Hunting In Memory

April 19, 2017

Jared Atkinson, Joe Desimone
Who are we

- Jared Atkinson
  - Defensive Service Technical Director - Specter Ops
  - Microsoft Cloud and Datacenter Management MVP (PowerShell)
  - Lead Developer PowerForensics

- Joe Desimone
  - Senior Malware Researcher - Endgame
  - Developer, Hunter, Reverse Engineer
Overview

- Why hunting in memory is important
- Memory based attacker techniques
- Existing tools and approaches
- New powershell tool for hunting at scale
Importance of Memory Hunting

- Memory resident malware has been in use for over a decade, and is now ubiquitous
- Once a staple of ‘APT’; now commonplace for crimeware
- Designed to evade PSPs and YOU
- Great signal to noise ratio; easy button hunting
Attacker Techniques

- Classic memory/shellcode injection
- Reflective DLLs
- Memory Module
- Process and Module Hollowing
- PEB Unlinking
- Gargoyle (ROP/APCs)
Classic Injection

- OpenProcess - Grab handle to target process
- VirtualAllocEx - Allocate a new chunk of memory in target
- WriteProcessMemory - Write the shellcode/payload into target
- CreateRemoteThread - Start a new thread to execute the payload
Classic Injection - Poison Ivy

```assembly
push 40
push 3000
push dword ptr ss:[ebp+10]
push 0
push dword ptr ss:[ebp+C]
call dword ptr ds:[esi+B1]
push eax
lea edi, dword ptr ss:[ebp-4]
push edi
push dword ptr ss:[ebp+10]
push dword ptr ss:[ebp+14]
push eax
push dword ptr ss:[ebp+C]
call dword ptr ds:[esi+B5]
```

```
[esi+B1]: VirtualAllocEx
[esi+B5]: WriteProcessMemory
```
Poison Ivy

```
push 0
push dword ptr ss:[ebp-F80]
call dword ptr ds:[esi+C9] [esi+C9]:CreateRemoteThread
push eax
push dword ptr ss:[ebp-F80]
call dword ptr ds:[esi+A1] [esi+A1]:CloseHandle
pop eax
```
<table>
<thead>
<tr>
<th>Address</th>
<th>Permissions</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x960000</td>
<td>Private</td>
<td>8 kB</td>
</tr>
<tr>
<td>0x970000</td>
<td>Private</td>
<td>4 kB</td>
</tr>
<tr>
<td>0x980000</td>
<td>Private</td>
<td>4 kB</td>
</tr>
<tr>
<td>0x990000</td>
<td>Private</td>
<td>4 kB</td>
</tr>
<tr>
<td>0x9a0000</td>
<td>Private</td>
<td>4 kB</td>
</tr>
<tr>
<td>0x9b0000</td>
<td>Private</td>
<td>4 kB</td>
</tr>
<tr>
<td>0x9c0000</td>
<td>Private</td>
<td>4 kB</td>
</tr>
<tr>
<td>0x9d0000</td>
<td>Private</td>
<td>4 kB</td>
</tr>
<tr>
<td>0x9f0000</td>
<td>Private</td>
<td>4 kB</td>
</tr>
</tbody>
</table>
Poison Ivy Thread

![Capture of Stack - thread 2520]

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0  ntkrnlpa.exe!KiDeliverApc+0xb3</td>
</tr>
<tr>
<td>1  ntkrnlpa.exe!ZwYieldExecution+0x19a4</td>
</tr>
<tr>
<td>2  ntkrnlpa.exe!NtWaitForSingleObject+0x9a</td>
</tr>
<tr>
<td>3  ntkrnlpa.exe!KeReleaseInStackQueuedSpinLockFromDpc...</td>
</tr>
<tr>
<td>4  ntdll.dll!KiFastSystemCallRet (No unwind info)</td>
</tr>
<tr>
<td>5  mswsock.dll+0x5f9f (No unwind info)</td>
</tr>
<tr>
<td>6  ws2_32.dll!select+0xa7 (No unwind info)</td>
</tr>
<tr>
<td>7  0x3b05f8 (No unwind info)</td>
</tr>
<tr>
<td>8  0x9603d8 (No unwind info)</td>
</tr>
<tr>
<td>9  0x3b0519 (No unwind info)</td>
</tr>
<tr>
<td>10 kernel32.dll!GetModuleFileNameA+0xb4 (No unwind info)</td>
</tr>
</tbody>
</table>
Reflective DLL Injection

- DLL that maps itself into memory - original design and code by Steven Fewer [1]
- Handy from attacker perspective - makes for a ‘dumb’ injector
- No longer have to code in assembly (like PI)
- Very common technique (ex: meterpreter, powershell empire)
- Allocate memory, map sections, resolve imports, fixup relocations, call entry

[1] https://github.com/stephenfewer/ReflectiveDLLInjection
Meterpreter

- Classic DLL Reflection, such as meterpreter, is easy to find
Meterpreter
Memory Module

