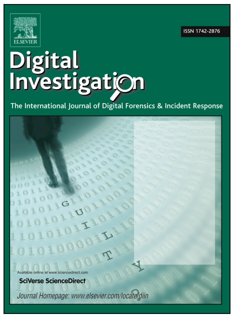

ISSN 1742-2876
Digital Investigation
The International Journal of Digital Forensics & Incident Response
SciVerse ScienceDirect
Journal Homepage: www.elsevier.com/locate/diin

Scanning memory with Yara

Michael Cohen

PII: S1742-2876(17)30059-2

DOI: 10.1016/j.diin.2017.02.005

Reference: DIIN 673

To appear in: *Digital Investigation*

Received Date: 1 September 2016

Revised Date: 5 January 2017

Accepted Date: 18 February 2017

Please cite this article as: Cohen M, Scanning memory with Yara, *Digital Investigation* (2017), doi: 10.1016/j.diin.2017.02.005.

Scanning Memory with Yara.

Michael Cohen[a]

[a] *Google Inc., 747 6th St. Kirkland, Washington, USA*

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.

1. 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 it's 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 suggests 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.

2. Background

2.1. 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

```
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*)
}
```

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

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 Figure 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.

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.

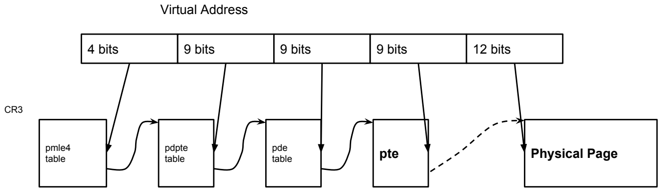


Figure 2: A simplified illustration of Virtual to Physical address resolution in the AMD64 architecture. The Virtual Address is divided into bit groups and each bit group represents an index into a different page table. The page table entry contains a pointer to the next level table, if the page is valid.

2.2. 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 Figure 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.

Figure 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 Figure 3: The single mapped file region

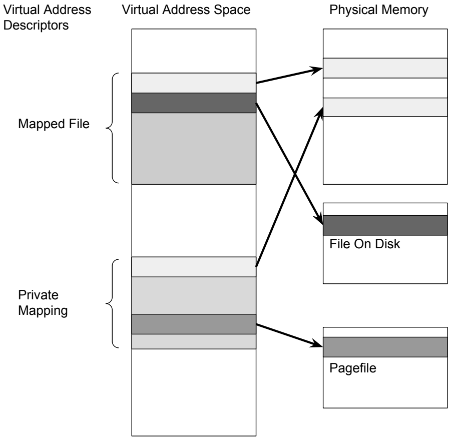


Figure 3: Examples of Virtual memory use between two processes.

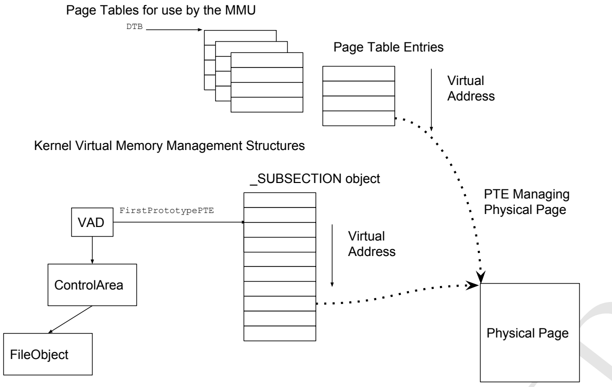


Figure 4: The kernel's virtual address space management structures.

consists of both resident pages in physical memory as well as virtual pages only backed by the file itself. If the process attempts to access these notional mapped pages, the kernel will map these pages from the file on demand - so that the entire file appears to be mapped into the process's virtual address space.

Figure 4 illustrates the major data structures used by the kernel to maintain information about the process's virtual address layout. There are two distinct structures - the page tables are managed by the kernel but are created for the sole benefit of the hardware's memory management unit (MMU). Typically page tables contain several levels which are traversed in the resolution process, but ultimately result in the discovery of a Page Table Entry (PTE) managing the particular virtual address.

The hardware PTE is read by the MMU, and if it is found to be in the Valid state, the MMU reads directly from the physical memory. If the Valid bit is not set, the MMU generates a *page fault* interrupt into the kernel's page fault handler. The remainder of the PTE bits can be freely used by the operating system for its own purposes.

Additionally, the kernel maintains a set of kernel data structures to keep track of the virtual address regions. Note that all kernel data structures are allocated from Non-paged pool resources and therefore are proceeded with their respective pool tags.

The main data structure representing the entire memory region is called the Virtual Address Descriptor (VAD). Each VAD represents a single contiguous range of virtual pages, and therefore each such virtual page is controlled via a *Prototype PTE*. Note that Prototype PTEs are allocated as a large single array from the pool area. Each VAD contains a pointer to a *Control Area* structure, in turn containing a reference to the file mapped into this region (in the case of a file mapping).

It is the interaction between the page tables and the VADs which ultimately control what data the process sees when addressing a particular virtual address. In order to reproduce the process's view of its own virtual memory, we must replicate the address translation process as faithfully as possible.

3. Scanning in the Virtual Address Space

When a process is started, it begins by mapping its own Portable Executable (PE) file into the virtual address space (i.e. a File Mapping is established between the file on disk and a certain Virtual Address range) (Russinovich et al., 2012).

The Windows kernel will then load all the DLLs the process needs into other Virtual Address ranges. Finally some private ranges will be created for the process heap and stack and the process will begin executing. As the process accesses different parts from its executables and DLLs, these pages will be read into memory (paged in) and the binary will execute code from there.

Figure 5 illustrates the Virtual Address Descriptors from a typical process on Windows.

There are some practical differences between scanning a single executable file using a Yara rule and scanning the virtual address space of a running process. The most obvious difference is that the process address space contains more than a single binary mapped into it - not only does it contain the original executable, but it also contains every DLL the binary loads. Since the process memory is examined during the process's runtime, even dynamically loaded DLLs (i.e. those loaded by runtime using the *LoadLibrary* API) will be visible. Some malware samples dynamically load DLLs which do not explicitly appear in their import table.

When scanning a process's virtual memory, the Yara rule can be made more sophisticated, as it can refers to strings in multiple regions. For example, combining artifacts from the process heap, code sections, and dependent DLLs. This additional information can be used to develop a stronger signature and reduce the false positive rate. For example, yara rules may be written that detect use of certain APIs or external DLLs indicative of behavioral traits. These indicators may not be visible in the original malware sample, especially if the original executable contains some form of obfuscation or packing.

