Vulpes: Obfuscating Memory Regions with Timers

mez0.cc/posts/vulpes-obfuscating-memory-regions

Table of Contents

- <u>Table of Contents</u>
- Introduction
- <u>Cleaning up</u>
- <u>Sleeping with Timers</u>
 - <u>The CORRECT region</u>
 - <u>DLLMain Bugs</u>
- <u>Conclusion</u>

Introduction

In this blog, I want to quickly document some bugs I squashed whilst playing with <u>Ekko</u> (from <u>5pider</u>). After looking into the technique, I figured it would be a cool addition to <u>Vulpes</u> and that's what I did. However, the purpose of this blog is to discuss the issues I had with:

- Reflective DLL Region Permissions
- Reflective DLL Region Tracking for Ekko to protect
- Threading in **DLLMain**
- General Cleanup

in <u>Maelstrom: Writing a C2 Implant</u>, specifically, <u>Safe Sleeping</u>, we document the background to this technique. Below is that excerpt:

On May 5th 2022, <u>Austin Hudson</u> posted a <u>tweet</u> with a blog: <u>Studying "Next Generation</u> <u>Malware" - NightHawk's Attempt At Obfuscate and Sleep</u>

This blog went through how Austin was able to identify a sample of <u>Nighthawk</u> which is a proprietary C2 from a UK-based Cyber Security Consultancy, <u>MDSec</u>. In this post, Austin discusses how the technique uses thread contexts and callbacks to flip the memory regions permissions (which we will discuss further in later posts).

For clarity, the research efforts for this technique, on behalf of <u>MDSec</u>, was <u>Peter Winter-Smith</u> and <u>modexp</u>.

I won't be detailing the technique, this blog is focusing on the aforementioned objective.

Cleaning up

In the following screenshot, <u>Vulpes</u> can be seen hanging out in memory in a RX region:

