</td>
<td>PoolIndex</td>
<td>Bitfield Pos 8, 8 Bits</td>
</tr>
<tr>
<td>+0x000</td>
<td>BlockSize</td>
<td>Bitfield Pos 16, 8 Bits</td>
</tr>
<tr>
<td>+0x000</td>
<td>PoolType</td>
<td>Bitfield Pos 24, 8 Bits</td>
</tr>
<tr>
<td>+0x000</td>
<td>Ulong1</td>
<td>Uint4B</td>
</tr>
<tr>
<td>+0x004</td>
<td>PoolTag</td>
<td>Uint4B</td>
</tr>
<tr>
<td>+0x008</td>
<td>ProcessBilled</td>
<td>Ptr64 to struct _EPROCESS</td>
</tr>
<tr>
<td>+0x008</td>
<td>AllocatorBackTraceIndex</td>
<td>Uint2B</td>
</tr>
<tr>
<td>+0x00a</td>
<td>PoolTagHash</td>
<td>Uint2B</td>
</tr>
</tbody>
</table>

**BlockSize**: size of allocation*

**PoolType**: paged pool, non-paged pool

**PoolTag**: 4 byte marker for this allocation type
ntdll！_POOL_TYPE

Enum _POOL_TYPE, 15 total enums

  NonPagedPool = 0n0
  PagedPool = 0n1
  NonPagedPoolMustSucceed = 0n2
  DontUseThisType = 0n3
  NonPagedPoolCacheAligned = 0n4
  PagedPoolCacheAligned = 0n5
  NonPagedPoolCacheAlignedMustS = 0n6
  MaxPoolType = 0n7
  NonPagedPoolSession = 0n32
  PagedPoolSession = 0n33
  NonPagedPoolMustSucceedSession = 0n34
  DontUseThisTypeSession = 0n35
  NonPagedPoolCacheAlignedSession = 0n36
  PagedPoolCacheAlignedSession = 0n37
  NonPagedPoolCacheAlignedMustSSession = 0n38
<table>
<thead>
<tr>
<th>Purpose</th>
<th>Pool Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>Driver Object</td>
<td>Driv</td>
</tr>
<tr>
<td>File Object</td>
<td>File</td>
</tr>
<tr>
<td>Kernel Module</td>
<td>MmLd</td>
</tr>
<tr>
<td>Logon Session</td>
<td>SeLs</td>
</tr>
<tr>
<td>Process</td>
<td>Proc</td>
</tr>
<tr>
<td>Registry Hive</td>
<td>CM10</td>
</tr>
<tr>
<td>TCP Endpoint</td>
<td>TcpE</td>
</tr>
<tr>
<td>TCP Listener</td>
<td>TcpL</td>
</tr>
<tr>
<td>Thread</td>
<td>Thre</td>
</tr>
<tr>
<td>UDP Endpoint</td>
<td>UdpA</td>
</tr>
</tbody>
</table>
Big (Large) Page Pool

Allocations over a certain size (~page size*) are made from the Big Page Pool

Info about allocations at nt!PoolBigPageTable

```c
struct _POOL_TRACKER_BIG_PAGES, 4 elements, 0x18 bytes
+0x000  Va : Ptr64 to Void
+0x008  Key : Uint4B
+0x00c  PoolType : Uint4B
+0x010  NumberOfBytes : Uint8B
```

**Va:** virtual address of the allocation  
**Key:** pool tag  
**PoolType:** type  
**NumberOfBytes:** size of allocation
Pool Tag Scanning

• Pool tags are handy for scanning through entire memory image
  – analogous to a file header
  – at least for smaller allocation sizes
• Also like file headers, need further verification to reduce false positives
  – use known constraints for structure type
  – other nearby structures like _OBJECT_HEADER
• What about Big Page Allocations?
  – pool tag stored at nt!PoolBigPageTable, not with the allocation itself
  – just enumerate the table
Pool Tag Quick Scanning

