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

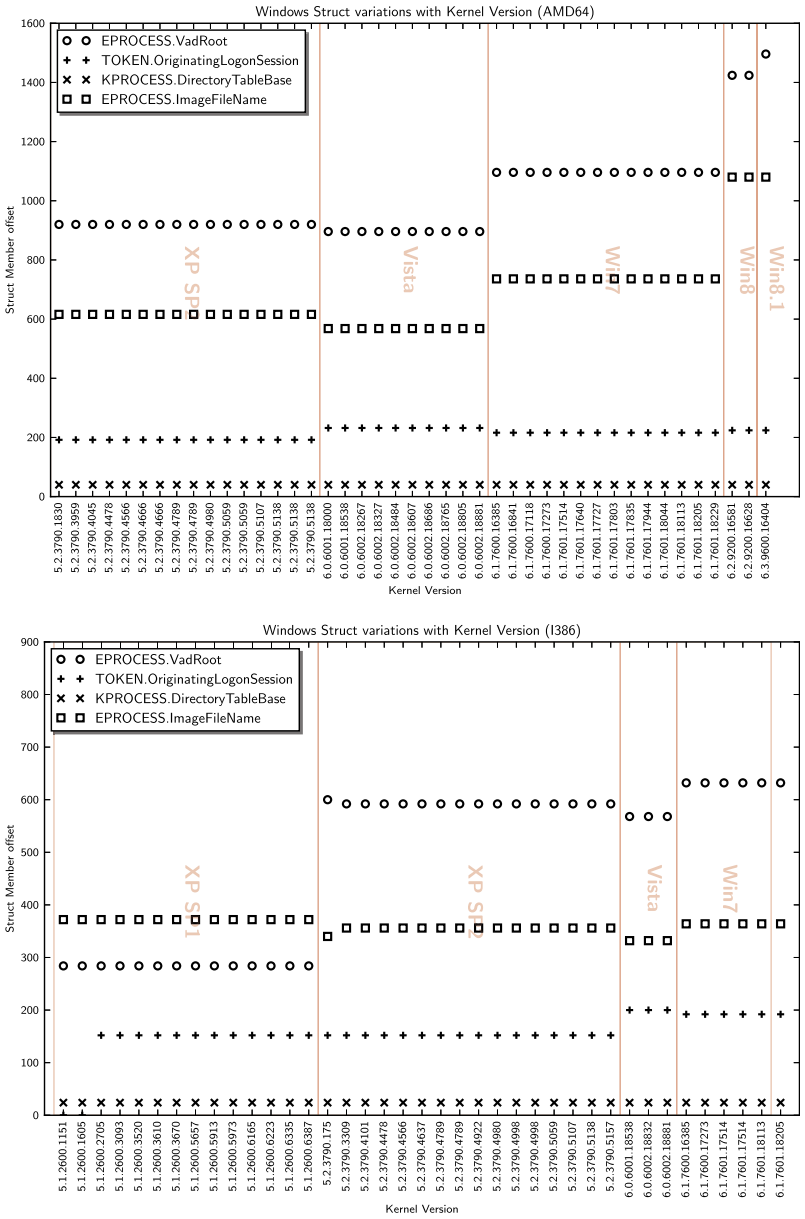

Fig. 1. Offsets for a few critical struct members across various versions of the Windows kernel. These offsets were derived by analyzing public debug information from the Microsoft debug server for the binaries in our collection.

register) which is critical in constructing the Virtual Address Space abstraction.

- The *_EPROCESS.ImageFileName* is the file name of the running binary. For example, this field might contain "csrss.exe".

Microsoft Windows kernel versions contain four parts: The major and minor versions, the revision and the build number. The build number increases for each build (e.g. security hotfix).

As can be seen in the figure, struct offsets do tend to remain stable across Windows versions. In most cases, with a single notable exception – version 5.2.3970.175 (GUID 466B4165EAA84AF88D29D617E86A95982), the struct offsets remain the same for all major Windows releases. Therefore, chances are good that the Volatility profile for a given Windows version would actually work most of the time for determining struct layout.

Kernel global constants variability

It is generally not sufficient to determine only the struct memory layout for memory analysis. For example, consider listing the running processes. One technique is to follow the doubly linked list of *EPROCESS.ActiveProcessLinks* in each process struct (Okolica and Peterson, 2010). This

technique needs to find the start of the list which begins at the global kernel constant *PsActiveProcessHead*. The location for this global constant in memory is determined statically by the compiler at compile time, and it is usually stored in one of the data sections in the PE file itself.

Since this information is also required by the debugger, the PDB file also contains information about global constants and functions (even if these are not actually exported via the Export Address Table). Rekall's *mspdb* plugin also extract this information into the profile.

Fig. 2 illustrates the memory addresses of some important kernel constants for the kernels in our collection:

- *NtBuildLab* is the location of the NT version string (e.g. "7600.win7_rtm.090713-1255"). This is used to identify the running kernel.
- *PsActiveProcessHead* is the head of the active process list. This is required in order to list the running processes.
- *NtCreateToken* is an example of a kernel function. This will normally exist in the *.text* section of the PE file.
- *str:FILE_VERSION* is literally the string "FILE_VERSION". Usually the compiler will place all literal strings into their own string table in the *.rdata* section of the PE file. The compiler will then emit debugging symbols for the location of each string – indicating that they are literal

