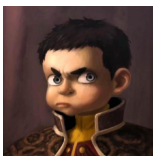
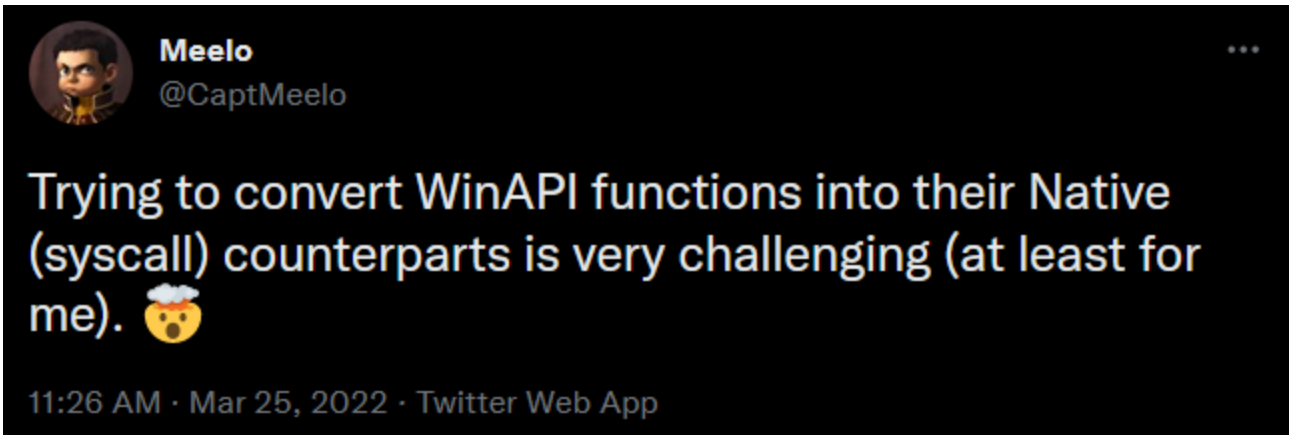
Making NtCreateUserProcess Work

captmeelo.com/redteam/maldev/2022/05/10/ntcreateuserprocess.html



Capt. Meelo

10 May 2022 » redteam, maldev
Last March, I tweeted something about converting WinAPI functions to their native counterparts. One of the WinAPIs I'm trying to convert is `CreateProcess`. Finally, after several months of on and off research, trials, and coding, I have successfully developed a PoC to launch a process using the native API `NtCreateUserProcess()`!

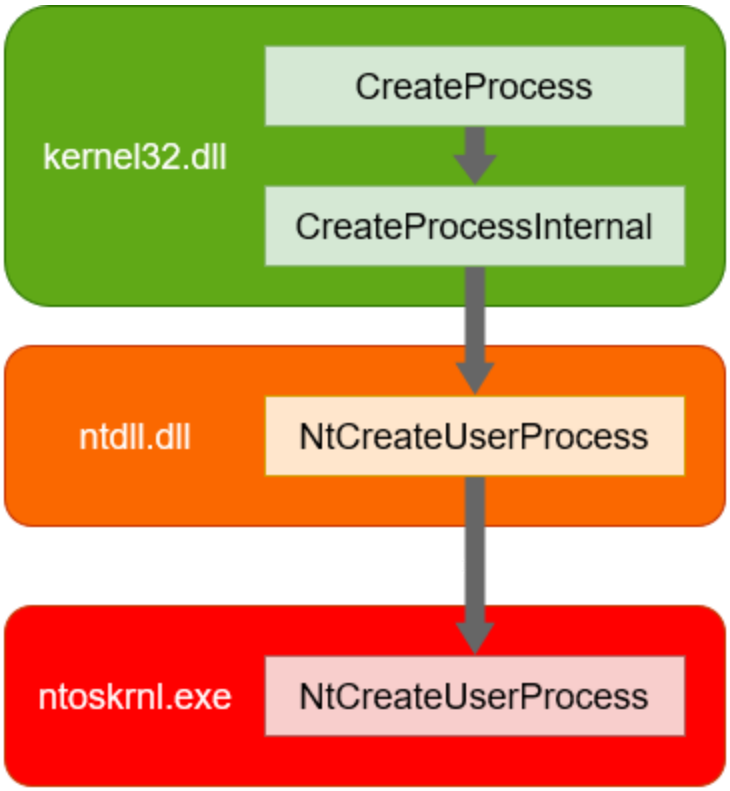


In this post, I'll share some of my notes and the journey that I took in the development of a "minimal" code, which is enough for my need of creating a process using native APIs.

> *I'm not an expert in Windows Internals so if you found some issues/mistakes, please let me know and I would be happy to make the necessary corrections.*

From `CreateProcess()` to `NtCreateUserProcess()`

One of the documented Windows APIs for creating processes is `CreateProcess()`. Using this API, the created process runs in the context (meaning the same access token) of the calling process. Execution then continues with a call to `CreateProcessInternal()`, which is responsible for actually creating the user-mode process. `CreateProcessInternal()` then calls the undocumented and native API `NtCreateUserProcess()` (located in `ntdll.dll`) to shift to kernel-mode.



But Why Use `NtCreateUserProcess()`?