• Crux: We know allocations for key kernel structures come from specific pools
  – non-paged pool
  – big page pool
• For non-paged pool, kernel keeps a VA allocation bitmap
  – what VAs are mapped to physical pages
• PTQS Process
  – get virtual address range of non-paged pool and use VA allocation bitmap to find those mapped physical pages
  – use big page table to find allocations backed by physical pages
  – use VAs/page tables to build range of physical pages to scan
  – scan only these pages
• Does it work?
  – Glad you asked.
Base Test Setup

• We are currently developing a new memory analysis framework (topic of coming paper)
• Developed two plugins to search for _EPROCESS allocations
  – \textit{psscan} to exhaustively search physical memory
  – \textit{psquickscan} to use the PTQS technique
• Ran a series of tests for accuracy, speed, etc.
  – Hardware: mid-2014 2.8 GHz MacBook Pro with 16 GiB RAM
  – Note: all times are average of 10 runs with highest and lowest removed
Scenario 1: Accuracy

- Win7SP1x64 16 GiB memory image
- Compare our `psscan` and `psquickscan`
- Compare to Volatility and Rekall

<table>
<thead>
<tr>
<th>Plugin</th>
<th>Type</th>
<th>Avg. Time</th>
<th>Running</th>
<th>Terminated</th>
<th>Prior Boot</th>
<th>Duplicate⁴</th>
</tr>
</thead>
<tbody>
<tr>
<td>psquickscan</td>
<td>Virtual</td>
<td>0.129s</td>
<td>128</td>
<td>21</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>psscan</td>
<td>Physical</td>
<td>15.584s</td>
<td>128</td>
<td>22</td>
<td>15</td>
<td>43</td>
</tr>
<tr>
<td>psscan (Rekall)</td>
<td>Physical</td>
<td>35.967s</td>
<td>128</td>
<td>22</td>
<td>15</td>
<td>43</td>
</tr>
<tr>
<td>psscan (Volatility)</td>
<td>Physical</td>
<td>25.448s</td>
<td>128</td>
<td>21</td>
<td>15</td>
<td>43</td>
</tr>
</tbody>
</table>

Notes

- All scan types found the same number of *running* processes
- Two anomalies in the number of *terminated* processes found
- `psquickscan` reported reading only **80 MiB** of the image
Scenario 2: Speed

- Memory images across multiple OSs, and RAM sizes
- Compare our *psscan* and *psquickscan*

<table>
<thead>
<tr>
<th>OS Version</th>
<th>Plugin</th>
<th>Data Scanned</th>
<th>RAM Size</th>
<th>Avg. Time</th>
<th>Running</th>
<th>Terminated</th>
<th>Duplicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vista SP0</td>
<td>psscan</td>
<td>1 GiB</td>
<td>0.083s</td>
<td>46</td>
<td>2</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>Vista SP0</td>
<td>psquickscan</td>
<td>60 MiB</td>
<td>1 GiB</td>
<td>0.073s</td>
<td>48</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Vista SP1</td>
<td>psscan</td>
<td>1 GiB</td>
<td>0.400s</td>
<td>48</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Vista SP1</td>
<td>psquickscan</td>
<td>76 MiB</td>
<td>1 GiB</td>
<td>0.236s</td>
<td>50</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Vista SP2</td>
<td>psscan</td>
<td>1 GiB</td>
<td>0.547s</td>
<td>50</td>
<td>1</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>Vista SP2</td>
<td>psquickscan</td>
<td>64 MiB</td>
<td>2 GiB</td>
<td>0.075s</td>
<td>43</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>7 SP0</td>
<td>psscan</td>
<td>2 GiB</td>
<td>0.712s</td>
<td>43</td>
<td>6</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>7 SP0</td>
<td>psquickscan</td>
<td>64 MiB</td>
<td>2 GiB</td>
<td>0.075s</td>
<td>50</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>7 SP1</td>
<td>psscan</td>
<td>2 GiB</td>
<td>0.691s</td>
<td>50</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7 SP1</td>
<td>psquickscan</td>
<td>44 MiB</td>
<td>4 GiB</td>
<td>0.054s</td>
<td>36</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>psscan</td>
<td>4 GiB</td>
<td>1.433s</td>
<td>36</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>psquickscan</td>
<td>244 MiB</td>
<td>8 GiB</td>
<td>0.170s</td>
<td>45</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8.1</td>
<td>psscan</td>
<td>8 GiB</td>
<td>2.977s</td>
<td>45</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Notes