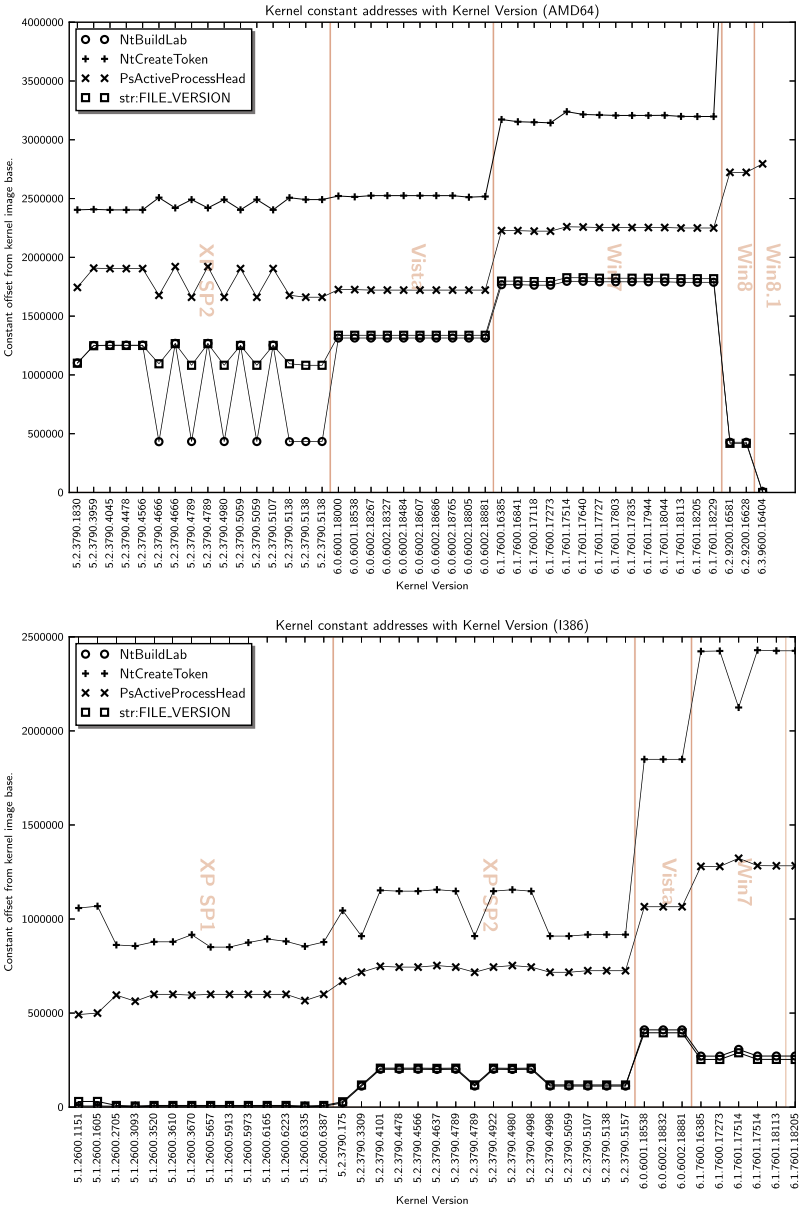

Fig. 2. Offsets for a few global kernel constants across various versions of the Windows kernel. These offsets were derived by analyzing public debug information from the Microsoft debug server for the binaries in our collection. Offsets are provided relative to the kernel image base address.

strings. The importance of this symbol will be discussed in the following sections.

As can be seen, the offsets of global kernel constants change dramatically between each build – even for the same version. This makes sense, since the compiler arranges global constants in their own PE section, so if any global constant is added or removed in the entire kernel, this affects the ordering of all other constants placed after it.

It is therefore clear that it is unreliable to directly obtain the addresses of kernel globals by simply relying on the version alone. The Volatility tool resorts to a number of techniques to obtain these globals:

- Many globals are obtained from the *KdDebuggerDataBlock* – another global kernel struct which contains pointers to many other globals. This structure is usually scanned for.
- Scanning for kernel objects which refer to global constants (e.g. via pool tag scanning or other signatures).
- Examining the export tables of various PE binaries for exported functions.
- Dynamically disassembling code to detect calls to non exported functions.

These techniques are complex and error prone. They are also susceptible to anti-forensics as signature scanners can trivially be fooled by spurious signatures (Williams and Torres, 2014). Scanning for signatures over very large memory images is also slow and inefficient.

The Rekall memory forensic framework (The Rekall Team, 2014a, b), a fork of the Volatility framework, takes a different approach. Instead of guessing the location of various kernel constants, the framework relies on a public profile repository which contains every known profile from every known build of the Windows kernel. This greatly simplifies memory analysis algorithms because the address of global kernel variables and functions is directly known from public debugging information provided by Microsoft. There is no need to scan or guess at all. Locating these globals is very efficient since there is no need to scan for signatures, making the framework fast and reducing the ability of attackers to subvert analysis.

Identifying binary versions

The Rekall profile repository contains, at the time of writing, 309 profiles for various Windows kernel versions (and this number is constantly increasing). Typically, users will simply report the GUID of the Windows kernel found in their image, but will not provide the actual kernel binary.

Previously, Rekall employed a scanning technique to locate the GUID of the NT kernel running within the image. Once the GUID is known, the correct profile can be fetched from the repository and analysis can begin. However, this technique is still susceptible to manipulation (It is easy for attackers to simply wipe or alter the GUID from memory). Sometimes the GUID is paged out of memory and in this case it is impossible to guess it. What we really need is a reliable way to identify the kernel version without relying on a single signature.

The problem of identifying kernel binaries in a memory image has been examined previously in the Linux memory analysis context (Roussev et al., 2014). In that paper, the authors used similarity hashing to match the kernel in a memory image with a corpus of known binaries.

In our case, we do not always have the actual binaries but have debugging symbols from these binaries. We therefore need a way for deducing enough information about the kernel binary itself (which we may not have) from the debug symbols. Consider the following information present in the PDB file:

- String Literals. As shown in the example above, the compiler generates string literals in the PE binary itself. These are then located using global debugging symbols. For example, in Fig. 2 we know the exact offsets in memory where we expect find the string “FILE_VERSION”.
- Function preamble. The PDB file also contains the locations of many functions. We note that each function is generally preceded by 5 NOP instructions in order to make room for hot patching (Chen, 2011). Thus, we can deduce that for each function in the PDB, the previous byte contains the value 0×90 (NOP instruction).