Despite these obvious advantages to scanning in memory, there are some practical challenges. When scanning through

```
-      VAD        lev   Start Addr      End Addr     com   ------- ------       Protect        Filename
  -------------- --- -------------- -------------- ------                -------------------- --------
-----------------------------------
0xfa8000e87490 winpmem_1.6.2.  3056
-----------------------------------
  0xfa8000f71a50   5        0x10000        0x1ffff      0 Mapped          READWRITE
  0xfa8000e6f210   6        0x20000        0x2ffff      0 Mapped          READWRITE
  0xfa80028aaf80   4        0x40000        0x40fff      0 Mapped  Exe     EXECUTE_WRITECOPY    \Windows\System32\apisetschema.dll
  0xfa80010215a0   6        0x50000        0x53fff      0 Mapped          READONLY
  0xfa8002b20530   5        0x60000        0x60fff      0 Mapped          READONLY
  0xfa8000fbf920   6        0x70000        0x70fff      1 Private         READWRITE
  0xfa8000fb1a00   3        0x80000       0x17ffff      5 Private         READWRITE
  0xfa80010cb370   6       0x180000       0x1e6fff      0 Mapped          READONLY             \Windows\System32\locale.nls
  0xfa8002e79580   5       0x220000       0x25ffff      7 Private         READWRITE
  0xfa80028a1430   4       0x350000       0x3cffff      6 Private         READWRITE
  0xfa8001dbc450   6       0x4c0000       0x5bffff     21 Private         READWRITE
  0xfa80010b1bb0   7       0x5c0000       0x6c0fff    257 Private         READWRITE
  0xfa8002d19ba0   5       0x6e0000       0x6effff      3 Private         READWRITE
  0xfa80026e13f0   6       0x810000       0x81ffff      6 Private         READWRITE
  0xfa8000fc7710   2       0xb70000       0xcb1fff      4 Mapped  Exe     EXECUTE_WRITECOPY    \cygwin\home\mic\winpmem_1.6.2.exe
  0xfa8001147470   5     0x75270000     0x752cbfff      6 Mapped  Exe     EXECUTE_WRITECOPY    \Windows\System32\wow64win.dll
  0xfa80010c2a70   4     0x752d0000     0x7530efff      3 Mapped  Exe     EXECUTE_WRITECOPY    \Windows\System32\wow64.dll
  0xfa80024984f0   3     0x75340000     0x75347fff      2 Mapped  Exe     EXECUTE_WRITECOPY    \Windows\System32\wow64cpu.dll
  0xfa80025d5b50   6     0x75370000     0x7537bfff      2 Mapped  Exe     EXECUTE_WRITECOPY    \Windows\SysWOW64\cryptbase.dll
  0xfa8001d6eef0   5     0x75380000     0x753dffff      2 Mapped  Exe     EXECUTE_WRITECOPY    \Windows\SysWOW64\sspicli.dll
  0xfa8002791a80   4     0x75760000     0x7580bfff      8 Mapped  Exe     EXECUTE_WRITECOPY    \Windows\SysWOW64\msvcrt.dll
  0xfa8001c4a590   6     0x75810000     0x758fffff      2 Mapped  Exe     EXECUTE_WRITECOPY    \Windows\SysWOW64\rpcrt4.dll
  0xfa8001e49990   5     0x75ff0000     0x76008fff      4 Mapped  Exe     EXECUTE_WRITECOPY    \Windows\SysWOW64\sechost.dll
  0xfa8002f67220   6     0x76520000     0x7662ffff      3 Mapped  Exe     EXECUTE_WRITECOPY    \Windows\SysWOW64\kernel32.dll
  0xfa8002ebd5a0   1     0x774a0000     0x7753ffff      5 Mapped  Exe     EXECUTE_WRITECOPY    \Windows\SysWOW64\advapi32.dll
  0xfa8000f3f5d0   4     0x77540000     0x77586fff      3 Mapped  Exe     EXECUTE_WRITECOPY    \Windows\SysWOW64\KernelBase.dll
```

Figure 5: Virtual memory layout of a typical windows executable. Several Virtual Address Descriptors specify distinct regions for mapping executable files as well as some private memory regions.

virtual memory, one must emulate the process's view of its own virtual memory. Since the Virtual Address space is sparse, the scanner needs to skip over unused regions. Currently the Yara library does not support streamed scanning. This means that a single finite sized buffer must be given to the library, over which a rule-set can be evaluated in whole. Therefore, we must calculate where each page in the virtual address range will be read from in the memory image, and then copy it into a single contiguous finite sized buffer.

The procedure can be outlined therefore:

1. Enumerate all virtual memory ranges from all the process VADs (Similar to those shown in Figure 5).
2. For each page in every VAD range, consult the page tables according to the algorithm described in Section 3.1 below. This provides the correct locations where data should be read from.
3. Read the data for each virtual page and concatenate it into a single large buffer.
4. Apply the Yara scanning engine on this buffer and report any results.

While some memory analysis tools have supported running Yara scanning for a while (The Volatility Foundation, 2015; The Rekall Team, 2016), they suffer from some limitations.

Volatility uses a 1MB buffer to evaluate the Yara rule. Therefore all hits must occur within this small buffer. Since the Yara library does not preserve state between calls, each buffer is treated independently. It is therefore impossible to specify a rule with hits further apart than the 1MB buffer. Volatility does not parse Prototype PTEs (as described in Section 3.1) and therefore is unable to add file mappings into the buffer - even when the data is available in the memory image. Therefore the likelihood that a string will be missed is increased.

Although Rekall can process Prototype PTE pages and has more complete virtual address processing implementation, the need to partition the virtual address space into finite sized buffers so they can be handed to the Yara library also necessitate operating on individual 10Mb buffers.

3.1. Windows specific address translation

In order to facilitate more accurate scanning of the virtual address space, we need to understand how Windows specifically manages the virtual memory and how to recover each virtual page from a process's virtual address space.

The hardware generates a page fault while translating a page, by calling an interrupt into the OS page fault handler, passing in the faulting PTE. At this point the page fault handler uses a number of flags to determine which state the PTE is in. Windows uses the *_MMPTE* struct to describe the PTE which is a union of all the possible states the PTE can be in (The correct member of the union is chosen based on the flags (Cohen, 2015)).

Figure 6 shows the algorithm used for resolution of the PTE passed into the page fault handler (Sometimes termed the Hardware PTE). The Rekall memory analysis framework implements this algorithm in order to emulate the page-fault handler and deduce the correct physical page to use.

In the first stage the PTE might represent one of the following states:

1. Valid PTE: If Bit 0 is set the PTE refers to a page in physical memory.
2. If the ProtoType bit is unset and the Transition bit is set the page is in the Transition state. Its content is still valid and therefore we can directly read the data from the image.
3. If both the ProtoType bit and the Transition bit are unset, the PTE refers to a Software PTE (i.e. the data exists in the pagefile). The offset into the pagefile can be calculated from the *PageFileHigh* field. Except if the offset into the pagefile is 0, in this case, the VAD must be consulted and the ProtoType PTE recovered from the VAD and analyzed through the second stage algorithm.

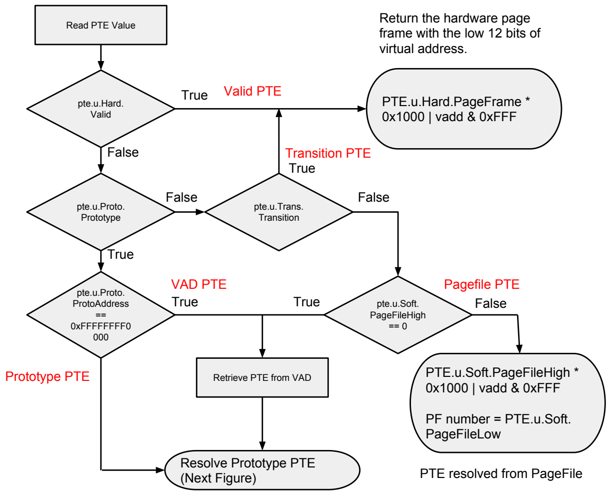

Read PTE Value
Return the hardware page frame with the low 12 bits of virtual address.
pte.u.Hard. Valid
True
Valid PTE
PTE.u.Hard.PageFrame * 0x1000 | vadd & 0xFFF
False
Transition PTE
True
pte.u.Proto. Prototype
False
pte.u.Trans. Transition
False
True
VAD PTE
pte.u.Proto. ProtoAddress == 0xFFFFFFFF0 000
True
True
pte.u.Soft. PageFileHigh == 0
Pagefile PTE
False
Prototype PTE
Retrieve PTE from VAD
PTE.u.Soft.PageFileHigh * 0x1000 | vadd & 0xFFF
PF number = PTE.u.Soft. PageFileLow
Resolve Prototype PTE (Next Figure)
PTE resolved from PageFile

Figure 6: Hardware PTE resolution algorithm.

