Surveying The User Space Through User Allocations

By

Andrew White, Bradley Schatz and Ernest Foo

From the proceedings of

The Digital Forensic Research Conference

DFRWS 2012 USA

Washington, DC (Aug 6th - 8th)

Contents lists available at SciVerse ScienceDirect

Digital Investigation

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

Surveying the user space through user allocations

Andrew White*, Bradley Schatz, Ernest Foo

Queensland University of Technology, Brisbane, Australia

Keywords:
Memory forensics
User space
Windows XP
Windows 7
Malware analysis

A B S T R A C T

Previous research into memory forensics has focused on understanding the structure and contents of the kernel space portions of physical memory, and mostly ignored the contents of the user space. This paper describes the results of a survey of user space virtual address allocations in the Windows XP and Windows 7 operating systems, comprehensively identifying the kernel and user space metadata required to identify such allocations. New techniques for determining the role and content of those allocations are identified, significantly increasing the proportion of allocations for which the role and function is understood. The validity of this approach is evaluated and a detailed analysis of the data structures involved provided. An implementation of this approach is presented which is capable of identifying all user space allocations, and for those allocations identifying for a high percentage, the role of those allocations, even for complex applications.

1. Introduction

Memory forensics provides a valuable way of analysing the contents of physical memory, in order to obtain transient information that would not necessarily be present on disk. Previous research in this area has focused on understanding and interpreting the layout and contents of the kernel portion of memory, in order to facilitate the development of memory analysis tools that recreate the capabilities of previously used live response tools. While this work has been invaluable, allowing the examination of processes, drivers, network sockets and other useful artefacts, it has not been taking into account the full picture.

Previous research has focused almost exclusively on the data the operating system itself is using, not the data of the user applications running on that operating system. As a result, the data and data structures used by these applications have not been examined, resulting in a lack of methodologies for understanding and interpreting this application data. Without such methods, it is not possible to extract useful information from the memory of an application, such as user credentials and chat logs, without first undertaking significant reverse engineering of the application itself.

In order to explain the data being used by an application, this paper presents an approach for describing the purpose of all memory allocations in the user space. This approach uses the Virtual Address Descriptors (VADs) proposed by previous research (Dolan-Gavitt, 2007) to identify all of these allocations, and then determines the purpose of these allocations using numerous sources of kernel and user space metadata. A sample implementation of this approach is then provided as a plugin for the Volatility framework (Volatile Systems, 2011), with an evaluation of both its validity and effectiveness. All of this research is based on extensive analysis of both the Windows XP SP3 32-bit and Windows 7 SP1 32-bit operating systems and their inner workings. Unless otherwise stated, all findings apply to both of the operating systems being studied.

The contributions of this paper are the detailed analyses of metadata sources that can be used to describe the purpose of user space allocations, and a plugin that implements this approach. The structure of this paper is as follows. Section 2 introduces the concept of user allocations, and Section 3 describes the ways in which metadata

* Corresponding author.
E-mail addresses: a13.white@qut.edu.au (A. White), b.schatz@qut.edu.au (B. Schatz), e.foo@qut.edu.au (E. Foo).

1742-2876/$ – see front matter © 2012 A. White, B. Schatz & E. Foo. Published by Elsevier Ltd. All rights reserved.
doi:10.1016/j.diin.2012.05.005

that describes these allocations can be located. Section 4 outlines the implementation of this approach as a Volatility plugin. Section 5 demonstrates the validity of this approach and its effectiveness, and these results are further discussed in Section 6. The related work is then presented in Section 7, before the paper is concluded in Section 8.

2. User allocations

Each process within Windows has its own virtual address space, and this virtual address space is divided into two parts, the user space, and the kernel space (Russinovich and Solomon, 2005). As the names suggest, the user space is where user application code and data is stored, and the kernel space is where code and data used by the kernel is stored.

Windows manages the use of user space memory by allocating memory in chunks of contiguous virtual address space ranges, which we term *user allocations*. The memory ranges which these user allocations occupy are described within a process by Virtual Address Descriptors (VADs). These VADs are stored in a self balancing binary tree, and each time a new user allocation is made, a new VAD is added to the tree to describe the memory ranges used by that allocation (Russinovich and Solomon, 2005). These user allocations are the highest level of abstraction possible within the user space of a process, and serve as a suitable framework within which to further explore the user space. An experiment to verify the suitability of VADs for locating these user allocations is described in Section 5.1.

While previous work has shown how these user allocations can be identified, techniques to determine the purpose of these allocations have been limited. Knowing the roles of these user allocations makes it possible to determine which allocations are relevant in a particular line of enquiry. For example, if the value of a user variable is the search, knowledge of where the application data is stored would significantly reduce the number of allocations requiring analysis. Without such knowledge, all the allocations used by the application would require analysis, including allocations which are not even relevant to the execution of the program.

Fig. 1 provides a comparison of which contents and roles of user allocations have been determined by previous research, and those newly determined by our research. The next section is dedicated to describing how each of these new roles can be determined.

Existing Research	Our Research
Files	Sections
Process Parameters (XP)	Environment (XP)
Stacks	Heaps
PEB	Heap Segments
TEB	Heap Virtual Allocs
	Shared Heaps
	Code Page
	Activation Contexts
	GDI Shared Handle Table
	User Shared Data
	Private Shim Data
	Error Reporting (Win7)

Fig. 1. Contents and roles capable of being determined.

3. User allocation metadata

To describe the contents and roles of these user allocations within the user space, sources of metadata which describe these allocations must be consulted. These sources of metadata can be divided based on whether they exist in the kernel space, or in the user space. Generally speaking, the sources of metadata in the kernel focus on the role of the user allocation, while the sources of metadata in the user space focus on the content of the user allocation. Unless otherwise stated, all of the following information has been determined through our own analyses of Windows XP SP3 and Windows 7 SP1 systems.

3.1. Kernel space metadata sources

The first source of metadata for a user allocation is of course the VAD that describes that allocation. In addition to providing the location of the user allocation, the VAD also provides two other pieces of information. The first is the permissions with which that memory was allocated, which are enforced in software by the Windows Memory Manager. These permissions can take on a variety of values, and are defined in *WinNT.h*.