The problem, therefore, boils down to identifying which of a finite set of kernel profiles is the one present in the memory image, based on known data that must exist at known offsets:

1. Begin by selecting a number of function names, or literal string names. We term these *Comparison Points* since we only compare the binaries at these known offsets.
2. Examine all available profiles, and record the offset of these symbols as well as the expected data to appear at this offset (either a NOP instruction or the literal string itself).
3. Build a decision tree around the known comparison points to minimize the number of string comparisons required for narrowing down the match. Note that at this stage it is possible to determine if there are sufficient number of comparison points to distinguish all profile selections. If profile selection is ambiguous, further comparison points are added and the process starts again.
4. Scan the memory image for the longest strings using the Aho-Corasick string matching algorithm (Aho and Corasick, 1975).
5. For each match, seek around the match to apply the decision tree calculated earlier. Within a few string comparisons, the correct profile is identified.
6. Load the profile from the profile repository and initialize the analysis.

In practice it was found that fewer than a dozen comparison points are required to characterize all the profiles in the Rekall profile repository, leading to extremely quick matching times. Also, binary identification is robust to manipulation since the choice of comparison points is rather arbitrary and can be changed easily.

Windows kernel binary identification

Section 3 described an efficient algorithm for identifying a binary match from a set of known binaries. However, in the memory analysis context, this comparison must be made in the Virtual address space. Modern CPUs operate in protected mode, and the exact memory accessible to the kernel does not necessarily need to be contiguous in the physical memory image.

Therefore, before we are able to apply the index classification algorithm, we must build a virtual address space, requiring us to identify the value of CR3, or the kernel's Directory Table Base (DTB).

The DTB can be captured during the acquisition process and stored in the image, but typically it must be scanned for. The Volatility memory forensic framework scans for the Idle process's *EPROCESS* struct. It first searches for the literal string "Idle", this should exist as the *EPROCESS.ImageFileName* member. Knowing the difference between the offsets of *EPROCESS.ImageFileName* and *EPROCESS.Pcb.DirectoryTableBase*, the framework reads the DTB and therefore locates the page tables.

The problem with this approach is that it requires knowing the exact offsets of two *EPROCESS* struct members. Fig. 1 shows how these relative offsets vary between Windows versions, so to know the offset we need to know the exact Windows version we are examining – but we can not identify the profile without applying the profile index, which requires a valid kernel address space – i.e. knowing the DTB first!

We solve this Catch-22 by noting that the total number of combinations of the *EPROCESS* member offsets is limited (4 combinations for 64 bit architectures and 6 combinations for 32 bit architectures). Therefore, it is possible to brute force all combinations in search of a valid DTB.

So in summary the complete Kernel Binary Autodetection algorithm, as implemented in Rekall, is:

- Scan the image for common Windows executable names (e.g. "csrss.exe", "cmd.exe" etc). This scan uses the Aho-Corasick algorithm to search for all strings at once.
- For each hit, brute force the DTB going through the 10 possible offsets. The DTB is validated using the *KUSER_SHARED_DATA.NtMajorVersion* and *KUSER_SHARED_DATA.NtMinorVersion* members. Since this struct must be found at a fixed location in memory and always have the same layout it is safe to hardcode it (Skape, 2005). Therefore, we can validate the DTB and kernel address space without knowing anything about the profile itself or the kernel version.
- Once a DTB is identified, we construct a virtual address space and scan for the kernel image in memory using the algorithm previously described.

Undocumented kernel structures

Section 2 examined the variability of documented kernel structures across different kernel versions. The question we try to answer now is, what is the variability of undocumented kernel structures of significance to the memory analyst?

One of the most interesting kernel drivers is the Windows 32 user mode GUI subsystem (Mandt, 2011; Yuan, 2001), implemented as "win32k.sys". The data structures used in this subsystem are required to detect many common hooks placed by malware (e.g. *SetWindowsHookEx()* style keyloggers (Sikorski and Honig, 2012)).

The Rekall profile repository currently contains profiles for 169 unique versions of this driver. However, only 33 versions include information about critical structures (e.g. *tagDESKTOP* and *tagWINDOWSTATION*). The remaining profiles only contain information about global constants and functions, but no structure information.

Our goal is to understand how various important structures evolved through the released versions. Since many of these versions are undocumented and do not have debugging information, previous research has manually reverse engineered several samples from different versions. However, we are unsure if there is internal variability within Windows versions and releases. Guided by our previous experience with the Windows Kernel versions, we hypothesize that the *win32k.sys* struct layout would not vary much between minor release versions.

Given our large corpus of binaries we can directly examine this hypothesis and evaluate the best approach for determining struct layout when analyzing the Win32k GUI subsystem.

Data driven reverse engineering

The literature contains a number of published systems for automatically detecting kernel objects from memory images (Sun et al., 2012). For example, the SigGraph system (Lin et al., 2011), is capable of building scanners for Linux kernel structures by analyzing their internal pointer graphs.

The SigGraph system specifically does not utilize incidental knowledge about the system to assist in the reversing task. However on Windows systems, there are some helpful observation one can make to facilitate type analysis from memory dumps.

In the Windows kernel all allocations come from one of the kernel pools (e.g. Paged, Non-Paged or Session Pool). Allocations smaller than a page are preceded by a *POOL_HEADER* object (Schuster, 2006, 2008).

The pool header contains a known tag as well as indications of the previous and next pool allocation (within the page). Thus, small pool allocations form a doubly linked list. Due to this property it is possible to validate the pool header and locate it in memory. A typical Windows kernel allocation is illustrated in Fig. 3.

