Metasploit: Reconstructing the Scene of the Crime

Metasploit: Reconstructing the Scene of the Crime, is a concept designed to demonstrate how memory analysis and acquisition can be combined with existing knowledge of tools to reveal data not previously easily accessed. The existing knowledge in this case is that of how exploit payloads, specifically Meterpreter, look in memory post-exploitation and after acquisition.

This paper discusses how to combine Memoryze ™[1], a freely available memory analysis and acquisition tool, and Metasploit forensic framework, an open source forensic framework designed to process acquired data from Memoryze, to reconstruct a Meterpreter session.

There are three things the reader needs to understand to make the concept of reconstructing a Meterpreter session a reality: i) access to physical memory ii) process acquisition iii) Metasploit payloads and Meterpreter

Accessing Physical Memory:

As a memory analysis tool Memoryze needs access to physical memory for two purposes: memory acquisition and memory analysis.

1. Memory acquisition is when Memoryze creates a complete image of the current state of physical memory or the current state of a specific process.
2. Memory Analysis is when Memoryze accesses physical memory with the intention of performing analysis, such as enumerating all processes and their handle tables.

To make accessing physical memory easy, Windows exposes a section object named \Device\PhysicalMemory. This section object can be opened and once opened an application will have a handle to physical memory. If an application reads from this handle, it is reading from physical memory. It is important to note that an application could access this section object in ring 3 from Windows 2000 thru Windows 2003 SP0. Windows 2003 SP1 and later changed the permissions on this section object and now require the application to be running at ring 0[2].

Understanding Windows Process Address Space

Once Memoryze has access to physical memory it has access to every processes' virtual address space. The address space can grow and shrink as memory is needed for things such as threads, dynamic storage, stacks, loaded binaries, etc. The Windows Memory Manager (MM) manages all of this information. The information is managed in a binary tree. A member variable within the Executive Process (EPROCESS) structure named VadRoot (Virtual Address Descriptor Root) is a pointer to the root of this binary tree. Every process has an EPROCESS structure that contains information the process manager needs to use to manage processes. The binary tree entries are actually structures named

[1] http://www.mandiant.com/software/memoryze.htm
[2] http://technet.microsoft.com/en-us/library/cc787565.aspx

Memory Manager Virtual Address Descriptors (MMVAD). These structures contain the information relevant to the MM such as the virtual start and the virtual size of the region of memory being described. To show the MMVAD structure and its members, type `dt` which in windbg is display type and then `_MMVAD`. It will then display the structure and all its members along with offsets and sizes of the variables, as demonstrated below (make sure symbols are loaded before you do this).

```
lkd> dt _MMVAD
          +0x000 StartingVpn      : Uint4B
          +0x004 EndingVpn        : Uint4B
          +0x008 Parent           : Ptr32 _MMVAD
          +0x00c LeftChild        : Ptr32 _MMVAD
          +0x010 RightChild       : Ptr32 _MMVAD
          +0x014 u                : __unnamed
          +0x018 ControlArea      : Ptr32 _CONTROL_AREA
          +0x01c FirstPrototypePte : Ptr32 _MMPTE
          +0x020 LastContiguousPte : Ptr32 _MMPTE
         +0x024 u2               : __unnamed
```

Figure 1:_ MMVAD Structure

As the reader can see, in Figure 1, this structure contains different variables. Memoryze is concerned with the following structures:

- StartingVpn - the virtual start address
- EndingVpn - the virtual size of the memory section
- LeftChild, RightChild - pointers to the children nodes in the tree
- Parent - a pointer to the parent node

If the MMVAD is describing a virtual address section that was created via a call to LoadLibrary or by the Windows process loader, that memory section will have a name. The name will be the actual name of the file stored in the virtual section described by the given MMVAD. To get the name or check if it exists, Memoryze references the ControlArea member variable which is a pointer to CONTROL_AREA structure.

```
lkd> dt _CONTROL_AREA
   +0x000 Segment          : Ptr32 _SEGMENT
   +0x004 DereferenceList  : _LIST_ENTRY
   +0x00c NumberOfSectionReferences : Uint4B
   +0x010 NumberOfPfnReferences : Uint4B
   +0x014 NumberOfMappedViews : Uint4B
   +0x018 NumberOfSubsections : Uint2B
   +0x01a FlushInProgressCount : Uint2B
   +0x01c NumberOfUserReferences : Uint4B
   +0x020 u                : __unnamed
   +0x024 FilePointer      : Ptr32 _FILE_OBJECT
   +0x028 WaitingForDeletion : Ptr32 _EVENT_COUNTER
   +0x02c ModifiedWriteCount : Uint2B
   +0x02e NumberOfSystemCacheViews : Uint2B
```