`NtCreateUserProcess()` is the lowest and the last function accessible in user-mode that we could call to evade the detection controls (such as User-land Hooking) set by an AV/EDR.

When I searched the web for a sample implementation of `NtCreateUserProcess()`, I came across the following repos:

However, none of them worked during the times I tested them. I spent hours modifying the codes that I found to try and make it work, but I keep on failing. So aside from the evasive purposes of this native function, **the main reason I dedicated my free time to this subject is for the fun and challenge**.

`NtCreateUserProcess()`

The native API `NtCreateUserProcess()` has the following syntax.

```
NTSTATUS
NTAPI
NtCreateUserProcess(
    _Out_ PHANDLE ProcessHandle,
    _Out_ PHANDLE ThreadHandle,
    _In_ ACCESS_MASK ProcessDesiredAccess,
    _In_ ACCESS_MASK ThreadDesiredAccess,
    _In_opt_ POBJECT_ATTRIBUTES ProcessObjectAttributes,
    _In_opt_ POBJECT_ATTRIBUTES ThreadObjectAttributes,
    _In_ ULONG ProcessFlags,
    _In_ ULONG ThreadFlags,
    _In_ PRTL_USER_PROCESS_PARAMETERS ProcessParameters,
    _Inout_ PPS_CREATE_INFO CreateInfo,
    _In_ PPS_ATTRIBUTE_LIST AttributeList
);
```

Let's start building this function and its parameter starting with both `ProcessHandle` and `ThreadHandle` which will store the handles to the created process and thread. These two arguments are too simple and can be initialized with the following:

```
HANDLE hProcess, hThread = NULL;
```

For `ProcessDesiredAccess` and `ThreadDesiredAccess` parameters, we need to supply them with `ACCESS_MASK` values that would identify the rights and controls we have over the process and thread we're creating. Different values could be assigned to `ACCESS_MASK` and they are listed in `winnt.h`. Since we're only dealing with process and thread objects, we can use the process- and thread-specific access rights `PROCESS_ALL_ACCESS` and `THREAD_ALL_ACCESS`.

For other process- and thread-specific access rights, refer to these documentations:

The next parameters are `ProcessObjectAttributes` and `ThreadObjectAttributes`, which are pointers to an `OBJECT_ATTRIBUTES`. This structure contains the attributes that could be applied to the objects or object handles that will be created. These parameters are optional hence we can simply assign `NULL` values to them.

The flags set within `ProcessFlags` and `ThreadFlags` determine how we want our process and thread to be created (e.g., if we want a suspended process/thread upon creation). They are similar to the `dwCreationFlags` argument of `CreateProcess()` but the flags documented in MSDN do not apply to `NtCreateUserProcess()`. At the same time, **SysWhisper2** (my go-to tool for generating the structs and typedefs of a function) does not support the generation of these flags. So where do we get the flags for these parameters? Good thing **Process Hacker** exists and open-source.

So based on ntpsapi.h header of **Process Hacker**, the valid values for `ProcessFlags` are:

```
#define PROCESS_CREATE_FLAGS_BREAKAWAY 0x00000001 // NtCreateProcessEx &
NtCreateUserProcess
#define PROCESS_CREATE_FLAGS_NO_DEBUG_INHERIT 0x00000002 // NtCreateProcessEx &
NtCreateUserProcess
#define PROCESS_CREATE_FLAGS_INHERIT_HANDLES 0x00000004 // NtCreateProcessEx &
NtCreateUserProcess
#define PROCESS_CREATE_FLAGS_OVERRIDE_ADDRESS_SPACE 0x00000008 // NtCreateProcessEx
only
#define PROCESS_CREATE_FLAGS_LARGE_PAGES 0x00000010 // NtCreateProcessEx only,
requires SeLockMemory
#define PROCESS_CREATE_FLAGS_LARGE_PAGE_SYSTEM_DLL 0x00000020 // NtCreateProcessEx
only, requires SeLockMemory
#define PROCESS_CREATE_FLAGS_PROTECTED_PROCESS 0x00000040 // NtCreateUserProcess only
#define PROCESS_CREATE_FLAGS_CREATE_SESSION 0x00000080 // NtCreateProcessEx &
NtCreateUserProcess, requires SeLoadDriver
#define PROCESS_CREATE_FLAGS_INHERIT_FROM_PARENT 0x00000100 // NtCreateProcessEx &
NtCreateUserProcess
#define PROCESS_CREATE_FLAGS_SUSPENDED 0x00000200 // NtCreateProcessEx &
NtCreateUserProcess
#define PROCESS_CREATE_FLAGS_FORCE_BREAKAWAY 0x00000400 // NtCreateProcessEx &
NtCreateUserProcess, requires SeTcb
#define PROCESS_CREATE_FLAGS_MINIMAL_PROCESS 0x00000800 // NtCreateProcessEx only
#define PROCESS_CREATE_FLAGS_RELEASE_SECTION 0x00001000 // NtCreateProcessEx &
NtCreateUserProcess
#define PROCESS_CREATE_FLAGS_CLONE_MINIMAL 0x00002000 // NtCreateProcessEx only
#define PROCESS_CREATE_FLAGS_CLONE_MINIMAL_REDUCED_COMMIT 0x00004000 //
#define PROCESS_CREATE_FLAGS_AUXILIARY_PROCESS 0x00008000 // NtCreateProcessEx &
NtCreateUserProcess, requires SeTcb
#define PROCESS_CREATE_FLAGS_CREATE_STORE 0x00020000 // NtCreateProcessEx only
#define PROCESS_CREATE_FLAGS_USE_PROTECTED_ENVIRONMENT 0x00040000 //
NtCreateProcessEx & NtCreateUserProces
```