If we were to ask, "What kernel object exists at a given virtual offset?", we can simply scan backwards for a suitable *POOL_HEADER* structure and deduce the type of object from the pool tag. We can further scan forward from this location for other heuristics, such as pointers to certain other pool allocations, or doubly linked lists. We wrote a Rekall plugin called *analyze_structs* to perform this analysis on arbitrary memory locations.

For example, Fig. 4 shows the analysis of the global symbol *grpWinStaList* which is the global offset of the head of the *tagWINDOWSTATION* list. We can see that at offset

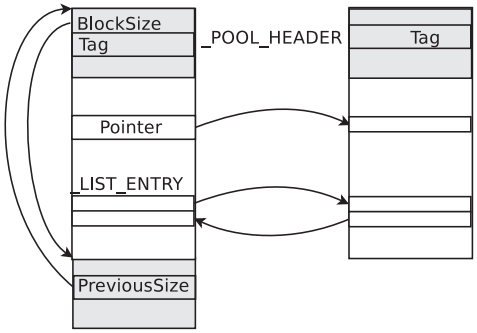

Fig. 3. An example of a typical Windows Kernel pool allocation. The *POOL_HEADER* indicates the type of the allocation. This header is also part of a doubly linked list with the next/previous allocation – a relation which may be used to validate it. By observing the type of allocations the struct members are pointing to it is possible to deduce the pointers and their target type.

0×10 there is a pointer to the *tagDESKTOP* object, at offset 0×18 there is a pointer to the global gTermIO object etc.

With Windows 7 we can find the complete struct information in the PDB file. This is also shown in Fig. 4. We can see that the detected pointers correspond with the *rpdeskList, pTerm, spklList, pGlobalAtomTable* and *psidUser* members.

An obvious limitation of this technique is that if a pointer in the struct is set to NULL, we are unable to say anything about it. Hence to reveal as many fields as possible we need to examine as many instances of the same object type as we can find (e.g. via pool scanning techniques).

Code based reverse engineering

The previous section demonstrates how we can deduce some struct layouts by observation of allocations we can find from the kernel pools. However, these observations are not sufficient to deduce all types of members. Specifically, only pointers are reliably deduced by this method. Additionally, we must observe allocated memory in a memory dump from a running system. Often we only have the executable binary (e.g. from disk) but not the full memory image.

In these cases, we need to resort to the more traditional reverse engineering approach. Previously, researchers have reverse engineered specific exemplars of the *win32k.sys* binary which is representative of a specific Windows version (The Volatility Foundation, 2014). However, manually reverse engineering every file in our large corpus of *win32k.sys* binaries is time consuming and error prone. Some forensic tools simply contain the reversed profile data as "Magic Numbers" embedded within their code (The Volatility Foundation, 2014) without an explanation of where these numbers came from, making forensic validation and cross checking difficult.

We wish to automatically extend this analysis to new binaries with minimal effort. We therefore want to express the required assembler pattern as a template which can be applied to the new file's disassembly. In practice, however, the compiler is free to mix use of registers in functions, or reorder branches. Often identical source code will generate assembler code using different registers, and different branching order.

Fig. 5 shows the same code segment from two different versions of the *xxxCreateWindowStation* function. As can be seen, although the general sequence of instructions is similar, the exact registers are different for each case (This

```
win7.elf 22:39:38> analyze_struct ``*win32k!grpWinStaList''
0xfa80022b0090 is inside pool allocation with tag 'Win\xe4' (0xfa80022b0000)
    Offset     Content
-------------- -------
          0x10 Data:0xfa8001853bc0 Tag:Des\xeb @0xfa8001853bc0
          0x18 Data:0xf960003af340 Const:win32k!gTermIO
          0x28 Data:0xf900c01369f0 Tag:Uskb @0xf900c01369f0
          0x78 Data:0xf8a002953880 Tag:AtmT @0xf8a002953880
          0x90 Data:0xf900c1aa0880 Tag:Usse ProcessBilled:winlogon.exe

win7.elf 22:42:14> print win32k_profile.tagWINDOWSTATION(0xFA80022B0090)
[tagWINDOWSTATION tagWINDOWSTATION] @ 0xFA80022B0090
  0x00 dwSessionId             [unsigned long:dwSessionId]: 0x00000001
  0x08 rpwinstaNext           <tagWINDOWSTATION Pointer to [0x00000000]>
  0x10 rpdeskList             <tagDESKTOP Pointer to [0xFA8001853BC0]>
  0x18 pTerm                  <tagTERMINAL Pointer to [0xF960003AF340]>
  0x20 dwWSF_Flags             [unsigned long:dwWSF_Flags]: 0x00000000
  0x28 spklList               <tagKL Pointer to [0xF900C01369F0]>
  0x30 ptiClipLock            <tagTHREADINFO Pointer to [0x00000000]>
  0x38 ptiDrawingClipboard    <tagTHREADINFO Pointer to [0x00000000]>
  0x40 spwndClipOpen          <tagWND Pointer to [0x00000000]>
  0x48 spwndClipViewer        <tagWND Pointer to [0x00000000]>
  0x50 spwndClipOwner         <tagWND Pointer to [0x00000000]>
  0x58 pClipBase              <Array Pointer to [0x00000000]>
  0x60 cNumClipFormats         [unsigned long:cNumClipFormats]: 0x00000000
  0x64 iClipSerialNumber       [unsigned long:iClipSerialNumber]: 0x00000000
  0x68 iClipSequenceNumber     [unsigned long:iClipSequenceNumber]: 0x00000000
  0x70 spwndClipboardListener <tagWND Pointer to [0x00000000]>
  0x78 pGlobalAtomTable       <_RTL_ATOM_TABLE Pointer to [0xF8A002953880]>
  0x80 luidEndSession         [_LUID luidEndSession] @ 0xFA80022B0110
  0x88 luidUser               [_LUID luidUser] @ 0xFA80022B0118
  0x90 psidUser               <Void Pointer to [0xF900C1AA0880]>
```

