Windows 10 20H1.19577 开始 System 进程内 Ntdll 的一点变化

背景

最近为了解决一个兼容问题，看了下 Windows 10 20H1 的内核，发现了一处有点意思的改动。

简单来说， ntdll.dll 过去映射在 System 进程以及其他所有进程内的时候，都是以可执行页面进行映射的。但从 19577 开始，在 System 进程内，NTDLL 开始映射为只读页面，而在其他所有进程内，NTDLL 仍映射为可执行页面。

初步检查 VADs

这点可以用 Windbg 来印证：

切换到 System 进程后，查看 VADs

```
1: kd> !vad
VAD     level      start      end    commit
9769a248 ( 2)         e0        e0        0 Mapped       READWRITE          Pagefile-backed section
9769a2a0 ( 1)         f0        f0        0 Mapped       READWRITE          Pagefile-backed section
94bdf6c0 ( 2)        100       100        0 Mapped       READWRITE          Pagefile-backed section
8c266770 ( 0)        3d0       579        0 Mapped  Exe  READONLY           \Windows\System32\ntdll.dll
86177288 ( 1)       7ffe0     7ffe0        1 Private      READONLY
```

```
8.  86177078 ( 2)       7ffe7     7ffe7         1 Private      READONLY
9.  Total VADs: 6, average level: 2, maximum depth: 2
```

可以看到，映射的 ntdll.dll 是 READ ONLY

那么我们切到其他进程，例如 smss，查看 VADs，可以看到这里 ntdll.dll 映射的是 EXECUTE_WRITECOPY

```
1.  1: kd> !vad
2.  VAD     level      start      end    commit
3.  90aff2a0 ( 3)         630       630         0 Mapped       READONLY           Pag
    efile-backed section
4.  8ec7afb0 ( 2)         650       66c         0 Mapped       READONLY           Pag
    efile-backed section
5.  90a72a08 ( 3)         670       6af         5 Private      READWRITE
6.  8ec7ada0 ( 1)         6b0       6e0         0 Mapped       READONLY           \Wi
    ndows\System32\C_936.NLS
7.  8ec7abe8 ( 3)         6f0       6f2         0 Mapped       READONLY           \Wi
    ndows\System32\l_intl.nls
8.  90a72cd8 ( 2)         700       708         1 Private      READWRITE
9.  90a72eb8 ( 4)         710       74f         5 Private      READWRITE
10. 90a72e58 ( 3)         750       78f         5 Private      READWRITE
11. a32a52b8 ( 4)         790       7cf         5 Private      READWRITE
12. 90a72a68 ( 0)         800       9ff         5 Private      READWRITE
13. 8ec7ab38 ( 3)         aa0       abb         3 Mapped  Exe  EXECUTE_WRITECOPY  \W
    indows\System32\smss.exe
14. 8ec7af58 ( 2)         ac0      2abf         8 Mapped       NO_ACCESS          Pag
    efile-backed section
15. 90a72d38 ( 3)        2c00      2dff        15 Private      READWRITE
16. 8ec7ab90 ( 1)       775f0     77799         9 Mapped  Exe  EXECUTE_WRITECOPY  \W
    indows\System32\ntdll.dll
17. 90a72d08 ( 4)       7f960     7f968         2 Private      READWRITE
18. 90a72d68 ( 3)       7f970     7f971         2 Private      READWRITE
19. 8ec7adf8 ( 4)       7f980     7f980         0 Mapped       READONLY           Pag
    efile-backed section
20. 90a72948 ( 2)       7ffe0     7ffe0         1 Private      READONLY
21. 90a72a98 ( 3)       7ffe7     7ffe7         1 Private      READONLY
22. Total VADs: 19, average level: 3, maximum depth: 4
```