While the following are the valid flags for `ThreadFlags` (based on ntpsapi.h):

```
#define THREAD_CREATE_FLAGS_CREATE_SUSPENDED 0x00000001 // NtCreateUserProcess &
NtCreateThreadEx
#define THREAD_CREATE_FLAGS_SKIP_THREAD_ATTACH 0x00000002 // NtCreateThreadEx only
#define THREAD_CREATE_FLAGS_HIDE_FROM_DEBUGGER 0x00000004 // NtCreateThreadEx only
#define THREAD_CREATE_FLAGS_LOADER_WORKER 0x00000010 // NtCreateThreadEx only
#define THREAD_CREATE_FLAGS_SKIP_LOADER_INIT 0x00000020 // NtCreateThreadEx only
#define THREAD_CREATE_FLAGS_BYPASS_PROCESS_FREEZE 0x00000040 // NtCreateThreadEx only
#define THREAD_CREATE_FLAGS_INITIAL_THREAD 0x00000080 // ?
```

What I like about the flags provided by **Process Hacker** is that they put some notes on what flags are supported by which native APIs. So be wary since not all flags are supported by `NtCreateProcess()`. For example, only `THREAD_CREATE_FLAGS_CREATE_SUSPENDED` could be applied to the `ThreadFlags` parameter.

So which among these flags should we use? Well, to create a “minimal” working PoC for `NtCreateUserProcess()`, we could simply set these parameters to `NULL`.

The next parameter (`ProcessParameters`) is optional but I found it to be mandatory. This parameter points to a `RTL_USER_PROCESS_PARAMETERS` structure which describes the startup parameters of the process to be created. We'll discuss more about this parameter in the next section.

Next in line is `CreateInfo` , which is a pointer to a `PS_CREATE_INFO` structure. There's not much information on the Internet (based on my searches) about `PS_CREATE_INFO` and this is the only post that I found that discusses this structure.

Based on my experiment, the `PS_CREATE_STATE` enumeration value that worked for me is `PsCreateInitialState` . So for a "minimal" working PoC, I set the value of `PS_CREATE_INFO` members to:

```
CreateInfo.Size = sizeof(CreateInfo);
CreateInfo.State = PsCreateInitialState;
```

