Digital Investigation 12 (2015) S38–S49

Contents lists available at ScienceDirect

Digital Investigation

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

DFRWS 2015 Europe

Characterization of the windows kernel version variability for accurate memory analysis

Michael I. Cohen

Google Inc., Brandschenkestrasse 110, Zurich, Switzerland

Keywords:
Memory analysis
Incident response
Binary classification
Memory forensics
Live forensics

A B S T R A C T

Memory analysis is an established technique for malware analysis and is increasingly used for incident response. However, in most incident response situations, the responder often has no control over the precise version of the operating system that must be responded to. It is therefore critical to ensure that memory analysis tools are able to work with a wide range of OS kernel versions, as found in the wild. This paper characterizes the properties of different Windows kernel versions and their relevance to memory analysis. By collecting a large number of kernel binaries we characterize how struct offsets change with versions. We find that although struct layout is mostly stable across major and minor kernel versions, kernel global offsets vary greatly with version. We develop a "profile indexing" technique to rapidly detect the exact kernel version present in a memory image. We can therefore directly use known kernel global offsets and do not need to guess those by scanning techniques. We demonstrate that struct offsets can be rapidly deduced from analysis of kernel pool allocations, as well as by automatic disassembly of binary functions. As an example of an undocumented kernel driver, we use the *win32k.sys* GUI subsystem driver and develop a robust technique for combining both profile constants and reversed struct offsets into accurate profiles, detected using a profile index.

Introduction

Memory analysis has become a powerful technique for the detection and identification of malware, and for digital forensic investigations (Ligh et al., 2010, 2014).

Fundamentally, memory analysis is concerned with interpreting the seemingly unstructured raw memory data which can be collected from a live system into meaningful and actionable information. At first sight, the memory content of a live system might appear to be composed of nothing more than random bytes. However, those bytes are arranged in a predetermined order by the running software to represent a meaningful data structure. For example consider the C struct:

```
typedef struct _EPROCESS {
   unsigned long long CreateTime;
   char[16] ImageFileName;
} EPROCESS;
```

The compiler will decide how to overlay the struct fields in memory depending on their size, alignment requirements and other consideration. So for example, the *CreateTime* field might get 8 bytes, causing the *ImageFileName* field to begin 8 bytes after the start of the *_EPROCESS* struct.

A memory analysis framework must have the same layout information in order to know where each field should be found in relation to the start of the struct. Early memory analysis systems hard coded this layout information which was derived by other means (e.g. reverse

E-mail address: scudette@google.com.

http://dx.doi.org/10.1016/j.diin.2015.01.009

engineering or simply counting the fields in the struct header file (Schuster, 2007)).

This approach is not scalable though, since the struct definition change routinely between versions of the operating system. For example, in the above simplified struct of an *_EPROCESS*, if additional fields are inserted, the layout of the field members will change to make room for the new elements. So for example, if another 4 byte field is added before the *CreateTime* field, all other offsets will have to increase by 4 bytes to accommodate the new field. This will cause all the old layout information to be incorrect and our interpretation of the struct in memory to be wrong.

Modern memory analysis frameworks address the variations across different operating system versions by use of a version specific memory layout template mechanism. For example in Volatility (The Volatility Foundation, 2014) or Rekall (The Rekall Team, 2014a, b) this information is called a *profile*.

The Volatility memory analysis framework (The Volatility Foundation, 2014) is shipped with a number of Windows profiles embedded into the program. The user chooses the correct profile to use depending on their image. For example, if analyzing a Windows 7 image, the profile might be specified as *Win7SP1x64*. In Volatility, the profile name conveys major version information (i.e. Windows 7), minor version information (i.e. Service Pack 1) and architecture (i.e. ×64). Volatility uses this information to select a profile from the set of built-in profiles.

Deriving profile information

The problem still remains how to derive this struct layout information automatically. The Windows kernel contains many struct definitions, and these change for each version, so a brute force solution is not scalable (Okolica and Peterson, 2010).

Memory analysis frameworks are not the only case where information about memory layout is required. Specifically, when debugging an application, the debugger needs to know how to interpret the memory of the debugged program in order to correctly display it to the user. Since the compiler is the one originally deciding on the memory layout, it makes sense that the compiler generates debugging information about memory layout for the debugger to use.

On Windows systems, the most common compiler used is the Microsoft Visual Studio compiler (MSVCC). This compiler shares debugging information via a *PDB* file (Schreiber, 2001), generated during the build process for the executable. The PDB file format is unfortunately undocumented, but has been reverse engineered sufficiently to be able to extract accurate debugging information, such as struct memory layout, reliably (Schreiber, 2001; Dolan-Gavitt, 2007a).

The PDB file for an executable is normally not shipped together with the executable. The executable contains a unique GUID referring to the PDB file that describes this executable. When the debugger wishes to debug a particular executable, it can then request the correct PDB file from a *symbol server*. This design allows production binaries to be debugged, without needing to ship bulky debug information with final release binaries.

The PDB file contains a number of useful pieces of information for a memory analysis framework:

- Struct members and memory layout. This contains information about memory offsets for struct members, and their types. This is useful in order to interpret the contents of memory.
- Global constants. The Windows kernel contains many important constants, which are required for analysis. For example, the *PsActiveProcessHead* is a constant pointer to the beginning of the process linked list, and is required in order to list processes by walking that list.
- Function addresses. The location of functions in memory is also provided in the PDB file – even if these functions are not exported. This is important in order to resolve addresses back to functions (e.g. in viewing the Interrupt Descriptor Table – IDT).
- Enumeration. In C an enumeration is a compact way to represent one of a set of choices using an integer. The mapping between the integer value and a human meaningful string is stored in the PDB file, and it is useful for interpreting meaning from memory.

Characterizing kernel version variability

As described previously, the Volatility tool only contains a handful of profiles generated for different major releases of the Windows kernel. However, each time the kernel is rebuilt by Microsoft (e.g. for a security hot fix), the code could be changed, and the profile could be different. The assumption made by the Volatility tool is that these changes are not significant and therefore, a profile generated from a single version of a major release will work on all versions from that release.

We wanted to validate this assumption. We collected the Windows kernel binary (*ntkrnlmp.exe, ntkrpamp.exe, ntoskrnl.exe*) from several thousand machines in the wild using the GRR tool (Cohen et al., 2011). Each of these binaries has a unique GUID, and we were therefore able to download the corresponding PDB file from the public Microsoft symbol server. We then used Rekall's *mspdb* parser to extract debugging information from each PDB file.

This resulted in 168 different binaries of the Windows kernel for various versions (e.g. Windows XP, Windows Vista, Windows 7 and Windows 8) and architectures (e.g. I386 and AMD64). Clearly, there are many more versions of the Windows kernel in the wild than exist in the Volatility tool. It is also very likely that we have not collected all the versions that were ever released by Microsoft, so our sample size, although large, is not exhaustive.

Fig. 1 shows sampled offsets of four critical struct members for memory analysis:

- The *_EPROCESS.VadRoot* is the location of the Vad within the process. This is used to enumerate process allocations (Dolan-Gavitt, 2007b).
- The *_KPROCESS.DirectoryTableBase* is the location of the Directory Table Base (i.e. the value loaded into the CR3