Fig. 4. Rekall analysis of the global symbol *grpWinStaList* which contains an allocation of type *tagWINDOWSTATION*. This is followed by the exact struct layout as extracted from the PDB file.

```
MOV RSI, [RIP+0x276b62]        0x0 win32k!gptiCurrent
MOV RBP, [RSI+0x178]
TEST RBP, RBP
```

```
MOV RSI, [RIP+0x2bbfdc]        0x0 win32k!gptiCurrent
MOV R15, [RSI+0x190]
TEST R15, R15
```

```
tagTHREADINFO:
  rpdesk:
  - - Disassembler
    - rules:
      - MOV $var1, *gptiCurrent
      - MOV $var2, [$var1+$out]
      - TEST $var2, $var2
      start: win32k!xxxCreateWindowStation
      target: Pointer
      target_args:
        target: tagDESKTOP
```

Fig. 5. Disassembled code for finding the *tagTHREADINFO.rpdesk* member offset. Even though the code is identical, different versions use different registers. We define a search template (Below) in YAML format to describe the required pattern regardless of the exact registers used.

function essentially checks the *rpdesk* pointer of the global variable *gptiCurrent*, a global *tagTHREADINFO* struct). We therefore construct our pattern match in such a way that exact register names are not specified. We only require the same register to be used for *$var1* throughout the pattern.

Additionally, the compiler may reorder Assembler code fragments from version to version. When a branch is reordered, the pattern match may be split into different parts of the branching instruction. In order to normalize the effect of branching, we unroll all branches in the assembly output. This means we follow all branches until we reach code that is already disassembled and then backtrack to resume disassembly from the branch onwards. This technique allows us to match our pattern against the complete code of each function.

For example consider Fig. 6. This shows a very short function *win32k!SetGlobalCursorLevel* which dereferences many pointers to a number of structs. The function iterates over all desktops (*tagDESKTOP*) and all threads (*tagTHREADINFO*) and sets their cursor level. It is quite simple to infer the structs and fields involved when reading the assembly code (for Windows 7) in conjunction with the struct definitions exported in the PDB files for Windows 7. The same templates can then be applied for other versions of the binary for which there are no exported symbols.

Our template can now be published and independently cross validated for accuracy. For example, in the event that investigators find a different version of the binary in the wild, they are able to apply the templates and re-derive the struct offsets directly from the binary – cross validating the resulting profile.

It must be noted that this technique does not work in every case since the code does change from version to version, sometimes dramatically. We therefore offer a number of possible templates (to different functions) that can be applied in turn until a match is found.

Results

We have collected 133 unique versions of the "win32k.sys" driver binary, and downloaded PDB files for these samples. We then generated assembler templates for many struct fields and ran these templates over these binaries in our collections.

Fig. 7 shows a summary of struct offsets across different versions of the win32k driver. As can be seen, the struct offsets are generally not changed between major and minor binary versions, although they do vary between each minor version.

Similarly, Fig. 8 shows that global constants vary wildly from build to build, hence version number alone is insufficient to provide reliable offsets for these constants.

Discussion

This study's main goal was to characterize what factors change between various binary versions, and how these are relevant to memory analysis. We found that generally, struct layout does not change within the same minor version, but global constants were found to vary wildly with version.

In our quest to characterize the variation we have developed a number of very useful techniques:

1. We have developed a technique to build a "profile index" – a mechanism to quickly detect which profile from a pre-calculated profile repository is applicable for a specific memory image. Our method is resilient to anti-forensic manipulation since it uses a random selection of comparison points chosen from the binary code and data segments themselves.
2. We have also demonstrated a data analysis technique for rapidly determining struct offsets by analyzing kernel pool allocations.
3. We have created an Assembler templating language which can be used to match sequences of assembler code in order to extract struct offsets for struct members. This technique can be applied for static binaries as well as binaries found in memory images.

