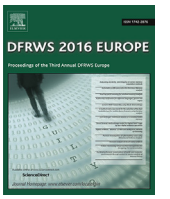
Digital Investigation 16 (2016) S25–S32

Contents lists available at ScienceDirect

Digital Investigation

journal homepage: www.elsevier.com/locate/diin



DFRWS 2016 Europe — Proceedings of the Third Annual DFRWS Europe

Pool tag quick scanning for windows memory analysis

Joe T. Sylve [a, b, *], Vico Marziale [a], Golden G. Richard III [b]

[a] Blackbag Technologies, Inc, San Jose, CA, USA
[b] Department of Computer Science, University of New Orleans, New Orleans, LA, USA

Keywords:
Microsoft windows
Memory analysis
Memory forensics
Live forensics
Pool tag scanning
Pool scanning
Incident response

A B S T R A C T

Pool tag scanning is a process commonly used in memory analysis in order to locate kernel object allocations, enabling investigators to discover evidence of artifacts that may have been freed or otherwise maliciously hidden from the operating system. The fastest current scanning techniques require an exhaustive search of physical memory, a process that has a linear time complexity over physical memory size. We propose a novel technique that we are calling “pool tag quick scanning” that is able to reduce the scanning space by 1–2 orders of magnitude, resulting in much faster discovery of targeted kernel data structures, while maintaining a high degree of accuracy.

Introduction

The Microsoft Windows operating system maintains several kernel mode heaps, known as “system memory pools” which store operating system kernel object allocations, e.g., _EPROCESS process descriptors, _FILE structures, etc. Since most pool allocations start with a _POOL_HEADER structure, a technique commonly known as “pool tag scanning” can be used to identify key OS-related forensic artifacts in physical memory images. Pool tag scanning was originally used for discovering structures associated with processes and threads, but is now widely used to target many kinds of data structures. It is particularly effective in detecting direct kernel object manipulation (DKOM), which is commonly used by malware to hide processes by removing references to the _EPROCESS allocation from other data structures. It can also be used to detect freed allocations that have not yet been overwritten. Yet another use for pool scanning is the recovery of kernel structures for which no better method has been developed, such as many structures associated with the Windows GUI subsystem. While pool tag scanning is effective, the most efficient existing techniques require a time consuming, exhaustive search of all physical memory to find structures of interest.

While current methods of pool scanning could be considered fast enough to analyze the majority of today's commodity systems, the process is linear over RAM size and sizes are quickly increasing. Windows 10 was recently released and supports physical RAM sizes of up to 2 TB for desktop systems. Modern versions of Windows Server support twice as much (Microsoft (2015b)).

As case loads increase, investigators often turn to batch processing of evidence. Reductions in the processing time of individual evidence sources can drastically reduce the overall analysis time.

During incident response scenarios time is also a critical factor and analysis is often done remotely over a network connection. A significant reduction in scanning time and network bandwidth requirements can make individual investigators better able to quickly detect and react to malicious behavior on a network. A fast enough scanning technique may also be useful for real-time detection of malware or other threats.

This paper presents a novel technique for pool tag scanning that limits scanning to only those physical memory pages that are identified as being a part of a

* Corresponding author.
E-mail addresses: joe.sylve@gmail.com (J.T. Sylve), vicodark@gmail.com (V. Marziale), golden@cs.uno.edu (G.G. Richard).

http://dx.doi.org/10.1016/j.diin.2016.01.005

system memory pool allocation. This technique greatly reduces scanning time by reducing the scanning space from the size of physical memory to the size of the allocated pool pages, which can easily be several orders of magnitude smaller. The method also significantly reduces the bandwidth requirements of preforming live memory analysis on a target system over a network.

Related work

Pool tag scanning

Schuster (2006) first introduced techniques for searching an entire image of physical memory for signatures associated with pool allocations to discover both currently active and freed (but not yet overwritten) kernel data structures, a technique now commonly referred to as "pool tag scanning" or just "pool scanning". Schuster (2008) showed that more than 90% of this information can often be retrieved even 24 h after process termination under optimum conditions. The two major open source memory analysis frameworks, Volatility[1] and Rekall[2] currently implement Schuster's scanning techniques.

