# **Unpacking Dexter POS "Memory Dump Parsing" Malware**

volatility-labs.blogspot.com/2012/12/unpacking-dexter-pos-memory-dump.html

Functions window 🗗 🗙 [	IDA View-A 🛛 🖸 Hex View-A	🛛 🗚 Structures 🛛 🗄 Enums 🔍	
Function name	nublic start		
<u>f</u> LoadMemberData	start proc near	Imports, Exports, Program Segmentation	8
F RenameCommand(IRootSt			
f RenameFortation(IRootSto	var_418= byte ptr -418h	Mail Imports	₽ ×
f RenameHerbal(IRootStorag	var_18- byce ptr -180 var 14= dword ntr -14b	Address Ordinal Name	Library
F RenameLoadMac(IRootSto	var 10= dword ptr -10h	00411410 GetKeyboardState	USER32
F RenameOptimize(IRootSto	var_C= dword ptr -0Ch	00411418 GetSystemWindowsDirectoryW	KERNEL32
F Rename lest(IRootStorage	var_8= dword ptr -8		
J start	var_4= dword ptr -4		4
f sub 401038	nush ehn	Line1 of 2	
f sub 401586	mov ebp, esp	P Exports	8×
f sub 401848	sub esp, 418h	Name Address	Ordinal
f sub 401F85	push 1C2h	LoadMemberData 00401696	5 1
f sub 402174	lea eax, [ebp+var_418]	RenameCommand(IRootStorage *.HUMPD *) 0040182F	F 2
f sub 402D03	call dword ptr ds:#11#18b	RenameFortation(IRootStorage *, HUMPD *) 0040164E	£ 3
f sub_4034AC	mov [ebp+var 14], 68FBE69Ch	RenameHerbal(IRootStorage *, HUMPD_*) 00401680	) 4
	lea eax, [ebp+var_18]	RenameLoadMac(IRootStorage *,HUMPD_ *) 00401000	) 5
	mov [ebp+var_C], 68FBE69Dh	RenameOptimize(IRootStorage *, HUMPD_ *) 00401667	/ 6
	mov [ebp+var_10], eax	RenameTest(IRootStorage *, HUMPD_ *) 00401816	i 7
	lea edx, [ebp+4]	📝 start 004016A	F
	mou dword 400340 eax		
	mov eax, dword 40A340	<	+
	mov eax, [eax+4]	Line 8 of 8	
	mov dword_40A334, eax		
	mov eax, dword_40A340	Program Segmentation	e x
	mov eax, [eax+8]	Name Start End R	W X D 🔺
	mov eax, dword 40A34A	data 0040A000 0040B000 R	w
	mov eax, [eax+0Ch]	.ts050 0040B000 0040C000 R	
👬 Graph overview 🗗 🗙	mov dword_40A33C, eax	.ts040 0040C000 0040D000 R	
	մեններ հոսոր հետ		=

I'm a big fan of Dexter. As I recently mentioned during an impromptu discussion with our <u>first</u> <u>group of memory analysis training attendees</u>, if there are only a few minutes left in an episode and he hasn't killed anyone yet, I start getting nervous. So when I heard there's malware named dexter that has also been "<u>parsing memory dumps</u>" of specific processes on POS (Point of Sale) systems, I was excited to take a look. How exactly does this memory dump parsing occur? Is it scanning for .vmem files on an infected VM host? Maybe walking directories of network shares to find collections of past memory dumps taken by forensic teams? Perhaps acquiring a crash dump or mini-dump of the POS system itself? Turns out its none of the above, and the memory dump parsing is just a <u>ReadProcessMemory</u> loop, but figuring that out was nonetheless a textbook example of how to use Volatility in a reverse-engineering malware scenario.

Getting started in the typical way, you can see dexter is packed. There are PE sections named .conas, .ts10, .ts20, .ts30, .ts40, and .ts50; suspiciously named exports like RenameHerbal, RenameFortation, and LoadMemberData; only two imported APIs - GetKeyboardState and GetSystemWindowsDirectoryW; and roughly 10% of the file is recognized by IDA as executable code (the rest is compressed/packed data).



