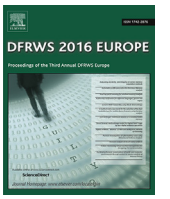
Digital Investigation 16 (2016) S25–S32

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.