How should these techniques be applied in order to improve the accuracy of memory analysis software?

```
Address                  Instruction          Comment
------------------ ------------------------ ------------------------------
------ win32k!SetGlobalCursorLevel ------: 0xf97fff1dc0a4
 0xf97fff1dc0a4    PUSH RBX
 0xf97fff1dc0a6    SUB RSP, 0x20
 0xf97fff1dc0aa    MOV RAX, [RIP+0x1b362f] win32k!grpdeskRitInput
 0xf97fff1dc0b1    MOV EBX, ECX
 0xf97fff1dc0b3    TEST RAX, RAX
 0xf97fff1dc0b6    JZ 0xf97fff1dc0da       win32k!SetGlobalCursorLevel+0x36
. 0xf97fff1dc0da   MOV RCX, [RIP+0x1b3827]  win32k!gpepCSRSS
. 0xf97fff1dc0e1   CALL QWORD [RIP+0x151ce9] win32k!imp_PsGetProcessWin32Process
. 0xf97fff1dc0e7   MOV RCX, [RAX+0x128]     ; tagPROCESSINFO.ptiList
. 0xf97fff1dc0ee   JMP 0xf97fff1dc11f       win32k!SetGlobalCursorLevel+0x7b
.. 0xf97fff1dc11f  TEST RCX, RCX
.. 0xf97fff1dc122  JNZ 0xf97fff1dc105       win32k!SetGlobalCursorLevel+0x61
... 0xf97fff1dc105 MOV RAX, [RCX+0x178]     ; tagTHREADINFO.pq
... 0xf97fff1dc10c MOV [RCX+0x290], EBX     ; tagTHREADINFO.iCursorLevel
... 0xf97fff1dc112 MOV [RAX+0x148], EBX     ; tagQ.iCursorLevel
... 0xf97fff1dc118 MOV RCX, [RCX+0x238]     ; tagTHREADINFO.ptiSibling
.. 0xf97fff1dc124  ADD RSP, 0x20
.. 0xf97fff1dc128  POP RBX
.. 0xf97fff1dc129  RET
 0xf97fff1dc0b8    MOV RAX, [RAX+0x18]       ; tagDESKTOP.rpwinstaParent
 0xf97fff1dc0bc    MOV RDX, [RAX+0x10]       ; tagWINDOWSTATION.rpdeskList
 0xf97fff1dc0c0    JMP 0xf97fff1dc0d5        win32k!SetGlobalCursorLevel+0x31
. 0xf97fff1dc0d5   TEST RDX, RDX
. 0xf97fff1dc0d8   JNZ 0xf97fff1dc0c2        win32k!SetGlobalCursorLevel+0x1e
.. 0xf97fff1dc0c2  LEA R8, [RDX+0xa0]        ; tagDESKTOP.PtiList
.. 0xf97fff1dc0c9  MOV RCX, [R8]             ; _LIST_ENTRY.Flink
.. 0xf97fff1dc0cc  CMP RCX, R8
.. 0xf97fff1dc0cf  JNZ 0xf97fff1dc0f0        win32k!SetGlobalCursorLevel+0x4c
... 0xf97fff1dc0f0 MOV RAX, [RCX-0x108]     ; tagTHREADINFO.PtiLink
... 0xf97fff1dc0f7 MOV [RCX+0x10], EBX
... 0xf97fff1dc0fa MOV [RAX+0x148], EBX      ; tagQ.iCursorLevel
... 0xf97fff1dc100 MOV RCX, [RCX]
... 0xf97fff1dc103 JMP 0xf97fff1dc0cc        win32k!SetGlobalCursorLevel+0x28
.. 0xf97fff1dc0d1  MOV RDX, [RDX+0x10]
```

```
tagDESKTOP:
  PtiList:
  - - Disassembler
    - rules:
      - MOV $var1, *grpdeskRitInput
      - TEST $var1, $var1
      - MOV $var1, [$var1+$rpwinstaParent]
      - MOV $pdesk, [$var1+$rpdeskList]
      - LEA *, [$pdesk+$out]
      start: win32k!SetGlobalCursorLevel
      target: Pointer
      target_args:
        target: _LIST_ENTRY
      max_separation: 300
```

Fig. 6. An example of matching an assembler pattern across a short function. First the function is unrolled such that all its branches are displayed. The pattern is then applied such that the same registers are used in a consistent manner. By comparing the assembly code to the struct field offsets in the exported PDB we can easily infer the types of structs used in this function. We can then extrapolate this inference to deduce struct offsets for binary versions we have no debugging information for.

As noted previously, some memory analysis frameworks currently use techniques such as pool scanning, disassembling and other heuristics to guess the locations of global kernel variables (The Volatility Foundation, 2014). This is especially problematic when trying to locate *win32k.sys* global parameters since the GUI subsystem has a different pool area for each session. Without contextual information, pool scanning techniques can not associate the correct kernel structures to the correct session, leading to many erroneous results.

It is therefore desirable to rely on accurate profile information in locating global structures. This warrants the creation and maintenance of a public profile repository with accurate symbol information for each version observed in the wild (The Rekall Team, 2014a, b). The problem remains however, how does one know which profile should be used for a specific memory image?

By applying the profile indexing technique, one can reliably detect the correct profile to use for each memory image. The profiles can then contain exact offsets of global variables and functions. This improves analysis because there is a large amount of accurate information available (for example it is possible to resolve addresses to function names – really helping with disassembly views).

Finally, we can address the problem of undocumented struct layouts. While the *win32k.sys* profiles do contain the addresses of global variables and functions, most do not contain struct layout.

Although we can apply the assembler templates to deduce the struct layouts directly within the memory image, this is not a reliable technique since in practice, many code pages will not be mapped into memory – causing the disassembly of the required functions to fail.

Instead we can collect *win32k.sys* binaries of all major and minor versions and apply the disassembly templates to the binaries themselves. Although we can never be absolutely sure that struct layouts are the same in all builds of the same version, our analysis suggests this is the case. That