If you needed further proof, you could check the strings:

### \$ strings -a ~/Desktop/dexter.exe

!This program cannot be run in DOS mode. IRich,

.text `.conas .const @.data .ts050 @.ts040 @.ts030 @.ts020 @.ts010 iopiio worG uNqkObyOqdrSDunixUVSmOFucsNpJUJKkmpmqIUW FvlLutksfHVJWIzigOJfTfFRxxUmwtdRKhmgjhdiXISq TZJ\_QaVg\_vGB OWMu\_wWH\_EHz SOU\_GTUQ PSOsqo\_Jk GetKeyboardState USER32.dll GetSystemWindowsDirectoryW KERNEL32.dll C:\Debugger.fgh

,vr1 rnyCsipvZnUURpjurWxiRqgauylOKfl3J owz{ tjpudajfQwdBCBGAtjpcrTlenAyHMz nuymGmpBownDvVIErgffsrBxQskLJu zn|c p}mOPSJqtFxbQlmrSPiThjdwfHxndtrP ModuleReplace.exe LoadMemberData

Nothing too interesting there. If we're going to understand how this malware parses memory dumps, we'll need to unpack it first. There's the manual option of finding OEP, dumping a sample with OllyDbg or LordPE, and fixing imports with ImpREC (or something similar), but I try to save that more time consuming and technical approach for when its really needed. In the case of dexter, and a majority of malware these days, all you need to do is run it and let it unpack itself. Being lazy never felt so good!

After copying the malware to a VM, it was executed and resulted in the creation of two new Internet Explorer processes. The code has to persist on the system in some way, so if the process (dexter.exe) doesn't stay running itself, you can bet it dissolves (i.e. injects) into another process. A reasonable first guess of the targets would be the two new IE instances: pids 1480 and 820.

mscorsvw.exe	1600			1.06 MB	NT AUTHORITY(SYSTEM	.NET Runtime
writeoisd.exe	1712			8.42 MB	NT AUTHORITY/SYSTEM	WMware Tools
TPAutoConn	1976	9.23		1.25 MB	NT AUTHORITY/SYSTEM	ThinPrint Auto
TPAutoC	732			1,024 kB	LORN-FASAF2F7211Larry	ThinPrint Auto
WPFFontCa	1104	4.62		772 kB	NT AU/LOCAL SERVICE	мpffontcache
🗖 alg.exe	1648			1.05 MB	NT AU/LOCAL SERVICE	Application La
🚽 msiexec.exe	3588			4.51 MB	NT AUTHORITY/SYSTEM	Windows@ ind
🗖 Isass.exe	696			3.59 MB	NT AUTHORITY/SYSTEM	LSA Shell (Exp
T DPCs				0		
Interrupts		1.54		0		
😑 😼 explorer.exe	424			8.95 MB	LORN-FASAF2F721(Larry	Windows Expl
rundl32.exe	496			2.13 MB	LORN-FASAF2F7211Larry	Run a DLL as
vmtoolsd.exe	504		684 B/s	8.56 MB	LORN-FASAF2F7211,Larry	W4ware Tools
ProcessHacker.exe	2532			1.68 MB	LORN-FASAF2F721(Larry	Process Hacke
E DPLORE.EXE	1460	76.92		2.39 MB	LORN-FASAF2F721 Larry	Internet Explo
EXPLORE.EXE	820		1	1.59 MB	LORN-FASAF2F721(Larry	Internet Explo
	<		U.	11		>

Now back in Volatility, working with the suspended VMs memory file, let's list processes just to orient ourselves with this new perspective:

## \$ ./vol.py pslist

Volatile Systems Volatility Framework 2.3_alpha								
Offset(V) Name	PID PP	r Di	hds	Hnds Start				
0x81bcc830 System	4	0	59	190				
0x81b27020 smss.exe	380	4	3	21 2012-12-03 (	05:35:49			
0x81a39660 csrss.exe	604	380	11	407 2012-12-03	3 05:35:51			

0x81710da0 IEXPL ORE EX	F	820	1480	2	30 2012-12	2-12 01-49-21
0x81b27558 IEXPLORE.EX	E 1	1480	968	7	115 2012-12	2-12 01:49:21
0x81783020 ProcessHacker	e 25	i32 4	424	3	79 2012-12-7	12 01:49:12
<snip></snip>						
0x81afd458 vmacthlp.exe	848	684	· 1	24	2012-12-03 (	05:35:54
0x81889150 lsass.exe	696	640	20	353	2012-12-03 (	)5:35:53
0x818e62a0 services.exe	684	640	15	287	2012-12-03	05:35:53
0x818fbd78 winlogon.exe	640	380	18	506	2012-12-03	05:35:53

The next thing I did since code injection was suspected is run <u>malfind</u> on the two IE pids. It located two memory segments - one in each IE process, same base address in both (0x150000), same protection (PAGE\_EXECUTE\_READWRITE), and according to the hexdump there's an MZ header at the base of the region.