Vulpes.x64.exe (6656) Proper	ties					<	vulpes
eneral Statistics Performance	Threads Token Modules Memory Environment H	landles GPU Comment					
Hide free regions					Strings Refresh		
Base address	Туре	Size Prot Us	e	Total WS	Private WS Shareable WS Sh ^		
0x7ffd4713e000	Image: Commit	8 k8 RW C:	\Windows\System32\rpcrt4.dll	8 kB	8 kB		
0x7ffd473b2000	Image: Commit	4k8 RW C:					-
x7ffd47584000	Image: Commit	4k8 RW C:	Vulpes.x64.exe (6656) (0x1ee1000 - 0x1fbd000)			-	
0x7ffd48095000	Image: Commit	4k8 RW C:					
1x7ffd48098000	Image: Commit	36 kB R.W C:	00000000 88 8d 0d f9 4f 11 00 e9 e4 5d 01 00 0f :				
x60a000	Private: Commit	12 kB RW+G St	00000020 89 c5 85 d2 75 7a 8b 15 ec 4f 11 00 31			45 31 e4 bfuz01~^HtE1	
xa2c000	Private: Commit	12 kB RW+G St	00000040 01 00 00 00 89 15 ce 4f 11 00 48 8b 2d 1 00000060 89 e0 f0 48 0f b1 3b 48 89 c6 48 85 c0			00 ff d5 4c0HuDL 84 ef 00 00H;HHu.H.=J	
xcfc000	Private: Commit	12 kB RW+G St				C3 0f 1f 00	
x107a000	Private: Commit	12 kB RW+G St				08 31 ff 48eH%0HH.p.1.H	
x1279000	Private: Commit	12 kB RW+G St	000000c0 8b 2d 46 75 11 00 eb 18 0f 1f 84 00 00	0 00 00 48 39		00 00 ff d5Fu	
x17bc000	Private: Commit	12 kB RW+G St	000000e0 48 89 f8 f0 48 0f b1 33 48 85 c0 75 e3	81 ff 48 8b 35		84 ef 00 00 HH3Hu.1.H.5	
x 19bb000	Private: Commit	12 kB RW+G St	00000100 00 8b 06 85 c0 0f 84 a5 00 00 00 8b 06	83 f8 01 0f 84		00 00 48 8bH.	
x1bba000	Private: Commit	12 kB RW+G St	00000120 05 4b 97 0e 00 48 8b 00 48 85 c0 74 0d	1d 89 e8 ba 02		d7 4e 11 00 .KHHt.MLN	
lx1ed9000	Private: Commit	12 kB RW+G St	00000140 01 b8 01 00 00 00 48 83 c4 28 5b 5e 5f			00 48 83 c4H([^_]A\A]DH	
x22fa000	Private: Commit	12 kB RW+G St	00000160 28 5b 5e 5f 5d 41 5c 41 5d c3 66 0f 1f 00000180 00 00 48 87 33 b8 01 00 00 00 e9 01 ff			1 C/ U/ UU UU ([^_]A\A].I.D.HNJ 1 f 44 00 00 .H.3	
0x401000	Image: Commit	20 kB RX C:	000001a0 31 c0 48 87 03 e9 74 ff ff ff 66 0f 1f			0e 00 c7 06 1.HtfDHiHR	
lx 1ee 1000	Private: Commit	880 kB RX	000001c0 01 00 00 00 e8 87 58 01 00 e9 3d ff ff			0e 00 e8 6dX=f.H)Hm	
x7ffd379a1000	Image: Commit	420 KB RX C:	000001e0 58 01 00 c7 06 02 00 00 00 e9 28 ff ff	ff 66 90 b9 1f	00 00 00 e8 76 58 01 00 e9 0	ff ff ff 90 X(fvXvX	
x7ffd40e61000	Image: Commit	12 kB RX C:	00000200 41 56 41 55 41 54 56 53 48 83 ec 20 48	3b 35 8d 96 0e	00 49 89 cd 89 16 41 89 d4 4	89 c3 85 d2 AVAUATVSH H.5IAL	
x7ffd40f41000	Image: Commit	832 kB RX C:	00000220 75 5e 8b 05 f0 4d 11 00 85 c0 74 35 e8	if 9b 00 00 49	89 d8 31 d2 4c 89 e9 e8 82 6	01 00 49 89 u^Mt5.0I1.LbI.	
0x7ffd42eb1000	Image: Commit	312 kB RX C:	00000240 d8 44 89 e2 4c 89 e9 e8 44 af 00 00 49	39 d8 44 89 e2	4c 89 e9 41 89 c6 e8 b3 fd f:	ff 85 c0 75 .DLDIDLAu	

For people who are familiar with the original Reflective Loader, it <u>allocates memory as RWX</u>:

// allocate all the memory for the DLL to be loaded into. we can load at any address because we will // relocate the image. Also zeros all memory and marks it as READ, WRITE and EXECUTE

// relocate the image. Also zeros all memory and marks it as READ, WRITE and EXECUTE
to avoid any problems.

```
uiBaseAddress = (ULONG_PTR)pVirtualAlloc( NULL, ((PIMAGE_NT_HEADERS)uiHeaderValue)-
>OptionalHeader.SizeOfImage, MEM_RESERVE|MEM_COMMIT, PAGE_EXECUTE_READWRITE );
```