Ligh (2013) introduced the -V and –virtual flags to Volatility. These flags enable pool tag scanning inside of the kernel's virtual address space, by performing an exhaustive search of the kernel's entire virtual address space. Since Volatility has no *a priori* mechanism for determining which pages are allocated, this approach requires page table lookups and address translation for every page in the kernel's address space, a process that reduces the amount of memory scanned, but is generally much *slower* than an exhaustive search of physical memory due to the lookup and translation overhead.

Cohen (2015) showed that Windows 10 obfuscates structures that are important to pool scanning with a value that is based off of the virtual address of the pool allocation. This makes pool scanning on physical memory ineffective against Windows 10 targets and thus requires a much slower exhaustive search of the kernel's virtual address space.

Kernel symbol lookups

Schreiber first described the internal structure of Microsoft program database (PDB) files as well as a methodology to look up and parse debug symbols (Schreiber, 2001, pp. 70–92).

Okolica and Peterson (2010) introduced the idea of using the debug information embedded in Microsoft's program database (PDB) files in a memory analysis tool to calculate symbol addresses in an arbitrary memory dump for any of the family of Windows NT operating systems.

Cohen and Metz (2014) introduced the functionality to parse PDB files and calculate kernel symbol addresses into Rekall.

[1] The Volatility Foundation, http://www.volatilityfoundation.org/.
[2] The Rekall Team, http://www.rekall-forensic.com/.

Table 1
Selected *non-paged pool* allocations.

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

Memory pools

The Windows kernel maintains several dynamically-sized memory pools, or heaps, that most kernel-mode components use to allocate system memory. The *non-paged pool* consists of ranges of system virtual addresses that are guaranteed to reside in physical memory at all times. The kernel also maintains more than one *paged pool* that can be paged into and out of the system. Both memory pools are located in the system part of the address space and are mapped in the virtual address space of every process. In addition to the *paged* and *non-paged* pools, there are a few other pools with special attributes or uses. For example, there is a pool region in session space, which is used for data that is common to all processes in the session (Russinovich et al., 2012, pp. 212–213).

The majority of key kernel structures, such as those shown in Table 1 are allocated on the *non-paged pool*. For example they include objects associated with running and terminated processes, network connections, and loaded kernel modules. Combined with the fact that *non-paged pool* pages are guaranteed to be resident in physical memory, it is evident that the *non-paged pool* is most relevant to memory analysts.

The remainder of this paper will focus on analysis of the *non-paged pool* for 64-bit versions of Windows from Windows Vista to Windows 8.1; however, the techniques described here can also be adapted to other pool types and operating system versions.

Pool sizes

The initial size of the *non-paged pool* is dependent on the amount of physical memory on the system, and is 3% of system RAM or 40 MiB[3] (whichever is larger). On 64-bit systems the pool can grow to a maximum of 75% of system RAM or 128 GiB (whichever is smaller) (Russinovich et al., 2012, p. 213).

64-bit versions of Windows dynamically allocate the memory reserved for the pool. While the initial pool sizes as described above are reserved by the memory manager, they are very sparsely allocated. The absolute minimum allocated amount is unknown, but we have observed *non-*

[3] The initial size of the *non-paged pool* is 10% of system RAM for systems with less than 400 MiB of RAM.

paged pools with as few as 64 MiB of allocated pages for systems with many gigabytes of RAM.

Big page pool

Pool allocations with sizes greater than 4064 bytes on 64-bit Windows and 4080 bytes on 32-bit Windows, regardless of type, are stored in the *big page pool*. This table is also commonly referred to as the "large pool allocation table". These allocations can not be found using pool tag scanning; however, the table is easy to enumerate.

The big page pool table is stored at the symbol nt!PoolBigPageTable and consists of an array of _POOL_TRACKER_BIG_PAGES structures as defined in Fig. 1. The size of this array is stored at the symbol nt!PoolBigPageTableSize.

