Windows 10 Parallel Loading Breakdown

blogs.blackberry.com/en/2017/10/windows-10-parallel-loading-breakdown

One of the unnoticed improvements of Window 10 is the parallel library loading support in *ntdll.dll*. This feature decreases process startup times by using multiple threads to load libraries from disk into memory.

How Windows 10 Implements Parallel Loading

Windows 10 implements parallel loading by creating a thread pool of worker threads when the process initializes. The parent process defines the number of worker threads by defining in the *PEB->ProcessParameters->LoaderThreads* (ULONG) field. *ntdll!LdrpInitializeExecutionOptions* can further override the LoaderThreads field by querying the Image File Execution Options (IFEO) registry key *HKLM\Software\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\<image.exe>\MaxLoaderThreads*.

Interestingly, Windows 10 contains a default entry for *chrome.exe* with *MaxLoaderThreads* set to 1 to disable parallel loading.

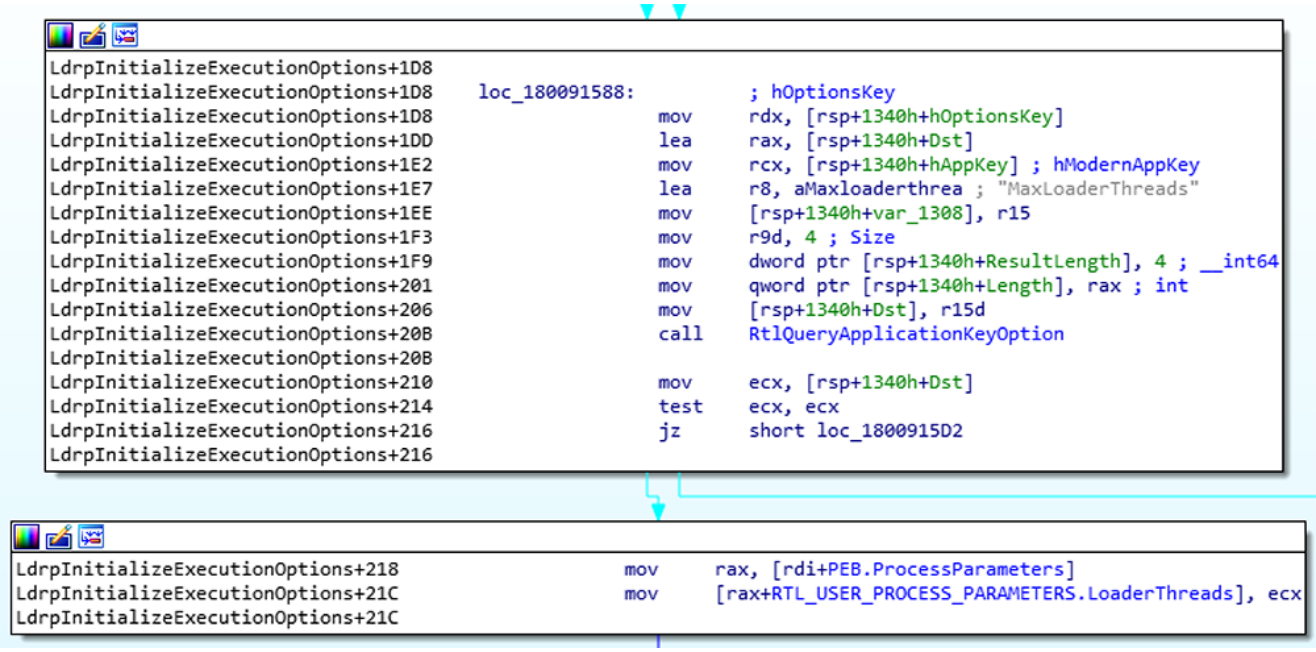


Figure 1: Querying the IFEO registry key for MaxLoaderThreads

The initial thread in the process executing *ntdll!LdrInitializeThunk* will be referred to as the master thread. Threads created by the master thread in the thread pool will be referred to as worker threads.

ntdll!LdrpInitParallelLoadingSupport and *ntdll!LdrpCreateLoaderEvents* are called to initialize the following structures:

