Digital Investigation 20 (2017) 34–43

Contents lists available at ScienceDirect

Digital Investigation

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

Scanning memory with Yara

Michael Cohen

Google Inc., 747 6th St., Kirkland, WA, USA

ARTICLE INFO

Article history:
Received 1 September 2016
Received in revised form
5 January 2017
Accepted 18 February 2017
Available online 21 February 2017

Key Words:
Memory analysis
Reverse engineering
Windows internals
Operating system
Forensic analysis
Malware detection
Intrusion detection

ABSTRACT

Memory analysis has been successfully utilized to detect malware in many high profile cases. The use of signature scanning to detect malicious tools is becoming an effective triaging and first response technique. In particular, the Yara library and scanner has emerged as the defacto standard in malware signature scanning for files, and there are many open source repositories of yara rules. Previous attempts to incorporate yara scanning in memory analysis yielded mixed results. This paper examines the differences between applying Yara signatures on files and in memory and how yara signatures can be developed to effectively search for malware in memory. For the first time we document a technique to identify the process owner of a physical page using the Windows PFN database. We use this to develop a context aware Yara scanning engine which can scan all processes simultaneously using a single pass over the physical image.

Introduction

Memory Scanning has been used as a quick and powerful way to detect anomalies or malicious software running on a system. For example, pool scanning techniques have been used to detect remnants of kernel objects such as exited processes, file handles and other kernel data structures – even after these have been freed from the active set (Sylve et al., 2016; Schuster, 2006). Scanning techniques can be used to identify and isolate encryption keys from process memory (Hargreaves and Chivers, 2008), and detect unique signatures for malware families (Oktavianto and Muhardianto, 2013).

There are a number of modes of applying scanning techniques – one can scan the process's virtualized view of memory, or the physical address space directly (i.e. the raw memory image itself). In general, scanning the physical address space tends to be faster because IO throughput is optimized (in the case where the user wants to exhaustively scan all processes). However scanning the Virtual Address Space may be more efficient when the user only wants to scan a targeted subset of running processes.

The Yara library and scanner has emerged as the defacto standard for communicating signatures used to identify malware files (Alvarez, 2016; Various, 2016). Popular memory forensic frameworks have provided the capabilities for applying Yara signatures directly on memory images (The Volatility Foundation, 2015; The Rekall Team, 2016).

In this paper we evaluate the existing state of the art in applying yara signatures within the memory analysis domain. In particular we consider the practical difference of scanning in the Virtual Process Address space, as opposed to scanning the Memory image directly.

We describe for the first time a technique, dubbed “Context Aware Scanning”, which uses the Windows PFN database to rapidly identify the owner of each physical page, and where that page is mapped in its virtual address space.

Using this technique provides sufficient context about each physical address to be able to associate related hits in a single coherent signature – even when the scan is performed over the physical address space. We demonstrate this technique as applied to the Yara scanning engine by implementing a powerful new context aware scanning methodology.

The novel scanning technique dubbed “Context-Aware” scanning, employs detailed understanding of the address translation process with optimized scanning of the physical address space, we are able to gain performance advantage over existing techniques and efficiently scan multiple processes simultaneously. Finally we suggest guidelines for constructing more robust, memory-centric signatures.

Finally we discuss the practical differences between the different scanning techniques discussed and their applicability in effective malware identification.

E-mail address: scudette@google.com.

http://dx.doi.org/10.1016/j.diin.2017.02.005
1742-2876/

Background

Malware identification through signature scanning

Identifying malware in files is a very common and established technique (Sathyanarayan et al., 2008). There are a number of approaches. On the one end of the scale the NSRL facilitates hash comparison analysis (Flaglien et al., 2011). This produces a high level of confidence if a hash matches that the file belongs to the suspected set. However, exact hash matching is very sensitive to small variations in the underlying file.

Commonly malware samples are not exactly identical, but rather are customized or are built from common source trees. Therefore malware samples can be clustered into malware *families*, suggesting that several samples are related to one another, although not identical.

Similarity hash matching is less sensitive to small variations in specific files and can be used to classify malware samples into respective families. However, calculating the similarity hash is resource intensive and less accurate than simpler approaches (Breitinger and Baier, 2012).

The YARA matching engine is commonly used to strike a balance between matching speed and matching accuracy (Griffin et al., 2009; Alvarez, 2016). The YARA signature rule format is an easy to understand domain specific language (DSL). A typical example of such a rule is given in Fig. 1 which is taken from a malware analysis report of the Mozart POS malware (Hoffman, 2015).

Each YARA rule contains several sections:

1. A metadata section is used to facilitate sharing and documenting the creation of the rule and the analysis.
2. The *strings* section lists named strings which may be encoded as hex, have wildcards or specify case insensitive matching or wide character match.
3. Finally the *condition* section specifies a logical match condition which, if evaluates to True, will trigger the rule's matching. In order to build in some flexibility into the signature, the condition may specify that only some of the strings should match, or a list of alternate matching conditions.