- Similar to Reflective technique, except loader does all the work [1]
- Payload DLL doesn’t need any special modifications
- Loader re-implements LoadLibrary(), but works on a buffer in memory
- Can map into local or remote process [2]
- Typical implementations avoid RWX

NetTraveler - Memory Layout

- Uses legitimate looking permissions

<table>
<thead>
<tr>
<th>Address</th>
<th>Access</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x3f0000</td>
<td>Private: Commit</td>
<td>4 kB</td>
</tr>
<tr>
<td>0x3f1000</td>
<td>Private: Commit</td>
<td>40 kB</td>
</tr>
<tr>
<td>0x3fb000</td>
<td>Private: Commit</td>
<td>8 kB</td>
</tr>
<tr>
<td>0x3fd000</td>
<td>Private: Commit</td>
<td>52 kB</td>
</tr>
<tr>
<td>0x40a000</td>
<td>Private: Reserved</td>
<td>4 kB</td>
</tr>
</tbody>
</table>
NetTraveler - Active Thread

14  winhttp.dll!WinHttpSendRequest+0x2fe6 (No unwind info)
15  winhttp.dll!WinHttpSendRequest+0x3069 (No unwind info)
16  winhttp.dll!WinHttpSendRequest+0x212 (No unwind info)
17  0x3f6e82 (No unwind info)
18  0x3f679e (No unwind info)
19  0x3f5d65 (No unwind info)
## Winnti - Memory Layout

<table>
<thead>
<tr>
<th>Address</th>
<th>Type</th>
<th>Commitment</th>
<th>Size</th>
<th>Permissions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x110000</td>
<td>Private</td>
<td></td>
<td>28kB</td>
<td>RWX</td>
</tr>
<tr>
<td>0x110000</td>
<td>Private: Commit</td>
<td></td>
<td>4kB</td>
<td>RX</td>
</tr>
<tr>
<td>0x111000</td>
<td>Private: Commit</td>
<td></td>
<td>12kB</td>
<td>RX</td>
</tr>
<tr>
<td>0x114000</td>
<td>Private: Commit</td>
<td></td>
<td>4kB</td>
<td>R</td>
</tr>
<tr>
<td>0x115000</td>
<td>Private: Commit</td>
<td></td>
<td>4kB</td>
<td>RW</td>
</tr>
<tr>
<td>0x116000</td>
<td>Private: Reserve</td>
<td></td>
<td>4kB</td>
<td></td>
</tr>
</tbody>
</table>
Winnti - Header Wipe
Process Hollowing

- Create new, suspended process
- Allocate new memory, unmap (hollow) existing code
- Write payload
- Redirect execution - SetThreadContext() and ResumeThread()
- Stealthy variants
  - Create/Map sections to avoid WriteProcessMemory
  - Modify entry point instead of SetThreadContext
DarkComet - Process Hollowing

CALL to CreateProcessA from dark.0040173E
ModuleFileName = "C:\Users\Joe\Desktop\dark.exe"
CommandLine = NULL
pProcessSecurity = NULL
pThreadSecurity = NULL
InheritHandles = FALSE
CreationFlags = CREATE_SUSPENDED
pEnvironment = NULL
CurrentDir = NULL
pStartupInfo = 0018FE5C
pProcessInfo = 0018FE4C

kernel32.CreateProcessA
### DarkComet

<table>
<thead>
<tr>
<th>Name</th>
<th>Base address</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dark.exe</td>
<td>0x400000</td>
<td>732 kB</td>
<td>WEmhu2 NXchu6</td>
</tr>
<tr>
<td>advani32.dll</td>
<td>0x76d60000</td>
<td>640 kB</td>
<td>Advanced Windows 32 bit</td>
</tr>
<tr>
<td>0x3e0000</td>
<td>Mapped: Commit</td>
<td>96 kB</td>
<td>R</td>
</tr>
<tr>
<td>0x400000</td>
<td>Private: Commit</td>
<td>732 kB</td>
<td>RWX</td>
</tr>
<tr>
<td>0x4c0000</td>
<td>Mapped: Commit</td>
<td>28 kB</td>
<td>R</td>
</tr>
</tbody>
</table>
Module Overwriting

- Up until now, all examples have led to non-image backed code executing
- Module Overwriting avoids this, making it more difficult to detect
- Flame and Careto are examples
- Map an unused module into target process
- Overwrite legitimate module with payload
- Odinaff had a similar trick but overwrote its own executable