Va will point to the virtual address of the allocation. Allocations that have been freed will have the least significant bit of Va set to 1. The allocation's pool tag will be stored in Key, its type in PoolType, and its size in NumberOfBytes.

Pool tag scanning

All Windows pool allocations smaller than 4064 bytes on 64-bit Windows and 4080 bytes on 32-bit Windows begin with a _POOL_HEADER structure as defined in Fig. 2 and are allocated on ChunkSize boundaries. On modern versions of Windows ChunkSize is defined as 8 bytes on 32-bit systems and 16 bytes on 64-bit systems. While scanning on ChunkSize boundaries there are three main fields that are suitable for validation at their respective offsets: BlockSize, PoolType, and PoolTag.

By multiplying BlockSize by ChunkSize we are able to determine the size of the pool allocation. The allocation size must be large enough to fit the _POOL_HEADER and the allocated object itself, including the _OBJECT_HEADER and any applicable optional headers.

A valid PoolType will contain a _POOL_TYPE value as defined in Fig. 3. In general *non-paged pool* allocations will have an even PoolType value. Allocations that have been freed will have a PoolType value of zero, but may still be stored in the *non-paged pool*.

Pool allocations are associated with a 4-byte pool tag. Some examples can be found in Table 1. PoolTag will contain the allocation's pool tag stored as a 32-bit, big-endian integer. Prior to Windows 8 the kernel marked "protected" allocations by setting the most significant bit of PoolTag, so care should be taken to scan for both variants.

Scanning with a signature based solely on these three fields may still produce false positives, so additional validation specific to the type of structures being scanned for should be used. For example valid _EPROCESS allocations store the physical address of the process's page table, commonly known as the DTB, in the field Pcb.DirectoryTableBase. The DTB should not be zero and must be page aligned, except on systems with page address extension (PAE) enabled, in which case the alignment is on a 32 byte boundary. Signatures can be significantly strengthened by introducing sanity checks based on this and other similar information.

```
kd> dt -v _POOL_TRACKER_BIG_PAGES
nt!_POOL_TRACKER_BIG_PAGES
struct _POOL_TRACKER_BIG_PAGES, 4 elements, 0x18 bytes
   +0x000 Va               : Ptr64 to Void
   +0x008 Key              : Uint4B
   +0x00c PoolType         : Uint4B
   +0x010 NumberOfBytes    : Uint8B
```

Fig. 1. Definition of _POOL_TRACKER_BIG_PAGES.

```
kd> dt -v _POOL_HEADER
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
```

Fig. 2. Definition of _POOL_HEADER on 64-bit windows.

```
kd> dt -v _POOL_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
```

Fig. 3. Definition of _POOL_TYPE.

Pool tag quick scanning

Pool tag scanning has a linear time complexity over its scanning space; however, since the fastest current methods require an exhaustive search of physical memory, this scanning space can be quite large. We have developed a method of reducing the required scanning space by multiple orders of magnitude by identifying and scanning only the memory pages associated with pool allocations, while still maintaining a high degree of accuracy.

In 64-bit Windows the virtual address ranges for each pool are managed by a Dynamic Virtual Address (DVA) subsystem. The DVA allows sparse allocation of virtual address space, enabling the kernel to reserve a large range of addressees for a pool, but to only allocate physical pages when needed. The kernel maintains a virtual address

Table 2
Relevant kernel symbols for identifying *non-paged pool* ranges.

Windows version	Static ranges	Start of dynamic allocation	Allocation bitmap
Vista SP0	MmNonPagedPoolStart – MmNonPagedPoolEnd0	MmNonPagedPoolExpansionStart	MiNonPagedPoolVaBitMap
Vista SP1 – 8	N/A	MiNonPagedPoolStartAligned	MiNonPagedPoolVaBitMap
8.1	N/A	MiNonPagedPoolStartAligned	MiDynamicBitMapNonPagedPool

allocation bitmap for each pool in order to keep track of which pages in the pool's virtual address range are backed by allocated physical pages. Each entry in the bitmap represents a 2 MiB physical allocation. By iterating through a pool's allocation bitmap, we can identify which pages in the pool's virtual address range are backed by allocated physical pages and limit our scanning to those pages.

