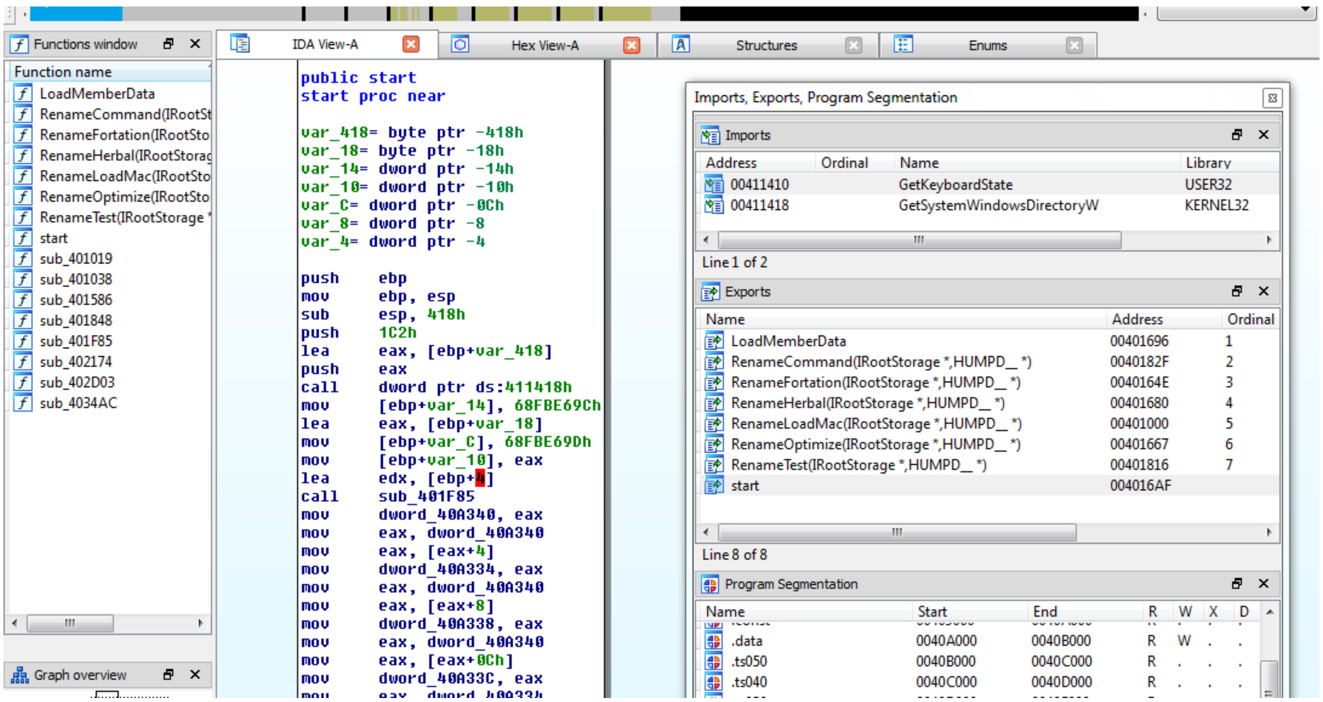
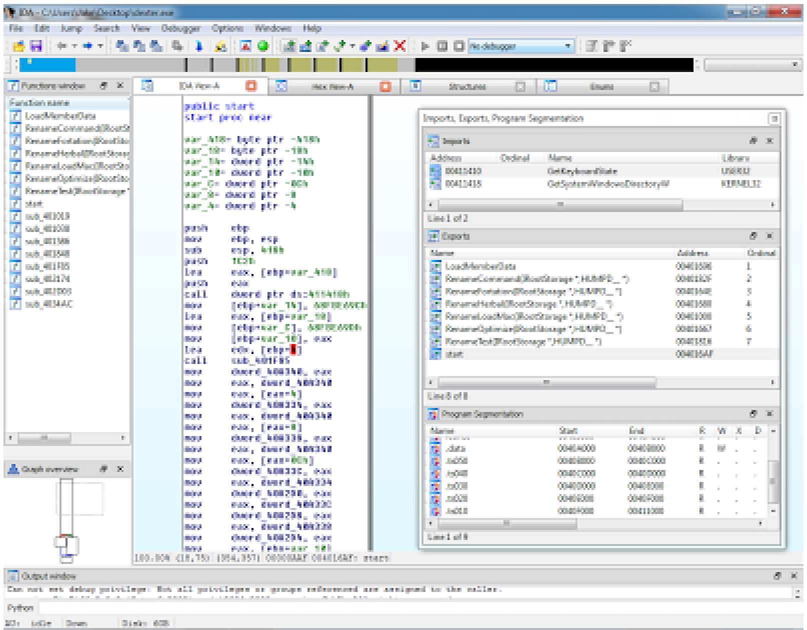
Unpacking Dexter POS "Memory Dump Parsing" Malware

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



I'm a big fan of Dexter. As I recently mentioned during an impromptu discussion with our first group of memory analysis training attendees, 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 "parsing memory dumps" 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 ReadProcessMemory 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).