Figure 2: CONTROL_AREA Structure

Within the CONTROL_AREA structure is another pointer called FilePointer that points to a FILE_OBJECT structure. The FILE_OBJECT structure, if valid (there are extensive checks done but outside the scope of this paper), will contain a UNICODE_STRING buffer in the member FileName that contains a pointer to the actual file path.

```
lkd> dt _FILE_OBJECT
   +0x000 Type             : Int2B
   +0x002 Size             : Int2B
   +0x004 DeviceObject     : Ptr32 _DEVICE_OBJECT
   +0x008 Vpb              : Ptr32 _VPB
   +0x00c FsContext        : Ptr32 Void
   +0x010 FsContext2       : Ptr32 Void
[…]
   +0x030 FileName         : _UNICODE_STRING
[…]
   +0x06c CompletionContext : Ptr32 _IO_COMPLETION_CONTEXT
```

It is very important to note that any MMVAD that does not have a name but whose contents contain an MZ/PE header can safely be considered injected or malicious.

By enumerating this binary tree of MMVAD entries, Memoryze can access the processes' heap(s), stack(s), executables, and DLLs. This gives a very complete and uninhibited view of the process address space without "tampering" or "touching" the process and utilizes no API calls. To help illustrate how the Windows memory manager uses the VAD tree, take a look at Figure 3, which is a screen of the memory map view from OllyDbg. The VAD tree is very similar to this view. Each virtual address in the address column would be its own entry in the VAD tree.

Address	Size	Owner	Section	Contains	Type	Access	Initial
00010000	00001000				Priv	RW	RW
00020000	00001000				Priv	RW	RW
00030000	00001000				Priv	RW	RW
0007B000	00001000				Priv	RW Gua	RW
0007C000	00004000			stack of ma	Priv	RW Gua	RW
00080000	00003000				Map	R	R
00090000	00002000				Map	R	R
000A0000	00010000				Priv	RW	RW
001A0000	00006000				Priv	RW	RW
001B0000	00003000				Map	RW	RW

Figure 3: OllyDbg Memory Map View

Process Acquisition

Now that the reader has an understanding of how a processes' virtual address space is laid out, understanding process acquisition is very straightforward. Recall that it all starts by identifying processes in memory. This is done by finding the EPROCESS structure and determining if the given EPROCESS structure matches the process the user has requested to acquire. Once a match is found, the VadRoot pointer is referenced and the binary tree is walked. Each MMVAD entry encountered is written to disk in a DD style format with no alterations. DD stands for disk duplicator, and was originally used to duplicate drives. The format is very simple: a buffer is read and then written to disk, if the buffer can't be read (in the case of a page being paged to disk and not the paging file) then a buffer of zeros, equal to the size of the attempted read would be written to disk to ensure the out-file is equal in size to the in-file. In this case the out-file would be an acquired VAD and the in-file would be the virtual section in memory. If the VAD has a name in its CONTROL_AREA structure, it will be given that name. Otherwise it is named after its virtual start and size. Memoryze is able to acquire a specific process' address space because it has a built-in ability to translate virtual addresses to physical addresses. Every virtual address encountered when performing the acquisition is translated to its physical memory address. The physical memory is then read into a buffer and written to disk. If the physical memory is marked as paged out, then Memoryze will parse the paging file(s) to determine if the page was paged to the paging file(s). If it is in the paging file(s) it will be read and then written to disk. The ability to utilize the paging file(s) during process acquisition gives a better and more complete picture of the processes' address space. Process acquisition through physical memory has a number of benefits, users can:

- Bypass any anti-debug routines a process is using to protect itself. By not attaching to a process to read the processes' virtual memory, the protected process has no idea it is being "acquired."
- Bypass debug register (DR) rootkits. DR rootkits work by setting traps on certain virtual addresses they don't want read, such as the virtual address of a hook. If you can't read the virtual address to check for the hook you don't know it's hooked. Memoryze only utilizes physical memory, therefore completely bypassing any DR.
- Captures a processes communication strings potentially pre- and post-encryption.
- C overcome most packing because the process is acquiring in an unpacked state.

- Acquire DLLs injected in memory that never touch disk.

Metasploit

Metasploit is an exploit framework originally developed by HD Moore in 2003[3]. Since conception it has grown exponentially in user base and developers. Metasploit provides penetration testers and researchers an advanced framework for binary exploitation, payloads and post-exploitation payloads[4].

Metasploit consists of developer and community contributed exploits for an array of operating systems, commercial and open source software. An exploit can be delivered in many forms, though the end goal remains clear: control program execution flow.