The second piece of useful information is the type of the user allocation, which can either be *private* or *mapped*. A private allocation only exists within the one user address space, whereas a mapped allocation exists in more than one address space. For example, these multiple address spaces could be the user and kernel portions of the one virtual address space, or multiple virtual address spaces. The type of an allocation is determined by the presence of a pointer in the *ControlArea* field, if a pointer exists, the allocation is mapped, otherwise it is private.

Aside from a VAD, a user allocation can potentially have two different types of objects that provide metadata about the allocation, a *_FILE_OBJECT* and a *_SECTION_OBJECT*. As the names suggest, a file object describes an allocation backed by a file on disk, while a section object describes a section, which is the term Windows uses to describe an allocation that is shared between multiple processes. Both of these objects are used to identify when physical memory pages are shared between multiple address spaces, which means they only ever apply to *mapped allocations*. In order to locate these objects, an understanding of the data structures involved in shared memory, the control area and the segment, are first required.

A control area, described by a *_CONTROL_AREA* data structure, is responsible for keeping track of how many address spaces an allocation has been mapped to. A segment, described by a *_SEGMENT* data structure, keeps track of the page table entries used by an allocation that is shared, such that for each address space it exists in, it always points to the same physical pages.

These two data structures are highly related, to the extent that a control area has a pointer to its corresponding segment in the *Segment* field, and a segment has a pointer its corresponding control area in the *ControlArea* field.

Control areas and segments always point to each other, essentially forming a pair of data structures.

As previously mentioned, a mapped allocation is one that has a valid pointer in the *ControlArea* field. This field, as the name suggests, points to a control area, making it easy to find the control area and corresponding segment that describe an allocation.

If a mapped allocation represents a file object, the control area will have a pointer to a valid file object in the *FilePointer* field, as noted by Dolan-Gavitt (2007). This file object will reveal several pieces of information about the file, including its address on disk, which allows more traditional techniques from disk forensics to be used.

If the *FilePointer* field of the control area is null, then the mapped allocation represents a section. Unlike with file objects, the control area does not have a field pointing to the section object, and neither does the segment. Instead, to locate relevant section objects, the object manager needs to be parsed.

The object manager is responsible for the creation and deletion of kernel objects, and allowing the retrieval of specific objects by their *handle* (Russinovich and Solomon, 2005). Handles are designed to be passed on to user code, as a method of allowing access to specific system resources, such as the file object for an open file. Each process has a link to its process handle table, which contains the handles that are in use by that process.

By parsing through the handle table of each process, section objects can be found. Section objects take the form of a *_SECTION_OBJECT*, which contains a variety of fields. However, in our experiments, the only field which is always used is the *Segment* field, which although the Windows symbol files indicate should point to a *_SEGMENT_OBJECT*, in fact points to a *_SEGMENT*.

This makes it possible to determine which shared memory sections are in use by a process, by relating the segments found by parsing the VAD to those found by interpreting the sections in the object manager. An example of this relationship is shown in Fig. 2. One key point to note is that in parsing the object manager, the handle table of all processes must be parsed, not just the process that owns the virtual address space in question. This is required as when one process maps shared memory into another, only one of these processes requires access to the handle.

Unlike the file object, the section object by itself provides limited information about the allocation, however it does facilitate the retrieval of two useful pieces of information. The first is the name of the section object, which is recorded in the object manager and can be determined while finding the section object. While the name of the section may reveal some information about its contents, in our observations it often contained a blank string or a string filled with non-printable characters. These types of section names are quite common, which could indicate that these non-printable characters are being used to store some other sort of information.

The second piece of useful information is the process that created the section. When a segment is pointed to by a section object, the *u1* field represents the *CreatingProcess* field. The value of this field is a pointer to an *_EPROCESS* structure, and as the name suggests, this allows the process that created the section to be determined. Since such sections are generally used for inter-process communication, it is often the case that the creating process is a different process to the process being examined. When the creating process is the same process, this could indicate that this section is being shared with other processes, or that the process is sharing information with part of the kernel.

One point to note is that it is possible for a VAD to have an associated file object and section object at the same time. Such an occurrence is commonly the case with the natural language support (.nls) files which are mapped as read-only into every process. The significance of this is that the same view of the file is being shared between multiple processes, not a unique view of the file as is typically achieved with the copy-on-write style permissions used for DLL files.

While a VAD entry describes which part of the virtual address space a user allocation occupies, it is the metadata of this entry obtained from file and section objects that allows the role of the allocation to be determined. Even though the description of these allocations is at a high level, with no explicit knowledge of the underlying data structures, the information can still be used in a variety of ways.

An example would be inferring the behaviour of a process based on the file and section objects present within its user space. If a process loads a DLL named *WINHTTP.dll* from the system root into the address space, one may hypothesise that the process is using the HTTP protocol. Likewise, if a section named *ShimSharedMemory* is being loaded, this would indicate that the Windows Shim Engine is being used, a part of Windows responsible for providing compatibility to older programs.

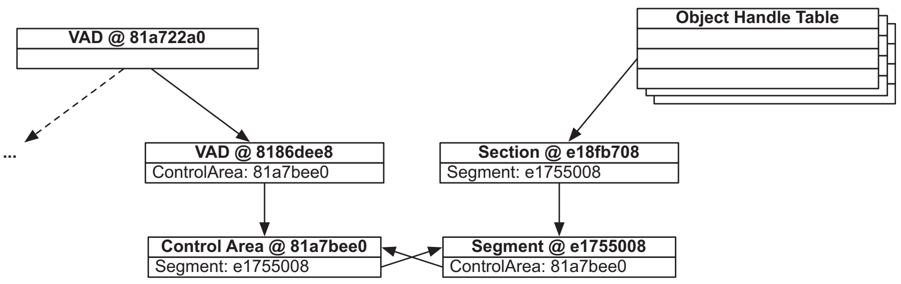

Fig. 2. Finding shared memory.