#### \$ ./vol.py malfind -p 1480,820

Volatile Systems Volatility Framework 2.3\_alpha

Process: IEXPLORE.EXE Pid: 1480 Address: **0x150000** Vad Tag: VadS Protection: PAGE\_EXECUTE\_READWRITE Flags: CommitCharge: 11, MemCommit: 1, PrivateMemory: 1, Protection: 6

Process: IEXPLORE.EXE Pid: 820 Address: **0x150000** Vad Tag: VadS Protection: PAGE\_EXECUTE\_READWRITE Flags: CommitCharge: 11, MemCommit: 1, PrivateMemory: 1, Protection: 6

Now that we've quite effortlessly identified where the unpacked code is hiding, let's dump it out of memory. We'll use the <u>dlldump</u> plugin for this. Although the PE at 0x15000 isn't necessarily a DLL, the dlldump plugin allows the extracting/rebuilding of any PE in process memory if you supply the --base address (which we know).

#### \$ mkdir dexter

### \$ ./vol.py dlldump -p 1480,820 --base=0x150000 -D dexter/ Volatile Systems Volatility Framework 2.3\_alpha Process(V) Name Module Base Name Result

----- -----

0x81b27558 IEXPLORE.EXE 0x000150000 UNKNOWN OK: module.1480.1b27558.150000.dll 0x81710da0 IEXPLORE.EXE 0x000150000 UNKNOWN OK: module.820.1710da0.150000.dll

For a quick understanding of how effective this approach can be in unpacking malware, take a look at the strings now:

#### \$ strings -a dexter/module.1480.1b27558.150000.dll

This program cannot be run in DOS mode. .text .data .rsrc wuauclt.exe alg.exe spoolsv.exe lsass.exe winlogon.exe csrss.exe smss.exe System explorer.exe iexplore.exe svchost.exe ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopgrstuvwxyz0123456789+/ SeDebugPrivilege NTDLL.DLL **NtQueryInformationProcess** /portal1/gateway.php 11e2540739d7fbea1ab8f9aa7a107648.com 7186343a80c6fa32811804d23765cda4.com e7dce8e4671f8f03a040d08bb08ec07a.com e7bc2d0fceee1bdfd691a80c783173b4.com 815ad1c058df1b7ba9c0998e2aa8a7b4.com 67b3dba8bc6778101892eb77249db32e.com fabcaa97871555b68aa095335975e613.com Windows 7 Windows Server R2

Windows Server 2008 Windows Vista Windows Server 2003 R2 Windows Home Server Windows Server 2003 Windows XP Professional x64 Windows XP Windows 2000 32 Bit 64 Bit http://%s%s Content-Type:application/x-www-form-urlencoded POST Mozilla/4.0(compatible; MSIE 7.0b; Windows NT 6.0) LowRiskFileTypes Software\Microsoft\Windows\CurrentVersion\Policies\Associations rpcrt4.dll gdi32.dll wininet.dll urlmon.dll shell32.dll advapi32.dll user32.dll IsWow64Process WindowsResilienceServiceMutex Software\Resilience Software Software\Microsoft\Windows\CurrentVersion\Run .DEFAULT\SOFTWARE\Microsoft\Windows\CurrentVersion\Run UpdateMutex: response= page= &ump= &opt= &unm= &cnm= &view= &spec= &query= &val= &var= DetectShutdownClass downloadupdatecheckin: scanin: uninstall

The strings output shows a list of process names, which makes sense - the Seculert Blog mentioned that it enumerates processes. You also see it references SeDebugPrivilege, likely for the ability to call OpenProcess and read/write the memory of other processes. The ABCDEF[....] is a base64 alphabet, so you can expect it to encode some or all of the data it POSTs to gateway.php on one of the randomly named .com domains. It would create the WindowsResilienceServiceMutex and make a run key in the Software\Resilience Software registry key.

To solve our real question - how does this malware parse memory dumps - we need to open the unpacked file in IDA. Its import table is already fixed up, so aside from switching the ImageBase value in the PE header so RVAs are interpreted correctly by IDA, we're done unpacking before we even really started. A quick look through the unpacked file's IAT shows it calls ReadProcessMemory, and cross-referencing that leads to one function, shown below:



What you see here is the "memory dump parsing" function. It iterates once for each active process on the system, calling OpenProcess() to obtain a handle, then using VirtualQueryEx() to determine which memory ranges are accessible to the process, and reading them into a local buffer with ReadProcessMemory(). The data is then passed off to two scanning sub-functions which do the real work of deciding which data to steal from the buffer.

In summary, though I'm slightly disappointed that the memory dump parsing function is just a ReadProcessMemory() loop, at least I didn't waste much time getting there. Unpacking the malware by leveraging Volatility was as easy as 1-2-3. Lastly, since some of our students in the Windows Memory Forensics training requested videos of common ways we use Volatility, here's an initial example in quicktime format showing the steps described in this blog: <u>http://www.mnin.org/dexter.mov.zip</u>.