Windbg 的!vad 指令是简单使用_MMVAD->VadFlags.Protection 来判断其页面的保护属性的。我们查看对应 VAD 结构也可以获得相同结论：

```
1: kd> dt _MMVAD 8c266770 -b
nt!_MMVAD
   +0x000 Core             : _MMVAD_SHORT
      +0x000 NextVad          : 0x9769a2a0
      +0x004 ExtraCreateInfo  : 0x86177288
      +0x000 VadNode          : _RTL_BALANCED_NODE
         +0x000 Children         :
          [00] 0x9769a2a0
          [01] 0x86177288
         +0x000 Left             : 0x9769a2a0
         +0x004 Right            : 0x86177288
         +0x008 Red              : 0y0
         +0x008 Balance          : 0y00
         +0x008 ParentValue      : 0
      +0x00c StartingVpn      : 0x3d0
      +0x010 EndingVpn        : 0x579
      +0x014 ReferenceCount   : 0n0
      +0x018 PushLock         : _EX_PUSH_LOCK
         +0x000 Locked           : 0y0
         +0x000 Waiting          : 0y0
         +0x000 Waking           : 0y0
         +0x000 MultipleShared   : 0y0
         +0x000 Shared           : 0y00000000000000000000000000000000 (0)
         +0x000 Value            : 0
         +0x000 Ptr              : (null)
      +0x01c u                : <unnamed-tag>
         +0x000 LongFlags        : 0xa0
         +0x000 VadFlags         : _MMVAD_FLAGS
            +0x000 Lock             : 0y0
            +0x000 LockContended    : 0y0
            +0x000 DeleteInProgress : 0y0
            +0x000 NoChange         : 0y0
            +0x000 VadType          : 0y010
            +0x000 Protection       : 0y00001 (0x1)
            +0x000 PreferredNode    : 0y000000 (0)
            +0x000 PageSize         : 0y00
            +0x000 PrivateMemory    : 0y0
```

```
38. 以下省略
```

可以看到 Protection 是 1 ， 这里的 Protection 是 Mm 内部转换为 Index 的 MM_XXXX 保护值，具体来说：

```
1.  MM_EXECUTE_WRITECOPY = 7
2.  MM_READONLY = 1
```

NTDLL 映射过程

那么，具体是哪里的代码影响了不同进程的 NTDLL 的保护属性呢？

这里要介绍下系统是如何加载映射 NTDLL 的。

在 Windows 操作系统启动过程中，OS LOADER 会将 OS KERNEL(ntoskrnel.exe），HAL 等等，以及 BOOT 驱动加载到内存中，但是 BOOT 驱动一开始不会得到执行， 而是会在 IO 系统初始化过程中，由 IoInitSystem 函数进行初始化并执行。

在执行 BOOT 和系统驱动的之前， 系统要为这些驱动创建 System 进程环境（在 Win10 中使用 IoInitSystemPreDrivers 实现），其中一项就是加载 NTDLL.DLL 到系统进程的用户内存中。

加载 NTDLL.DLL 使用的是 PsLocateSystemDll，该函数遍历 PspSystemDlls 常量内的 DLL(X64 下还需要加载 WOW64 的 NTDLL)，找到 NTDLL 路径后，为其创建 Section，最后使用 PsMapSystemDll 进行映射，实现加载。

这是 NTDLL.DLL 第一次被加载，其后其他进程加载 NTDLL 则是使用另一条路径，当非 system 进程创建时，系统会调用 MiMapProcessExecutable，将进程的可执行文件映射到内存中，同时，也会调用 PspMapSystemDlls->PsMapSystemDll 将 NTDLL 映射到内存中，此时的 section 会使用第一次加载时，保存的 section object（使用 fast reference 存储和管理）。

在 WRK 中，我们可以看到，PsMapSystemDll 的原型为：

```
1.  NTSTATUS
2.  PsMapSystemDll (
3.      IN PEPROCESS Process,
4.      OUT PVOID *DllBase OPTIONAL,
5.      IN LOGICAL UseLargePages
6.      )
```

然而在 Windows 10 中，这个函数在最后新增了一个参数，可以称为 bFirstInit 吧，指示了 PsMapSystemDll 是从 IoInitSystem 过来的初次加载，还是从 MiMapProcessExecutable 过来的后续映射。

首先我们可以看看，上个版本（19564）是如何处理这个参数的：

```
1.  status = MmMapViewOfSectionEx(
2.                  Section,
3.                  Process,
4.                  (int)&MappedBase,
5.                  (int)&ZeroBits,
6.                  (int)&CommitSize,
7.                  UserLargePages != 0 ? MEM_LARGE_PAGES : 0,
8.                  4,
9.                  (int)&v12,
10.                 2,
11.                 0,
12.                 0,
```

```
13.             UserLargePages != 0 ? MEM_LARGE_PAGES : 0);
14.   ObFastDereferenceObject(v5, Section);
15.   if ( status != 1073741827 )
16.     goto LABEL_5;
17.   if ( Process != PsInitialSystemProcess )
18.   {
19.     status = STATUS_CONFLICTING_ADDRESSES;
20. LABEL_5:
21.     if ( status < 0 )
22.       return status;
23.   }
24.   if ( bFirstInit )
25.   {
26.     status = 0;
27.     ntheaders = RtlImageNtHeader(MappedBase);
28.     _DllInfo = DllInfo;
29.     *(_DWORD *)(DllInfo + 20) = ntheaders->OptionalHeader.ImageBase;
30.     *(_DWORD *)(_DllInfo + 24) = MappedBase;
31.     return status;
32.   }
```