Just like `ProcessFlags` , I found that the last argument (`AttributeList`) is not optional (in my case). This parameter is used to set up the attributes for process and thread creation. An example of this is when implementing PPID Spoofing where the `PROC_THREAD_ATTRIBUTE_PARENT_PROCESS` attribute is set (if the WinAPI `CreateProcess()` is used). While the attributes for `CreateProcess()` is documented here, the valid attributes for `NtCreateProcess()` is not. So where do we get these "native attributes"? Again, **Process Hacker**'s ntpsapi.h to the rescue.

```
#define PS_ATTRIBUTE_PARENT_PROCESS PsAttributeValue(PsAttributeParentProcess, FALSE,
TRUE, TRUE)
#define PS_ATTRIBUTE_DEBUG_PORT PsAttributeValue(PsAttributeDebugPort, FALSE, TRUE,
TRUE)
#define PS_ATTRIBUTE_TOKEN PsAttributeValue(PsAttributeToken, FALSE, TRUE, TRUE)
#define PS_ATTRIBUTE_CLIENT_ID PsAttributeValue(PsAttributeClientId, TRUE, FALSE,
FALSE)
#define PS_ATTRIBUTE_TEB_ADDRESS PsAttributeValue(PsAttributeTebAddress, TRUE, FALSE,
FALSE)
#define PS_ATTRIBUTE_IMAGE_NAME PsAttributeValue(PsAttributeImageName, FALSE, TRUE,
FALSE)
#define PS_ATTRIBUTE_IMAGE_INFO PsAttributeValue(PsAttributeImageInfo, FALSE, FALSE,
FALSE)
#define PS_ATTRIBUTE_MEMORY_RESERVE PsAttributeValue(PsAttributeMemoryReserve, FALSE,
TRUE, FALSE)
#define PS_ATTRIBUTE_PRIORITY_CLASS PsAttributeValue(PsAttributePriorityClass, FALSE,
TRUE, FALSE)
#define PS_ATTRIBUTE_ERROR_MODE PsAttributeValue(PsAttributeErrorMode, FALSE, TRUE,
FALSE)
#define PS_ATTRIBUTE_STD_HANDLE_INFO PsAttributeValue(PsAttributeStdHandleInfo,
FALSE, TRUE, FALSE)
#define PS_ATTRIBUTE_HANDLE_LIST PsAttributeValue(PsAttributeHandleList, FALSE, TRUE,
FALSE)
#define PS_ATTRIBUTE_GROUP_AFFINITY PsAttributeValue(PsAttributeGroupAffinity, TRUE,
TRUE, FALSE)
#define PS_ATTRIBUTE_PREFERRED_NODE PsAttributeValue(PsAttributePreferredNode, FALSE,
TRUE, FALSE)
#define PS_ATTRIBUTE_IDEAL_PROCESSOR PsAttributeValue(PsAttributeIdealProcessor,
TRUE, TRUE, FALSE)
#define PS_ATTRIBUTE_UMS_THREAD PsAttributeValue(PsAttributeUmsThread, TRUE, TRUE,
FALSE)
#define PS_ATTRIBUTE_MITIGATION_OPTIONS
PsAttributeValue(PsAttributeMitigationOptions, FALSE, TRUE, FALSE)
#define PS_ATTRIBUTE_PROTECTION_LEVEL PsAttributeValue(PsAttributeProtectionLevel,
FALSE, TRUE, TRUE)
#define PS_ATTRIBUTE_SECURE_PROCESS PsAttributeValue(PsAttributeSecureProcess, FALSE,
TRUE, FALSE)
#define PS_ATTRIBUTE_JOB_LIST PsAttributeValue(PsAttributeJobList, FALSE, TRUE,
FALSE)
#define PS_ATTRIBUTE_CHILD_PROCESS_POLICY
PsAttributeValue(PsAttributeChildProcessPolicy, FALSE, TRUE, FALSE)
#define PS_ATTRIBUTE_ALL_APPLICATION_PACKAGES_POLICY
PsAttributeValue(PsAttributeAllApplicationPackagesPolicy, FALSE, TRUE, FALSE)
#define PS_ATTRIBUTE_WIN32K_FILTER PsAttributeValue(PsAttributeWin32kFilter, FALSE,
TRUE, FALSE)
#define PS_ATTRIBUTE_SAFE_OPEN_PROMPT_ORIGIN_CLAIM
PsAttributeValue(PsAttributeSafeOpenPromptOriginClaim, FALSE, TRUE, FALSE)
#define PS_ATTRIBUTE_BNO_ISOLATION PsAttributeValue(PsAttributeBnoIsolation, FALSE,
TRUE, FALSE)
#define PS_ATTRIBUTE_DESKTOP_APP_POLICY PsAttributeValue(PsAttributeDesktopAppPolicy,
FALSE, TRUE, FALSE)
#define PS_ATTRIBUTE_CHPE PsAttributeValue(PsAttributeChpe, FALSE, TRUE, TRUE)
#define PS_ATTRIBUTE_MITIGATION_AUDIT_OPTIONS
PsAttributeValue(PsAttributeMitigationAuditOptions, FALSE, TRUE, FALSE)
```

```
#define PS_ATTRIBUTE_MACHINE_TYPE PsAttributeValue(PsAttributeMachineType, FALSE,
TRUE, TRUE)
```

Table 3-7: Process Attributes of "Windows Internals, Part 1 (7th Edition)" presents a nice table on which "native attribute" corresponds to its Win32 equivalent.

Going back to `AttributeList`, I initialize this parameter with the following code:

```
PPS_ATTRIBUTE_LIST AttributeList =
(PS_ATTRIBUTE_LIST*)RtlAllocateHeap(RtlProcessHeap(), HEAP_ZERO_MEMORY,
sizeof(PS_ATTRIBUTE));
AttributeList->TotalLength = sizeof(PS_ATTRIBUTE_LIST) - sizeof(PS_ATTRIBUTE);

AttributeList->Attributes[0].Attribute = PS_ATTRIBUTE_IMAGE_NAME;
AttributeList->Attributes[0].Size = NtImagePath.Length;
AttributeList->Attributes[0].Value = (ULONG_PTR)NtImagePath.Buffer;
```

Here, the attribute `PS_ATTRIBUTE_IMAGE_NAME` specifies the name of the process to be created, and the `NtImagePath` variable holds the path of the image/binary from which the process will be created.

So here's what the code for `NtCreateProcess()` would look like:

```
NtCreateUserProcess(&hProcess, &hThread, PROCESS_ALL_ACCESS, THREAD_ALL_ACCESS, NULL,
NULL, NULL, NULL, ProcessParameters, &CreateInfo, AttributeList);
```

`RtlCreateProcessParametersEx()`

If you have observed, `NtCreateProcess()` does not accept any argument that contains the path to the process to be created. This is where `RtlCreateProcessParametersEx()` comes into action. As the name suggests, its purpose is to populate the structure that will hold the parameters of the process to be created. This undocumented (but not native) function has the following syntax.

```
NTSTATUS
NTAPI
RtlCreateProcessParametersEx(
    _Out_ PRTL_USER_PROCESS_PARAMETERS* pProcessParameters,
    _In_ PUNICODE_STRING ImagePathName,
    _In_opt_ PUNICODE_STRING DllPath,
    _In_opt_ PUNICODE_STRING CurrentDirectory,
    _In_opt_ PUNICODE_STRING CommandLine,
    _In_opt_ PVOID Environment,
    _In_opt_ PUNICODE_STRING WindowTitle,
    _In_opt_ PUNICODE_STRING DesktopInfo,
    _In_opt_ PUNICODE_STRING ShellInfo,
    _In_opt_ PUNICODE_STRING RuntimeData,
    _In_ ULONG Flags
);
```

`pProcessParameters` points to the `RTL_USER_PROCESS_PARAMETERS` structure, which will hold the process parameter information as a result of executing `RtlCreateProcessParametersEx()`. Any information stored in the structure is then used as an input to `NtCreateProcess()`. MSDN has poor and very limited documentation of the `RTL_USER_PROCESS_PARAMETERS` structure so I recommend the structure provided by **Process Hacker**.

