Pool Tag Quick Scanning for Windows Memory Analysis

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

DFRWS EU 2016

BlackBag TECHNOLOGIES

THE UNIVERSITY of NEW ORLEANS

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

```
nt!_POOL_HEADER
struct _POOL_HEADER, 9 elements, 0x10 bytes
   +0x000 PreviousSize        : Bitfield Pos 0, 8 Bits
   +0x000 PoolIndex           : Bitfield Pos 8, 8 Bits
   +0x000 BlockSize           : Bitfield Pos 16, 8 Bits
   +0x000 PoolType            : Bitfield Pos 24, 8 Bits
   +0x000 Ulong1              : Uint4B
   +0x004 PoolTag             : Uint4B
   +0x008 ProcessBilled       : Ptr64 to struct _EPROCESS
   +0x008 AllocatorBackTraceIndex : Uint2B
   +0x00a PoolTagHash         : Uint2B
```

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
```

Purpose	Pool Tag
Driver Object	Driv
File Object	File
Kernel Module	MmLd
Logon Session	SeLs
Process	Proc
Registry Hive	CM10
TCP Endpoint	TcpE
TCP Listener	TcpL
Thread	Thre
UDP Endpoint	UdpA

Big (Large) Page Pool

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

Info about allocations at nt!PoolBigPageTable

```
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
  - *psscan* to exhaustively search physical memory
  - *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

Plugin	Type	Avg. Time	Running	Terminated	Prior Boot	Duplicate[4]
psquickscan	Virtual	0.129s	128	21	0	0
psscan	Physical	15.584s	128	22	15	43
psscan (Rekall)	Physical	35.967s	128	22	15	43
psscan (Volatility)	Physical	25.448s	128	21	15	43

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*

OS Version	Plugin	Data Scanned	RAM Size	Avg. Time	Running	Terminated	Duplicate
Vista SP0	psquickscan	38 MiB	1 GiB	0.083s	46	2	15
Vista SP0	psscan	1 GiB	1 GiB	0.356s	46	2	15
Vista SP1	psquickscan	60 MiB	1 GiB	0.073s	48	0	0
Vista SP1	psscan	1 GiB	1 GiB	0.400s	48	0	0
Vista SP2	psquickscan	76 MiB	1 GiB	0.236s	50	1	0
Vista SP2	psscan	1 GiB	1 GiB	0.547s	50	1	11
7 SP0	psquickscan	64 MiB	2 GiB	0.075s	43	4	0
7 SP0	psscan	2 GiB	2 GiB	0.712s	43	6	4
7 SP1	psquickscan	64 MiB	2 GiB	0.075s	50	5	0
7 SP1	psscan	2 GiB	2 GiB	0.691s	50	5	0
8	psquickscan	44 MiB	4 GiB	0.054s	36	3	0
8	psscan	4 GiB	4 GiB	1.433s	36	3	0
8.1	psquickscan	244 MiB	8 GiB	0.170s	45	0	0
8.1	psscan	8 GiB	8 GiB	2.977s	45	0	0

typo in paper!

Notes

- About an order of magnitude speedup

Scenario 3: Network Data Transfer

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

RAM Size	Plugin	Scanned	Time	Transferred
2 GiB	psquickscan	102 MiB	9.489s	116.115 MiB
2 GiB	psscan	2 GiB	28.132s	2.014 GiB
4 GiB	psquickscan	122 MiB	9.640s	177.367 MiB
4 GiB	psscan	4 GiB	56.971s	4.027 GiB
8 GiB	psquickscan	246 MiB	15.360s	299.648 MiB
8 GiB	psscan	8 GiB	3m26.449s	8.132 GiB

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

Plugin	Data Scanned	Avg. Time
psquickscan	5.76 GiB	5.797s
psscan	192 GiB	3m8.421s
psscan (Rekall)	192 GiB	6m7.207s
psscan (Volatility)	192 GiB	4m42.412s

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