There are many different attack vectors in which to take advantage of bugs, whether, it be in form of local buffer overflows, heap corruption, integer overflows, format string vulnerabilities, etc. These can take place in the form of local or remote attacks, depending on the vulnerable software and its function. Each exploit in Metasploit's repository comes with a plethora of payload options. Payloads are the actual machine code to be executed on the victim machine. A payload can be as simple as adding a user to the system or as advanced as injecting a VNC server into the victim processes. Metasploit's value to the community is its continuation of advanced payloads. Metasploit provides a user a simple interface to execute powerful exploits with even more advanced payloads, all with little to no user knowledge of how the machine is being compromised. The ease of use, documentation and resources for a user means that it is likely some attackers will utilize Metasploit to compromise a network. One of the most popular payloads used by attackers is Meterpreter.

The Meterpreter (short for Meta Interpreter) payload will give an attacker a presence in memory only, and reduces the attackers need to touch disk to zero. Metasploit will upload a DLL (Meterpreter) to the remote host; the uploaded DLL will be stored in the compromised processes' heap. Traditionally an uploaded DLL would be written to disk. This is because LoadLibrary, which loads modules, will only load modules from disk or from a network share. To get around this, Metasploit hooks the underlying API calls that LoadLibrary makes. These API calls are:

- `NtMapViewOfSection`
- `NtQueryAttributesFile`
- `NtOpenFile`
- `NtCreateSection`
- `NtOpenSection`

These hooks allow Meterpreter to be loaded from memory and not from disk. Meterpreter once loaded offers the attacker a plethora of options. The community has also added custom scripts to expand Meterpreter's capability. [5]j

[3] http://www.metasploit.com/
[4] http://metasploit.com:55555/PAYLOADS
[5] http://hick.org/code/skape/papers/remote-library-injection.pdf

Meterpreter

Once Meterpreter's staged shellcode has been executed and Meterpreter has been loaded, communication begins. Meterpreter's communication and extensibility are what makes it so valuable to an advanced attacker. For the purposes of this paper think about the attacker as the client, and the victim as the server. Meterpreter's protocol is well documented[6] by HD Moore and Skape. This paper touches on the finer points of the protocol that will eventually be leveraged to reconstruct communication. Meterpreter uses a protocol called Type Length Value (TLV). Type and length are 4 bytes and the value is N bytes. One caveat of Meterpreter usage of the traditional TLV protocol is it flips the TL making a length type value protocol, however as in the Meterpreter documentation the authors will continue to refer to the protocol as TLV.

The client will send a request to the server specifying a type. This tells the server how to process the request, the length and the value, all of which help the server perform some request. A response is formed using the same principles of TLV: the response has a type a length and finally a value. The value can be another TLV. The nesting of TLVs allows for dynamic responses and representation of complex data structures.

The easiest way to understand how a client would make a request to a server is in an example. Let us say the client wants to get the process id of the currently exploited process. The client would form a request with a type of PACKET_TYPE_REQUEST, the value would be of `stdapi_sys_process_getpid`. This string represents a method exposed by the server. By specifying this method once the server processes the request, the server will call this method and respond with the results. This is somewhat similar to how RPC works, where a client would specify an opcode (function number) to call on the server and get the results marshaled backed.

Once the server receives a request, it looks up the method in its table. The function table has entries that look like this:

```
{ "stdapi_sys_process_getpid",
  { request_sys_process_getpid,                    { 0 }, 0 },
  { EMPTY_DISPATCH_HANDLER                                  },
},
```

The information in this array represents the method name and a pointer to the function that should be called if such a method is requested. In this case the client wants `request_sys_process_getpid` to be called. The server will execute this function and respond with the results. The resulting response is where the interest of a forensic investigator would lie. Let's take a look at the response of a `stdapi_sys_process_getpid` request. The result is a complex TLV that looks like:

[6] http://www.nologin.org/Downloads/Papers/meterpreter.pdf

TLV Header	
Length	sizeof(TLV Header)
Type	PACKET_TLV_TYPE_PLAIN_RESPONSE
Value	Length: Sizeof(Value) Type: TLV_TYPE_METHOD 0x00010001 Value: stdapi_sys_process_getpid Length: 41 Type: TLV_TYPE_REQUEST_ID 0x00010002 Value: 316481384670289912891653753639 9 Length 12 Type:TLV_TYPE_PID 0x000208FC Value: 0x03EC Length: 12 Type: TLV_TYPE_RESULT Value: 0x00000000

Table 1: TLV Packet from getpid request

As the reader can see, the response value is actually three nested TLVs. This is a basic response; most responses are more complicated and get really nasty with grouped results within nested TLVs. Leveraging Meterpreter's protocol and its easy to understand structure, it is possible to reconstruct what an attacker did by parsing out the structures left over in memory.