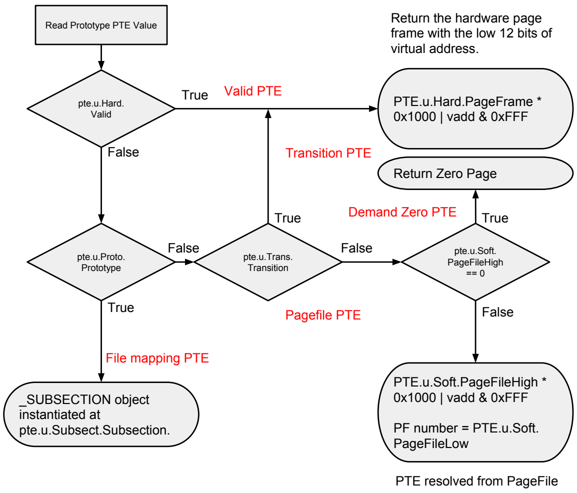

Read Prototype PTE Value
Return the hardware page frame with the low 12 bits of virtual address.
pte.u.Hard. Valid
True
Valid PTE
PTE.u.Hard.PageFrame * 0x1000 | vadd & 0xFFF
Transition PTE
Return Zero Page
False
True
Demand Zero PTE
True
pte.u.Proto. Prototype
False
pte.u.Trans. Transition
False
pte.u.Soft. PageFileHigh == 0
True
Pagefile PTE
False
File mapping PTE
_SUBSECTION object instantiated at pte.u.Subsect.Subsection.
PTE.u.Soft.PageFileHigh * 0x1000 | vadd & 0xFFF
PF number = PTE.u.Soft. PageFileLow
PTE resolved from PageFile

Figure 7: Prototype PTE resolution algorithm.

4. If the ProtoType bit is set, and the ProtoAddress field is 0xFFFFFFFF0000 then the ProtoType PTE must be fetched from the VAD entry corresponding with the relevant address. The ProtoType PTE is then found by using the *_MMVAD.FirstPrototypePte* pointer. The retrieved PTE is then processed further as a ProtoType PTE in the second stage.

If a ProtoType PTE was retrieved in the first stage, it is analyzed again to try to resolve the page. ProtoType PTEs are allocated from system pool and are not part of the hardware page tables (i.e. The MMU never reads a ProtoType PTE directly), but they share the same kernel structs and add some additional PTE states. Figure 7 illustrates the algorithm for resolving a prototype PTE. One major difference from the previous algorithms is the case where the prototype bit is set in a ProtoType PTE. The PTE then represents a Subsection Object (i.e. a File Mapping). When the page fault handler encounters a Subsection PTE it simply reads the data from the file into a new physical page. However, in a memory analysis framework we are unable to resolve this page without access to the corresponding disk image.

Additionally if the ProtoType PTE is a Software PTE (i.e. refers to the pagefile) but the offset into the pagefile is 0, the PTE is considered a “Demand Zero PTE”. In this case the kernel will re-purpose a zeroed page and remap it into memory on demand.

When scanning the virtual address space one must resolve the location of each virtual page independently. A typical example of such resolution is shown in Figure 8. As the Figure illustrates, contiguous virtual pages are rarely contiguous in the physical address space and are typically interleaved with file mapped pages (i.e. for these pages, the data is not present in memory at all, but can be read from the file on disk on demand) and pagefile pages (i.e. the needed data is in the pagefile). When Rekall assembles these pages into a single buffer it needs to seek to random locations on disk and therefore disk IO becomes a bottleneck, especially for very large images.

```
0x7cbb8000     0x3000 File Mapping
C:\WINDOWS\system32\shell32.dll @ 0x1f7400 (P)
0x7cbbb000     0x1000 Valid                PhysAS @ 0x551c000 (P)
0x7cbbc000     0x1000 Valid                PhysAS @ 0xe18000
0x7cbbd000     0x3000 Pagefile             PF 0 @ 0x4fe000
0x7cbc0000     0x1000 Valid                PhysAS @ 0x682b000
0x7cbc1000     0x6000 Transition           PhysAS @ 0x282b000 (P)
0x7cbc7000    0x12000 Demand Zero          (P)
```

Figure 8: An example of the output of Rekall’s *vadmap* plugin which explains the location of consecutive virtual pages.

4. Scanning physical memory

In many ways scanning memory is analogous to feature extraction in the traditional disk forensics domain. Each page in the physical image is analogous to a sector on the raw disk. Just as the filesystem ties related pages in the correct order into logical “files” (Cohen, 2007), so do the page tables tie physical pages into a logical “virtual address space”.

As we saw in the previous section, scanning the virtual address space of each process is inefficient since each buffer scanned must be assembled from a collection of non-contiguous pages fetched from all over the physical address space. This is akin to scanning each logical file from the filesystem, by recombining the discrete clusters which make up the logical file.

Taking guidance from the disk forensics domain, we can recognize that scanning in the physical address space is much faster since we can read large contiguous buffers from the image. For example *Bulk Extractor*, carves salient details from the raw disk, without regard to the filesystem at all (Garfinkel, 2013). Garfinkel (2013) describes how processing speed of hard disks increases linearly with the number of available cores demonstrating that direct feature extraction is a highly parallelizable problem.

When extracting features from the raw disk, the analyst lacks important context about those features. For example, if a certain email address is recovered, the analyst just knows that the email address once existed on this disk, but they do not know the logical file from which the email address was read. Typically in the disk forensic domain, one needs to build a reverse mapping between all files in the filesystem and their clusters (or perform an exhaustive search) in order to answer the question "which file does this cluster belong to?" (Carrier, 2016).

In an analogous way, memory analysis frameworks aim to answer a similar question "which process does this physical page belong to?". Current memory analysis frameworks tackle this problem in a similar way - each page in every process's virtual address space is enumerated and the corresponding physical page is calculated. A large reverse mapping is constructed mapping physical pages to their virtual pages. This algorithm is used by Rekall's *pas2vas* plugin (The Rekall Team, 2016). However building this reverse map is slow.

In the disk forensic domain, one can leverage some filesystem analysis to get some partial information. For example, most filesystems have a bitmap that can quickly identify if a cluster is in use at all (Carrier, 2005).

Luckily, in Windows there is an internal kernel data structure called the *Page File Number Database (PFN DB)* which can be used to quickly get very accurate information about every physical page (Russinovich et al., 2012). As we will see below, this makes matters much simpler for the case of carving physical memory, with no equivalent analog in the disk forensic domain.

4.1. Using the PFN DB to describe a physical page

Given a physical address where a possible signature matched, we would like to answer the following questions:

1. Which processes contain this page in their virtual address space?
2. Where is this page mapped in each process's address space?
3. If the page is mapped from a file, what is the filename and path?
4. If the page is a file mapping, where in the file is it from?