3.2. User space metadata sources

Although the role of numerous user allocations can be described using kernel sources of metadata, it is not possible to describe all user allocations in this way. However, additional sources of metadata exist in the user space, which can be located from known structures in the kernel space. For the most part, these user space metadata sources describe the content of the user allocations, in particular those that are created when a process is loaded. These default user allocations are often required for the application to function, or are used by some Windows component to provide helpful functionality.

The two main sources of metadata with which the content of all other user allocations can be described are the Process Environment Block, *_PEB*, and the Thread Environment Block, *_TEB*. While previous work has described how these can be used to list the modules loaded by a process, find the process parameters (Betz, 2005), and locate the stacks of a process (Arasteh and Debbabi, 2007), the PEB contains numerous more fields that, prior to this work, have not been identified in the literature.

The useful fields of the PEB are summarised in Fig. 3. This information has been taken from the Windows XP SP3 symbol files, and updated with some type specific information for some of the fields.

The *ImageBaseAddress*, *Ldr* and *ProcessParameters* fields were all fields first used by the *Memparser* tool (Betz, 2005). These fields can be used to find the virtual address of the process executable, list the loaded modules and retrieve the process parameters respectively. Although the list of loaded modules could potentially be identified using file objects from the VAD, this provides an alternative method with which to verify that information.

There are two key pieces of information which have not been noted by previous work about the process parameters. The first is that the data structure it uses, *_RTL_USER_PROCESS_PARAMETERS*, always resides within its own user allocation. The second is that the *Environment* field of this data structure can be used to find another user allocation, which contains a series of strings relating to environment variables. While this allows two additional allocations to be explained on Windows XP, the same does not apply to Window 7. Although Windows 7 continues to use the same data structures to store this information, these data structures are stored in the default heap rather than in their own allocations.

```
ntdll!_PEB
   ...
   +0x008 ImageBaseAddress : Ptr32 Void
   +0x00c Ldr              : Ptr32 _PEB_LDR_DATA

   +0x010 ProcessParameters : Ptr32
                              _RTL_USER_PROCESS_PARAMETERS
   ...
   +0x018 ProcessHeap      : Ptr32 _HEAP
   ...
   +0x04c ReadOnlySharedMemoryBase : Ptr32 _HEAP
   +0x050 ReadOnlySharedMemoryHeap : Ptr32 _HEAP
   +0x058 AnsiCodePageData : Ptr32 Void
   +0x05c OemCodePageData  : Ptr32 Void
   +0x060 UnicodeCaseTableData : Ptr32 Void
   ...
   +0x088 NumberOfHeaps    : Uint4B
   ...
   +0x090 ProcessHeaps     : Ptr32 Ptr32 _HEAP
   +0x094 GdiSharedHandleTable : Ptr32 Void
   ...
   +0x1e8 pShimData : Ptr32 Void
   ...
   +0x1f8 ActivationContextData : Ptr32 Void
   ...
   +0x200 SystemDefaultActivationContextData : Ptr32 Void
```

Fig. 3. Relevant fields of the PEB (XP).

The *ProcessHeap*, *NumberOfHeaps* and *ProcessHeaps* allow the location of the heaps used by the process, each of which reside in their own allocation. These heaps are found from the *ProcessHeaps* field, which is a pointer to a list of pointers to heaps, the size of which is given by *NumberOfHeaps*. The first heap of this list is also referenced by the *ProcessHeap* field, which is considered to be the default heap. There is also one heap that does not appear in this list, which is the read-only shared heap, pointed to by the values of *ReadOnlySharedMemoryBase* and *ReadOnlySharedMemoryHeap*.

Each of these heaps begin with a *_HEAP* object, which contains two fields that can be used to describe further user allocations. The first is the field which gives the heap segments. On Windows XP, this is the *Segments* field, which is an array of pointers to *_HEAP_SEGMENT* data structures on Windows XP, whereas on Windows 7, this is the *SegmentListEntry* field, which is a double linked list of the same data structure. A heap segment describes the heap within its current allocation, and as such, every heap always has at least one segment. If a heap extends to a separate user allocation however, that allocation will have its own heap segment. The second is the *VirtualAllocdBlocks* field, which is a double linked list of *_HEAP_VIRTUAL_ALLOC_ENTRY* data structures. A heap virtual alloc entry is used when a heap receives a request for memory exceeding a threshold size, at which point rather than the data being stored in the existing heap allocation it is stored within its own separate user allocation. As can be seen, the location and parsing of these heaps potentially allows the content of even more user allocations to be determined.

Returning to the PEB, the next significant fields are the *AnsiCodePageData*, *OemCodePageData* and *UnicodeCaseTableData* fields. These fields all point to different locations within the same user allocation, which is the code page. This code page is present in every process, and contains a series of tables containing information relating to ANSI, OEM and Unicode character encoding information. While the specific data structure used to store this information is not known, it is likely to be a simple array.

If a process has a GUI element, it will have a GDI shared handle table, which is pointed to by the *GdiSharedHandleTable* field of the PEB. This user allocation is a simple array of *GDITableEntry* data structures (Nasarre, 2003), and as the name suggests, is shared between all processes with a GUI element. Since these entries give both a virtual memory address and an owning process for each GDI object, it is possible to parse these entries to determine which allocations of the process contain local GDI information.

When a process uses the Windows Shim Engine, the PEB will potentially contain a valid pointer in the *pShimData* field. This field will point to a private allocation containing a small amount of data relating to the Shim Engine in an unknown data format.

Towards the end of the PEB are the *ActivationContextData* and *SystemDefaultActivationContextData* fields. These fields store an activation context, which is a mechanism used by Windows to redirect an application to load specific versions of resources, such as DLL files. An application can potentially have two user allocations as a result of this functionality, one for the system default activation context, and one for the activation context specified by the process. Although the data structure used for this purpose is not known, it begins with a distinct magic value of *Actx*.