This is something that <u>Paranoid Ninja</u> demonstrates in <u>PE Reflection: The King is Dead</u>, <u>Long Live the King</u> and in his course: <u>Malware on Steroids</u>. From the blog, the following code is shown:

```
numberOfSections = ((PIMAGE_NT_HEADERS)pOldNtHeader)->FileHeader.NumberOfSections;
pSectionHeader = ((ULONG_PTR) & ((PIMAGE_NT_HEADERS)pOldNtHeader)->OptionalHeader +
((PIMAGE_NT_HEADERS)pOldNtHeader)->FileHeader.SizeOfOptionalHeader);
while (numberOfSections--) {
    void* thisSectionVA = (void*) (dllNewBaseAddress +
((PIMAGE_SECTION_HEADER)pSectionHeader)->VirtualAddress);
    ULONG_PTR thisSectionVirtualSize = ((PIMAGE_SECTION_HEADER)pSectionHeader)-
>Misc.VirtualSize;
    DWORD ulPermissions = 0;
    if (((PIMAGE_SECTION_HEADER)pSectionHeader)->Characteristics &
IMAGE_SCN_MEM_WRITE) {
        ulPermissions = PAGE_WRITECOPY;
    }
    if (((PIMAGE_SECTION_HEADER)pSectionHeader)->Characteristics &
IMAGE_SCN_MEM_READ) {
        ulPermissions = PAGE_READONLY;
    }
    if ((((PIMAGE_SECTION_HEADER)pSectionHeader)->Characteristics &
IMAGE_SCN_MEM_WRITE) && (((PIMAGE_SECTION_HEADER)pSectionHeader)->Characteristics &
IMAGE_SCN_MEM_READ)) {
        ulPermissions = PAGE_READWRITE;
    }
   if (((PIMAGE_SECTION_HEADER)pSectionHeader)->Characteristics &
IMAGE_SCN_MEM_EXECUTE) {
        ulPermissions = PAGE_EXECUTE;
    }
    if ((((PIMAGE_SECTION_HEADER)pSectionHeader)->Characteristics &
IMAGE_SCN_MEM_EXECUTE) && (((PIMAGE_SECTION_HEADER)pSectionHeader)->Characteristics &
IMAGE_SCN_MEM_WRITE)) {
        ulPermissions = PAGE_EXECUTE_WRITECOPY;
    }
    if ((((PIMAGE_SECTION_HEADER)pSectionHeader)->Characteristics &
IMAGE_SCN_MEM_EXECUTE) && (((PIMAGE_SECTION_HEADER)pSectionHeader)->Characteristics &
IMAGE_SCN_MEM_READ)) {
        ulPermissions = PAGE_EXECUTE_READ;
    }
    if ((((PIMAGE_SECTION_HEADER)pSectionHeader)->Characteristics &
IMAGE_SCN_MEM_EXECUTE) && (((PIMAGE_SECTION_HEADER)pSectionHeader)->Characteristics &
IMAGE_SCN_MEM_WRITE) && (((PIMAGE_SECTION_HEADER)pSectionHeader)->Characteristics &
IMAGE_SCN_MEM_READ)) {
        ulPermissions = PAGE_EXECUTE_READWRITE;
   }
    pVirtualProtect(thisSectionVA, thisSectionVirtualSize, ulPermissions,
&ulPermissions);
    pSectionHeader += sizeof(IMAGE_SECTION_HEADER);
}
```

To quote the blog:

The below screenshot shows the newly rebased PE section which does not have any RWX regions anymore, and the RX section only contains the executable code i.e. the .text section since all other remaining sections are allocated to other regions now.

This allows the Reflective DLL's .text section to be converted to RX, and that is what the screenshot earlier on was showing.

For the eagle-eyed, there was only one memory region. As this was demonstrated in <u>Malware</u> <u>on Steroids</u> and I cannot find any reference online showing how to determine which region to free. However, <u>PE Reflection: The King is Dead, Long Live the King</u> does show *how* to free it:

```
#include "badger.h"
BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD dwReason, LPVOID lpReserved)
{
    BOOL bReturnValue = TRUE;
    switch (dwReason)
    {
    case DLL_PROCESS_ATTACH: {
        struct DLL_SWEEPER *dllSweeper = (struct DLL_SWEEPER*)lpReserved;
        CHAR* newlpParam = NULL;
        task_crealloc(&newlpParam, (CHAR*)dllSweeper->lpParameter);
        VirtualFree((LPVOID)dllSweeper->lpParameter, 0, MEM_RELEASE);
        VirtualFree((LPVOID)dllSweeper->dllInitAddress, 0, MEM_RELEASE);
        badger_main(newlpParam);
        break;
    }
    case DLL_PROCESS_DETACH:
    case DLL_THREAD_ATTACH:
    case DLL_THREAD_DETACH:
        break;
    }
    return bReturnValue;
}
```

This is not something I will be showing, though.