Yara signatures allow for constructing flexible indicators which can be used to recognize a sample as potentially belonging to a particular malware family. There are a number of public sources of Yara signatures (Various, 2016), however these are often designed to work on static executable files, rather than operate on the memory image of the running executable. Indeed Yara provides for constructs which do not easily translate to memory analysis (such as dereferencing data as file offsets, and PE specific indicators). These specialized rules should be avoided when writing signatures suitable for memory analysis. In this paper we do not consider signatures with more complex constructs than simple string matches evaluated in simple logical conditionals.

```
rule Mozart {
   meta:
        author = ''Nick Hoffman''
        description = ''Detects samples of the Mozart POS RAM scraping utility''
   strings:
        $pdb = ''z:\\Slender\\mozart\\mozart\\Release\\mozart.pdb'' nocase wide ascii
        $output = {67 61 72 62 61 67 65 2E 74 6D 70 00}
        $service_name = ''NCR SelfServ Platform Remote Monitor'' nocase wide ascii
        $service_name_short = ''NCR_RemoteMonitor''
        $encode_data = {B8 08 10 00 00 E8 ?? ?? ?? ?? A1 ?? ?? ?? ?? 53 55
 8B AC 24 14 10 00 00 89 84 24 0C 10 00 00 56 8B C5 33 F6 33 DB 8D 50 01 8D
 A4 24 00 00 00 00 8A 08 40 84 C9 ?? ?? 2B C2 89 44 24 0C ?? ?? 8B 94 24 1C
 10 00 00 57 8B FD 2B FA 89 7C 24 10 ?? ?? 8B 7C 24 10 8A 04 17 02 86 E0 BA
 40 00 88 02 B8 ?? ?? ?? ?? 46 8D 78 01 8D A4 24 00 00 00 00 8A 08 40 84 C9
 ?? ?? 2B C7 3B F0 ?? ?? 33 F6 8B C5 43 42 8D 78 01 8A 08 40 84 C9 ?? ?? 2B
 C7 3B D8 ?? ?? 5F 8B B4 24 1C 10 00 00 8B C5 C6 04 33 00 8D 50 01 8A 08 40
 84 C9 ?? ?? 8B 8C 24 20 10 00 00 2B C2 51 8D 54 24 14 52 50 56 E8 ?? ?? ??
 ?? 83 C4 10 8B D6 5E 8D 44 24 0C 8B C8 5D 2B D1 5B 8A 08 88 0C 02 40 84 C9
 ?? ?? 8B 8C 24 04 10 00 00 E8 ?? ?? ?? ?? 81 C4 08 10 00 00}
   condition:
      any of ($pdb, $output, $encode_data) or
      all of ($service*)
}
```

Fig. 1. A YARA rule used to detect the Mozart POS Malware.

Before we can discuss the differences between scanning a stand alone file and a process's memory image, we need to understand how an executable is loaded into a process's virtual memory.

The Windows virtual memory

The following is a brief introduction to the concept of virtual memory. While this is a widely understood concept in operating system design, recent advances in memory analysis techniques have made it possible to reconstruct a process's virtual memory view more accurately than previously possible (Cohen, 2015; Gruhn, 2015).

Modern computer systems use a memory management unit (MMU) to mediate access between the CPU's memory bus and the physical address bus. When the CPU attempts to access a memory address, the MMU performs a transformation in hardware on this address converting it to a Linear Address (Physical Address). It is this physical address which is used to index into the RAM chips in order to retrieve the data stored in that location.

The transformation performed by the MMU is guided by the use of *page tables*, which are configured and managed by the operating system. When resolving a virtual address to its physical address, the MMU divides the virtual address into bit groups and each bit group is used to index a different array of page table entries (PTE). A simplified lookup process for 64 bit AMD CPUs is illustrated in Fig. 2 (Intel, 2015).

It is important to realize that each process and the kernel itself has its own unique set of page tables — and therefore, each process has a unique view of virtual memory specific to itself. In fact, each process is free to address its entire virtual address space, relying on the MMU to route virtual address references to physical pages or else to generate the appropriate page fault interrupts for the kernel to resolve.

Fig. 3 illustrates a typical process's virtual memory layout. The virtual address space is broadly divided into large contiguous regions dedicated to specific uses by the process. For example, a file mapping (such as an executable mapped into the process's address space) dedicates a specific range of virtual addresses as backed by a file on disk. Alternatively the process may allocate memory to be privately used by itself (for example to be used by the heap or stack).

In order to keep track of the virtual address layout, the kernel maintains a set of kernel data structures called the Virtual Address Descriptors (VAD) (Dolan-Gavitt, 2007; Russinovich et al., 2012). While each VAD represents a single contiguous region, each page within this region can take on different states. This is illustrated in Fig. 3: The single mapped file region consists of both resident pages in physical memory as well as virtual pages only backed by the file

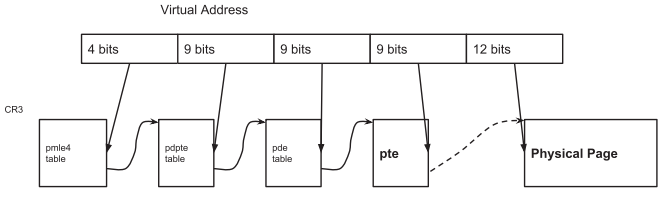

Fig. 2. A simplified illustration of Virtual to Physical address resolution in the AMD64 architecture. The Virtual Address is divided into bit groups and each bit group represents an index into a different page table. The page table entry contains a pointer to the next level table, if the page is valid.