The Windows Page File Number database (PFN Db) is simply an array of *MMPFN* structs which starts at the symbol "nt!MmPfnDatabase" and has a single entry for every physical page on the system. The MMPFN struct must be as small as possible and so consists of many unions and can be in several states as indicated by the *MMPFN.u3.e1.PageLocation* field. Depending on the state, different fields must be interpreted in different ways. Below we discuss some of the states the PFN entry can be in.

4.1.1. Free, Zero and Bad lists

Windows maintains three linked lists of available physical pages. Free pages are those which are not used by anything and can utilized at a later stage. Zero pages are those that have been zeroed and may be immediately used. Bad pages are pages not to be used by anything since Windows suspects they are backed by faulty hardware.

4.1.2. Active pages

Active pages are physical pages currently in use by something. But what exactly?

The most important thing to realize is that each valid physical page (frame) must be managed by a PTE of some sort (Refer to the kernel's management structures illustrated in Figure 4). Since that PTE record must also be accessible to the kernel, it must be mapped in the kernel's virtual address space.

When the PFN entry is in the Active state, it contains 3 important pieces of information:

1. The virtual address of the PTE that is managing this physical page (in *MMPFN.PteAddress*).
2. The Page Frame (Physical page number) of the PTE that is managing this physical page (in *MMPFN.u4.PteFrame*). Note the above values provide both the virtual and physical address of the managing PTE.
3. The *OriginalPte* value (usually the prototype PTE which controls this page). When Windows installs a hardware PTE from a prototype PTE, it will copy the original prototype PTE into this field.

Note that the managing PTE structure can be either a Hardware PTE or a Prototype PTE (Figure 4).

4.1.3. Hardware managing PTE

If the managing PTE is a Hardware PTE, the field *MMPFN.u3.e1.PrototypePte* will be clear and the managing PTE will reside in the System PTE area (i.e. the managing PTE belongs in the hardware page tables region). This is the case when the virtual page is valid and can be immediately used by the hardware. Since the managing PTE itself must also be managed by the kernel, its own managing PTE will reside in the hardware page table area.

We can use this property to find the PTE which manages each PTE all the way up to the top level. Each time we do this, we will go back up the page tables until we reach to top most PTE which manages itself.

The interesting thing is that in this case, the PTE that is managing this page will belong in the Hardware page tables created for the process which is using this page. That PTE, in turn will also be accessible by a PDE inside that process's page tables, and so forth. This occurs all the way up to the root of the page table (DTB or CR3) which is its own PTE.

Therefore if we keep following the PTE which controls each PTE 4 times we will discover the physical addresses of the DTB, PML4E, PDPTE, PDE and PTE belonging to the given physical address. Since a DTB is unique to a process we immediately know which process owns this page because it is the process which owns this particular set of page tables.

Consulting the illustration in Figure 2 reveals that each bit grouping in the virtual address is really an index into the relevant page table. Knowing the DTB we can work forward and discover the start of each page table and since we know the actual PTE in each table that is used, we can calculate the required index that would have made the forward translation point to the correct PTE. Thus from the index of each table we can

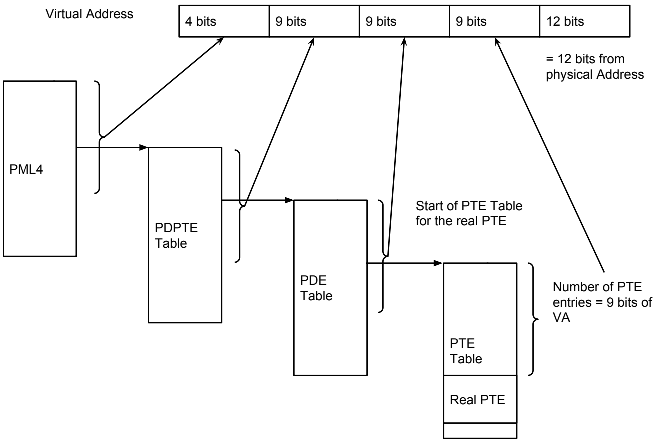

Virtual Address
4 bits
9 bits
9 bits
9 bits
12 bits
= 12 bits from physical Address
PML4
PDPTE Table
PDE Table
Start of PTE Table for the real PTE
PTE Table
Real PTE
Number of PTE entries = 9 bits of VA

Figure 9: Algorithm for reconstructing the Virtual Address from a Physical Address.

assign these into the relevant bit grouping and just work our way through the virtual address filling bit groupings in until the entire virtual address is known. This algorithm is illustrated in Figure 9.

Therefore in the case that the managing PTE is in the hardware page tables, we can calculate directly the virtual address and the process which owns the physical page. Note that by its nature, such a virtual address represents a page private to this process. If the page is shared with another process or file mapping, the controlling PTE will be a prototype PTE.

4.1.4. *Prototype managing PTE*

If the flag *MMPFN.u3.e1.PrototypePte* is set, the managing PTE is a prototype PTE. This means it is allocated from kernel pool memory. Typically Prototype PTEs are used to manage mapped memory (either shared memory, file mapping or private mapping).

If one attempts to use the algorithm described in Section 4.1.3 to find the DTB for the virtual address space, one will find that the DTB will always be the kernel's own DTB. This makes sense since the managing PTE is allocated from kernel pool and therefore it is only mapped from the kernel's page tables.

However, we do know that the managing PTE must belong inside a *SUBSECTION* object. As Figure 4 illustrates, mapped memory is managed by the *SUBSECTION* kernel data structure. The *SUBSECTION* object contains a linear array of prototype PTEs, each manage a single page from the specified mapping. Note that the file mapping managed by the *SUBSECTION* object does not need to begin from the middle of the file. The mapping can represent a small view into the file, starting from *SUBSECTION.StartingSector*.

As can be seen in Figure 4, when a process maps a file, it receives a new VAD descriptor. The *MMVAD* struct stores the first mapped PTE in *MMVAD.FirstPrototypePte* inside the *SUBSECTION* object as well as a pointed into the *SUBSECTION.ControlArea*, in turn pointing at the *FILE_OBJECT* with its filesystem path. Therefore by enumerating all VAD descriptors from all processes, we can immediately identify all the VAD descriptors which contain the managing PTE in their Prototype PTE arrays. If a VAD descriptor represents a file mapping we can get the full filename to the mapped file.

Building the initial lookup table is very fast since we just need to traverse the VAD trees for each process and identify the prototype PTE ranges for each descriptor.

The Rekall *ptov* plugin implements this algorithm, and it is able to reveal extra information about specific physical addresses.

```
[1] win7.elf 13:45:07> ptov 0x15bf0000
File Mapping (C:\Windows\System32\oleaut32.dll @ 0x8a600
Mapped in 0xfa80024f85d0 svchost.exe   236 @ 0x7feff37b000
Mapped in 0xfa80028a1640 WmiPrvSE.exe   592 @ 0x7feff37b000
Mapped in 0xfa80023f6770 svchost.exe   608 @ 0x7feff37b000
Mapped in 0xfa8002522b30 svchost.exe   624 @ 0x7feff37b000
Mapped in 0xfa800242a350 svchost.exe   716 @ 0x7feff37b000

[1] win7.elf 13:48:29> ptov 0x15bf2000
DTB 0xfd06000 Owning process: 0xfa80025b4060 svchost.exe  1092
PML4E@ 0xfd06f68 = 0xfd06863
PDPTE@ 0xfd06000 = 0x1a30000016da7867
PDE@ 0x16da7000 = 0x300000016cea867
PTE@ 0x16cea008 = 0x3c0000000f89c867
Physical Address 0x15bf2000
Virtual Address       0x2e3000 (DTB      0xfd06000)
```

Figure 10: Rekall's *ptov* plugin. Top: The physical address belongs to a mapped DLL (oleaut32.dll) at offset 0x8a600 in the DLL file. The page is shared between many different processes on the system and it happened to be mapped at the same virtual address in all of them. Bottom: The private page is owned by the *svchost.exe* executable and mapped at virtual address 0x2e3000.