While Windows 7 adds numerous new fields to the PEB, only two commonly provide useful metadata. These are the *pContextData* and *WerRegistrationData* fields. Although the name of the first field would suggest it is related to activation contexts, this does not seem to be the case, as it does not start with the magic value of *Actx*. As such, the purpose of this allocation is not known. The second field relates to data used for Windows Error Reporting functionality, and while the data structure used for this purpose is unknown, it is commonly paged out or for the most part unpopulated.

As demonstrated by the vast amount of metadata the PEB provides, it is invaluable source for determining the content of user allocations. Although a portion of these allocations relate to default data structures located in every process, this knowledge allows the exclusion of these allocations when their information is not required. Many of the allocations identified however are directly related to application data, such as the heap, allowing these allocations to be searched for key pieces of user information, such as login credentials.

4. Implementation

The interpretation techniques discussed in the previous sections have been implemented as a plugin for the Volatility Framework (Volatile Systems, 2011). Volatility was chosen for the implementation as it not only provides the base functionality required for memory analysis, such as memory address translation, but also provides numerous plugins covering functionality described in previous research. This allowed the development to focus only on implementing new functionality, rather than spending time reimplementing previous research.

Our plugin, *Userspace.py*, uses all of the aforementioned metadata sources to describe the contents of allocations within user space memory. This is achieved by following three main steps. Firstly, the specified process is located and its VAD Tree is parsed, using previous Volatility functionality. Secondly, the PEB and TEBs of the process are located, and the metadata within them is used to describe the relevant allocations. Finally, the control areas of each VAD are parsed to locate file objects, and then the handle table parsed to link the remaining undescribed user allocations to relevant section objects.

The development of this tool was conducted in parallel with the research, in order to provide an easy way of testing and validating hypotheses formed about the contents of various data structures. Small, custom programs with known functionality were created for the purposes of analysis, before moving onto more complex, existing applications. Windows XP SP3 and Windows 7 SP1 were the versions of Windows on which this research was based, and as such numerous memory dumps of this operating system were made to facilitate development. These memory dumps were all created using multiple virtual machines, with a variety of different memory sizes.

An example of the output from this implementation is shown in Fig. 4. This demonstrates the plugin running against *malloc.exe*, a simple C application that allocates a few KBs of memory, fills it with data and then sleeps. A more complex example using the Windows 7 version of *notepad.exe* is shown in Fig. 5.

The plugin is available at http://github.com/a-white/.

5. Tool evaluation

The evaluation of the aforementioned implementation was undertaken in three ways. Firstly, the assumption that a VAD tree completely describes a user space was tested, as the plugin relies on the VADs when identifying the user allocations. Secondly, the correctness of the plugin was validated through the extensive comparison of the results obtained against manual analysis of the involved data structures using existing tools. Lastly, once the results were known to be accurate, the plugin was then compared against existing approaches in order to investigate the benefit obtained from the use of the plugin.

All evaluations took place using Windows XP SP3 and Windows 7 SP1 virtual machines and the memory dumps made using these virtual machines. A variety of different virtual machine software was employed for this purpose.

5.1. VAD tree completeness

The outlined approach and subsequent plugin rely on the assumption that the VADs can be used to identify all user allocations within a virtual address space. If this were not the case, the VAD tree would not provide a suitable framework within which to analyse user space memory, as it would not allow the complete identification of user allocations. Without first identifying these user allocations, their purpose cannot be determined, affecting the completeness of the results from the plugin.

In order to validate this assumption, a simple experiment to test the inverse was devised. A simple Volatility plugin, *VadCompleteness.py* was created, which checks every process to see if a virtual address not described by a VAD maps to a physical memory page. This plugin was then used to analyse a series of Windows XP SP3 and Windows 7 SP1 memory images.

The surprising result of this analysis was that the VADs never completely described the user space of a process. All processes running on a Windows XP SP3 or Windows 7 SP1 system were found to have a single page mapped at the address 0x7FFE0000 which was not described by a VAD. In addition to being mapped at the same virtual address in every process, this virtual address always corresponded to the same physical page, meaning it was identical in each process.

```
Start    End      Used     Size     Permission        Type     Description
-------- -------- -------- -------- ----------------- -------- ---------------------------------------------
00010000 00010fff 00001000 00001000 READWRITE         Private  Environment
00020000 00020fff 00001000 00001000 READWRITE         Private  Parameters *
00030000 0012ffff 00002000 00100000 READWRITE         Private  Stack (Thread 1) *
00130000 00132fff 00002000 00003000 READONLY          Mapped   SystemDefaultActivationContextData
                                                               Section - PID 00584, Name ‘’
00140000 0023ffff 00004000 00100000 READWRITE         Private  Heap 0
00240000 0024ffff 00003000 00010000 READWRITE         Private  Heap 1
00250000 0025ffff 00002000 00010000 READWRITE         Mapped   Heap 2
00260000 00275fff 00005000 00016000 READONLY          Mapped   \WINDOWS\system32\unicode.nls *
                                                               Section - PID 00584, Name ‘NlsSectionUnicode’
00280000 002c0fff 00001000 00041000 READONLY          Mapped   \WINDOWS\system32\locale.nls *
                                                               Section - PID 00584, Name ‘NlsSectionLocale’
002d0000 00310fff 00000000 00041000 READONLY          Mapped   \WINDOWS\system32\sortkey.nls *
                                                               Section - PID 00584, Name ‘NlsSectionSortkey’
00320000 00325fff 00004000 00006000 READONLY          Mapped   \WINDOWS\system32\sorttbls.nls *
                                                               Section - PID 00584, Name ‘NlsSectionSortTbls’
00330000 0033ffff 00005000 00010000 READWRITE         Private  Heap 3
00340000 00342fff 00002000 00003000 READONLY          Mapped   \WINDOWS\system32\ctype.nls *
                                                               Section - PID 00584, Name ‘NlsSectionCType’
00350000 003d0fff 00081000 00081000 READWRITE         Private  Virtual Alloc of Heap 3
00400000 0040efff 0000e000 0000f000 EXECUTE_WRITECOPY Mapped   \Documents and Settings\xp\malloc.exe *
7c800000 7c8f5fff 0002b000 000f6000 EXECUTE_WRITECOPY Mapped   \WINDOWS\system32\kernel32.dll *
7c900000 7c9b1fff 0002f000 000b2000 EXECUTE_WRITECOPY Mapped   \WINDOWS\system32\ntdll.dll *
7f6f0000 7f7effff 00002000 00100000 EXECUTE_READ      Mapped   Shared Heap
                                                               Section - PID 00584, Name ‘’
7ffb0000 7ffd3fff 00005000 00024000 READONLY          Mapped   Code Page
7ffd7000 7ffd7fff 00001000 00001000 READWRITE         Private  PEB *
7ffdf000 7ffdffff 00001000 00001000 READWRITE         Private  TEB (Thread 1) *
7ffe0000 7ffe0fff 00001000 00001000 N/A               N/A      User Shared Data

Unreferenced Pages
Start    Size
```