- ntdll!LdrpWorkQueue (LIST_ENTRY)
- ntdll!LdrpWorkQueueTail (LIST_ENTRY)
- ntdll!LdrpWorkQueueLock (CRITICAL_SECTION)
- ntdll!LdrpRetryQueue (LIST_ENTRY)
- ntdll!LdrpRetryQueueTail (LIST_ENTRY)
- ntdll!LdrpLoadCompleteEvent (HANDLE)
- ntdll!LdrpWorkCompleteEvent (HANDLE)

```
LdrpInitParallelLoadingSupport
LdrpInitParallelLoadingSupport
LdrpInitParallelLoadingSupport
LdrpInitParallelLoadingSupport          LdrpInitParallelLoadingSupport proc near
LdrpInitParallelLoadingSupport                   sub     rsp, 28h
LdrpInitParallelLoadingSupport+4                 lea     rax, LdrpWorkQueue
LdrpInitParallelLoadingSupport+B                 xor     r8d, r8d
LdrpInitParallelLoadingSupport+E                 mov     cs:LdrpWorkQueueTail, rax
LdrpInitParallelLoadingSupport+15                lea     rcx, LdrpWorkQueueLock
LdrpInitParallelLoadingSupport+1C                mov     cs:LdrpWorkQueue, rax
LdrpInitParallelLoadingSupport+23                xor     edx, edx
LdrpInitParallelLoadingSupport+25                lea     rax, LdrpRetryQueue
LdrpInitParallelLoadingSupport+2C                mov     cs:LdrpRetryQueueTail, rax
LdrpInitParallelLoadingSupport+33                mov     cs:LdrpRetryQueue, rax
LdrpInitParallelLoadingSupport+3A                call    RtlInitializeCriticalSectionEx
LdrpInitParallelLoadingSupport+3A
LdrpInitParallelLoadingSupport+3F                add     rsp, 28h
LdrpInitParallelLoadingSupport+43                jmp     $+5
LdrpInitParallelLoadingSupport+43
LdrpInitParallelLoadingSupport+43      LdrpInitParallelLoadingSupport endp
LdrpInitParallelLoadingSupport+43
```

Figure 2: Initializing the work queue structures

```
LdrpCreateLoaderEvents
LdrpCreateLoaderEvents
LdrpCreateLoaderEvents
LdrpCreateLoaderEvents          LdrpCreateLoaderEvents proc near
LdrpCreateLoaderEvents
LdrpCreateLoaderEvents          InitialState    = byte ptr -18h
LdrpCreateLoaderEvents
LdrpCreateLoaderEvents                           sub     rsp, 38h
LdrpCreateLoaderEvents+4                        mov     r9d, 1 ; EventType
LdrpCreateLoaderEvents+A                        mov     [rsp+38h+InitialState], 0 ; InitialState
LdrpCreateLoaderEvents+F                        xor     r8d, r8d ; ObjectAttributes
LdrpCreateLoaderEvents+12                       lea     rcx, LdrpLoadCompleteEvent ; EventHandle
LdrpCreateLoaderEvents+19                       mov     edx, 1F0003h ; DesiredAccess
LdrpCreateLoaderEvents+1E                       call    ZwCreateEvent
LdrpCreateLoaderEvents+1E
LdrpCreateLoaderEvents+23                       test    eax, eax
LdrpCreateLoaderEvents+25                       js      short loc_18003C596
LdrpCreateLoaderEvents+25
```

```
LdrpCreateLoaderEvents+27                       mov     r9d, 1 ; EventType
LdrpCreateLoaderEvents+2D                       mov     [rsp+38h+InitialState], 0 ; InitialState
LdrpCreateLoaderEvents+32                       xor     r8d, r8d ; ObjectAttributes
LdrpCreateLoaderEvents+35                       lea     rcx, LdrpWorkCompleteEvent ; EventHandle
LdrpCreateLoaderEvents+3C                       mov     edx, 1F0003h ; DesiredAccess
LdrpCreateLoaderEvents+41                       call    ZwCreateEvent
LdrpCreateLoaderEvents+41
```

Figure 3: Creating the synchronization events

After *ntdll* loads *kernel32.dll* and *kernelbase.dll* are loaded, *ntdll!LdrpEnableParallelLoading* is called to set up the necessary events and worker pool. One interesting thing to note is that *kernel32.dll* and *kernelbase.dll* are loaded even if the process does not require it.