Virtual address range identification

Before we can start scanning we must first identify the virtual address ranges associated with the *non-paged pool*. The size of the pool (in bytes) is stored at the symbol nt!MmMaximumNonPagedPoolInBytes.

For Windows Vista SP0 the pool is separated into two distinct address ranges. First, we must scan the ranges between the addresses stored at the kernel symbols nt!MmNonPagedPoolStart and nt!MmNonPagedPoolEnd0. These pages are guaranteed to be backed by physical pages, so we do not have to reference the virtual address bitmap; however, the bytes allocated in these ranges are included in nt!MmMaximumNonPagedPoolInBytes. The kernel symbol nt!MmNonPagedPoolExpansionStart stores the address of the beginning of the dynamically-allocated portion of the *non-paged pool*. To scan this range, we must reference the allocation bitmap to locate the allocated pages.

For later versions of Windows there is only a single, dynamically-allocated range and it starts at the address stored at nt!MiNonPagedPoolStartAligned. To scan this range, we must reference the allocation bitmap to locate the allocated pages.

The information in this section is summarized in Table 2.

Bitmap location

Starting with Windows Vista, and in versions of Windows prior to Windows 8.1 the *non-paged pool* DVA allocation bitmap can be found at the address provided by symbol nt!MiNonPagedPoolVaBitMap. Starting with Windows 8.1 the allocation bitmap is located at the address provided by nt!MiDynamicBitMapNonPagedPool. Memory analysis tools with the ability to parse debug symbols from PDBs can easily look up the offsets of these symbols inside of the kernel image and calculate their addresses based on the kernel's base address.

_RTL_BITMAP iteration

Fig. 4 shows the very simple structure of the bitmaps used throughout the kernel. SizeOfBitMap specifies the number of bits in the bitmap and Buffer points to the beginning of the bitmap data. The bitmap data is stored as a sequence of 32-bit integers. The first 32 bits of the bitmap are stored in bits 0–31 of the first integer in the buffer. The next 32 bits are stored in bits 0–31 of the second integer in the buffer and so on.

```
kd> dt -v _RTL_BITMAP
ntdll!_RTL_BITMAP
struct _RTL_BITMAP, 2 elements, 0x10 bytes
   +0x000 SizeOfBitMap     : Uint4B
   +0x008 Buffer           : Ptr64 to Uint4B
```

Fig. 4. Definition of _RTL_BITMAP.

In the case of the pool bitmaps, a set bit represents 2 MiB of virtual address space that is backed by allocated physical memory. An unset bit represents an unbacked 2 MiB range of virtual address space.

Scanning

Once the relevant memory regions are identified, scanning is just a matter of iterating through the bitmap to determine which 2 MiB pages should be scanned using the techniques described in Schuster (2006).

Metrics

In order to determine the efficacy of quick scanning we developed both a *psquickscan* and *psscan* plugin in our experimental memory analysis framework. The *psquickscan* plugin uses the quick scan technique to search the *non-paged pool* for *_EPROCESS* allocations, while the *psscan* plugin performs an exhaustive search of physical memory. The architecture of the experimental framework itself will be the subject of a future publication. All tests were performed on a mid-2014, 2.8 GHz MacBook Pro with 16 GiB of RAM. All reported time measurements are averages over 10 executions with the highest and lowest values removed.

Comparison to pool scanning (physical)

A memory image was taken from a moderately-used Windows 7 SP1 x64 laptop with 16 GiB of RAM. We then compared the results from *psquickscan* to those of *psscan* as well as the output of the *psscan* plugins from both Rekall[4] and Volatility[5] using the default configuration of scanning physical memory. A summary of the results can be found in Table 3 along with the average execution time of each

[4] Rekall 1.4.1, commit 72405db3ded58de43cf9564aa53c155963404536.

[5] Volatility 2.4, commit 2c84ee4370eec4ed20ca9ccd6ca1f91423be121f.

Table 3
A comparison of *psquickscan* and *psscan* results.