Fig. 4. *Userspace.py* running against *malloc.exe* on Windows XP. Asterisks indicate allocations described by *Existing.py*.

This page is in fact the *_KUSER_SHARED_DATA* data structure, which is shared between all virtual memory address spaces. Numerous system wide variables are stored within this structure, such as the system time, the timezone, and the location of the system root.

Aside from this one page however, the VADs correctly described which virtual memory ranges mapped to physical memory pages for every process. While this demonstrates that the VAD tree provides a suitable framework for the analysis of user space memory, special care must be taken to first ensure that no additional addresses outside the memory ranges described by the VADs map to physical memory pages. For this reason, *Userspace.py* checks to see if any virtual addresses that are not described by a VAD map to a physical address, and outputs any such addresses.

5.2. Plugin validation

Once the accuracy of the VAD based approach had been verified, the validation process then turned to the plugin itself. Validating the results of the plugin required the in depth analysis of the data structures involved, and some existing tools were employed to achieve this.

The main tool employed was WinDbg (Microsoft, 2010), a Microsoft debugging program, which was used to analyse the data structures contained within Windows memory. This tool was used as it provides a way to access the numerous operating system data structures described by the Windows symbol files, and use these data structures to dump the specified address in memory. All validation of the results obtained from the plugin was performed by using WinDbg in this way to manually analyse specific data structures, and ensure that the resulting output from the plugin accurately represented these data structures. One limitation of WinDbg is that it only operates on crash dumps, a Windows specific memory image format, which prevented the use of other memory capture methods.

A selection of tools from the Sysinternals Suite (Russinovich, 2012) were also used in validating the results of the plugin. Since each of the tools were designed for live response, not memory analysis, this required that they be run on the virtual machine while simultaneously experimenting with test applications. Of these tools, VMMap, WinOBj and NotMyFault were the most heavily used. The VMMap tool provided a high-level view of the contents of each user allocation, which served as a useful starting point for analysis. The WinObj tool provided a method of exploring the object manager, allowing the validation of the information retrieved from the object manager through Volatility. Finally, the NotMyFault tool, which can cause a variety of system errors, was essential for creating the memory dumps in crash dump format for WinDbg.