How Windows 10 Mitigates Parallel Loading Hazards

There are a lot of hazards when it comes to parallel loading and code hooking. In order to mitigate against corrupting memory or compatibility issues, Windows detects if a process is hooked before enabling parallel loading.

ntdll!LdrpEnableParallelLoading calls *ntdll!LdrpDetectDetour* to determine if the process being hooked. If a hook is detected, *ntdll!LdrpDetourExist* is set to true and the thread pool is drained and released.

Hooks are detected by examining the first 16 bytes of the functions defined in *ntdll!LdrpCriticalLoaderFunctions:*

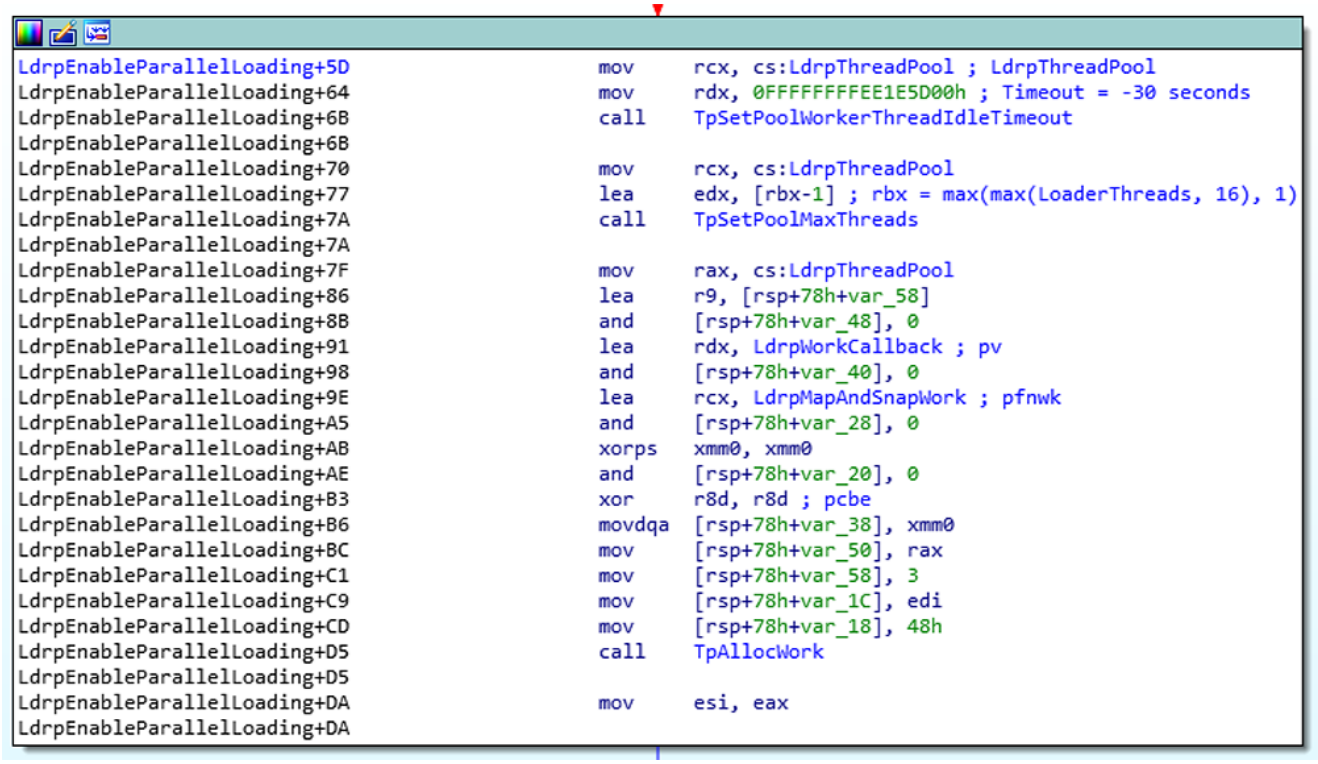
- ntdll!NtOpenFile
- ntdll!NtCreateSection
- ntdll!ZqQueryAttributes
- ntdll!NtOpenSection
- ntdll!ZwMapViewOfSection

The first 16 bytes of these functions are compared to *ntdll!LdrpThunkSignature*.