The second parameter `ImagePathName` holds the full path (in NT path format) of the image/binary from which the process will be created. For example:

```
UNICODE_STRING NtImagePath;
RtlInitUnicodeString(&NtImagePath, (PWSTR)L"\\??\\C:\\Windows\\System32\\calc.exe");
```

The `RtlInitUnicodeString()` function, which has the following syntax, is necessary to initialize the `UNICODE_STRING` structure.

```
VOID NTAPIRtlInitUnicodeString(
    _Out_ PUNICODE_STRING DestinationString,
    _In_opt_ PWSTR SourceString
);
```

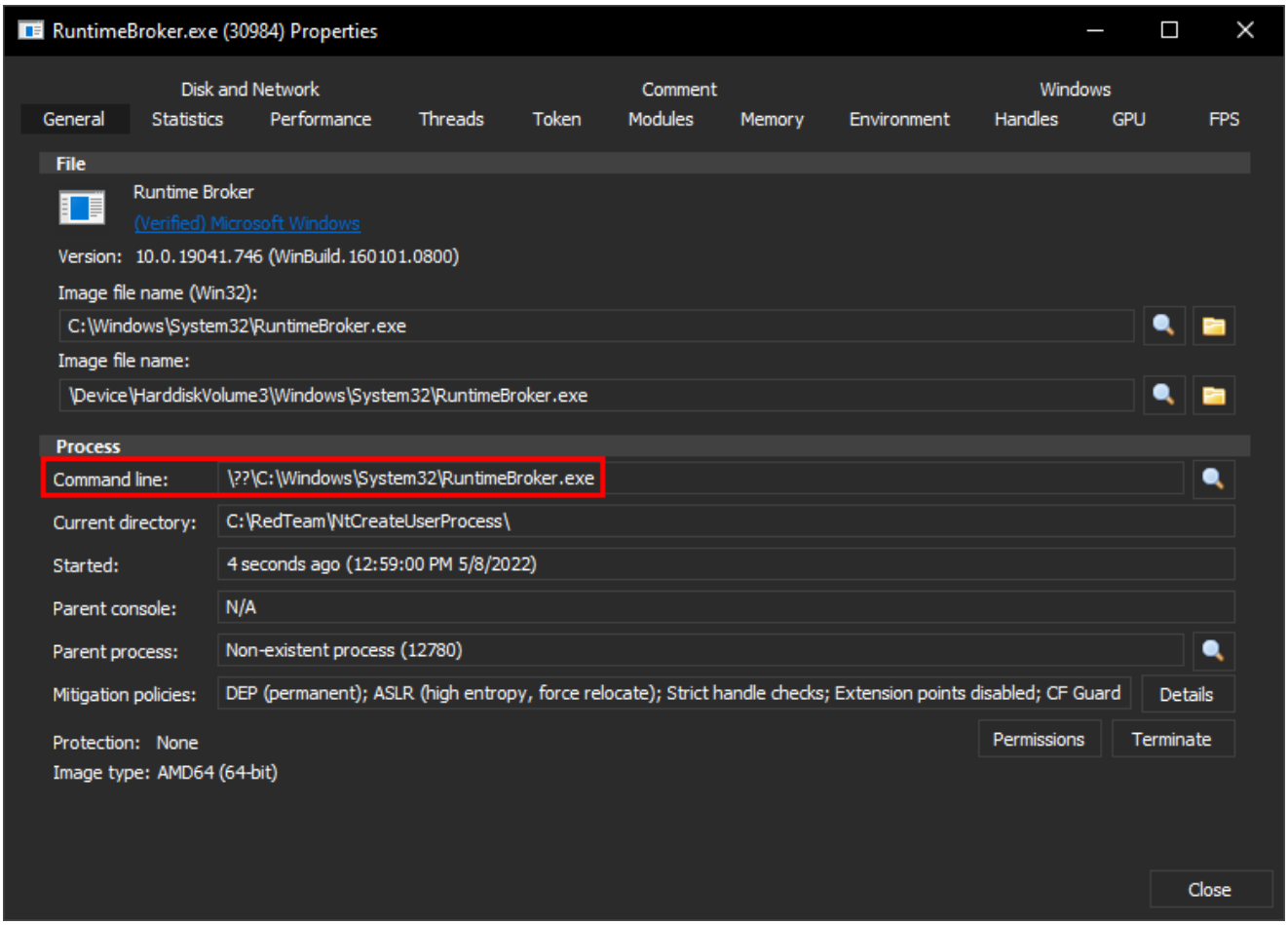
The initialization of the `UNICODE_STRING` structure is done by:

- Setting the `Length` and `MaximumLength` members to the length of the `SourceString`
- Setting the `Buffer` member to the address of the string passed in `SourceString`

```
typedef struct _UNICODE_STRING
{
  USHORT Length;
  USHORT MaximumLength;
  PWSTR Buffer;
} UNICODE_STRING, * PUNICODE_STRING;
```