5.3. Plugin comparison

In order to compare the *Userspace.py* to the existing approaches, this required a tool that implemented all of

```
Start    End      Used     Size     Permission        Type     Description
-------- -------- -------- -------- ----------------- -------- ---------------------------------------------
00010000 0001ffff 00001000 00010000 READWRITE         Mapped   Heap 1
00020000 00022fff 00003000 00003000 WRITECOPY         Mapped   \Windows\System32\en-US\notepad.exe.mui
00030000 00033fff 00004000 00004000 READONLY          Mapped   SystemDefaultActivationContextData
                                                               Section - PID 00356, Name ‘’
00040000 00041fff 00002000 00002000 READONLY          Mapped   ActivationContextData
                                                               Section - PID 00356, Name ‘’
00050000 00050fff 00001000 00001000 READWRITE         Private  pContextData
00060000 00060fff 00001000 00001000 READWRITE         Private  (GDI Data)
00070000 00070fff 00001000 00001000 READWRITE         Private  (GDI Data)
00080000 00080fff 00001000 00001000 READWRITE         Mapped   Section - PID 00308, Name ‘windows_shell_global_counters’
00090000 00091fff 00002000 00002000 READONLY          Mapped   Section - PID 00356, Name ‘’
000c0000 001bffff 00027000 00100000 READWRITE         Private  Heap 0 (GDI Data)
001d0000 0020ffff 00003000 00040000 READWRITE         Private  Stack of Thread 0
00210000 00276fff 00030000 00067000 READONLY          Mapped   \Windows\System32\locale.nls
00280000 00347fff 00005000 000c8000 READONLY          Mapped
00390000 0039ffff 00001000 00010000 READWRITE         Private  Heap 3
003a0000 0041ffff 00001000 00080000 READWRITE         Private
00440000 0044ffff 00003000 00010000 READWRITE         Private  Heap 2
00450000 00550fff 0000b000 00101000 READONLY          Private  GdiSharedHandleTable
00560000 0063efff 00053000 000df000 READONLY          Mapped   Section - PID 00308, Name ‘’
00700000 0073ffff 00010000 00040000 READWRITE         Private  Heap 4
00830000 0086ffff 00014000 00040000 READWRITE         Private  Heap 5
00870000 00b3efff 0001f000 002cf000 READONLY          Mapped   \Windows\Globalization\Sorting\SortDefault.nls
00c00000 00c2ffff 0000f000 00030000 EXECUTE_WRITECOPY Mapped   \Windows\System32\notepad.exe
00c30000 0182ffff 0000e000 00c00000 READONLY          Mapped
01830000 0215ffff 0003e000 00930000 READONLY          Mapped   \Windows\Fonts\StaticCache.dat
                                                               Section - PID 00308, Name ‘’
72e00000 72e50fff 00013000 00051000 EXECUTE_WRITECOPY Mapped   \Windows\System32\winspool.drv
73a00000 73a12fff 00009000 00013000 EXECUTE_WRITECOPY Mapped   \Windows\System32\dwmapi.dll
73d30000 73d6ffff 0001c000 00040000 EXECUTE_WRITECOPY Mapped   \Windows\System32\uxtheme.dll
73eb0000 7404dfff 0003c000 0019e000 EXECUTE_WRITECOPY Mapped   \Windows\winsxs\x86_microsoft.windows.common-[...]\comctl32.dll
74420000 74428fff 00007000 00009000 EXECUTE_WRITECOPY Mapped   \Windows\System32\version.dll
74e70000 74e7bfff 00008000 0000c000 EXECUTE_WRITECOPY Mapped   \Windows\System32\cryptbase.dll
74fd0000 75019fff 0001d000 0004a000 EXECUTE_WRITECOPY Mapped   \Windows\System32\KernelBase.dll
75220000 752bcfff 00043000 0009d000 EXECUTE_WRITECOPY Mapped   \Windows\System32\usp10.dll
752c0000 7536bfff 0001c000 000ac000 EXECUTE_WRITECOPY Mapped   \Windows\System32\msvcrt.dll
75370000 754cbfff 00019000 0015c000 EXECUTE_WRITECOPY Mapped   \Windows\System32\ole32.dll
754d0000 7551dfff 00015000 0004e000 EXECUTE_WRITECOPY Mapped   \Windows\System32\gdi32.dll
75520000 75576fff 00010000 00057000 EXECUTE_WRITECOPY Mapped   \Windows\System32\shlwapi.dll
758f0000 75990fff 00013000 000a1000 EXECUTE_WRITECOPY Mapped   \Windows\System32\rpcrt4.dll
75a00000 75ad3fff 00029000 000d4000 EXECUTE_WRITECOPY Mapped   \Windows\System32\kernel32.dll
75ae0000 75b6efff 0000b000 0008f000 EXECUTE_WRITECOPY Mapped   \Windows\System32\oleaut32.dll
75b70000 767b9fff 00017000 00c4a000 EXECUTE_WRITECOPY Mapped   \Windows\System32\shell32.dll
767c0000 76888fff 00028000 000c9000 EXECUTE_WRITECOPY Mapped   \Windows\System32\user32.dll
76a90000 76b0afff 00008000 0007b000 EXECUTE_WRITECOPY Mapped   \Windows\System32\comdlg32.dll
76cb0000 76d4ffff 00011000 000a0000 EXECUTE_WRITECOPY Mapped   \Windows\System32\advapi32.dll
76d50000 76d59fff 00007000 0000a000 EXECUTE_WRITECOPY Mapped   \Windows\System32\lpk.dll
76d60000 76d7efff 0000a000 0001f000 EXECUTE_WRITECOPY Mapped   \Windows\System32\imm32.dll
76dd0000 76f0bfff 0005d000 0013c000 EXECUTE_WRITECOPY Mapped   \Windows\System32\ntdll.dll
76f10000 76f28fff 00008000 00019000 EXECUTE_WRITECOPY Mapped   \Windows\System32\sechost.dll
76f30000 76ffbfff 00025000 000cc000 EXECUTE_WRITECOPY Mapped   \Windows\System32\msctf.dll
77010000 77010fff 00001000 00001000 EXECUTE_WRITECOPY Mapped   \Windows\System32\apisetschema.dll
7f6f0000 7f7effff 00005000 00100000 READONLY          Mapped   Heap 6 (Shared)
                                                               Section - PID 00356, Name ‘SharedSection’
7ffb0000 7ffd2fff 00013000 00023000 READONLY          Mapped   Code Page
7ffd4000 7ffd4fff 00001000 00001000 READWRITE         Private  PEB
7ffdf000 7ffdffff 00001000 00001000 READWRITE         Private  TEB (Thread 0)
7ffe0000 7ffe0fff 00001000 00001000 N/A               N/A      KSHARED_USER_DATA

Unreferenced Pages
Start    Size
```

Fig. 5. *Userspace.py* running against *notepad.exe* on Windows 7. OS.

these approaches. Unfortunately, no such tool existed, although the majority of the functionality required for these approaches had been implemented in Volatility. This meant a plugin to easily replicate this behaviour could be created. For the purposes of this section, we created *Existing.py*. This plugin identifies all user allocations using the VAD tree, checks each of these VADs for control areas with file objects, locates the process parameters from the PEB, and locates the stacks from the TEBs.

Since the plugins describe the contents of user space memory for a particular process, a set of processes were required for analysis. Four processes of increasing complexity were chosen for this purpose. The first, *malloc.exe*, was a simple C program created during development of the plugin. The behaviour of this program is very simple, it allocates a few KBs of memory, fills it with constant values, then sleeps for 20 s. The second, *calc.exe*, is the default calculator program for Windows. For each memory image, the calculator was opened, and a simple calculation performed. The third, *notepad.exe*, is the simple text editing program that comes preinstalled with Windows. In each captured memory image, notepad was used to open a new document, enter a paragraph of text, and then save the file. The final and most complex program chosen was *iexplore.exe*, the preinstalled web browser on Windows, which was was opened and allowed to completely load the default home page for each memory image. For the programs included by default on Windows, the default version after installation was used, no updates were applied.

The results of comparing *Existing.py* and *Userspace.py* over a series of memory images on both Windows XP and Windows 7 can be seen in Fig. 6. For each analysed program, the average number of user allocations for this program across the memory images is shown, followed by on average how many of those entries were able to be described by the *Existing.py* and *Userspace.py* plugins.

6. Discussion

As can be seen from Fig. 6, our plugin provides a significant improvement on the existing approaches in terms of determining the contents and roles of user allocations. Since our plugin implements a superset of the existing approaches, there is no scenario in which the existing approaches could outperform our plugin.

Comparing the results across the two operating systems provides some interesting results. Since the default versions of each Windows program were used in the experiments, this meant that the custom malloc.exe program was the only program to remain identical for both operating systems. For this simple program, the number of memory allocations decreased, partially due to the lack of dedicated allocations for the environment and process parameters data structures and partially due to reduction in loaded .nls files.

