

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

By Chetan Nayak

Published: 2021-06-01 · Archived: 2026-04-06 00:11:16 UTC

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 positives 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 gets 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 VirtualAlloc with RWX, WriteProcessMemory after rebasing the PE and then calls the DllMain function as a function pointer.

The diagram illustrates the execution flow of a reflective loader. It starts with 'primary_stage.exe' which performs several system calls: 'CreateProcessW | Hidden Window | Suspended', 'WriteProcessMemory', 'VirtualAllocEx PAGE_READWRITE', 'VirtualProtectEx PAGE_EXECUTE_READ', and 'CreateRemoteThread ExportedFunction pointer'. These actions lead to the execution of 'notepad.exe'. A 'reflective thread endpoint' is established, which then triggers 'Rebase the PE Parse kernel32.dll'. This step leads to 'VirtualAlloc RWX WriteProcessMemory', which is associated with a red box containing 'PAGE_EXECUTE_READWRITE' and '???'. The process then calls 'DllMain()' and finally reaches the 'FINAL PAYLOAD'. A screenshot of a memory dump shows the 'loader's RWX' region at 0070120000 and the 'Original RX' region at 0070120000, with various memory addresses and permissions listed.

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 its 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)->SectionHeaderAddressOfFirstSectionHeader);
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;
    }
}
```

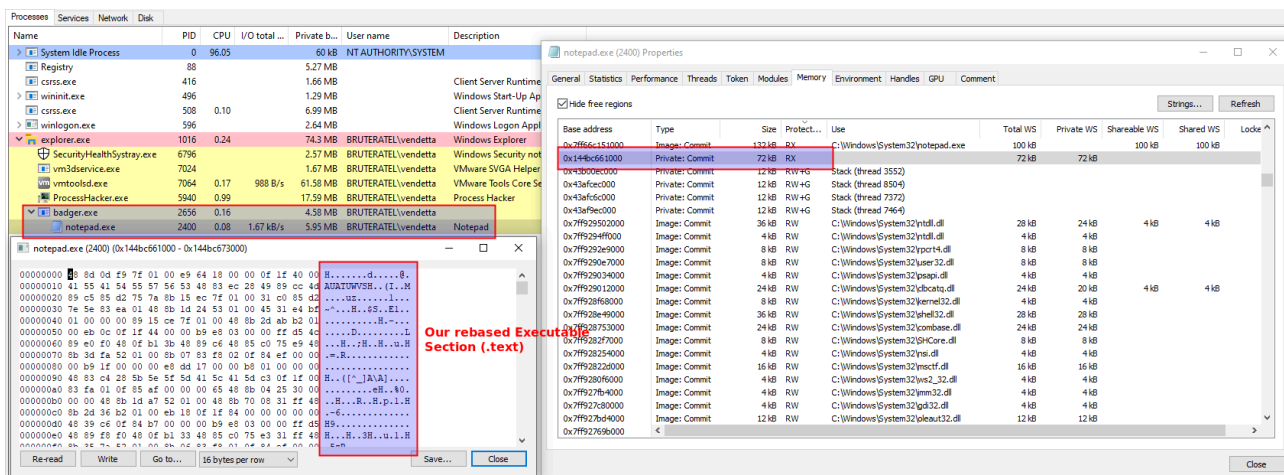
```

if (((PIMAGE_SECTION_HEADER)pSectionHeader)->Characteristics & IMAGE_SCN_MEM_EXECUTE) && (((PIM
    ulPermissions = PAGE_EXECUTE_READ;
}
if (((PIMAGE_SECTION_HEADER)pSectionHeader)->Characteristics & IMAGE_SCN_MEM_EXECUTE) && (((PIM
    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.



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.

Source: <https://bruteratel.com/research/feature-update/2021/06/01/PE-Reflection-Long-Live-The-King/>