5. Context aware signature scanning

We have seen in Section 2.1 how signatures can be written to balance false positives and flexibility and resilience to malware evolution. Specifically we saw that good Yara signatures typically contain more than one string and potentially some "fuzziness" to allow for only some of the signatures to match. For example a common technique is to match on Ṉ out of M̱ strings.

The problem is that when scanning in the physical address space, one collects some physical pages into a scan buffer and then applies the signature to that. However, the scan buffer actually contains a random set of pages from different processes, mapped files, and kernel code. On the other hand the signature is written with the view that the strings will be matched on the same binary file, or process virtual address space. In other words, each signature expects an implied *Context* around the signature (e.g. 3 of 5 strings must match within a particular process), but scanning in the physical address space mixes up the contexts from many processes within the same buffer.

On the other hand, applying the signature one page at the time is unlikely to match any reasonable signature because Yara will consider each scan buffer independently from other buffers (The yara library keeps no context between independent scans), only exasperating the problems noted in Section 3.

We have therefore developed a novel algorithm for applying Yara signatures on the physical address space - achieving both optimized scanning throughput and accurate signature matching.

5.1. Applying Yara Signatures over physical memory

Armed with the algorithms described in Section 4.1 we can rapidly extract context information from every physical address, such as name of process, mapped file etc. The challenge is to use this context information with existing Yara signatures.

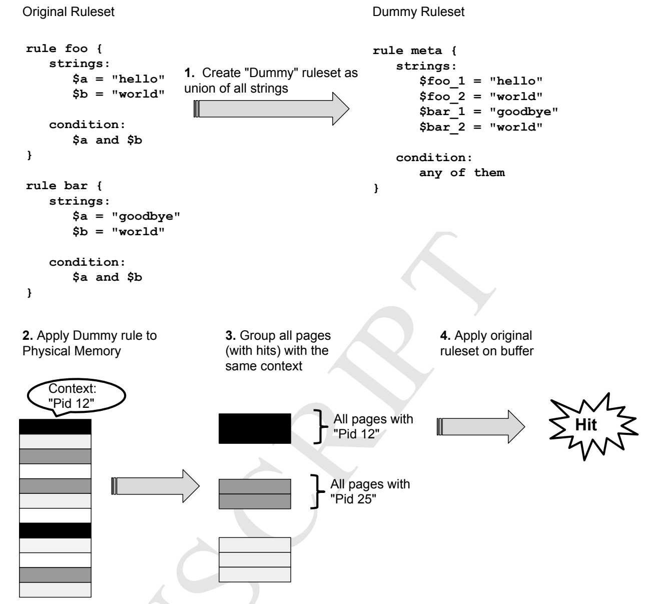
Our approach, illustrated in Figure 11, is to convert the original set of Yara signatures to a large, single, dummy signature by extracting all the strings in the original signature into the new single rule. The new rule contains a single condition "any of them" which will fire whenever any of the strings are found anywhere.

When any string is found in a physical page, we apply the algorithm in Section 4.1 to reveal a set of context strings about this physical address. These strings include all the unique contexts in which we wish to evaluate the original yara ruleset. For example, using the *ptov* plugin described earlier we can derive all the owning processes (in case of a shared page) and the mapped filename (in case of a mapped file). So a set of context strings may describe a particular page as "Pid 12", "Pid 22", "File:notepad.exe".

We then build a map between each context string and the set of hits found. For example, all the pages which contain the string "Pid 12" are collected together. Note that if a page is shared between multiple processes it will appear once for each of its contexts.

Finally we create a small buffer for each unique context string by copying the pages which produced hits into this buffer. Since these pages matched the dummy rule we know that at least one string of interest appears within them. Additionally, since all these pages contain the same context we know these are related (i.e. they are all part of the same process or mapped file). Since the Yara ruleset does not care where in the buffer the strings must match or their order, we can simply apply the original Yara rule over this small buffer and allow the yara library to evaluate the condition over them. The Yara signature would again match the same strings, but this time the full condition is evaluates, and if the condition fires, then the context is said to match the signature and we report the hits.

In a sense, the reduced buffer is simply a relevant subset of the original virtual address space, only containing pages of possible interest, concatenated in a random order.


Original Ruleset
Dummy Ruleset
1. Create "Dummy" ruleset as union of all strings
2. Apply Dummy rule to Physical Memory
3. Group all pages (with hits) with the same context
4. Apply original ruleset on buffer
Context: "Pid 12"
All pages with "Pid 12"
All pages with "Pid 25"
Hit

```
rule foo {
   strings:
      $a = "hello"
      $b = "world"

   condition:
      $a and $b
}

rule bar {
   strings:
      $a = "goodbye"
      $b = "world"

   condition:
      $a and $b
}
```

```
rule meta {
   strings:
      $foo_1 = "hello"
      $foo_2 = "world"
      $bar_1 = "goodbye"
      $bar_2 = "world"

   condition:
      any of them
}
```

Figure 11: Algorithm for applying Yara signatures over the physical address space. First the Yara ruleset is converted to a dummy ruleset which will match on any of the strings of interest. The dummy ruleset is then applied to the physical memory, and for each hit a set of context strings is derived. Finally all the hits with the same context string are concatenated together into a single pseudo-buffer and the original ruleset evaluated over them.

6. Evaluation

This paper argues that by considering the entire Virtual Address space, one can devise better signatures than simply considering each executable in turn. This is especially important when there is nothing especially suspicious about the executable itself, rather in the way it is being used by an attacker. This trend has been recently observed in the tactics used by elite threat actors, and has been dubbed by some as "living off the land" - or in other words actors which operate using credentials, systems, and tools which already exist on the compromised systems (Counter Threat Unit Research Team, 2015).

To illustrate this point we consider the common case of an attacker attempting to ex-filtrate large amount of data from the system. However, rather than installing their own Remote Access Trojan (RAT), the attacker chooses to simply use the existing Python installation on the target system. Python is a very popular language (Van Rossum and Drake, 2011), and many systems already have it installed. Python also comes with many built in modules and one of the most useful is called *SimpleHTTPServer* (Python Software Foundation, 2016). The attacker can use this module to instantly create a web server which will server all files from arbitrary directories. The attacker can then use any browser to transfer the files to another system.

In order to emulate this scenario we used a Windows 10 Virtual Machine with 1Gb RAM. The machine had Python 2.7.12 installed from the official distribution site. We opened a command prompt and changed directory to the user's *Documents and Settings* folder. Once there, we ran the command *python.exe -m SimpleHTTPServer 8000*.

We then started the Microsoft Edge browser and navigated to *http://127.0.0.1:8000/* to verify that the server was working. We were greeted with a directory index listing all the files in the users folder. We then used the Rekall forensic tool to acquire memory into an AFF4 memory image.

While this ex-filtration technique is very effective, it is challenging to write an appropriate Yara rule using the traditional memory analysis approach. The problem is that the Python program is not suspicious at all - it is a very common executable, and so we can not just trigger on its presence. Instead it is the combination of how it is being invoked and its runtime state which is suspicious.

```
rule python_httpserver {
  strings:
    $a = /python.+-m.+SimpleHTTPServer/ wide
    $b = "<title>Directory listing for %s"
    $c = "<title>Directory listing for /"

  condition:
    $a and $b and $c
}
```

Figure 12: A YARA rule used to detect python http servers. String $a indicates a malicious command line invocation while string $b matches the template code producing the directory listing. Either of these indicate that the python interpreter is running. The directory listing string is produced by the server and should be found in the process heap.