This data is an array of the first 16 bytes of each function copied by *ntdll!LdrpCaptureCriticalThunks* which is called near the start of *ntdll!LdrpInitializeProcess*.

ntdll!LdrpEnableParallelLoading validates the number of worker threads to be between [1, 16] and creates a thread pool with one less than *LoaderThreads*workers threads since the master thread will also perform the work of loading DLLs. If *LoaderThreads* is 0, it will be set to the default value of 4; if the *LoaderThreads* is larger than 16, it is set to the max value of 16.

The worker thread idle timeout is set to 30 seconds. Programs which execute in less than 30 seconds will appear to hang due to *ntdll!TppWorkerThread*waiting for the idle timeout before the process terminates.



```
LdrpEnableParallelLoading+5D    mov     rcx, cs:LdrpThreadPool ; LdrpThreadPool
LdrpEnableParallelLoading+64    mov     rdx, 0FFFFFFFFEE1E5D00h ; Timeout = -30 seconds
LdrpEnableParallelLoading+6B    call    TpSetPoolWorkerThreadIdleTimeout
LdrpEnableParallelLoading+6B
LdrpEnableParallelLoading+70    mov     rcx, cs:LdrpThreadPool
LdrpEnableParallelLoading+77    lea     edx, [rbx-1] ; rbx = max(max(LoaderThreads, 16), 1)
LdrpEnableParallelLoading+7A    call    TpSetPoolMaxThreads
LdrpEnableParallelLoading+7A
LdrpEnableParallelLoading+7F    mov     rax, cs:LdrpThreadPool
LdrpEnableParallelLoading+86    lea     r9, [rsp+78h+var_58]
LdrpEnableParallelLoading+8B    and     [rsp+78h+var_48], 0
LdrpEnableParallelLoading+91    lea     rdx, LdrpWorkCallback ; pv
LdrpEnableParallelLoading+98    and     [rsp+78h+var_40], 0
LdrpEnableParallelLoading+9E    lea     rcx, LdrpMapAndSnapWork ; pfnwk
LdrpEnableParallelLoading+A5    and     [rsp+78h+var_28], 0
LdrpEnableParallelLoading+AB    xorps   xmm0, xmm0
LdrpEnableParallelLoading+AE    and     [rsp+78h+var_20], 0
LdrpEnableParallelLoading+B3    xor     r8d, r8d ; pcbe
LdrpEnableParallelLoading+B6    movdqa  [rsp+78h+var_38], xmm0
LdrpEnableParallelLoading+BC    mov     [rsp+78h+var_50], rax
LdrpEnableParallelLoading+C1    mov     [rsp+78h+var_58], 3
LdrpEnableParallelLoading+C9    mov     [rsp+78h+var_1C], edi
LdrpEnableParallelLoading+CD    mov     [rsp+78h+var_18], 48h
LdrpEnableParallelLoading+D5    call    TpAllocWork
LdrpEnableParallelLoading+D5
LdrpEnableParallelLoading+DA    mov     esi, eax
LdrpEnableParallelLoading+DA
```

Figure 4: Creating the thread pool

ntdll!LdrpWorkCallback is registered as the thread pool work callback function. When work is available, the worker thread will call *ntdll!LdrpWorkCallbac*k which calls *ntdll!LdrpProcessWork*.

The thread will either map (*ntdll!LdrpMapDllSearchPath* or *ntdll!LdrpMapDllFullPath*) a DLL or snap (*ntdll!LdrpSnapModule*) a DLL based on the value *_LDR_DDAG_NODE.State*.

Mapping is the process of loading a file from disk into memory. Snapping is the process of resolving the library's import address table.

At the end of every mapping procedure, *ntdll!LdrpSignalModuleMapped* is called which will queue the snap action by calling *ntdll!LdrpQueueWork*.

The work queue is defined by *ntdll!LdrpWorkQueue* which is a doubly linked list (LIST_ENTRY) of an opaque structure, LDRP_LOAD_CONTEXT.

This structure is allocated by *ntdll!LdrpAllocatePlaceHolder* and contains a variety of information such as the DLL name, a _LDR_DATA_TABLE_ENTRY structure, a pointer to the import address table (IAT), the activation context, and the control flow guard (CFG) function pointer [1][2].

```
00000000 LDRP_LOAD_CONTEXT struc ; (sizeof=0xA1, mappedto_957)
00000000                                        ; XREF: LdrpDrainWorkQueue:loc_180048708/o
00000000 BaseDllName     UNICODE_STRING ?       ; XREF: LdrpAllocatePlaceHolder+84/w
00000000                                        ; LdrpAllocatePlaceHolder+93/w ...
00000010 pUnknown        dq ?                   ; XREF: LdrpAllocatePlaceHolder+8C/w
00000010                                        ; LdrpAllocateModuleEntry+86/r
00000018 Flags           dd ?                   ; XREF: LdrpProcessWork:loc_18004482B/r
00000018                                        ; LdrpProcessWork+58/r ...
0000001C                 db ? ; undefined
0000001D                 db ? ; undefined
0000001E                 db ? ; undefined
0000001F                 db ? ; undefined
00000020 pState          dq ?                   ; XREF: LdrpMapAndSnapDependency:loc_1800225FA/r
00000020                                        ; LdrpProcessWork+16/r ...
00000028 ParentEntry     dq ?                   ; XREF: LdrpMapAndSnapDependency+205/r
00000028                                        ; LdrpProcessWork:loc_1800448FD/r ...
00000030 Entry           dq ?                   ; XREF: LdrpSnapModule+1A/r
00000030                                        ; LdrpMapAndSnapDependency+17/r ... ; offset
00000038 WorkQueueListEntry LIST_ENTRY ?        ; XREF: LdrpQueueWork+2F/o
00000048 ReplacedEntry   dq ?                   ; XREF: LdrpHandlePendingModuleReplaced+1D/r
00000048                                        ; LdrpHandlePendingModuleReplaced:loc_18006EF16/w ; offset
00000050 ImportLdteArray dq ?                   ; XREF: LdrpSnapModule+6F/r
00000050                                        ; LdrpSnapModule:loc_1800178E4/r ... ; offset
00000058 ImportDllCount  dd ?                   ; XREF: LdrpSnapModule+64/r
00000058                                        ; LdrpMapAndSnapDependency+FC/w
0000005C TaskCount       dd ?                   ; XREF: LdrpMapAndSnapDependency+107/w
0000005C                                        ; LdrpMapAndSnapDependency+1D6/w ...
00000060 pvIAT           dq ?                   ; XREF: LdrpMapAndSnapDependency+44/r
00000060                                        ; LdrpMapAndSnapDependency:loc_18002259A/r ...
00000068 SizeOfIAT       dd ?                   ; XREF: LdrpPrepareImportAddressTableForSnap+D/o
0000006C CurrentDll      dd ?
00000070 pImageImportDescriptor dq ?            ; XREF: LdrpSnapModule:loc_1800178B1/r
00000078 pImageImportDescriptor2 dq ?           ; XREF: LdrpMapAndSnapDependency+F1/w
00000080 OriginalIATProtect dd ?                ; XREF: LdrpPrepareImportAddressTableForSnap+DF/o
00000084                 db ? ; undefined
00000085                 db ? ; undefined
00000086                 db ? ; undefined
00000087                 db ? ; undefined
00000088 CFGCheckFunction dq ?                  ; XREF: LdrpAllocateModuleEntry+A7/r
00000088                                        ; LdrpPrepareImportAddressTableForSnap+A0/w
00000090 field_90        dq ?
00000098 field_98        dq ?
```

Figure 5: Partially documented LDRP_LOAD_CONTEXT structure

At this point, worker threads in the thread pool will pull work off the queue and perform the appropriate action (mapping or snapping). If the worker finds a new dependency, it will queue up more work. Work is added and removed from the queue in a last-in first-out (LIFO) manner.

Once the thread pool has been initialized, the master thread continues on with *ntdll!LdrpMapAndSnapDependency* which will map the first level of explicit imports with a call to *ntdll!LdrpLoadDependentModule*. As the master thread loads imports, the work queue is filled up with secondary library dependencies for worker threads to process.

The master thread will perform the same map and snap work actions as the worker thread by calling *ntdll!LdrpDrainWorkQueue*.

```
LdrpInitializeProcess+11C2   loc_180092D46:                        ; CODE XREF: LdrpInitializeProcess+11AB↑j
LdrpInitializeProcess+11C2                                         ; LdrpInitializeProcess+11B1↑j
LdrpInitializeProcess+11C2                   mov     rax, gs:60h
LdrpInitializeProcess+11CB                   mov     edx, dword ptr [rax+PEB.AppCompatFlags]
LdrpInitializeProcess+11D1                   bt      rdx, 16h
LdrpInitializeProcess+11D6                   movzx   eax, cs:RtlpForceCSDebugInfoCreation
LdrpInitializeProcess+11DD                   mov     ecx, 1
LdrpInitializeProcess+11E2                   cmovb   eax, ecx
LdrpInitializeProcess+11E5                   mov     cs:RtlpForceCSDebugInfoCreation, al
LdrpInitializeProcess+11EB                   mov     rax, [rsp+428h+ProcessParameters]
LdrpInitializeProcess+11F3                   mov     ecx, [rax+RTL_USER_PROCESS_PARAMETERS.LoaderThreads] ; ulLoaderThreads
LdrpInitializeProcess+11F9                   call    LdrpEnableParallelLoading
LdrpInitializeProcess+11FE                   mov     edx, 1
LdrpInitializeProcess+1203                   mov     cs:LdrInitState, edx
LdrpInitializeProcess+1209                   mov     rax, cs:LdrpImageEntry
LdrpInitializeProcess+1210                   mov     rcx, [rax+LDR_DATA_TABLE_ENTRY.DdagNode]
LdrpInitializeProcess+1217                   mov     [rcx+_LDR_DDAG_NODE.State], r14d ; r14d = 2 = LdrModulesMapped
LdrpInitializeProcess+121B                   mov     rax, [rsp+428h+pStatus]
LdrpInitializeProcess+1223                   xor     ecx, ecx
LdrpInitializeProcess+1225                   mov     [rax], ecx
LdrpInitializeProcess+1227                   cmp     byte ptr [rsp+428h+bIsDotNET], cl
LdrpInitializeProcess+122B                   jnz     loc_180092F5E
LdrpInitializeProcess+1231                   or      [rsp+428h+var_1F0], edx
LdrpInitializeProcess+1238                   lea     rcx, [rsp+428h+LoadContext] ; LoadContext
LdrpInitializeProcess+1240                   call    LdrpMapAndSnapDependency
LdrpInitializeProcess+1245                   mov     ecx, 1          ; dwDrainTask
LdrpInitializeProcess+124A                   call    LdrpDrainWorkQueue
LdrpInitializeProcess+124F                   mov     eax, [rsp+428h+status]
LdrpInitializeProcess+1253                   test    eax, eax
LdrpInitializeProcess+1255                   jns     loc_18009306D
LdrpInitializeProcess+125B
```

Figure 6: Overview of master thread enabling parallel loading, mapping and snapping first level of dependencies, and joining with slave threads

ntdll!LdrpDrainWorkQueue serves as a synchronization point for the master thread as it joins in performing work added to *ntdll!LdrpWorkQueue* and returns when there is no more work to be completed. At this point, all of the dependencies have been resolved and loaded.

After all of the dependencies are mapped, *ntdll!LdrpPrepareModuleForExecution* is called which condenses the dependency graph with a call to *ntdll!LdrpCondenseGraph*. As the graph is traversed, callbacks are notified with *ntdll!LdrpSendPostSnapNotifications* which execute any callbacks registered with AppCompat (Shim Engine) or Application Verifier.

Once the callbacks are completed, *ntdll!LdrpInitializeNode* is called which initializes thread local storage (TLS) with a call to *ntdll!LdrpCallTlsInitializers* and finally every library's entry point (typically *DllMain*) is called by *ntdll!LdrpCallRoutine*.

File: C:\Windows\System32\ntdll.dll
Version: 10.0.15063.447
SHA256:
2B8D65907A2811121EA75DB44BC540D0AF198C1991C30886A365001123F16B7D

References:

[1] https://stackoverflow.com/questions/42789199/why-there-are-three-unexpected-worker-threads-when-a-win32-console-application-s

[2] https://conference.hitb.org/hitbsecconf2017ams/materials/D2T1%20-%20Bing%20Sun%20and%20Chong%20Xu%20-%20Bypassing%20Memory%20Mitigation%20Using%20Data-Only%20Exploitation%20Techniques.pdf