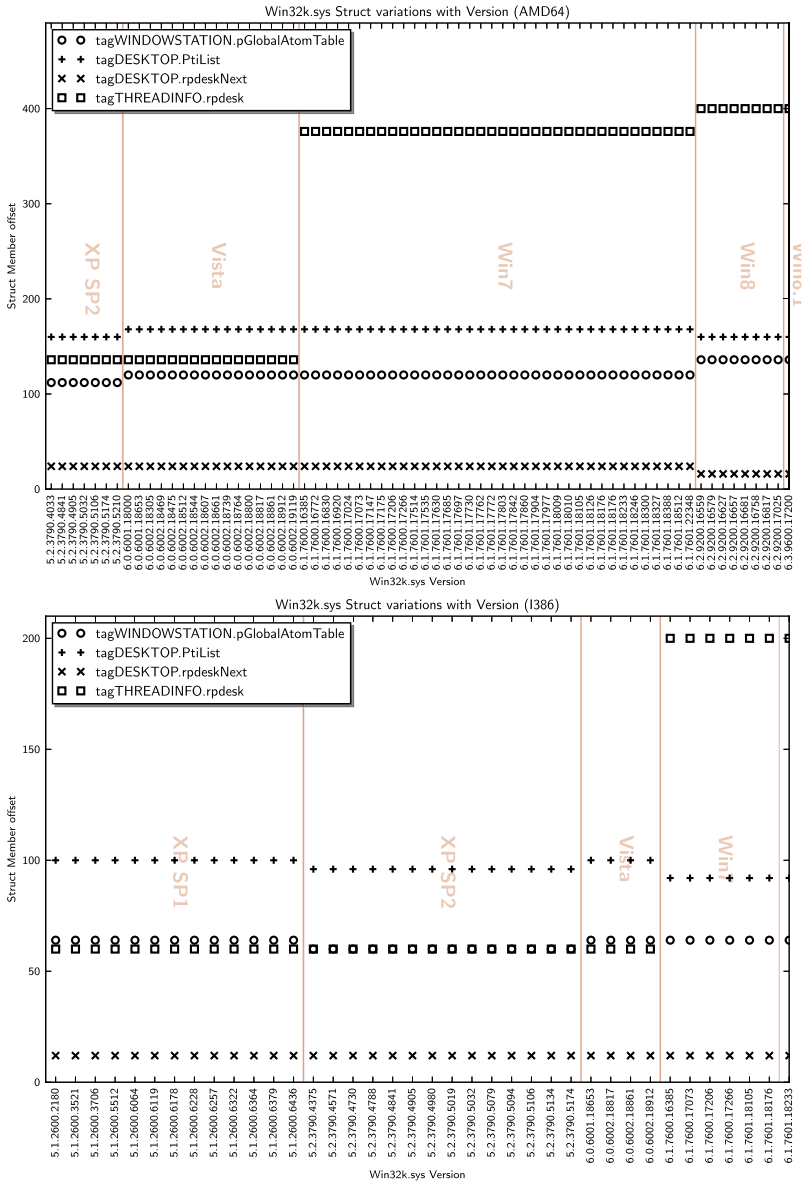

Fig. 7. Offsets for a selection of struct members across various versions of the Windows GUI subsystem. These offsets were derived by applying the automated disassembly templates on the driver executable.

is, the struct layout for *win32k.sys* depends only on the major and minor version numbers of the *win32k.sys* binary itself. We therefore make the assumption that struct layout does not vary between major and minor versions (this assumption seems to hold well as a result of this research).

Therefore, we construct a profile for all *win32k.sys* binaries by merging the global constants and functions found in the PDB files provided by Microsoft with the canonical struct layout for the specific major and minor version. We then similarly create a "profile index" for all known *win32k.sys* profiles and apply it on in the memory image to detect the correct profile to use.

Once the correct profile is found (containing both accurate constants and accurate struct layouts) we can use it to conduct analysis of the memory image without problems.

Limitations of symbol based memory analysis

In this paper we find that kernel constants vary greatly between kernel builds. We advocate locating the kernel constants directly from the debugging symbols distributed by Microsoft. While this approach makes for an efficient analysis, which is less susceptible to manipulation, it does have some shortcomings.

The main problem is that we require the PDB files for the exact versions of the kernel we are dealing with to be available. While Microsoft typically publishes PDB files for

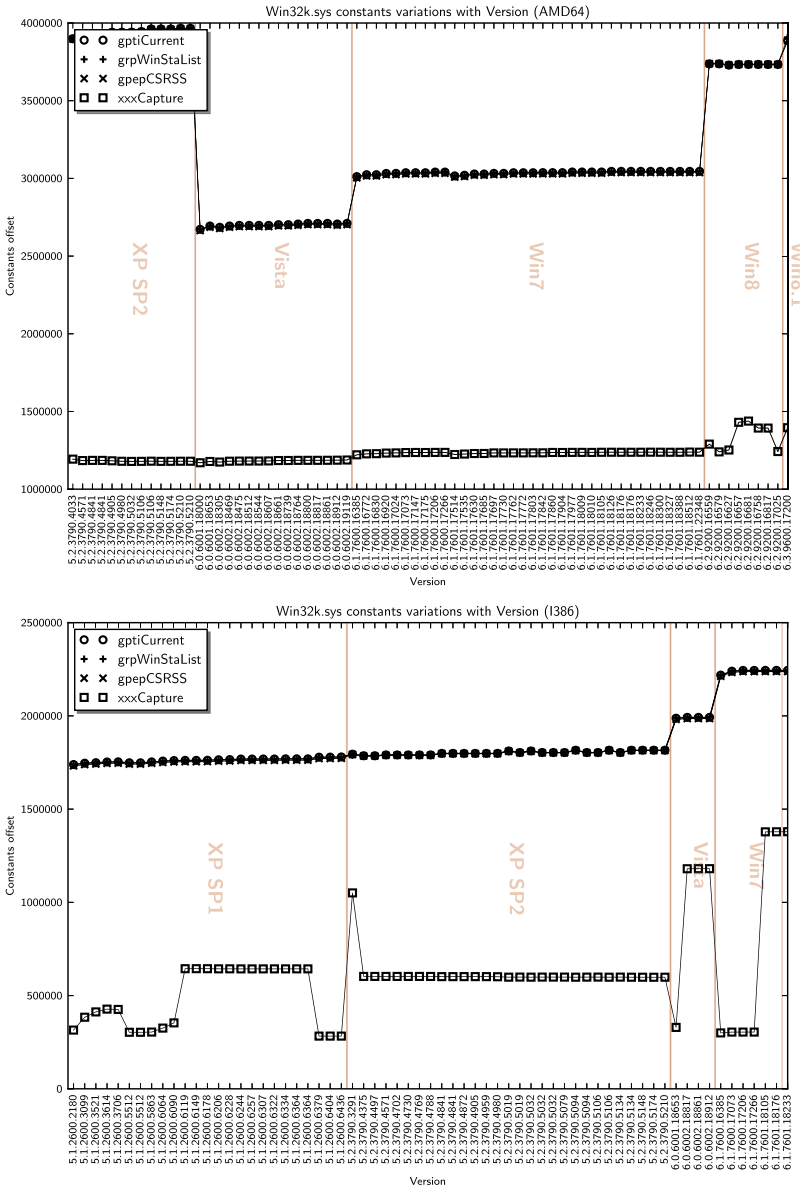

Fig. 8. Offsets for a selection of global constants across various versions of the Windows GUI subsystem. These offsets were derived by parsing the provided PDB files for these binary versions.

publicly released versions of the operating system, it is possible that PDB files for private, or development versions of the operating system are not published.