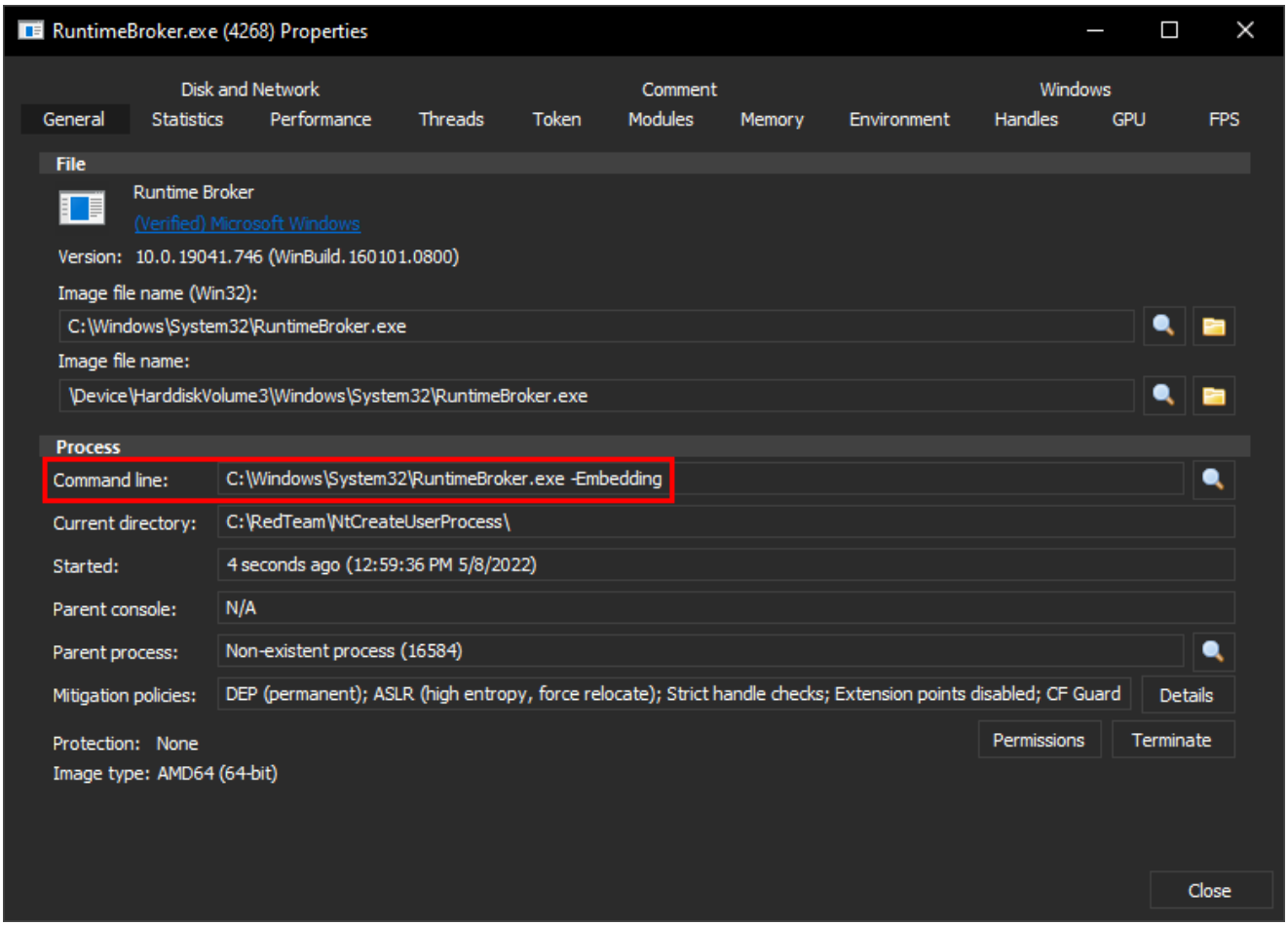
The other arguments (`DllPath` , `CurrentDirectory` , `CommandLine` , `Environment` , `WindowTitle` , `DesktopInfo` , `ShellInfo` , `RuntimeData`) are optional. For our goal, we can simply set them all to `NULL` .

A scenario in which these parameters can be useful is when “blending in” to help avoid detections. As an example, if we set `CommandLine` to `NULL` , the value that will be set upon process creation is the same with what’s passed in `ImagePathName` .



If we want to mimic/spoof the actual command line when `C:\Windows\System32\RuntimeBroker.exe` gets created normally, we can do it using the following code:

```
UNICODE_STRING CommandLine;
RtlInitUnicodeString(&CommandLine, (PWSTR)L"\C:\\Windows\\System32\\RuntimeBroker.exe
-Embedding");
```



The last parameter (`Flags`) is used to normalize the parameters by setting the value `RTL_USER_PROCESS_PARAMETERS_NORMALIZED` . When a process is created, some inputs are not even fully initialized yet. If this happens, there's a chance wherein the memories being accessed are just relative offsets of the structure describing the process and not the actual memory addresses.