Metasploit Forensic Framework

Metasploit Forensic Framework (MSFF) is a tool set written to aid in the post-exploitation analysis of a process. Before MSFF can be run, the analyst must acquire the process they suspect has been compromised. This is done using Memoryze and the batch scripts that accompany it. Acquisition is as easy as:

ProcessDD.bat –pid [process id]

Once the process has been acquired, MSFF can be run against the files written to disk. Remember each file represents a memory section that was loaded in the processes' address space.

How MSFF Works

MSFF works because memory that has been freed is not actually gone or erased. When memory is freed, it is marked as reusable but what was contained on those pages is not erased or zeroed out. This means that an analyst can go back and examine memory and find the freed pages.

Meterpreter packet dispatching internals indicate that it is correctly freeing its payload. Meterpreter will send the response TLV by calling `packet_transmit_response`, which calls `packet_transmit`. This function is what actually sends the response TLV back to the client. Once the response is sent, Meterpreter calls `destroy_packet`. This function frees the "payload". In our example it would free the whole response. However, freeing the memory does not mean it is gone and unreachable.

MSFF works by scanning each acquired memory section looking for known "method" strings that Meterpreter would have sent as a response while an attacker was in communication. Some common methods are:

- `priv_passwd_get_sam_hashes` – method use to tell Meterpreter to dump sam hashes
- `stdapi_sys_process_getpid` – method used to retrieve the currently exploited processes
- `core_channel_write` – method used to write data back to the Metasploit console, also utilized when an attacker does "execute –i"

There are plenty more supported by MSFF and some unsupported. MSFF works scanning each VAD for these known strings, if it finds the string it starts to parse out the TLV structure. Since the TLV structure contains types that indicate how to parse its data, MSFF leverages this information to pull out the responses. Also because Meterpreter responds with the method that called the response, MSFF can indicate what the attacker requested that resulted in the response. All this information is pulled from memory and can be used to reconstruct what occurred.

```
08 74 04 06 00 01 00 01 73 74 64 61 70 69 5F 73 ; .t......stdapi_s
79 73 5F 70 72 6F 63 65 73 73 5F 67 65 74 70 69 ; ys_process_getpi
64 00 00 00 00 29 00 01 00 02 33 31 36 34 38 31 ; d....)....316481
33 38 34 36 37 30 32 38 39 39 31 32 38 39 31 36 ; 3846702899128916
35 33 37 35 33 36 33 39 39 34 00 00 00 00 0C 00 ; 5375363994......
02 08 FC 00 00 03 EC 00 00 00 0C 00 02 00 04 00 ; ..ü...ì.........
00 00 00 01 48 05 98 01 0B 00 0E 00 C7 01 0E 00 ; ....H.˜.....Ç...
```

Figure 4: Acquired VAD from exploited processes

The final walk-through will reference Figure 4 above. Figure 4 is part of a VAD that was acquired after Meterpreter exploited a process. Recall that MSFF scans for `stdapi_sys_process_get_pid`, which is the method called. In this case the method is found. Continuing to parse this specific memory region results in retrieving the values seen in Table 1: TLV Packet from getpid request.

Caveats and Gotchas

A significant caveat and gotcha is that memory is volatile. In a heavily active system, memory will eventually be reused. Meterpreter does free its packets, but the windows memory manager functions in such a way that a free memory page is not immediately reused. This means that freed memory can be lying around a system for quite some time. There are many variables that factor into how long the memory stays in the system, but it has been observed hours after freeing.

There are some issues when acquiring memory from certain exploited process situations. Metasploit provides three different types of exit routines: SEH, process, and thread. If the attacker were to use the exit functions of SEH or process, with a browser exploit that was vulnerable to heap corruption, the entire process memory would be removed when the attacker disconnected the Meterpreter socket connection. This would prevent process memory acquisition of the browser because it has been closed. In testing the MS09-002 IE 7 uninitialized memory corruption vulnerability, if the attacker used an `EXITFUNC` variable set to the thread, the Internet Explorer process would not terminate after a Meterpreter socket close, it would just exit the thread used during the exploit. However, this would render the functionality of Internet Explorer unusable to any end user on the exploited host. Not all exploits kill the currently exploited process. As seen in MS08-067, the memory acquisition of the exploited svchost.exe process had Meterpreter data present hours after the Meterpreter client disconnected from the server.

Conclusion

The techniques and research discussed here are just a starting point for memory forensics. Specifically how memory forensics can better start utilizing the discovery of artifact memory. The fact that the windows memory manager functions in such a way that freed memory pages are not immediately reclaimed is useful to a forensic analyst. This behavior also resembles traditional file system forensics. When a file is deleted it is unlinked from its table but not initially overwritten, allowing the analyst to recover deleted files days after deletion. This is a first step in demonstrating that memory forensics has a true place in a forensic analysts' toolkit. Further research needs to be done to better know the limits of freed memory.