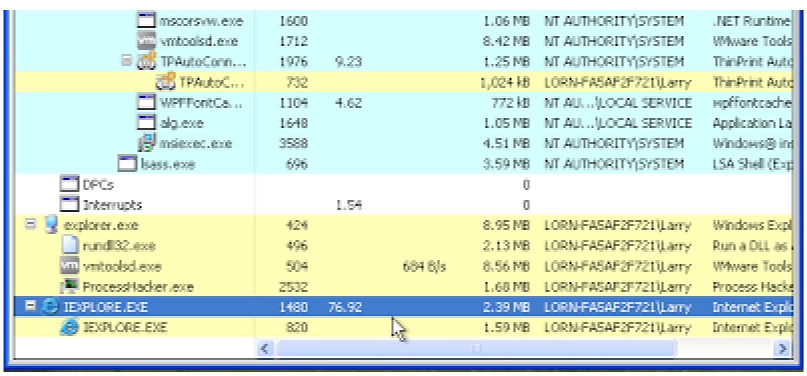
If you needed further proof, you could check the strings:

$ **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
uNqkObyOqdrSDunixUVSmOFucsNpJUJKkmpmqlUW
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.



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 PPID Thds 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 05:35:51

```
0x818fbd78 winlogon.exe        640    380     18      506  2012-12-03 05:35:53
0x818e62a0 services.exe        684    640     15      287  2012-12-03 05:35:53
0x81889150 lsass.exe           696    640     20      353  2012-12-03 05:35:53
0x81afd458 vmacthlp.exe        848    684      1       24  2012-12-03 05:35:54
<snip>
0x81783020 ProcessHacker.e    2532    424      3       79  2012-12-12 01:49:12
0x81b27558 IEXPLORE.EXE       1480    968      7      115  2012-12-12 01:49:21

0x81710da0 IEXPLORE.EXE        820   1480      2       30  2012-12-12 01:49:21
```

The next thing I did since code injection was suspected is run malfind 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

0x00150000  4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00   MZ..............
0x00150010  b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00   ........@.......
0x00150020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x00150030  00 00 00 00 00 00 00 00 00 00 00 00 c0 00 00 00   ................

Process: IEXPLORE.EXE Pid: 820 Address: 0x150000
Vad Tag: VadS Protection: PAGE_EXECUTE_READWRITE
Flags: CommitCharge: 11, MemCommit: 1, PrivateMemory: 1, Protection: 6

0x00150000  4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00   MZ..............
0x00150010  b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00   ........@.......
0x00150020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x00150030  00 00 00 00 00 00 00 00 00 00 00 00 c0 00 00 00   ................
```

Now that we've quite effortlessly identified where the unpacked code is hiding, let's dump it out of memory. We'll use the dlldump 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:

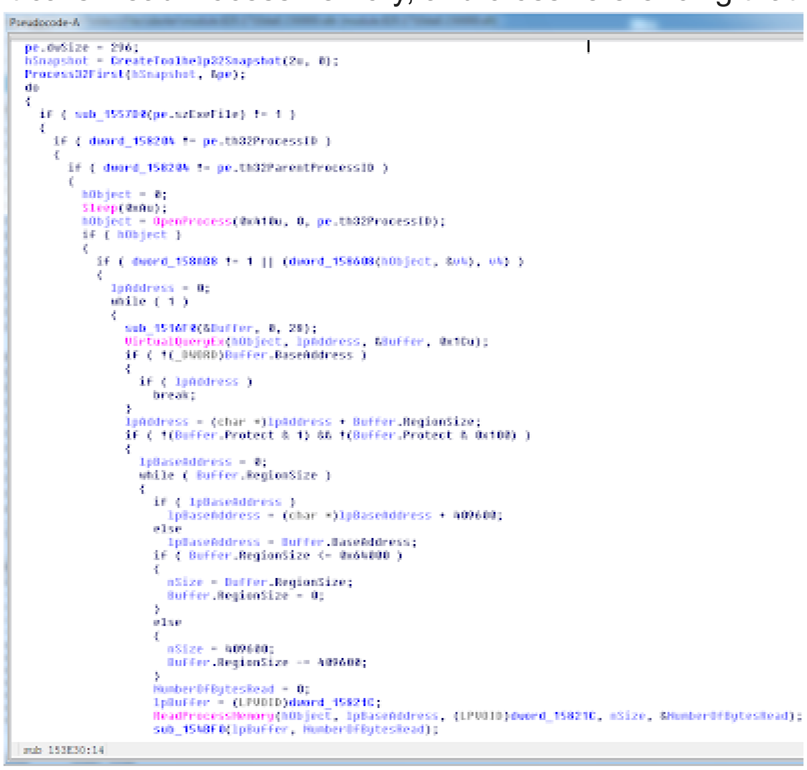
```
$ strings -a dexter/module.1480.1b27558.150000.dll
!This program cannot be run in DOS mode.
.text
.data
.rsrc
wuauclt.exe
alg.exe
spoolsv.exe
lsass.exe
winlogon.exe
csrss.exe
smss.exe
System
explorer.exe
iexplore.exe
svchost.exe
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/
SeDebugPrivilege
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
download-

update-
checkin:
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: http://www.mnin.org/dexter.mov.zip.