Plugin	Type	Avg. time	Running	Terminated	Prior boot	Duplicate[a]
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

[a] Duplicate results are redacted in other columns.

plugin over 10 runs, excluding the highest and lowest values.

Our *psquickscan* plugin reported that it scanned only 80 MiB of the 16 GiB memory image, while the *psscan* plugins all performed an exhaustive search of the entire 16 GiB memory image. The significantly reduced scanning space resulted in a two order of magnitude decrease in scanning time. Our *psscan* implementation produced comparable results to the Rekall and Volatility implementations.

More than 20% of the *psscan* results reported were exact duplicates caused by two or more identical _EPROCESS allocations being found in physical memory at different offsets. This is likely caused by allocations being copied as they are relocated in physical memory, possibly as a result of the memory manager using compacting garbage collection techniques. As these duplicate results are not considered useful to investigators, we consider them to be invalid and only focus on unique results. *psquickscan* did not report any duplicate results.

There is a minor discrepancy in the Volatility *psscan* results as compared to the other *psscan* implementations. This can likely be attributed to slight differences in the way that candidate _EPROCESS structures are validated after scanning. In this case, Volatility's validation methods resulted in a single false-negative.

Chow et al. (2005) noted that the effects on RAM resident data across reboots varies depending on the hardware used. In some cases, volatile evidence may survive even a "hard" reboot, when all power is removed from the machine for a moderate period of time. While this phenomenon is hardware dependent, our testing suggests the laptop that we used as the target of our efficacy testing does exhibit this behavior. While comparing the *psquickscan* and *psscan* results we noted that the *psquickscan* results were a subset of those from *psscan*. We also noted that all but one of the remaining *psscan* results had a creation timestamp that was several days prior to the system boot time (as shown by the creation timestamp of the System process). We can infer that because *psscan* searches all of physical memory it is able to locate pool allocations from previous system boots that have not been overwritten or lost during the reboot process. Because *psquickscan* is limited to searching only the virtual address space of the *non-paged pool* it is not able to find these allocations. Given the increased speed of our technique, we consider this a reasonable trade-off. In non-time critical situations where analysis is conducted on machines where data remanence is possible, a complete scan may be more appropriate; however, starting in Windows 10 this may not be possible as all existing techniques for pool scanning on that system require a search of virtual memory (Cohen (2015)).

A single unique result from the current system boot was found by all of the *psscan* implementations and not *psquickscan*. In order to determine the cause, further analysis was needed.

Fig. 5 shows select *psscan* output from Rekall for the allocation that was not found by *psquickscan*. Rekall attempts to map the physical address of the _EPROCESS struct to a virtual address. By looking at the Offset(V) column we can see that it fails, providing no address. Since our memory dump is in the Microsoft Crash Dump format, we can use WinDbg[6] to try to determine the reason.

The WinDbg !pfn command gives us information about a physical memory page by parsing its entry in the nt!MmPfnDatabase. Fig. 5 shows us that the _EPROCESS allocation of interest resides at physical address 0x9392890. By dividing this address by the page size (4096) we can obtain its page frame number (PFN), 0x9392. Passing this number to the WinDbg !pfn command results in the output in Fig. 6. We notice that the reference count is 0 and that the page is on the standby list. This tells us that the page is not mapped in any virtual address space (Russinovich et al., 2012, pp. 315–316). This demonstrates another limitation with scanning in the virtual address space. We can not detect freed allocations that reside in pages that are no longer mapped into the kernel's address space. Because Windows zeroes process pages during idle time (Chow et al. (2005)), allocations like these are rare, but further research is needed to attempt to quantify the impact of this limitation.

Comparison to pool scanning (virtual)

As stated above, kernel virtual memory scanning techniques are slower than exhaustive physical memory searches, but for completeness, we present metrics here showing the speed of kernel virtual memory scanning versus quick scanning. To compare, we reran our experiments using the –scan_in_kernel flag in Rekall and the –virtual flag in Volatility. Each of these flags enables pool scanning in the kernel's virtual memory space.