At this point, the Reflective DLL looks okay in memory. It has one region cleaned up and freed, and then the other operating out of RX.

Sleeping with Timers

Again, we discussed <u>Ekko</u> in <u>Maelstrom: Writing a C2 Implant</u>:

Once the proof-of-concept was made public by Austin, <u>C5pider</u> then built it out into an opensource tool called <u>Ekko</u>. However, this proof-of-concept uses the base address of the entire image as the region to protect, this only works when the malware is the entire EXE on disk, or loaded as a proper DLL. This can be seen on line <u>36</u>:

```
ImageBase = GetModuleHandleA( NULL );
```

In the event that malware wants to load in the implant entirely through memory, so something like a Reflective DLL, this technique will not work as the **GetModuleHandleA** call will get the base address of the image the DLL is being loaded into. For example, say the DLL is being reflectively loaded into **calc.exe**, then the **GetModuleHandleA** will be the base of **calc.exe**.

For this to work with a proper Reflective DLL, the code needs to be changed slightly. The easiest way to redefine the function is as such:

```
VOID EkkoObf(DWORD SleepTime, DWORD64 ImageBase, DWORD ImageSize);
```

Whilst also removing the call to **GetModuleHandleA** :

```
ImageBase = GetModuleHandleA( NULL );
ImageSize = ( ( PIMAGE_NT_HEADERS ) ( ImageBase + ( ( PIMAGE_DOS_HEADER ) ImageBase
)->e_lfanew ) )->OptionalHeader.SizeOfImage;
```

The CORRECT region

The next thing is to figure out which region. Well, the region we have is the **RX** one. I spent some time debugging and <u>Paranoid Ninja</u> pointed out that it should be the rebased .text section, which is obvious in hindsight:

```
if ((((PIMAGE_SECTION_HEADER)pSectionHeader)->Characteristics &
IMAGE_SCN_MEM_EXECUTE) && (((PIMAGE_SECTION_HEADER)pSectionHeader)->Characteristics &
IMAGE_SCN_MEM_READ)) {
    ulPermissions = PAGE_EXECUTE_READ;
}
So, in my Reflective Loader:
if (dwPermissions == PAGE_EXECUTE_READ)
{
    Caller.Region = lpCurrentSection;
    Caller.Size = dwCurrentSection;
}
```

Where **Caller** is:

```
struct CALLER
{
    LPVOID Region;
    DWORD Size;
    LPVOID Release;
};
```

The struct is then passed to **DLLMain** as seen in <u>PE Reflection: The King is Dead, Long Live</u> <u>the King</u>:

```
((DLLMAIN)uiValueA)
(
    (HINSTANCE)uiBaseAddress,
    DLL_PROCESS_ATTACH,
    &Caller
);
Where DLLMain is:
BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpReserved)
{
    CALLER* Caller = { 0 };
    switch (fdwReason) {
    case DLL_PROCESS_ATTACH:
        if (lpReserved != nullptr)
        {
            Caller = (CALLER*)lpReserved;
            VirtualFree(Caller->Release, 0, MEM_RELEASE);
            StartVulpes(Caller->Region, Caller->Size);
            break;
        }
        break;
    case DLL_THREAD_ATTACH:
        break;
    case DLL_THREAD_DETACH:
        break;
    case DLL_PROCESS_DETACH:
        break;
    }
    return TRUE;
}
```

At this point, I had the correct region. But this then led to a few *days* of debugging.

DLLMain Bugs

For the longest time, my DLLMain had created a thread on DLL_PROCESS_ATTACH :

```
NtCreateThreadEx(&hThread, GENERIC_EXECUTE, NULL, (HANDLE)(HANDLE)-1, StartVulpes,
nullptr, FALSE, 0, 0, 0, nullptr);
```

But this only caused issued because:

- 1. The Reflective Loader creates a thread pointing to the export function
- 2. The export function does some stuff and then calls **DLLMain**. So, that call will remain in the context of the thread from the Loader.
- 3. **DLLMain** is called and a subsequent thread is created pointing to the implants core function, then it breaks.
- 4. The DLLMain returns and the NtWaitForSingleObject call returns, and the implant exits with ERROR_SUCCESS.