For the default Windows applications however, the number of allocations doubled for calc.exe and notepad.exe, and increased by 80% for iexplorer.exe. As is quite obvious from running both versions of Internet Explorer on the two operating systems, iexplorer.exe has changed dramatically between the two versions of Windows, and can almost be treated as a different program. Despite the differences in code between these versions however, the results for both versions of Internet Explorer remained almost the same.

Unlike Internet Explorer, Notepad seemed identical between Windows versions save for the updated visual theme on Windows 7, making the increase in allocations seem as purely the result of additional operating system complexity introduced by Windows 7. Even with this additional complexity however, the results favoured the Windows 7 version of notepad with a 10% increase in identified allocations for userspace.py over the Windows XP version. Given the similar increase in the results for existing.py however, this is likely due to an increase in the DLL files required for the Windows 7 version.

While for Internet Explorer and Notepad the identified allocations both improved across versions of Windows, the Windows 7 version of Calculator demonstrated a significant drop. While like Notepad, the visual theme of Calculator was updated for Windows 7, unlike Notepad it also introduced new functionality. Since none of this new functionality was used when taking the memory images however, it is unlikely that this is the cause of the increase in unexplained allocations. Instead, this would suggest that some alternate method is being used to allocate or manage memory, for which no metadata has yet been identified.

Being able to accurately describe the contents and roles of user allocations allows the significant reduction of the search space when looking for specific information. Instead of analysing the whole user space, only a small subset now requires analysis. Depending on the information being searched for, this subset could range from a few allocations right down to a specific data structure.

In addition, it facilitates the analysis of data not previously possible. For example, some newly identified data structure could be used to identify some artefact previously overlooked, or the shared memory sections could reveal information about process or driver communication that could prove useful in investigations.

6.1. Limitations

While the tool is capable of providing information about the contents of a high percentage of user allocations, it can not explain every possible allocation. With the increase in the complexity of the application, it can be seen that the completeness of the descriptions offered by the plugin decreases. Since it was not clear how the increased complexity of the program was affecting the results, a small experiment focussing on *notepad.exe* was performed.

Program	OS	User Allocations	Existing.py	Userspace.py	Percentage
malloc.exe	XP	22.0	11.0	22.0	100%
calc.exe	XP	52.0	28.0	46.4	89.2%
notepad.exe	XP	94.8	52.0	78.6	82.9%
iexplorer.exe	XP	149.4	79.6	111.0	74.3%
malloc.exe	Win7	19.0	9.0	19.0	100%
calc.exe	Win7	118.4	42.4	67.4	56.9%
notepad.exe	Win7	187.0	134.8	173.8	92.9%
iexplorer.exe	Win7	267.8	146.4	201.6	75.3%

Fig. 6. Allocations identified by existing.py and userspace.py.

For each memory image, as before, notepad was opened and a paragraph of text was entered. This time however, a memory capture was taken prior to saving the file, then another memory capture taken after the file had been saved. The results of this experiment can be seen in Fig. 7. As can be seen from the results, saving the file caused significantly more user allocations to be made, particularly on Windows 7. After saving the file, in terms of known user allocations, more files were mapped, more threads created and one additional section was allocated when compared to before saving. Numerous more unknown user allocations were also created. In terms of functionality, saving required significantly more GUI activity, such as selecting where on disk to save the file, and device driver interaction, in order to save the file to the disk.

When comparing these results to *malloc.exe*, which as a terminal application has no GUI component, it can be seen that increased GUI activity is one reason for the decrease in completeness of the results. Manual analysis of unknown allocations also confirms this, as GUI related terms such as *Combobox* appear in many of them.

Aside from concerns about the location of GUI related information, there were also other potential causes of unexplained user allocations.

As with any sort of memory analysis, paging is an issue. Since Volatility does not support accessing an accompying page file, this means that any data that has been paged to disk is inaccessible. While the majority of the metadata sources will always remain in memory, there are some sources, such as TEBs and heaps, that can be paged to disk, preventing the extraction of their related metadata. Combining the page file with the memory image would allow this problem to be overcome, but would likely introduce new inconsistency issues between the state of the two data sources.

One potential issue could the fact that data structures are being captured while they are being modified. Although virtual machines were being used, which are capable of taking a snapshot of memory at an instantaneous point in time, it is likely that at this point, some part of memory was in the process of being freed. Since there are potentially many data structures that require modification to remove a single user allocation, the lack of information could potentially be the result of a lack of consistency between these data structures. For example, although rare, instances of control areas were found where the segment pointer did not point to a segment but pointed to some other unknown data structure. The same kind of inconsistency could potentially occur with the virtual to physical address translation process. While these sorts of inconsistencies can be detected through sanity checks of the object references, there is not enough redundancy in the metadata to allow their correction.

A more likely cause of these unexplained user allocations however is the use of some undocumented Windows API that has not accurately been accounted for. This could mean that metadata about these allocations exist, but they simply reside in a data structure that has yet to be understood, or the link to the data structure that provides the metadata has not been found.

One example whereby the metadata exists but is referenced by an unknown data structure is with mapped allocations. Although the vast majority of these mapped allocations can be explained through the use of a file object or section object as described in Section 3.1, there exist VAD structures, which point to a valid pair of control area and segment objects, for which no file or section objects exist. This could potentially be caused by two factors, either a type of object that has not been accounted for is referencing the segment, or an unknown source of section objects has not been examined. More research into the internals of the Windows memory manager would be required to overcome this issue.

7. Related work

The majority of previous research in memory forensics has focused on describing the kernel, and few contributions have been made that further the ability to describe the contents of user space.

Memparser (Betz, 2005) was one of the first memory analysis tools, and was capable of retrieving some user space information, such as the modules loaded and process parameters by using the PEB. Although the tool was capable of dumping this information, it provided no further user space analysis except for dumping the accessible pages.

Dolan-Gavitt (2007) was the first to allow the exploration of the user space by retrieving the VADs from a memory image. Aside from describing the VAD data structures and how to determine if a VAD represented a file, the analysis of these VADs was taken no further.