As shown in Table 4, quick scanning was just as accurate as an exhaustive search of virtual memory, while being two orders of magnitude faster. Enabling virtual scanning caused the scanning time to double in both Volatility and Rekall as compared to physical scanning as seen in Table 3. The exhaustive searches performed by Volatility and Rekall did report a single duplicate value. While duplicate values are not common when scanning virtual memory space, in

[6] Microsoft, 2015a, https://msdn.microsoft.com/en-us/library/windows/hardware/ff551063(v=vs.85).aspx.

```
_EPROCESS (P)    Name         PID   Offset(V)  PPID  DTB
--------------   ----------   ----  ---------  ----  --------------
...
0x000009392890   chrome.exe   4168  -          4852  0x00008be6e000
...
```

Fig. 5. Select Rekall *psscan* output.

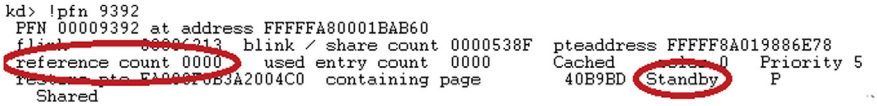


Fig. 6. WinDbg output for PFN 9392.

Table 4
A comparison of virtual scanning.

Plugin	Type	Avg. time	Running	Terminated	Prior boot	Duplicate[a]
psquickscan	Virtual	0.129s	128	21	0	0
psscan (Rekall)	Virtual	71.513s	128	21	0	1
psscan (Volatility)	Virtual	60.526s	128	19	0	1

[a] Duplicate results are redacted in other columns.

Table 5
A sample of *psquickscan* and *psscan* results among different OS versions.

OS version	Plugin	Data scanned	RAM size	Avg. time	Running	Terminated	Duplicate
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

rare circumstances they may occur. These results can be attributed to analysis during the short window of time between an allocation being copied for relocation and its original location being freed from virtual memory.

Sample of results

Comparisons of *psscan* and *psquickscan* results across multiple versions of Windows can be found in Table 5. All images come from lightly used systems with varying amounts of physical RAM.

Because the first ranges scanned in Windows Vista SP0 are not associated with dynamically allocated memory, many duplicate results can be found. The pools in later versions of Windows are entirely dynamically allocated and thus do not show this behavior. All other observed results were as described in the previous sections.

Large memory testing

In order to compare relative scanning speed of each plugin on systems with large amounts of memory we collected a memory image from a fresh install of Windows 7 SP1 with the maximum of 192 GiB of RAM. The results of running the various plugins can be found in Table 6.

The *psquickscan* plugin reported that it scanned 5.76 GiB of data and had an average scanning time of under 6 s, while the *psscan* plugins scanned the entire 192 GiB and took several minutes.

It is worth noting that at the time of this writing the Rekall implementation of *psscan* does not seem to produce complete results with large memory images. This is due to a bug where the PoolIndex field of the _POOL_HEADER is not being validated properly. This bug has been reported to the developers and only affects the reporting of results. Since

Table 6
Comparison of scanning speed on a 192 GiB Memory Image.

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

Table 7
Bandwidth comparison using F-Response to scan a Windows target.

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

the entire image is still scanned, the timing data should not be affected.

Bandwidth testing

In incident response scenarios investigators are often tasked with locating and triaging indicators of malware across multiple systems as quickly as possible. A common solution is to use a system like F-Response[7] to directly access RAM on a target system remotely. Investigators can then use standard memory analysis tools to analyze the live RAM of the target system from a remote, host machine. In these situations time is often a critical factor and network bandwidth limitations may be a bottleneck, especially when analyzing systems that are geographically located far away from the investigator. Since quick scanning does not have to perform an exhaustive search of RAM, it is very well suited to these scenarios.

In order to measure the relative impact of quick scanning on network bandwidth we ran F-Response on a number of target Windows systems. For each target system, we mounted the RAM as an iSCSI device on a host Linux analysis machine located on the same gigabit LAN. While collecting network packet captures we then ran either our *psquickscan* or *psscan* plugin against the target's live RAM. After scanning we analyzed the network traffic and noted the amount of data transferred between the target and host systems. The results of our analysis can be found in Table 7. In all instances quick scanning performed much faster with significantly reduced bandwidth requirements.