TL;DR: DLLMain shouldn't create a thread because the loader will do the thread creation.

Also, don't be like me and use a Parent Process Id spoof in the loader which injects into a suspended process because the process hasn't finished setting up. This left the thread created by the loader with a base address of 0×0 , crashing within Ekko.

By simply removing the thread creation in **DLLmain**, and just calling the function, the timers work:

Payloads Listeners	Commands Web Log	gs								O Settings 🗸
Implant Id	Listener	Operating System	Hostname	Address	Username	Integrity	Process Name	Process Id	Architecture	Last Seen
Insignificant Basset Hound	Debugging Listener	10.0.19044	WIZARD	10.10.11.222	mez0	Medium	Vulpes.x64.exe	10112	x64	Wednesday, 29 June 2022 at 15:40:40
ently cher General Statistics Performance	erties e Threads Token Modules Memory I	Environment Handles GPU Co	omment							
ntly chec								Strings	Refresh	
ntly chec esponse General Statistics Performanc			size Prot Use	:			Total WS Pri	Strings	Refresh	
esponse General Statistics Performanc	e Threads Token Modules Memory i						Total WS Pri		Refresh	
esponse Hide free regions Base address	e Threads Token Modules Memory i Type		Size Prot Use	ck (thread 23244)			Total WS Pri		Refresh	
ntly chec sponse Hide free regions Base address 0x1ce8000	e Threads Token Modules Memory I Type Private: Commit		Size Prot Use 12 kB RW+G Sta	ck (thread 23244) ck (thread 20276)			Total WS Pri		Refresh	
ntly chec esponse Hide free regions Base address 0x10e8000 0x210e000	e Threads Token Modules Memory y Type Private: Commit Private: Commit		Size Prot Use 12 kB RW+G Sta 12 kB RW+G Sta	ck (thread 23244) ck (thread 20276) ck (thread 27312)			Total WS Pri		Refresh	
tily cher sponse General Statistics Performance → Hide free regions Base address 0x1208000 0x2308000 0x2508000 0x2508000 0x2508000 0x2108000 0x2108000 0x2108000 0x2108000 0x2108000 0x2108000 0x2108000 0x2108000 0x2108000 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x210800 0x2108000 0x210800 0x210800 0x210800 0x2108000 0x210800 00	e Threads Token Modules Memory Type Private: Commit Private: Commit		Size Prot Use 12.k8 RW+G Sta 12.k8 RW+G Sta 12.k8 RW+G Sta 12.k8 RW+G Sta 20.k8 RX C:W	ck (thread 23244) ck (thread 20276) ck (thread 27312) ck (thread 21840) Users\mez0\Downloads\Vulp			20 k8	vate WS Shareable W	Refresh S Sh ^	
ntly chee esponse ⊡ Hide free regions Base address 0x1208000 0x2208000 0x250600	e Threads Token Modules Memory Type Private: Commit Private: Commit Private: Commit	4	Size Prot Use 12.k8 RW+G Sta 12.k8 RW+G Sta 12.k8 RW+G Sta 12.k8 RW+G Sta 12.k8 RW+G Sta 20.k8 RX C:V 20.k8 RX C:V	ck (thread 23244) ck (thread 20276) ck (thread 27312) ck (thread 21840)	di			vate WS Shareable W	Refresh IS Sh ^	

Conclusion

All in all, this took a few days of my life. The Timers technique is a interesting and is a cool way to hide malicious memory regions. With that said, <u>Patriot</u> is a tool put together by <u>Joe</u> <u>Desimone</u> to detect this method by searching memory for timers which point to **NtContinue** !

Thanks to:

- <u>Peter Winter-Smith</u> and <u>modexp</u>: Original authors of the technique
- <u>Austin Hudson</u>: For identifying a sample and Reverse Engineering the technique 👀
- <u>5pider</u>: For proof-of-concepting the research
- <u>Paranoid Ninja</u>: For helping me understand Reflective DLLs properly and debugging the memory region setup