- About an order of magnitude speedup

*typo in paper!*
Scenario 3: Network Data Transfer

- Use F-Response to mount RAM over network (gigabit)
- Compare our psscan and psquicksan

<table>
<thead>
<tr>
<th>RAM Size</th>
<th>Plugin</th>
<th>Scanned</th>
<th>Time</th>
<th>Transferred</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 GiB</td>
<td>psquicksan</td>
<td>102 MiB</td>
<td>9.489s</td>
<td>116.115 MiB</td>
</tr>
<tr>
<td>2 GiB</td>
<td>psscan</td>
<td>2 GiB</td>
<td>28.132s</td>
<td>2.014 GiB</td>
</tr>
<tr>
<td>4 GiB</td>
<td>psquicksan</td>
<td>122 MiB</td>
<td>9.640s</td>
<td>177.367 MiB</td>
</tr>
<tr>
<td>4 GiB</td>
<td>psscan</td>
<td>4 GiB</td>
<td>56.971s</td>
<td>4.027 GiB</td>
</tr>
<tr>
<td>8 GiB</td>
<td>psquicksan</td>
<td>246 MiB</td>
<td>15.360s</td>
<td>299.648 MiB</td>
</tr>
<tr>
<td>8 GiB</td>
<td>psscan</td>
<td>8 GiB</td>
<td>3m26.449s</td>
<td>8.132 GiB</td>
</tr>
</tbody>
</table>

Notes

- Data transferred just greater than data scanned
- Slower networks will just make the wait more frustrating
Scenario 4: Large Memory Image

- Test with significantly larger memory image
- Compare our psscan and psquickscan
- Compare to Volatility and Rekall

<table>
<thead>
<tr>
<th>Plugin</th>
<th>Data Scanned</th>
<th>Avg. Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>psquickscan</td>
<td>5.76 GiB</td>
<td>5.797s</td>
</tr>
<tr>
<td>psscan</td>
<td>192 GiB</td>
<td>3m8.421s</td>
</tr>
<tr>
<td>psscan (Rekall)</td>
<td>192 GiB</td>
<td>6m7.207s</td>
</tr>
<tr>
<td>psscan (Volatility)</td>
<td>192 GiB</td>
<td>4m42.412s</td>
</tr>
</tbody>
</table>

Notes
- About 2 orders of magnitude speedup versus other methods
- psscan linear in RAM size, not psquickscan
A Note on Limitations

• Our limitations are inherent to scanning in virtual address space
• Starting in Windows 10 Microsoft obfuscates _OBJECT_HEADERs using the VA of the allocation
• Must scan in kernel’s virtual address space
• tl;dr - Existing tools may have the same limitations as us starting with Windows 10
Conclusions

• New technique: limit pool tag scanning to pools where allocations for these objects are made
• Significantly more efficient
  – time: order of magnitude+ speedup
  – network bandwidth
• Minimal loss of accuracy
  – no processes from previous boot
  – terminated processes in deallocated pages not found
  – we’d have these limitations in Windows 10+ anyway
Future Work

• More testing of pool sizes with different workloads
• Quantify the incidence of objects in deallocated pages
• Find a way to scan a subset of deallocated pages that might hold fun stuff
Questions?

Vico Marziale
vico@blackbagtech.com
@vicomarziale

Joe Sylve*
joe@blackbagtech.com
@jtsylve

*after today, ask him