Arasteh and Debbabi (2007) used the user and kernel stacks to recreate the execution history of a process. Although they provide a method of locating the user stack of a thread, the location of other user space data structures was not in the scope of their research. However, they do present an alternative method of finding the executable of a process, locating the executable file through the use of a section object.

Other research, while not expanding on the ability to describe the user space, has focused on analysing the user space without explicit knowledge of the data structures involved.

Hejazi et al. (2009) analysed API call traces on the stack to locate useful data structures, such as those dealing with

Program	OS	User Allocations	Existing.py	Userspace.py	Percentage
Notepad.exe pre-save	XP	54.0	29.0	48.6	90.0%
Notepad.exe post-save	XP	93.8	52.0	77.6	82.9%
Notepad.exe pre-save	Win7	54.0	33.0	51.6	95.6%
Notepad.exe post-save	Win7	187.0	134.8	173.8	92.9%

Fig. 7. Analysis of the effect of saving on the results.

encryption. This however was performed without any knowledge of how data was stored in the user space, limiting their ability to retrieve user data without significant reverse engineering of the API call.

Specific applications have also been analysed, such as Skype (Simon and Slay, 2010) and Pidgin (Simon and Slay, 2011) for data of interest, such as chat logs. One key process in the analysis of Pidgin was the comparison of how individual pages within an address space changed over time. The use of VAD allocations could have significantly improved this process, allowing the removal of irrelevant pages such as those related to known DLL files. Given the time required to analyse individual applications however, this approach does not scale to the huge range of potential applications that a target computer could be running.

To foster the development of memory analysis, numerous toolkits have been developed over the years (Okolica and Peterson, 2010; Petroni et al., 2006; Walters and Petroni, 2007). The Volatility Framework (Volatile Systems, 2011) however, has become the de facto standard for memory analysis, with its extensive plugin support allowing the simple creation of new analysis tools. Numerous of the previously mentioned techniques have already been reimplemented as plugins for Volatility, and it is for this reason that we chose to implement our work as a Volatility plugin.

8. Conclusion

This paper has presented an approach to determine the contents and roles of user allocations. The user allocations described by the VAD tree have been shown to be a reliable framework upon which to describe the user space, and a detailed analysis of how various sources of metadata can be used to further describe these user allocations has been shown. A sample implementation of this approach was then provided, the results of which were validated and then demonstrated to be able to consistently describe the contents and roles of a high percentage of user allocations even when dealing with complex applications.

8.1. Future work

While the provided plugin is capable of describing the majority of the user allocations of a process, it is not capable of describing all of them. More research is required to locate the reasons for these unexplained user allocations, and update the plugin accordingly. Analysis of the susceptibility of these techniques to malicious modification is also required.

In addition, the presented work needs to be extended to support Windows Vista and the soon to be released Windows 8, as well as the 64-bit versions of the currently supported operating systems. This will involve the location of any new artefacts, as well as the verification that the existing artefacts have remained the same. The plugin can then be updated to account for these additional versions of Windows and validated against them.

References

Arasteh AR, Debbabi M. Forensic memory analysis: from stack and code to execution history. Digital Investigation 2007;4(S1):114–25.

Betz C. MemParser, http://www.dfrws.org/2005/challenge/memparser.shtml; 2005.

Dolan-Gavitt B. The VAD tree: a process-eye view of physical memory. Digital Investigation 2007;4(S1):62–4.

Hejazi SM, Talhi C, Debbabi M. Extraction of forensically sensitive information from windows physical memory. Digital Investigation 2009; 6(S1):121–31.

Microsoft. Debugging tools for Windows 32-bit version, http://msdn.microsoft.com/en-us/windows/hardware/gg463016; 2010.

Nasarre C. Detect and Plug GDI Leaks in your code with two powerful tools for Windows XP, http://msdn.microsoft.com/en-au/magazine/cc188782.aspx; 2003.

Okolica J, Peterson GL. Windows operating systems agnostic memory analysis. Digital Investigation 2010;7:48–56.

Petroni NL, Walters A, Fraser T, Arbaugh WA. FATKit: a framework for the extraction and analysis of digital forensic data from volatile system memory. Digital Investigation 2006;3(4):197–210.

Russinovich M. Windows sysinterals, http://technet.microsoft.com/en-us/sysinternals; 2012.

Russinovich ME, Solomon DA. Microsoft Windows internals. 4th ed. Redmond, Washington: Microsoft Press; 2005.

Simon M, Slay J. Recovery of Skype application activity data from physical memory. In: Proceedings of the 5th International Conference on Availability, Reliability, and Security; 2010. p. 283–8.

Simon MP, Slay J. Recovery of pidgin chat communication artefacts from physical memory: a pilot test to determine feasibility. In: Proceedings of the Sixth International Conference on Availability, Reliability and Security. IEEE; 2011. p. 183–8.

Volatile Systems. The volatility framework: volatile memory artifact extraction utility framework, https://www.volatilesystems.com/default/volatility; 2011.

Walters A, Petroni N. Volatools: integrating volatile memory forensics into the digital investigation process. Black Hat DC; 2007.

Andrew White is currently a full-time PhD Student at the Information Security Institute within the Queensland University of Technology, located in Brisbane, Australia. His research primarily focuses on memory analysis techniques, and how they can be utilised to find evidence of malware.

Dr. Bradley Schatz is the director of the independent digital forensics consultancy Schatz Forensic, and an adjunct Associate Professor at the Queensland University of Technology (QUT). Dr. Schatz' forensic practice provides forensic services primarily to the legal sector, where his advice is sought in relation to matters ranging from intellectual property theft to computer intrusions. Bradley's research currently focuses on digital evidence in control systems, enterprise environments, and volatile memory.

Dr. Ernest Foo is an active researcher in the area of information and network security. Dr. Foo has worked extensively in the field of electronic commerce protocols investigating secure protocols for electronic tendering and electronic contracting in the Australian construction industry. Dr. Foo has broad interests having published in the area of formal analysis of privacy and identity management protocols as well as proposing secure reputation systems for wireless sensor networks. Recently Dr. Foo has been looking into research in the area of secure SCADA systems and memory forensics.