When Rekall encounters a windows kernel version which does not exist in the repository, the user may follow a procedure to add it to the repository by downloading the corresponding debug information from the Microsoft symbol server. However, if this is not possible (perhaps because the PDB file is not published), the user is unable to proceed at all. Rekall does not employ scanning or guessing techniques for locating kernel global constants without having the profile information (e.g. like Volatility does).

Conclusions and future work

Although this paper concentrates specifically on the Windows kernel binary and the win32k.sys GUI subsystem driver, the techniques presented are applicable for other drivers and binaries.

Specifically, the *tcpip.sys* driver manages the network stack and is largely undocumented. The same techniques we develop for constructing profiles from a mixture of documented and undocumented (reversed) information can be applied to this case.

Identifying which of a set of known binaries matches the exact running binary in a memory image is a critical

first step to memory analysis of all operating systems. For example, we have extended this method to auto-detect the exact kernel running on an OSX system.

The ability to generate profiles with more accurate information allows one to abandon using scanning and guessing techniques for determining this information from the potentially compromised memory image itself. The less the framework relies on the memory image to derive analysis information, the more resilient it is to malicious manipulation. For example, the literature has noted that the Kernel Debugger Block can be easily overwritten by malware in such a way that memory analysis can fail to find it (Haruyama and Suzuki, 2012).

Finally, this paper presents the groundwork for ultimately addressing the difficult problem of Linux memory analysis. Linux kernel struct layouts vary wildly based on kernel configuration as well as purely on kernel version. Only recently has it become possible to acquire memory on a Linux system in a kernel version agnostic manner (Stüttgen and Cohen, 2014), but there is a wide need to reliably determine the correct profile for unknown kernels – often encountered during incident response situations.

Previously, systems were proposed that attempted to derive all kernel struct offsets by examining the specific assembly instructions. However these systems, failed to take into account register swapping and function re-branching (Case et al., 2010), making them less reliable for matching real kernels in practice. This paper's proposed assembler templates are much more robust to these variations. Previous dynamic analysis platforms attempt to build a complete profile from the reversed parameters. However, as shown in this paper, we only need to gather just enough information to select the correct profile from a finite set of known profile variations. Future work can apply the techniques discussed in this paper to auto-detecting a Linux profile from an unknown kernel.

References

Aho AV, Corasick MJ. Efficient string matching: an aid to bibliographic search. Commun ACM 1975;18(6):333–40.

Case A, Marziale L, Richard III GG. Dynamic recreation of kernel data structures for live forensics. Digit Investig 2010;7:S32–40.

Chen R. Why do windows functions all begin with a pointless mov edi, edi instruction?. 2011. URL, http://blogs.msdn.com/b/oldnewthing/archive/2011/09/21/10214405.aspx.

Cohen M, Bilby D, Caronni G. Distributed forensics and incident response in the enterprise. Digit Investig 2011;8:S101–10.

Dolan-Gavitt B. Push the red button: the types stream. 2007. http://moyix.blogspot.de/2007/10/types-stream.html.

Dolan-Gavitt B. The vad tree: a process-eye view of physical memory. Digit Investig 2007b;4:62–4.

Haruyama T, Suzuki H. One-byte modification for breaking memory forensic analysis. Black Hat Europe; 2012.

Ligh M, Adair S, Hartstein B, Richard M. Malware analyst's cookbook and DVD: tools and techniques for fighting malicious code. Wiley Publishing; 2010.

Ligh MH, Case A, Levy J, Walters A. The art of memory forensics: detecting malware and threats in Windows, Linux, and Mac memory. 1st ed. Wiley Publishing; 2014.

Lin Z, Rhee J, Zhang X, Xu D, Jiang X. Siggraph: brute force scanning of kernel data structure instances using graph-based signatures. In: NDSS symposium; 2011.

Mandt T. Kernel attacks through user-mode callbacks. 2011. URL, http://media.blackhat.com/bh-us-11/Mandt/BH/_US/_11/_Mandt/_win32k/_WP.pdf.

Okolica J, Peterson GL. Windows operating systems agnostic memory analysis. Digit Investig 2010;7:S48–56.

Roussev V, Ahmed I, Sires T. Image-based kernel fingerprinting. Digit Investig 2014;11:S13–21.

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

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

Schuster A. Ptfinder (version 0.3.05). 2007. http://computer.forensikblog.de/en/2007/11/ptfinder-version-0305.html.

Schuster A. The impact of Microsoft Windows pool allocation strategies on memory forensics. Digit Investig 2008;5:S58–64.

Sikorski M, Honig A. Practical malware analysis: the hands-on guide to dissecting malicious software. 1st ed. San Francisco, CA, USA: No Starch Press; 2012.

Skape. Temporal return addresses, exploitation chronomancy. 2005. Uninformed 2. URL, http://www.uninformed.org/?v=2&a=2.

Stüttgen J, Cohen M. Robust Linux memory acquisition with minimal target impact. Digit Investig 2014;11:S112–9.

Sun XX, Chen H, Wen Y, Huang MH. Reversing engineering data structures in binary programs: overview and case study. In: Innovative mobile and internet services in ubiquitous computing (IMIS), 2012 Sixth international conference on IEEE; 2012. p. 400–4.

The Rekall Team. The rekall memory forensic framework. 2014. URL, http://www.rekall-forensic.com/.

The Rekall Team. The rekall profile repository. 2014. URL, https://github.com/google/rekall-profiles.

The Volatility Foundation. The volatility framework. 2014. URL, http://www.volatilityfoundation.org/.

Williams J, Torres A. Add – complicating memory forensics through memory disarray. 2014. URL, https://archive.org/details/ShmooCon2014/_ADD/_Complicating/_Memory/_Forensics/_Through/_Memory/_Disarray.

Yuan F. Windows graphics programming: Win32 GDI and DirectDraw. Prentice Hall Professional; 2001.