Conclusion

Pool quick scanning is a novel technique for pool tag scanning that limits scanning to only those physical memory pages that are identified as being a part of a system memory pool allocation. This technique greatly reduces scanning time by reducing the scanning space from the size of physical memory to the size of the allocated memory pool pages, which can easily be multiple orders of magnitude smaller. The method also significantly reduces the bandwidth requirements of performing live memory analysis on a target system over a network.

Quick scanning is limited in the sense that it can not locate allocations that are not mapped in the kernel's virtual address space, such as allocations from previous system boots that were not overwritten during the reboot process. This limitation is not unique to quick scanning, but is inherent to all methods of scanning virtual memory, including the existing methods needed to scan Windows 10 systems. Allocations that have not been freed and have been hidden by DKOM are always mapped in the virtual address space and can be detected by quick scanning.

Quick scanning is especially useful in incident response situations where time is a critical factor or in the case of remote analysis where network bandwidth is limited. Due to the inherent speed of quick scanning, it may also be useful for real-time detection of malware or other threats.

Future work

We know that scanning time is linear over scanning space and that quick scanning allows us to reduce that space significantly by limiting scanning to only allocated pool pages. While the theoretical limit of the pool size can be as high as 75% of system RAM, we have observed allocations ranging from 36 MiB to 246 MiB in size. Future work will include empirical testing over a wide variety of OS versions, system RAM sizes, and work loads.

References

Agile Risk Management LLC, 2003–2016. F-response. https://www.f-response.com.

Chow J, Pfaff B, Garfinkel T, Rosenblum M. Shredding your garbage: reducing data lifetime through secure deallocation. In: Proceedings of the 14th conference on USENIX security symposium – volume 14. SSYM'05. USENIX association, Berkeley, CA, USA; 2005. 22–22.

Cohen M. Rekall memory forensics blog: adding rekall's windows 10 support. June 2015. http://rekall-forensic.blogspot.com/2015/06/adding-rekalls-windows-10-support.html.

Cohen M, Metz J. Implemented the ms pdb parser. google/rekall@89f4f28. January 2014. https://github.com/google/rekall/commit/89f4f2832d99eac3b783b02ce9025806eaca6bd8.

Ligh M. Update the scanning framework to support win 8 and server 2012. implevolatilityfoundation/volatility@1b8e547. October 2013. https://github.com/volatilityfoundation/volatility/commit/1b8e54783fb7b650e105a10750317d0f55733c6c#diff-ec2ad15e066f752d0fc8606485c76a88.

Microsoft. Debugging tools for windows (windbg, kd, cdb, ntsd) – windows 10 hardware dev. 2015. https://msdn.microsoft.com/en-us/library/windows/hardware/ff551063(v=vs.85).aspx.

Microsoft. Memory limits for windows and windows server releases (windows). 2015. https://msdn.microsoft.com/en-us/library/windows/desktop/aa366778(v=vs.85).aspx#physical_memory_limits_windows_10.

Okolica J, Peterson GL. Windows operating systems agnostic memory analysis. In: Digital investigation 7, supplement, S48 – S56, the proceedings of the tenth annual DFRWS conference; 2010.

Russinovich M, Solomon D, Ionescu A. Windows internals. No. 2 in developer reference series. Microsoft Press; 2012.

Schreiber SB. Undocumented windows 2000 secrets: a programmer's cookbook. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.; 2001.

Schuster A. Pool allocations as an information source in windows memory forensics. In: IMF; 2006. p. 104–15.

[7] Agile Risk Management LLC, https://www.f-response.com/.

Schuster A. The impact of microsoft windows pool allocation strategies on memory forensics. In: Digital investigation 5, supplement, S58 – S64, the proceedings of the eighth annual DFRWS conference; 2008.

The Rekall Team, 2013–2016. The rekall memory forensic framework. http://www.rekall-forensic.com/.

The Volatility Foundation, 2007–2016. The volatility framework. http://www.volatilityfoundation.org/.