<table>
<thead>
<tr>
<th>In Memory</th>
<th>On Disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>MajorLinkerVersion</td>
<td>MajorLinkerVersion</td>
</tr>
<tr>
<td>MinorLinkerVersion</td>
<td>MinorLinkerVersion</td>
</tr>
<tr>
<td>SizeOfCode</td>
<td>SizeOfCode</td>
</tr>
<tr>
<td>SizeOfInitializedData</td>
<td>SizeOfInitializedData</td>
</tr>
<tr>
<td>SizeOfUninitializedData</td>
<td>SizeOfUninitializedData</td>
</tr>
<tr>
<td>AddressOfEntryPoint</td>
<td>AddressOfEntryPoint</td>
</tr>
<tr>
<td>FileAlignment</td>
<td>FileAlignment</td>
</tr>
<tr>
<td>MajorOperatingSystemVersion</td>
<td>MajorOperatingSystemVersion</td>
</tr>
<tr>
<td>MinorOperatingSystemVersion</td>
<td>MinorOperatingSystemVersion</td>
</tr>
</tbody>
</table>

**In Memory**
- MajorLinkerVersion: 0A
- MinorLinkerVersion: 00
- SizeOfCode: 00001200
- SizeOfInitializedData: 00001200
- SizeOfUninitializedData: 00000000
- AddressOfEntryPoint: 00002180
- FileAlignment: 00000200
- MajorOperatingSystemVersion: 0005
- MinorOperatingSystemVersion: 0001

**On Disk**
- MajorLinkerVersion: 06
- MinorLinkerVersion: 00
- SizeOfCode: 00002000
- SizeOfInitializedData: 00006000
- SizeOfUninitializedData: 00000000
- AddressOfEntryPoint: 00001D04
- FileAlignment: 00001000
- MajorOperatingSystemVersion: 0004
- MinorOperatingSystemVersion: 0000
PEB Unlinking

- Not an in memory technique, but rather an evasion
- Hide loaded DLL from security products, admins, hunters
- HackingTeam used this technique in their RAT
- Flame also unlinked shell32.dll
- To find peb unlinking, you could compare what the Win32 API reports as ‘loaded’ versus what you find is actually loaded with VirtualQuery/GetSectionName
Gargoyle

- Technique developed by Josh Lospinoso to hide injected code from security products
- Payload lies dormant, with read only permissions
- Periodically ‘wakes up.’ Sets payload executable with an asynchronous procedure call and ROP. Permissions reverted, cycle repeats.

Available Tools

- Volatility / malfind
- GRR
- Rekall
- inVtero
Detecting Injection
w/ PowerShell
PSReflect

- PowerShell module written by Matt Graeber (@mattifestation)
  - https://github.com/mattifestation/PSReflect

- Avoids the compilation artifacts associated with P/Invoke
  - IMO the cleanest way to deal with Win32 API from PowerShell

- Library to abstract the complexities of calling Win32 functions via Reflection

- Intuitive “domain specific language” for defining enums, structs, and P/Invoke function signatures

- Must include PSReflect code in your scripts/modules
Get-InjectedThread

- Built on PSReflect

- PowerShell function to identify injected threads via detection methodology:
  - Use Windows Toolhelp API to get all threads
  - Iterate through each thread
  - Identify the thread’s Base (Memory) Address
  - Query the memory page for which the Base Address belongs to
  - Check if the memory page’s state is MEM_COMMIT
  - Check if the memory page’s type is not MEM_IMAGE

- Returns details regarding offending process and thread
  - Check memory page permissions
  - Look for unnecessary privileges or integrity level
  - Identify abnormal user tokens
Get-InjectedThread Output

- Injected Process Information
  - Process Id
  - Name
  - File Path (PEB and EPROCESS)
  - Command Line

- Thread Information
  - Thread Id
  - Unique Thread Token
  - Base Priority
  - Does thread have unique token?

- Memory Segment
  - Base Address
  - Size
  - Protection
  - State
  - Type
  - First 100 Bytes

- Token (Thread or Process)
  - Integrity Level
  - Enabled Privileges
  - SID / UserName
  - Logon Session Start Time
  - Logon Type
  - Authentication Package
Our Threat - 9002 Trojan

- Delivered as .zip file via Google Drive

- Zip archive contains one file
  - 2nd Myanmar Industrial Resource Development Symposium.exe
  - File has PowerPoint icon to trick users into opening

- Drops files upon execution
  - %USERPROFILE%\<random>\RealNetwork.exe (Legitimate Application)
  - %USERPROFILE%\<random>\main.dll (Loaded by MPAMedia.dll)
  - %USERPROFILE%\<random>\mpaplugins\MPAMedia.dll (DLL Side Loading)
  - Ppt
    - Opened in PowerPoint to keep up the ruse
Response

- Kill Thread
  - Stop-Thread
  - Built on Window’s TerminateThread API

- Process Minidump
  - Out-Minidump (PowerSploit)
    - https://github.com/PowerShellMafia/PowerSploit/blob/master/Exfiltration/Out-Minidump.ps1

- Thread Dump
  - Dump-Thread
```sql
Get-InjectedThread | select ProcessName, ThreadId | fl
```
Questions?