> *If you're going to use `RtlCreateProcessParameters()` , which is the non-extended version, a call to `RtlNormalizeProcessParameters()` should be made to normalize the parameters.*

So here's what the code for `RtlCreateProcessParametersEx()` I ended up with:

```
RtlCreateProcessParametersEx(&ProcessParameters, &NtImagePath, NULL, NULL, NULL,
NULL, NULL, NULL, NULL, NULL, RTL_USER_PROCESS_PARAMETERS_NORMALIZED);
```

Cleanup Code

As a cleanup, we can use `RtlFreeHeap()` to free the memory that was allocated by `RtlAllocateHeap()` , and `RtlDestroyProcessParameters()` to deallocate the process parameters stored in the `RTL_USER_PROCESS_PARAMETERS` structure.

```
RtlFreeHeap(RtlProcessHeap(), 0, AttributeList);
RtlDestroyProcessParameters(ProcessParameters);
```

The Header File

Aside from developing the code, another challenge working with native APIs is finding the right structures and definitions to use. I tried to manually create the header file by scouring the Internet and compiling what I have found until every structure and definition that I needed are satisfied. However, it cost me some headaches so I just gave up.

As you're already aware, I keep on referencing the ntpsapi.h header file from **Process Hacker**. However, it does not contain every structure and definition that I needed. Even the whole phnt native API header files caused an error during compilation due to missing structures.

After a ton of searches, I ended up with the ntdll.h header file provided by x64dbg/TitanEngine. After including this header in my code, I'm relieved that it did not cause any compilation error.

While it contains everything I need for my "minimal" PoC, there are still some missing definitions that I found. For example, it only has the following flags for process creation compared to what **Process Hacker** has (listed above):

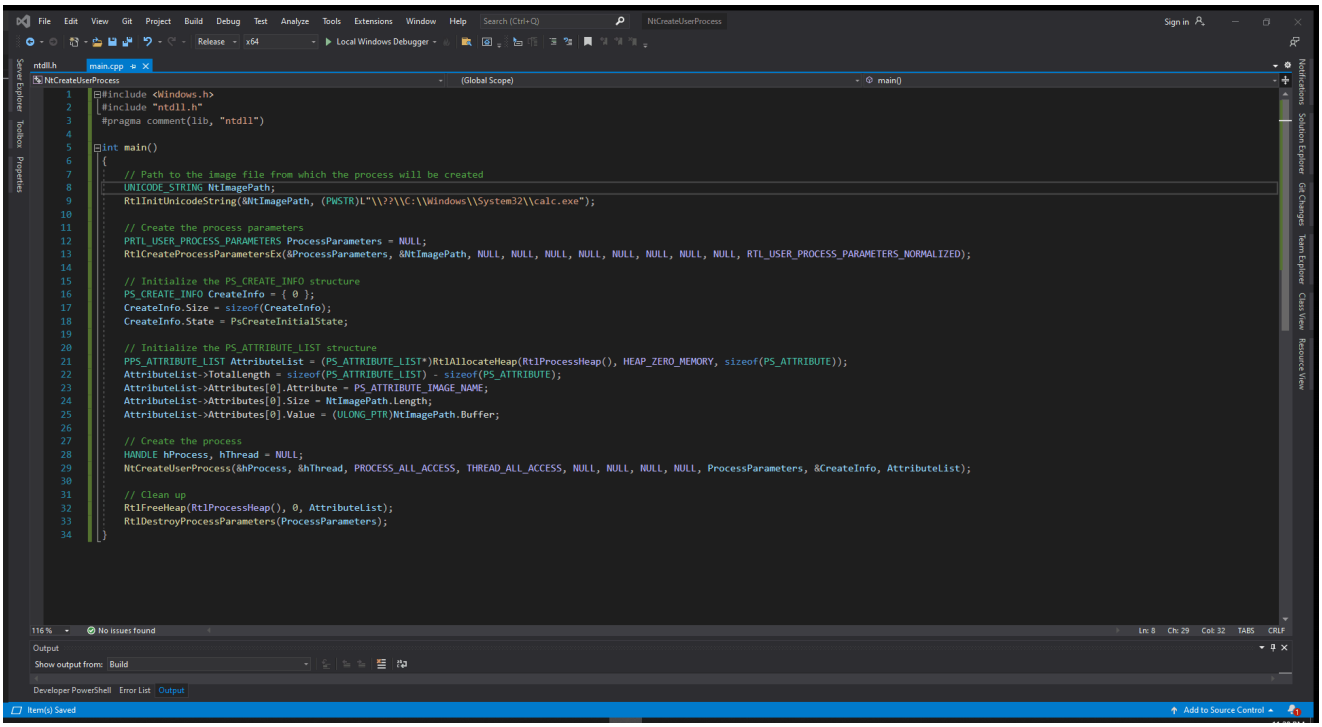
```
#define PROCESS_CREATE_FLAGS_BREAKAWAY              0x00000001
#define PROCESS_CREATE_FLAGS_NO_DEBUG_INHERIT       0x00000002
#define PROCESS_CREATE_FLAGS_INHERIT_HANDLES        0x00000004
#define PROCESS_CREATE_FLAGS_OVERRIDE_ADDRESS_SPACE 0x00000008
#define PROCESS_CREATE_FLAGS_LARGE_PAGES            0x00000010

// Only usable with NtCreateUserProcess (Vista+):
#define PROCESS_CREATE_FLAGS_LARGE_PAGE_SYSTEM_DLL  0x00000020
#define PROCESS_CREATE_FLAGS_PROTECTED_PROCESS      0x00000040 // Only allowed if the
calling process is itself protected
#define PROCESS_CREATE_FLAGS_CREATE_SESSION         0x00000080
#define PROCESS_CREATE_FLAGS_INHERIT_FROM_PARENT    0x00000100
```