Invoking the SimpleHTTPServer module from the command line is particularly strange because the attacker is not running a normal python script. Although this invocation is unusual, it is not sufficient to just search for this string - this may produce too many false positives (e.g. the documentation of the SimpleHTTPServer might mention this as a valid way to run it or a web page might describe this method). Note that the command line string with which the process is started only exists in the process's environment block (which is a private memory mapping in a separate VAD region.), it is not part of the binary itself.

We also note that the SimpleHTTPServer serves a directory index HTML document when someone browses the web site. We can recognize the string "<title>Directory listing for /" as fairly unique for this page, making it an obvious choice for a signature. This string is generated for each page by the python interpreter and so we expect to find this string in the process heap region.

The string itself is generated from a template which is found in the code ("<title>Directory listing for %s" - the %s is a placeholder for the directory path), so seeing this template in process memory is a very strong indication that this process has the relevant python code loaded.

In isolation each of these strings is a very weak signal, but together the strings add up to a strong signal. If a process is started with the "-m SimpleHTTPServer" command line, imports the SimpleHTTPServer python module code and also has allocated in its heap memory the expanded string from the template we can be quite confident that it ran this code. Such a rule is illustrated in Figure 12.

Unfortunately, each string is expected to appear in different virtual memory regions. As noted in section 3, current memory analysis tools can not apply the same Yara signature on multiple different Virtual Address Ranges. This is primarily due to the practical need to partition the Virtual Address space into reasonably sized buffers for scanning, and the Yara library's inability to keep state between scanning operations.

To demonstrate this shortcoming we used Rekall's *yarascan* plugin. We chose to scan the Virtual Address space in all processes (there were 61 processes). Rekall was called with the command "rekall yarascan –yara_file=test.yara –scan_process_memory". When invoked with these arguments, Rekall applies the yara rule to every process's virtual address space. As noted previously, Rekall ends up scanning many DLLs multiple times because they are shared with multiple processes, therefore they appear in multiple Virtual Address Spaces. The complete scan was therefore very time consuming taking 4 minutes 58 seconds on this 1GB image. As expected the yara rule did not fire. We repeated the same analysis using Volatility's *yarascan* plugin, and since it worked in a similar way, it too did not return any hits.

To investigate why the rule did not fire we changed the rule's condition to "any of them" so that yarascan can report all matches, and then we reran the scan only on the python process in question. This time the rule fired many times for each string, demonstrating that all the strings were in fact present in memory (See Figure 13. Closer inspection of the hits reveals that every string hit was located in a different VAD range. When Volatility or Rekall assembled the buffer to be scanned, only data from the same memory range was passed, and therefore only one of the strings was presents at the time - not enough to fire to original ruleset.

6.1. Scanning physical memory

We implemented the algorithm described in Section 5.1 in a new Rekall plugin named *yarascan_physical*. We then applied this to the same physical memory image and saw that the python process matched the signature, yet no other process matched. The total scanning time was 56 seconds. Repeating the scan only took 16 seconds since the *SUBSECTION* lookup map remains cached between executions. Therefore it took 40 seconds to build the *SUBSECTION* map by walking all VAD trees from all processes.

In order to illustrate how the algorithm can discriminate between the correct match and possible false positives we again modified the rule to match on any of the strings (this makes it equivalent to the dummy rule described in Section 5.1). Sample output from the plugin is shown in Figure 14 (Some similar hits were removed for brevity).

While many processes did match one or more of the strings, only the rogue python process matched all of them at the same time. Each string was matched in a different VAD region. String $a matched in the process environment block, while string $b (string expansion template) matched in the process heap, where python loaded the relevant code file. String $c matched in a different heap region because it was served by the python process when the Edge browser requested a directory listing.

Unsurprisingly the Edge browser's heap also contains the generated directory listing, however the full signature does not match it because it does not contain the string template (i.e. it has no code to generate that string).

Finally we note that *csrss.exe* and *cmd.exe* both matched the command line launching the SimpleHTTPServer, unsurprisingly. This is expected since they both contain the console history buffers (Stevens and Casey, 2010).

6.2. Limitations

Although the technique described above works well for typical yara signatures, it has some limitations. Since we scan the

```
[1] test.raw 15:47:12> yarascan yara_file=''/tmp/yara.rule'', scan_process_memory=1

Owner                              Rule       Offset                   hexdump                           Context
---------------------------------- ---------- ------------ --------------------------------- -------------------------
0xe000ed89f200 python.exe     3712 python_htt 0x11c1000    70 00 79 00 74 00 68 00  p.y.t.h.  vad_0x11c0000+0x1000
                                   pserver                 6f 00 6e 00 2e 00 65 00  o.n...e.
                                                           78 00 65 00 00 00 70 00  x.e...p.
                                                           79 00 74 00 68 00 6f 00  y.t.h.o.
                                                           6e 00 20 00 20 00 2d 00  n.....-.
                                                           6d 00 20 00 53 00 69 00  m...S.i.
                                                           6d 00 70 00 6c 00 65 00  m.p.l.e.
                                                           48 00 54 00 54 00 50 00  H.T.T.P.
0xe000ed89f200 python.exe     3712 python_htt 0x2e4f000    3c 74 69 74 6c 65 3e 44  <title>D  vad_0x2d60000+0xef000
                                   pserver                 69 72 65 63 74 6f 72 79  irectory
                                                           20 6c 69 73 74 69 6e 67  .listing
                                                           20 66 6f 72 20 25 73 3c  .for.%s<
                                                           2f 74 69 74 6c 65 3e 0a  /title>.
                                                           00 00 00 00 00 00 00 00  ........
                                                           00 02 00 00 00 00 00 00  ........
                                                           00 c0 47 ed 5f 00 00 00  ..G._...
0xe000ed89f200 python.exe     3712 python_htt 0x2ce2000    3c 74 69 74 6c 65 3e 44  <title>D  vad_0x2c10000+0xd2000
                                   pserver                 69 72 65 63 74 6f 72 79  irectory
                                                           20 6c 69 73 74 69 6e 67  .listing
                                                           20 66 6f 72 20 2f 3c 2f  .for./</
                                                           74 69 74 6c 65 3e 0a 00  title>..
                                                           00 00 c8 6d 17 03 00 00  ...m....
                                                           00 00 c8 60 17 03 00 00  ...`....
                                                           00 00 88 62 17 03 00 00  ...b....
0xe000ed99d780 MicrosoftEdgeC 1816 python_htt 0x608259b000 3c 74 69 74 6c 65 3e 44  <title>D  vad_0x6082580000+0x1b000
                                     pserver               69 72 65 63 74 6f 72 79  irectory
                                                           20 6c 69 73 74 69 6e 67  .listing
                                                           20 66 6f 72 20 2f 3c 2f  .for./</
                                                           74 69 74 6c 65 3e 0a 3c  title>.<
                                                           62 6f 64 79 3e 0a 3c 68  body>.<h
                                                           32 3e 44 69 72 65 63 74  2>Direct
                                                           6f 72 79 20 6c 69 73 74  ory.list
0xe000ed293080 csrss.exe       420 python_htt 0x6ee9e2e000 70 00 79 00 74 00 68 00  p.y.t.h.  vad_0x6ee9e20000+0xe000
                                   pserver                 6f 00 6e 00 20 00 20 00  o.n.....
                                                           2d 00 6d 00 20 00 53 00  -.m...S.
                                                           69 00 6d 00 70 00 6c 00  i.m.p.l.
                                                           65 00 48 00 54 00 54 00  e.H.T.T.
                                                           50 00 53 00 65 00 72 00  P.S.e.r.
                                                           76 00 65 00 72 00 20 00  v.e.r...
                                                           38 00 30 00 30 00 30 00  8.0.0.0.
```