这里可以看到对于 bFirstInit 的处理，仅仅是在映射后（可看到对 UseLargePages 的处理），如果是第一次加载，要将 ImageBase 和映射后的 Base 保存到 DLL 信息中。

19577 的代码改动

但是检查 19577 开始（包括以后版本）的 PsMapSystemDll 代码，可以看到：

```
1.  AllocationType = UseLargePages != 0 ? MEM_LARGE_PAGES : 0;
2.    CommmitSize = 0;
3.    v14 = 5;
4.    v15 = 32;
5.    v16 = 0;
6.    v19 = 0;
7.    if ( bFirstInit )
8.      AllocationType |= MEM_MAPPED;
9.    if ( !(*(v5 + 8) & 8) )
```

```
10.     v18 = MmHighestUserAddress;
11.   status = MmMapViewOfSectionEx(
12.              Section,
13.              Process,
14.              &MappedBase,
15.              &ZeroBits,
16.              &CommmitSize,
17.              AllocationType,
18.              2,
19.              &v12,
20.              2,
21.              0,
22.              0,
23.              AllocationType);
24.   ObFastDereferenceObject(v5, Section);
25.   if ( status != STATUS_IMAGE_NOT_AT_BASE )
26.     goto LABEL_9;
27.   if ( Process != PsInitialSystemProcess )
28.   {
29.     status = STATUS_CONFLICTING_ADDRESSES;
30. LABEL_9:
31.     if ( status < 0 )
32.       return status;
33.   }
34.   if ( bFirstInit )
35.   {
36.     MappedBase_1 = MappedBase;
37.     status = 0;
38.     v9 = RtlImageNtHeader(MappedBase);
39.     v10 = v25;
40.     *(v25 + 0x14) = v9->OptionalHeader.ImageBase;
41.     *(v10 + 0x18) = MappedBase_1;
42.     return status;
43.   }
```

这里 bFirstInit 除了用于标记是否保存 ImageBase 和映射后的 base 外，当=TRUE 时，还会将 AllocationType 增加 MEM_MAPPED

Mm/Mi 映射的处理

那么使用 MEM_MAPPED 又会如何影响 MmMapViewOfSectionEx 的行为呢？

我们一路追下去看 MmMapViewOfSectionEx->MiMapViewOfSectionExCommon->MiMapViewOfSection->MiMapViewOfImageSection 的行为

在映射镜像的处理中，我们可以追到相应的代码：（这部分改动早于 19577）

```
1.  if ( !MapSectionParameters || *(MapSectionParameters + 32) & MEM_RESET || *(v8
    + 20) & MEM_MAPPED )
2.    {
3.      if ( a7 != 1 )
4.        return STATUS_INVALID_PAGE_PROTECTION;
5.      _memflags |= 0x800u;
6.      mem_flags = _memflags;
7.    }
```

可以看到，这里（MapSectionParameters 是 Mi 处理过程中将参数 AllocationType 整理打包到栈上）判断 AllocationType 如果含有 MEM_MAPPED 或 MEM_RESET，则会将函数的一个内部标记 增加 0x800。

而对于这个 0x800 的标记，其后是这样处理的：

```
1.  flags = v25 & 0xFFFFF0FF | 0x80;
2.    vad->Core.PushLock.Value = 0;
3.    vad->Core.u.LongFlags = flags;
4.    if ( mem_flags < 0x800 )
5.    {
6.      flags |= 0x380u;
7.      vad->Core.u.LongFlags = flags;
```

```
8.      vad->Core.u1.LongFlags1 ^= (BugCheckParameter2 ^ vad->Core.u1.LongFlags1)
    & 0x7FFFFFFF;
9.  }
```

在这里代码我们可以看到，首先会将 VadFlags 初始化添加 0x80，这里因为 Protection 的位置在 Flags 的第 7 位开始，所以 0x80>>7 = 1，即为 MM_READONLY

如果 mem_flags 没有 0x800 的标记，也就是正常映射，没有 MEM_MAPPED 的情况下，会为 VadFlags 添加 0x380, 0x380>>7 = 7，即为 MM_EXECUTE_WRITECOPY

至此，我们就弄明白了为什么 System 进程（首次加载）中的 NTDLL 会是 Read Only 的属性，而其他进程中的，会是 Execute Write Copy

想法

至于为什么会有这个改动，猜测有可能是为了避免在 System 进程地址空间中遗留可定位的可执行内存？

为什么不干脆取消 NTDLL 在 system 进程的映射？猜测是还有一些第三方内核驱动还需要 NTDLL 中的数据。