

Creating Processes Using System Calls

 coresecurity.com/core-labs/articles/creating-processes-using-system-calls

When we think about EDR or AV evasion, one of the most widespread methods adopted by offensive teams is the use of system calls (syscalls) to carry out specific actions. This technique is so common and effective simply because most AVs/EDR have userland hooks to track and intercept requests userland processes make. However, we found that a key userland API, *CreateProcess*, is still extensively used even in offensive tools to create processes.

There has been some work on weaponizing *NtCreateUserProcess* so that it can be used on defended environments, but the reality is that few of these projects out there have managed to implement it in a way that is reliable and useful.

The problem is that creating the process is half the battle; if we try to create a notepad process using the *NtCreateUserProcess* syscall, we will quickly realize that it dies instantaneously. The reason for that is that if we want our newly created processes to function normally, we first need to notify the Windows Subsystem about it. If we do not do so, the application will *segfault* because it fails when calling the Win32 API.

When analyzing *CreateProcessW* WIN32 API call, it calls *kernelbase!CreateProcessInternalW*, which is where all the process-creation logic takes place. This includes notifying the Windows Subsystem about the newly created process. An excerpt of the code making the notification can be seen in the following figure.

Image

```
1583         local_1a0 = local_d68;
1584     }
1585     ret_val = CsrCaptureMessageMultiUnicodeStringsInPlace
1586             (&CapturedBuffer, uVar30, ppuVar33);
1587 }
1588 uVar26 = local_14ac;
1589 ntstatus = ret_val;
1590 if ((int)ret_val < 0) goto LAB_18000a5a8;
1591 CsrClientCallServer(ApiMsg, CapturedBuffer, 0x1001d, 0x218);
1592 if ((int)local_dbc < 0) {
1593     FUN_18002b300();
1594     local_14ac = local_dbc;
1595     goto LAB_18000b1c4;
1596 }
```

The API *ntdll!CsrClientCallServer* is responsible for sending a message to the CSRSS process, notifying it of the existence of the new process.

The notification occurs after the process has been created (in suspended mode) and before it is resumed. So, if we want to have a usable implementation of *NtCreateUserProcess*, we need to handle the notification process.

At first glance, it might seem that calling *ntdll!CsrClientCallServer* would be the solution. However, calling it is not that straightforward because it takes two complex structures as parameters.

There are a few open-source projects that come very close to achieving this, including [this one](#) and [this one](#). Also, ReactOS [contains](#) a lot of extremely useful code snippets that relate very closely to what Windows does.

After a lot of copying and reworking other people's code, reverse engineering, and debugging, I managed to successfully call *ntdll!CsrClientCallServer*, register the new process, and make the whole thing look like *NtCreateUserProcess*.

After doing that, I looked at the actual implementation of *ntdll!CsrClientCallServer* using Ghidra.

Image

```

Decompile: CsrClientCallServer - (ntdll.dll)
21  if (LdrpIsSecureProcess == '\0') {
22      if ((int)DataLength < 0) {
23          DataLength = -DataLength;
24          *(undefined2 *) ((longlong)ApiMessage + 4) = 0;
25      }
26      else {
27          *(undefined4 *) ((longlong)ApiMessage + 4) = 0;
28      }
29      *(undefined8 *) ((longlong)ApiMessage + 0x28) = 0;
30      *(ULONG *) ((longlong)ApiMessage + 0x30) = ApiNumber & 0xffffffff;
31      *(ULONG *) ApiMessage = (DataLength << 0x10 | DataLength) + 0x400018;
32      lVar1 = CsrPortMemoryRemoteDelta;
33      if (((*(byte *) ((longlong *) (in_GS_OFFSET + 0x60) + 3) & 2) == 0) ||
34          (((ApiNumber & 0xffff0000) != 0x20000 &&
35            ((ApiNumber & 0x100000000) == 0 ||
36              ((*(byte *) ((longlong *) (in_GS_OFFSET + 0x60) + 3) & 0x40) != 0)))))) {
37          if (CsrClientProcess == '\0') {
38              lVar1 = *(longlong *) (in_GS_OFFSET + 0x30);
39              uVar2 = *(undefined4 *) (lVar1 + 0x44);
40              uVar3 = *(undefined4 *) (lVar1 + 0x48);
41              uVar4 = *(undefined4 *) (lVar1 + 0x4c);
42              *(undefined4 *) ((longlong)ApiMessage + 8) = *(undefined4 *) (lVar1 + 0x40);
43              *(undefined4 *) ((longlong)ApiMessage + 0xc) = uVar2;
44              *(undefined4 *) ((longlong)ApiMessage + 0x10) = uVar3;
45              *(undefined4 *) ((longlong)ApiMessage + 0x14) = uVar4;
46              iVar5 = (*CsrServerApiRoutine)();
47          }
48          else {
49              if (CaptureBuffer != (_CSR_CAPTURE_BUFFER *)0x0) {
50                  *(longlong *) ((longlong)ApiMessage + 0x28) =
51                      (longlong)&CaptureBuffer->Size + CsrPortMemoryRemoteDelta;
52                  PointerCount = CaptureBuffer->PointerCount;
53                  CaptureBuffer->BufferEnd = (PVOID)0x0;
54                  PointerOffsetArray = &CaptureBuffer->PointerOffsetArray;
55                  while (currPointer = PointerOffsetArray, PointerCount != 0) {
56                      plVar7 = (longlong *)*currPointer;
57                      PointerCount = PointerCount - 1;
58                      PointerOffsetArray = currPointer + 1;
59                      if (plVar7 != (longlong *)0x0) {
60                          *plVar7 = *plVar7 + lVar1;
61                          *currPointer = (ULONG_PTR) ((longlong)plVar7 - (longlong)ApiMessage);
62                      }
63                  }
64              }
65              uVar6 = NtAlpcSendWaitReceivePort();
66              lVar1 = CsrPortMemoryRemoteDelta;
67              iVar5 = (int)uVar6;
68              if (CaptureBuffer != (_CSR_CAPTURE_BUFFER *)0x0) {

```

I found that it had the potential to be an interesting challenge to create my own version of CsrClientCallServer. Additionally, doing this would also have another benefit: if the NtAlpcSendWaitReceivePort syscall (line 65) was hooked I could bypass it.

The first obstacle I ran into is that the *ntdll!CsrClientCallServer* API uses two global variables, *CsrPortHandle* and *CsrPortMemoryRemoteDelta*.

Both are set by *ntdll!CsrpConnectToServer*, which is in charge of making the first connection to the CSRSS process. The technical details of how this is done are extremely well explained in Windows CSRSS Write Up: Inter-process Communication (part 1/3) and

Windows CSRSS Write Up: Inter-process Communication (part 2/3) blog posts by [Jooru](#).

In a nutshell, these two blogposts explain that Windows has a mechanism, named LPC, which allows local processes to communicate with one another.

So, to communicate with the Csr, you first need to open a connection to a “named port” by calling *ntdll!NtSecureConnectPort*. This will give you a port handle named *CsrPortHandle*.

To exchange large amounts of information, we need to create a section that will be mapped in both our process and the Csr. The difference between the local address and the remote address of this section is the *CsrPortMemoryRemoteDelta*.

I decided to implement *ntdll!CsrpConnectToServer*. However, after I implemented it I noticed that when I called *NtSecureConnectPort* the CSRSS refused my connection request with the status code 0xc0000041, which means STATUS_PORT_CONNECTION_REFUSED.

There are two plausible explanations for why this happened. The first one is that I messed up my implementation in some way and the second one is that the CSRSS knows that our process already has an existing connection established, so it refuses to open a new one.

You can find my faulty implementation of *CsrpConnectToServer* [here](#).

Either way, I decided to abandon that path, which meant that I needed to use the existing connection to the Csr to move forward. In order to do so, I needed to know the values of both *CsrPortHandle* and *CsrPortMemoryRemoteDelta*.

These two global variables have a fixed (relative) address inside the ntdll library, but that address changes from version to version. There are (at least) two ways of obtaining this address. The first option is to save the offset where they are stored for each version of ntdll, but this is hardly a practical approach. the second option is to parse the code section of ntdll, find instructions that reference these global variables, and, by using that reference, find their absolute addresses. Since I was inspired by [Revisiting a Credential Guard Bypass](#) by [itm4n](#), I went with the latter.

Instead of scanning the entire code section of ntdll, I only searched the beginning of the exported API that I was interested in. In this case, this was *ntdll!CsrClientCallServer*. I simply looked for the bytes that preceded the relative address of the global variables, then added the address of the next instructions (RIP) and got the absolute address of both global variables.

As an example, take these two instructions inside of *ntdll!CsrClientCallServer*.

Image

00007ffb`16a78a58 488b0de9311600	mov	rcx, qword ptr [ntdll!CsrPortHandle (7ffb16bdbc48)]
00007ffb`16a78a5f 488d442440	lea	rax, [rsp+40h]

We would then just need to find the bytes { 0x48, 0x8b, 0x0d } within *ntdll!CsrClientCallServer*, parse the next four bytes as an unsigned 32-bit number in little endian (e9311600 -> 0x1631e9), and add that number to the address of the next instruction (0x7ffb16a78a5f). This would give us the absolute address of *CsrPortHandle*, which is 0x7ffb16bdbc48.

After completing this step, I needed to deal with some more internal structures, which turned out to be fairly easy to do because the code from ReacOS was of great help. After that was done, I called the syscall *NtAlpcSendWaitReceivePort*, since I already had my own implementation of *CsrClientCallServer*.

Finally, I decided to implement *ntdll!CsrCaptureMessageMultiUnicodeStringsInPlace*, which is called by *kernelbase!CreateProcessInternalW* before calling *ntdll!CsrClientCallServer*.

This meant finding some more global variables and dealing with some more structures. Once again, ReacOS had almost all the code that I needed. Once I finished coding *CsrCaptureMessageMultiUnicodeStringsInPlace*, I had my own working implementation of *kernelbase!CreateProcessInternalW*, which relies exclusively on system calls and can be used to spawn virtually any process—sweet!

I also included several useful features like spoofing the parent process id, specifying the working directory, process parameters, and blocking non-Microsoft DLLs.

Just when I thought I was done, I read this article by Microsoft, *Using Process Creation Properties to Catch Evasion Techniques*.

To quickly summarize, it explained that the kernel-based process creation callback routine, which EDRs use to be notified of every new process so they can inspect it, is not actually triggered when the process is created, but rather when the first thread is inserted in the process.

Because the syscall *NtCreateUserProcess* does most of the work required to create a new process within the kernel, it also creates the first thread. This means that EDRs are notified of the new process before the syscall finishes.

The article also explains that the legacy syscall, named *NtCreateProcessEx*, does not create the initial thread, which means it doesn't trigger the callback right away. This allows several techniques like process doppelganging, process herpaderping and process ghosting.

Since there are already several high-quality implementations of all the techniques described above, I decided to instead focus on creating a regular process using this specific syscall and registering it with the Csr as before.

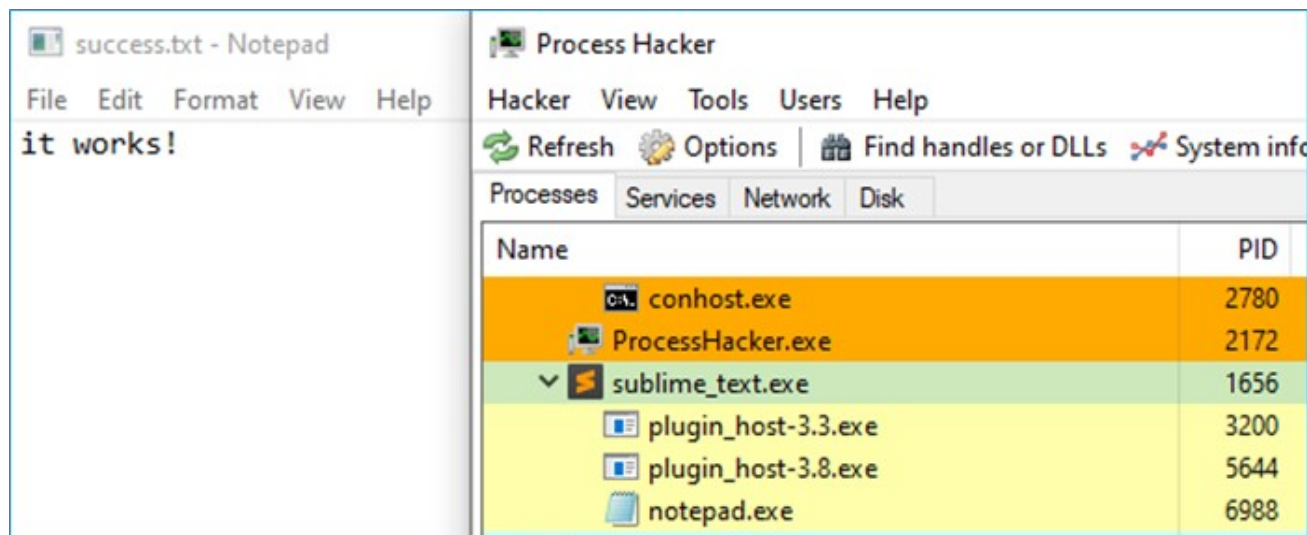
When you create a process using this syscall, you are responsible for, among other things, setting the process parameters. I followed the same approach as most public implementations and created the `RTL_USER_PROCESS_PARAMETERS` structure locally and then wrote it to the remote process. But to actually make it work, I noticed you need to adjust all the pointers that exist within that structure so that they make sense in the context of the new process instead of our preexisting one. Once that small caveat is sorted out, the implementation is standard. Luckily, registering the new process with the Csr is the same as when using `NtCreateUserProcess`.

You can find the final implementation [here](#).

Image

```
PS Z:\> .\createProcess.x64.exe --binary C:\Windows\notepad.exe --syscall NtCreateProcessEx --params "success.txt" --directory C:\Windows\Temp --ppid 1656
[+] Parent process
[i] PPID = 1656
[i] hProcess = 0x00000000000000a4
[+] New process created
[i] hProcess = 0x00000000000000a8
[i] hThread = 0x00000000000000ac
[i] PID = 6988
[i] TID = 5704
[i] PEB = 0x00000000e9976100
[+] Registered new process with the CSRSS
[+] Resumed process
PS Z:\>
```

Image



Conclusion

The hard truth is that `kernelbase!CreateInternalProcessW` is a very complex function that handles a lot of edge cases that I don't deal with. Consequently, every custom implementation of it will always have limitations. It is up to you if you want to operate

within those limitations in order to increase your opsec capabilities. It should be considered as another offensive resource for well-defended networks where the use of `CreateProcess` is not an option.