Figure 13: Output from Rekall's traditional *yarascan* plugin, after the yara rule is modified to hit on any of the strings. While all the strings appear in the same python process (pid 3712), each of them occurs in a different VAD region. Note that Rekall notation names each vad region by its start address so for example "vad_0x11c0000+0x1000" means the hit occurs 0x1000 bytes into the VAD regions which started at virtual address 0x11c0000. Because each hit occurs in a different part of the process, the traditional plugin does not scan all VAD regions at the same time, hence failing to trigger the rule which requires all strings to be present.

physical address space we rely on the entirety of the string to exist within a single memory page (typically 4kb). If the signature is split across a page boundary, it is unlikely that the next physical page will also be contiguous in the virtual address space. Therefore, the signature will fail to match.

Another potential problem which may be observed when scanning memory is that some pages in the virtual address space are paged into the page file - or indeed consist of file mappings without physical memory backing. Since our technique only scans the physical address space, those pages which are not present will not yield a match which might lead to the signature failing to match.

The above limitations are inherent to scanning in memory as opposed to a static, unchanging file. The Yara rule engine allows signature authors to build in some redundancy into the signature. Typically this is used to accommodate some variability in samples (e.g. detecting slightly modified versions of the known malware). For this reason failing to match a single string across page boundaries is not necessarily a false negative, providing enough alternate strings are present in the memory sample. We can therefore devise some general guidelines as to writing more effective signatures for memory:

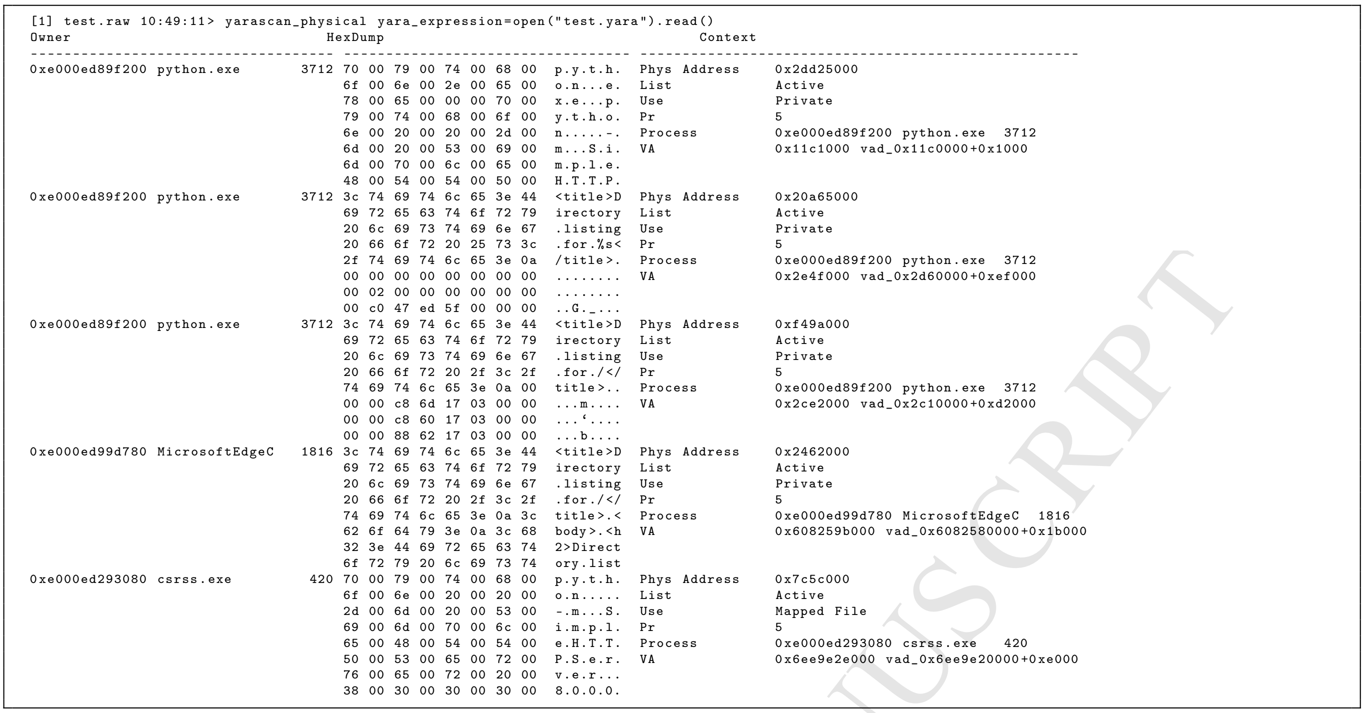
1. Signature strings should be short compared with the 4kb page size.
2. Authors should devise a set of signatures and match some percentage of them (e.g. using a condition such as "5 of them").
3. Authors should avoid more complex Yara constructs (such as PE header checks or specific pointer following directives), since these are not easily translated to the memory domain.

7. Conclusions

Signature scanning is a quick and easy technique for identifying malware. With careful signature creation rules it is possible to balance the false positive rate with detection resilient to variations in malware families. Yara is a powerful tool allowing searching of signatures on large bodies of data quickly.

This paper examined the application of Yara signatures to the domain of Memory Forensics. We discover that scanning memory is a different problem than scanning a file on disk and that some signatures written for files do not work well when searching memory. Conversely, we demonstrated that signatures specifically designed to work in memory can be stronger than signatures initially developed on files. By considering the entire address space layout, it is possible to introduce an element of behavior analysis to the signature as it can be made to match the process heap as well as just the binary. This helps to distinguish between benign executables used in a benign way and those same executables used in malicious ways.

We have developed a very fast way to determine what process each physical page belongs to, and where in that process it is mapped, using the Windows PFN database. Armed with



```
[1] test.raw 10:49:11> yarascan_physical yara_expression=open("test.yara").read()
Owner                              HexDump                                      Context
------------------------------------- ---------------------------------  ----------------------------------------------------
0xe000ed89f200 python.exe        3712 70 00 79 00 74 00 68 00  p.y.t.h.  Phys Address    0x2dd25000
                                      6f 00 6e 00 2e 00 65 00  o.n...e.  List            Active
                                      78 00 65 00 00 00 70 00  x.e...p.  Use             Private
                                      79 00 74 00 68 00 6f 00  y.t.h.o.  Pr              5
                                      6e 00 20 00 20 00 2d 00  n.....-.  Process         0xe000ed89f200 python.exe  3712
                                      6d 00 20 00 53 00 69 00  m...S.i.  VA              0x11c1000 vad_0x11c0000+0x1000
                                      6d 00 70 00 6c 00 65 00  m.p.l.e.
                                      48 00 54 00 54 00 50 00  H.T.T.P.
0xe000ed89f200 python.exe        3712 3c 74 69 74 6c 65 3e 44  <title>D  Phys Address    0x20a65000
                                      69 72 65 63 74 6f 72 79  irectory  List            Active
                                      20 6c 69 73 74 69 6e 67  .listing  Use             Private
                                      20 66 6f 72 20 25 73 3c  .for.%s<  Pr              5
                                      2f 74 69 74 6c 65 3e 0a  /title>.  Process         0xe000ed89f200 python.exe  3712
                                      00 00 00 00 00 00 00 00  ........  VA              0x2e4f000 vad_0x2d60000+0xef000
                                      00 02 00 00 00 00 00 00  ........
                                      00 c0 47 ed 5f 00 00 00  ..G._...
0xe000ed89f200 python.exe        3712 3c 74 69 74 6c 65 3e 44  <title>D  Phys Address    0xf49a000
                                      69 72 65 63 74 6f 72 79  irectory  List            Active
                                      20 6c 69 73 74 69 6e 67  .listing  Use             Private
                                      20 66 6f 72 20 2f 3c 2f  .for./</  Pr              5
                                      74 69 74 6c 65 3e 0a 00  title>..  Process         0xe000ed89f200 python.exe  3712
                                      00 00 c8 6d 17 03 00 00  ...m....  VA              0x2ce2000 vad_0x2c10000+0xd2000
                                      00 00 c8 60 17 03 00 00  ...‘....
                                      00 00 88 62 17 03 00 00  ...b....
0xe000ed99d780 MicrosoftEdgeC    1816 3c 74 69 74 6c 65 3e 44  <title>D  Phys Address    0x2462000
                                      69 72 65 63 74 6f 72 79  irectory  List            Active
                                      20 6c 69 73 74 69 6e 67  .listing  Use             Private
                                      20 66 6f 72 20 2f 3c 2f  .for./</  Pr              5
                                      74 69 74 6c 65 3e 0a 3c  title>.<  Process         0xe000ed99d780 MicrosoftEdgeC  1816
                                      62 6f 64 79 3e 0a 3c 68  body>.<h  VA              0x608259b000 vad_0x6082580000+0x1b000
                                      32 3e 44 69 72 65 63 74  2>Direct
                                      6f 72 79 20 6c 69 73 74  ory.list
0xe000ed293080 csrss.exe          420 70 00 79 00 74 00 68 00  p.y.t.h.  Phys Address    0x7c5c000
                                      6f 00 6e 00 20 00 20 00  o.n.....  List            Active
                                      2d 00 6d 00 20 00 53 00  -.m...S.  Use             Mapped File
                                      69 00 6d 00 70 00 6c 00  i.m.p.l.  Pr              5
                                      65 00 48 00 54 00 54 00  e.H.T.T.  Process         0xe000ed293080 csrss.exe   420
                                      50 00 53 00 65 00 72 00  P.S.e.r.  VA              0x6ee9e2e000 vad_0x6ee9e20000+0xe000
                                      76 00 65 00 72 00 20 00  v.e.r...
                                      38 00 30 00 30 00 30 00  8.0.0.0.
```

Figure 14: An extract from the output of the yarascan plugin ran over the physical address space. The rule was modified to match on any of the strings. The real python server matches all strings, but they are found in different VAD regions. Other processes match one or more of the strings but not all of them.

the powerful technique we are able to collect sufficient context about potential signature hits to perform context aware signature matching on the physical address space. This not only improves matching speed, but also improves matching accuracy since we do not need to reconstruct the virtual address space of every process in the scanning phase.

Alvarez, V. M., Aug 2016. YARA. http://virustotal.github.io/yara/, [Online; accessed 29-Aug-2016].

Breitinger, F., Baier, H., 2012. Similarity preserving hashing: Eligible properties and a new algorithm mrsh-v2. In: International Conference on Digital Forensics and Cyber Crime. Springer, pp. 167–182.

Carrier, B., 2005. File system forensic analysis. Vol. 3. Addison-Wesley Reading.

Carrier, B., Aug 2016. ifind - Find the meta-data structure that has allocated a given disk unit or file name. http://www.sleuthkit.org/sleuthkit/man/ifind.html, [Online; accessed 29-Aug-2016].

Cohen, M., 2015. Forensic analysis of windows user space applications through heap allocations. In: 2015 IEEE Symposium on Computers and Communication (ISCC). IEEE, pp. 237–244.

Cohen, M. I., 2007. Advanced carving techniques. Digital Investigation 4 (3), 119–128.

Counter Threat Unit Research Team, May 2015. Living off the land. https://www.secureworks.com/blog/living-off-the-land.

Dolan-Gavitt, B., 2007. The vad tree: A process-eye view of physical memory. digital investigation 4, 62–64.

Flaglien, A., Franke, K., Arnes, A., 2011. Identifying malware using cross-evidence correlation. In: IFIP International Conference on Digital Forensics. Springer, pp. 169–182.

Garfinkel, S. L., 2013. Digital media triage with bulk data analysis and bulk_extractor. Computers & Security 32, 56 – 72.
URL http://www.sciencedirect.com/science/article/pii/S0167404812001472

Griffin, K., Schneider, S., Hu, X., Chiueh, T.-C., 2009. Automatic generation of string signatures for malware detection. In: International Workshop on Recent Advances in Intrusion Detection. Springer, pp. 101–120.

Gruhn, M., 2015. Windows nt pagefile. sys virtual memory analysis. In: IT Security Incident Management & IT Forensics (IMF), 2015 Ninth International Conference on. IEEE, pp. 3–18.

Hargreaves, C., Chivers, H., 2008. Recovery of encryption keys from memory using a linear scan. In: Availability, Reliability and Security, 2008. ARES 08. Third International Conference on. IEEE, pp. 1369–1376.

Hoffman, N., January 2015. The Mozart RAM Scraper. http://securitykitten.github.io/the-mozart-ram-scraper/, [Online; accessed 29-Aug-2016].

Intel, 2015. Intel 64 and IA-32 Architectures Developer's Manual: Vol. 3A. Vol. 3A. Intel Corporation, Ch. Chapter 3.

Oktavianto, D., Muhardianto, I., 2013. Cuckoo Malware Analysis. Packt Publishing Ltd.

Python Software Foundation, Aug 2016. SimpleHTTPServer - Simple HTTP request handler. https://docs.python.org/2/library/simplehttpserver.html, [Online; accessed 29-Aug-2016].

Russinovich, M., Solomon, D., Ionescu, A., 2012. Windows Internals. No. pt. 1 in Developer Reference. Pearson Education.
URL https://books.google.ch/books?id=w65CAwAAQBAJ

Sathyanarayan, V. S., Kohli, P., Bruhadeshwar, B., 2008. Signature generation and detection of malware families. In: Australasian Conference on Information Security and Privacy. Springer, pp. 336–349.

Schuster, A., 2006. Searching for processes and threads in microsoft windows memory dumps. digital investigation 3, 10–16.

Stevens, R. M., Casey, E., 2010. Extracting windows command line details from physical memory. digital investigation 7, S57–S63.

Sylve, J. T., Marziale, V., Richard, G. G., 2016. Pool tag quick scanning for windows memory analysis. Digital Investigation 16, S25–S32.

The Rekall Team, 2016. Rekall memory forensic framework. http://www.rekall-forensic.com/, [Online; accessed 09-Aug-2016].

The Volatility Foundation, 2015. Volatility. https://github.com/volatilityfoundation/volatility, [Online; accessed 09-Feb-2015].

Van Rossum, G., Drake, F. L., 2011. The python language reference manual. Network Theory Ltd.

Various, 2016. Repository of yara rules. https://github.com/Yara-Rules/rules, [Online; accessed 29-Aug-2016].