The Minimal Code

After all the headaches and struggles, here's the "minimal" working PoC that I came up with:

```
#include <Windows.h>
#include "ntdll.h"
#pragma comment(lib, "ntdll")

int main()
{
        // Path to the image file from which the process will be created
        UNICODE_STRING NtImagePath;
        RtlInitUnicodeString(&NtImagePath, (PWSTR)L"\\??
\\C:\\Windows\\System32\\calc.exe");

        // Create the process parameters
        PRTL_USER_PROCESS_PARAMETERS ProcessParameters = NULL;
        RtlCreateProcessParametersEx(&ProcessParameters, &NtImagePath, NULL, NULL,
NULL, NULL, NULL, NULL, NULL, NULL, RTL_USER_PROCESS_PARAMETERS_NORMALIZED);

        // Initialize the PS_CREATE_INFO structure
        PS_CREATE_INFO CreateInfo = { 0 };
        CreateInfo.Size = sizeof(CreateInfo);
        CreateInfo.State = PsCreateInitialState;

        // Initialize the PS_ATTRIBUTE_LIST structure
        PPS_ATTRIBUTE_LIST AttributeList =
(PS_ATTRIBUTE_LIST*)RtlAllocateHeap(RtlProcessHeap(), HEAP_ZERO_MEMORY,
sizeof(PS_ATTRIBUTE));
        AttributeList->TotalLength = sizeof(PS_ATTRIBUTE_LIST) -
sizeof(PS_ATTRIBUTE);
        AttributeList->Attributes[0].Attribute = PS_ATTRIBUTE_IMAGE_NAME;
        AttributeList->Attributes[0].Size = NtImagePath.Length;
        AttributeList->Attributes[0].Value = (ULONG_PTR)NtImagePath.Buffer;

        // Create the process
        HANDLE hProcess, hThread = NULL;
        NtCreateUserProcess(&hProcess, &hThread, PROCESS_ALL_ACCESS,
THREAD_ALL_ACCESS, NULL, NULL, NULL, NULL, ProcessParameters, &CreateInfo,
AttributeList);

        // Clean up
        RtlFreeHeap(RtlProcessHeap(), 0, AttributeList);
        RtlDestroyProcessParameters(ProcessParameters);
}
```

And here's a demo of launching `C:\Windows\System32\calc.exe` using the above code.



```
#include <Windows.h>
#include "ntdll.h"
#pragma comment(lib, "ntdll")

int main()
{
    // Path to the image file from which the process will be created
    UNICODE_STRING NtImagePath;
    RtlInitUnicodeString(&NtImagePath, (PWSTR)L"\\??\\C:\\Windows\\System32\\calc.exe");

    // Create the process parameters
    PRTL_USER_PROCESS_PARAMETERS ProcessParameters = NULL;
    RtlCreateProcessParametersEx(&ProcessParameters, &NtImagePath, NULL, NULL, NULL, NULL, NULL, NULL, NULL, RTL_USER_PROCESS_PARAMETERS_NORMALIZED);

    // Initialize the PS_CREATE_INFO structure
    PS_CREATE_INFO CreateInfo = { 0 };
    CreateInfo.Size = sizeof(CreateInfo);
    CreateInfo.State = PsCreateInitialState;

    // Initialize the PS_ATTRIBUTE_LIST structure
    PPS_ATTRIBUTE_LIST AttributeList = (PS_ATTRIBUTE_LIST*)RtlAllocateHeap(RtlProcessHeap(), HEAP_ZERO_MEMORY, sizeof(PS_ATTRIBUTE));
    AttributeList->TotalLength = sizeof(PS_ATTRIBUTE_LIST) - sizeof(PS_ATTRIBUTE);
    AttributeList->Attributes[0].Attribute = PS_ATTRIBUTE_IMAGE_NAME;
    AttributeList->Attributes[0].Size = NtImagePath.Length;
    AttributeList->Attributes[0].Value = (ULONG_PTR)NtImagePath.Buffer;

    // Create the process
    HANDLE hProcess, hThread = NULL;
    NtCreateUserProcess(&hProcess, &hThread, PROCESS_ALL_ACCESS, THREAD_ALL_ACCESS, NULL, NULL, NULL, NULL, ProcessParameters, &CreateInfo, AttributeList);

    // Clean up
    RtlFreeHeap(RtlProcessHeap(), 0, AttributeList);
    RtlDestroyProcessParameters(ProcessParameters);
}
```

The full project can be found here.

Conclusion

That's it for this post! Again, I'm not an expert in Windows Internals so I'm happy to hear some corrections and additional information.

Before I end this, if you want to know the detailed and step-by-step procedure on how a process gets created, I suggest reading the **Flow of CreateProcess** section in "Windows Internals, Part 1 (7th Edition)".

A summary of what's in the book could also be read in this 2-part series: