Windows Process Injection Techniques Catch Them All

Rajiv Kulkarni[1], Xiaofei (Rex) Guo[2], and Sushant Paithane[2]

Palo Alto Networks[1]
Confluera[2]

Abstract. Process injection (PI) in Windows has been a well known security topic for many years. It is used to gain more stealth because it does not create additional processes in the system that could attract unwanted attention from the defender. It is also used to bypass security products that have limited visibility into the injection behaviors. Since PI techniques use legitimate windows APIs, detecting them becomes a challenging task.

We provide a comprehensive detection for true PI techniques in real time. True PIs inject into target processes which are already running. Pre-execution injections such as AppInit and process hollowing are not in scope. The most comprehensive and up-to-date collection of true PIs was presented recently in 2019 Blackhat USA. Our detection can detect all PIs mentioned in that talk.

We classify PI behavior invariants that are unique compared to benign program behaviors but are common between various PI behaviors. Based on the behavior invariants, we divide PIs into five classes: (1) stateless detection (2) stateful detection (3) payload execution by DLL files (4) ROP gadgetwrite and execution (5) execution without unique syscalls We propose three detection algorithms to detect these three behavior invariants classes. Our detection algorithm relies on four detection primitives: (1) System call interception (2) Floating code detection (3) Library baselining (4) ROP detection Our detection is compatible with the latest Windows 10 x64 and previous versions. Evaluation results show that the detection is effective and has low overhead.

1 Introduction

Process injection (PI) has gained a lot of popularity due to its stealthy nature, ability to remain fileless (memory resident) and ability to evade security products. A comprehensive summary of PIs was given in Blackhat USA 2019 [15]. However, we could not find a comprehensive and realistic real time detection proposal for these PI techniques. We aim to fill this gap and our main contributions are:

- We comprehensively analyzed the PI behaviors by analyzing the documented and undocumented Windows APIs used by true PIs.
- We extracted PI behaviors that is different from benign program behaviors but is common among PI behaviors.
- We provide a comprehensive real-time PI detection technique for the publicly known PI techniques. Our technique detects all the publicly known "true" injections presented in the 2019 Blackhat USA talk[1].

[1] In [15], `MessagePassing` Write primitive and `DnsQuery_A` Callback execution method cannot be reliably achieved by an injection. Since they are not recommended by the authors, we did not discuss them in this paper. With the proper syscall hooked, our detection may be able to detect these since our detection is generic to the injection mechanisms

Although our detection covers the true and reliable PI techniques in the comprehensive presentation in Blackhat USA 2019 [15], we understand the PI techniques we consider in this paper may not cover all PI techniques and new techniques are invented or publicized over time. We elaborate the supported platforms for our detections in the paper and understand the scale of our experiments can be limited since we did not test the detection on all platforms and all environments across the world. We encourage the readers to send us corrections and we can improve our detection.

2 Related Work

To the best of our knowledge, we are not aware of public research that detect all publicly known PIs in real time. Memhunter was proposed to perform threat hunting in the memory using various heuristics to detect PIs [18]. The goal is to detect various process injections. However, it requires the defender to periodically scan the memory. Therefore it does not provide real-time detection and gives the attacker more time for evasion. If the defender scans more frequently, it will incur more CPU overhead.

3 Detection Primitives

This subsection describes our detection primitives. Based on our PI behavior analysis, we proposed four generic primitives to detect PIs. They are:

- System call (syscall) hooking
- Floating code detection
- Library baselining
- ROP detection

We will present the detection algorithm that leverages these primitives in the next subsection.

A naive way to perform real-time detection is to poll information from the memory continuously. There are several drawbacks:

- A significant amount of CPU overhead due to the continuous polling
- Considering most systems should not be infected during most their life time, lots of CPU cycles are wasted

Due to the above reasons, this approach is not practical in most scenarios.

A second approach is to monitor each and every primitive as it happens. One could have trigger-based monitoring for all the APIs and/or behaviors used in the process injection. There are several drawbacks:

- Process injection leverages existing windows APIs and many APIs are used by legitimate programs such as VirtualAllocEx, VirtualProtect, WriteProcessMemory, QueueUserAPC, etc.. This will again cause lots of CPU overhead due to the continuous event trigger.
- State tracking is required since the detection cannot conclude if the behavior is indeed process injection attempt before all the key APIs are called in the right sequence. This is especially problematic in heavily loaded servers.

Our approach is real-time, low CPU overhead, and low false positive. We uses a trigger based detection, i.e., we monitor a small set of events and we perform a sequence of analysis when such events happen.

We selected the triggered events with following characteristics:

– They need to be a core part of the process injection behavior
– They should be moderately or infrequently used by a benighn program
– When they are triggered, the system should be in a state where we can make a detection decision with high confidence

3.1 System Call Hooking

MS Windows implements syscalls in the OS kernel to allow user mode applications to perform operations on privilege resources. System call implementations can be found in `ntoskrnl.exe` (for native syscalls) and `win32k.sys` (for graphics related syscalls). `win32k.sys` has stubs pointing to actual implementations in `win32kfull.sys`, etc. Each syscall has a syscall number used by user mode programs to refer to. A mapping between a syscall number and the address of its implementation is stored in a syscall table. Windows contains two syscall tables: `KeSystemServiceDescriptorTable`, i.e., SSDT (for native syscalls) and `KeSystemServiceDescriptorTableShadow`, i.e., SSDT Shadow (for win32k syscalls). A user mode program call a `WriteProcessMemory` syscall to access kernel functionality as shown in the example below:

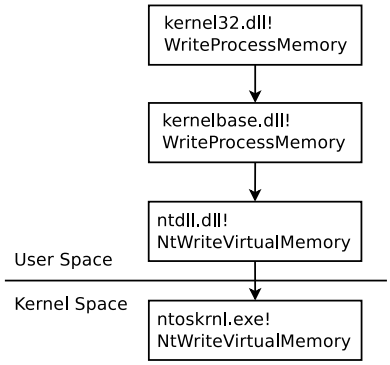

Fig. 1: Windows `WriteProcessMemory` call flows from user space to syscall

If a malicious process wants to perform process injection, it uses syscalls [15]. Therefore we intercept syscalls that are required to perform PI in real time by hooking these syscalls. Although syscalls are maintained in SSDT in the kernel, hooking SSDT (modifying the function pointers in the SSDT) are difficult in newer Windows versions due to PatchGuard.

We hook syscalls by leveraging Event Tracing for Windows (ETW) [16]. ETW is a tracing and debugging framework that can log Kernel and/or application defined events. There are three components in the framework: event provider, event consumer, and event controller. Event providers provide real-time data streaming and write to files. Event consumer consumes the data. Event controller binds a consumer and provider to a trace session and is responsible for managing the provider. There are default ETW sessions in the kernel and one of them is Circular Kernel Context Logger which logs syscalls. However, we did not find a way to monitor syscall arguments using existing allowed configurations in ETW.

There are different ways to hook system calls. As a proof of concept, we use the technique described in ByePG [10] When an ETW session is created, each active session has an associated `WMI_LOGGER_CONTEXT` structure. One member of this structure is called `GetCPUClock` which is a function pointer that is called whenever an event needs to be logged, since the current time is logged. We change this function pointer for the `WMI_LOGGER_CONTEXT` structure of CKCL (Circular Kernel Context Logger) and make it point to a stub in our driver.

We also enable CKCL to log all syscalls and this forms the basis of our hook. Other system call hooking implementation should work as well since our algorithm is generic.

3.2 Floating Code Detection

We detect if an executable memory region is backed either an executable or a DLL. We achieve this by obtaining the `PIMAGEHLP_MODULE64 ModuleInfo` using the target address in victim's memory. For PIs that executes without unique syscalls, i.e., Service Control method, we scan the service process memory periodically for floating code [2].

3.3 Library Baselining

We build a library baseline when the process started. When we get the library path, we check if it is the same as the path list of previously loaded libraries of that process. We alert if the path is not in the list. Since we can get the library path, one can certainly create a whitelist or blacklist based on the library path to make the baseline more accurate if he/she understand their environment.

3.4 ROP Detection

ROP gadgets are typically short code sequences ended with a `ret` instruction. Since these `ret` instructions does not have corresponding `call` instructions and the return address is random from branch predictor's perspective, branch prediction will almost certainly fail. Therefore one can see a huge differences in terms of the number of branch prediction failure per instruction between the benign program and ROP payload. We use three hardware performance counters (HPCs) to measure the occurrences of hardware events, i.e., the total number of instructions, the total number of branch instructions, and the number of mipredicted branch instructions. We use the total number of misprected branch instructions as the trigger counter which triggers a call back. Our heuristic is similar to the one used in [4].

4 Process Injections Techniques and Detections

This section briefly explains each process injection technique and the corresponding detection algorithms. We break down the PIs into three attack primitives:

- Allocation: memory is allocated in the target memory
- Writing: payload is written to the target memory
- Execution: payload is executed in the target memory

We observe commonalities between different PI techniques. Based on which and how the primitives are used, we classify the injection techniques into five behavior categories. In this section, we briefly describe the behavior categories and the corresponding PI techniques. For each PI technique, we also describe the detection algorithm. For full details of the PI techniques, please refer to the 2019 Blackhat USA talk [15].

These PIs can use non-ROP shellcode and/or ROP shellcode. The descriptions and detection algorithms written in this section are based on non-ROP shellcode except Ghost-Writing technique. For all PI variations that uses ROP shellcode, the reader can refer to our detailed explanation in section 4.4 which describes Ghost-writing.

By analyzing the behavior categories, our detection is generic and can potentially detect future injection methods. We focus on 64-bit processes on Windows 10 machines with the latest security features.

[2] This PI method can only inject into service processes

Table 1: PI Behavior Categories and Techniques

Technique	Pre-execution (without Allocation)[a]	Execution
PI techniques (Stateless Detection)		
CreateRemoteThread [15]	`WPM(payload)`	`CreateRemoteThread`

Suspend-Inject-Resume (SIR)[c][15]	`SuspendThread`
	`WPM(payload)`[b]
		`SetThreadContext` or
		`NtQueueApcThread(SetThreadContext)`
		`ResumeThread`

QueueUserAPC	`WPM(payload)`	`QueueUserAPC`
		`NtQueueApcThread`

Atom Bombing [3]	`GlobalAddAtomA`
	`SuspendThread`
	`NtQueueAPCThread(GlobalGetAtomNameA)`
		`NtSetContextThread`
		`ResumeThread`

Stack Bombing [15]	`SuspendThread`
	`NtQueueAPCThread(memset)`
	`NtQueueAPCThread(memove)`
		`ResumeThread`

PI techniques (Stateful Detection)		
PROPagate [5]	`WPM(payload)` `SetPropA`	`PostMessage`

SetWindowLong [7]	`WPM(payload)`
	`WPM(CTray)`
	`SetWindowLongPtr`
		`SendNotifyMessageA`

Unmap + Overwrite [12]	`NtSuspendProcess`
	`NtUnmapViewOfSection`
	(Patched APIs in memory)[d]
	`NtMapViewOfSection`
	`FlushInstructionCache`
		Victim to call
		the patched API[c]

Kernel Control Table [14]	(`NtQueryInformationProcess`)
	`WPM(target_payload)`
	`WPM(target_kct)`
	`WPM(pbi.PebBaseAddress`
	`+ OFFSETOF(PEB, KernelCallbackTable))`
		`SendMessage`

USERDATA [6]	`WPM(payload)`
	`WPM(target_cw)`
	`WPM(udptr)`
		`SendMessage`

Ctrl-Inject [14]	`WPM(payload)` `WPM(kernelbase!singlehandler)`	`PostMessageA` `SendInput`
ALPC Callback [11]	`WPM(payload)` `WPM(TP_CALLBACK_ENVIRON)`	`NtConnectPort`
WNF Callback [9]	`WPM(Payload)` `WPM(WNF_USER_SUBSCRIPTION)`	`NtUpdateWnfStateData`
Payload execution by loading DLLs from Disk		
CreateRemoteThread with LoadLibrary [15]	`WPM(payload)`	`CreateRemoteThread` `(LoadLibrary(DLLPath))`
Windows Hook [15]	`SetWindowsHookExA`	`PostThreadMessage`
Repetitive ROP gadget write and execution		
Ghost-Writing [1]	`SuspendThread`	`SetThreadContext` `ResumeThread`
Payload execution without syscall		
Service Control [8]	`WPM(payload)`	`ControlService`

[a] We did not include allocation primitives in pre-execution to avoid repetitiveness since allocation primitives used in these injections are similar or interchangeable.
[b] We use WPM for `WriteProcessMemory` for brevity.
[c] SIR stands for Suspend-Inject-Resume and is also known as Thread Execution Hijacking.
[d] As an example, an attacker can patch `NtClose` in ntdll.dll in victim's memory [15].

Algorithm 1 PI Detection Algorithm

```
1: // aPid is attacker pid, vPid is victim pid. alert() API will exit the program and send alerts.
2: recordHistoryList = [WriteProcessMemory, NtUnmapViewOfSection, NtUserSetWindowLongPtr, NtSuspendThread, NtUserSetProp]
3: statelessExecutionList = [CreateRemoteThread, NtSetContextThread, NtQueueAPCThread]
4: statefulExecutionList = [NtUserMessageCall, NtConnectPort, NtUpdateWnfStateData, NtMapViewOfSection]
5: function DetectionFromKernelCallback
6:     if syscall in recordHistoryList then
7:         if syscall == WriteProcessMemory then
8:             if callerPid != targetPid then
9:                 save(callerPid, targetPid, targetMemAddr); // saves memory address to a size N list indexed by callerPid
10:            end if
11:        else if syscall == NtUnmapViewOfSection then
12:            save(callerPid, targetPid); // saves pids to a size N list indexed by targetPid
13:        else if syscall == NtSuspendThread then
14:            save(callerPid, targetPid);
15:        else if syscall == NtUserSetProp then
16:            save(callerPid, targetMemAddr);
17:        else
18:            save(callerPid, handle, targetMemAddr);
19:        end if
20:    else if syscall in statelessExecutionList then
21:        if syscall == NtQueueAPCThread then
22:            // check if the second argument APC routine maps to GlobalGetAtomA in kernel32.dll
23:            if funcMapTo("GlobalGetAtomA", vPid, addr) == true then
24:                alert(aPid, vPid, addr, "AtomBombing");
25:            end if
26:            if funcMapTo("memset", vPid, addr) || funcMapTo("memmove", vPid, addr) then
27:                alert(aPid, vPid, addr, "StackBombing");
28:            end if
29:        end if
30:        if isFloatingCode(vPid, addr) == true then
31:            alert(aPid, vPid, addr, "FloatingCode", syscallName);
32:        end if
33:    else if syscall in statefulExecutionList then
34:        if syscall == NtMapViewOfSection then
35:            // check the NtMapViewOfSection has a corresponding NtUnmapViewOfSection
36:            if isFloatingCode(pid, addr) == true && hasMatchingUnmap(aPid, vPid) == true then
37:                alert(aPid, vPid, addr, "FloatingCode", syscallName);
38:            end if
39:            if callerPid != targetPid then
40:                save(callerPid, targetPid, targetMemAddr); // saves memory address to a size N list indexed by callerPid
41:            end if
42:        end if
43:        tuples = findvPidAddr(aPid) // find the victim pid and target address
44:        for int i = 0; i < sizeof(tuples); i++ do
45:            vPid, addr = tuples[i]
46:            if isFloatingCode(vPid, addr) == true then
47:                alert(aPid, vPid, addr, "FloatingCode", syscallName);
48:            end if
49:        end for
50:    end if
51:    if syscall == CreateRemoteThread || syscall == NtUserSetWindowsHookEx then
52:        if withinLibraryBaseline() == false then
53:            alert(aPid, vPid, addr, "Malicious DLL")
54:        end if
55:    end if
56:    if syscall == NtUserPostMessage then
57:        if isKeyPress == 'c' && isCtrlSentFromNtUserSendInput == true then
58:            vPid = getWindowThreadProcessId(handle)
59:            if isFloatingcode(vPid, addr) == false then
60:                alert(aPid, vPid, addr, "FloatingCode", syscallName);
61:            end if
62:        end if
63:        tuples = findvAddr(aPid) // find the victim pid and target address
64:        for int i = 0; i < sizeof(tuples); i++ do
65:            addr = tuples[i]
66:            if isFloatingCode(vPid, addr) == true then // we get vPid from NtUserPostMessage
67:                alert(aPid, vPid, addr, "FloatingCode", syscallName);
68:            end if
69:        end for
70:    end if
71: end function
72: function DetectionFromROP
73:    tuples = findaPidAddr(vPid) // find the attacker pid and target address
74:    for int i = 0; i < sizeof(tuples); i++ do
75:        vPid, addr = tuples[i]
76:        if isFloatingCode(vPid, addr) == true then
77:            alert(aPid, vPid, addr, "FloatingCode", syscallName);
78:        end if
79:    end for
80: end function
```

4.1 PI techniques (Stateless Detection)

This class of PIs can be detected with syscalls and the syscall arguments that are unique to the technique. Table 1 contains the full list of techniques in this class.

CreateRemoteThread An attacker can use a classic allocation and write primitive such as `VirtualAllocEx` and `WriteProcessMemory` before calling `CreateRemoteThread` to execute the malicious payload [15] as shown in Table 1. A variation of this technique is the infamous reflective DLL injection [13].

Related logic in algorithm 1:

- Call back on `CreateRemoteThread` to get the process handle and target execution address
- Use the victim process' pid and the target execution address to check floating code
- If it sees floating code, it sends an alert indicating the offending process with attack process' pid.

Suspend-Inject-Resume (SIR) An attacker can use a classic allocation primitive to allocate a memory for the payload [15]. Then it suspends the thread using `SuspendThread` and then set the registers in target thread's context, especially the `RIP` and/or `RSP` registers in the target thread's context. Then it resumes the thread using `ResumeThread`.

Detection algorithm:

- From `SetThreadContext` call back, we get the attacker and victim pids and the thread context.
- We dereferences the `RIP` in the thread context and check if it is pointing to floating code.

QueueUserAPC An attacker can use allocation and write primitives to write the payload [15]. Then use `QueueUserAPC` to control RCX or `ntdll!NtQueueAPCThread` to control RCX, RDX, and R8 to ask the victim to execute the payload as an APC routine.

Detection algorithm:

- From `NtQueueAPCThread`, we get the attacker and victim pid and the target address
- We dereferences the address and check if it is pointing to floating code.

Atom Bombing Although this attack is limited to threads that are in alertable state, it is goal is to avoid the usage of `WriteProcessMemory` to achieve more stealth [3]. The attack repetitively writes the ROP payload into the target using `NtQueueAPCThread` to register `GlobalGetAtomNameA`. After the full payload is written, it uses `NtSetContextThread` to set up the RIP and RSP to use stack pivot to execute the full ROP payload.

This attack eliminates `WriteProcessMemory` as a write primitive which is often monitored by security software. Attacker uses global Atom tables to get malicious code into target process memory. Attacker writes the malicious payload as strings into Atom tables using `GlobalAddAtom`. Attacker then queues an APC to the target thread with the APC routine pointing to `GlobalGetAtomA`. Thus, when the APC is run in the target, the payload is copied into target memory. To complete the attack, the attacker then uses `SetThreadContext` to execute the payload.

Detection algorithm:

- `NtQueueAPCThread` triggers a call back and we obtain victim pid and address pointing to the APC routine.
- Then we check if the address is pointing to `GlobalGetAtomA` [3] in kernel32.dll

[3] We also check if it is pointint to `GlobalGetAtomW`

Stack Bombing In this attack, the attacker performs the following actions. Attacker suspends the thread using `SuspendThread`. Attacker writes the payload one byte at a time using `NtQueueAPCThread(VictimThread, GetProcAddress(ntdll, ‘‘memset"), targetExecution, payloadSrc)`. The attacker can overwrite the stack (especially the stack) and write a ROP chain. Note that the attacker can also use `memmove` to achieve atomic writing. Attacker resumes the thread using `ResumeThread`. Thus when the execution is resumed, ROP chain written by the attacker is executed.

Detection algorithm:

- From `NtQueueUserAPCThread` call back, we get the second argument and check if it is map to `memset` or `memmove` in kernel32.dll

4.2 PI techniques (Stateful Detection)

This class of PIs can be detected by performing a historical lookup of syscalls based on a unique syscall trigger. This differs from the previous class of PI techniques in the way we detect them. The previous section of PI techniques were detected by analyzing unique syscalls in a stateless manner. However, for this class of PIs, we would have to look back at the system call activity of a suspected attacking process after a particular system call is encountered which acts as a trigger.

This class of PIs follow the general pattern:

1. Allocating memory in the target process
2. Writing payload to target memory
3. Overwriting callback function pointers to point to the payload memory
4. Triggering the callback via specific APIs

PROPagate An attacker can use allocation and write primitives to write the payload [5]. Then the attacker finds a subclassed window in the target process and reads its `UxSubclassInfo` property [4]. The attacker clones this structure and modifies the clone to set its virtual function to point to malicious payload. Attacker writes this property structure back into victim. Then the attacker sets the target window's `UxSubclassInfo` to point to this new structure using `SetProp`. The payload is then triggered used `PostMessage`

Detection algorithm:

- `PostMessage` uses `NtUserPostMessage` syscall which triggers a callback.
- We use `GetWindowThreadProcessId` using the handle.
- Then we look back the dialog item and the address to subclass item saved by `NtUserSetProp` syscall used by `SetProp`. Dereference this to find the pointer to payload and check for floating code

SetWindowLong An attacker can use allocation and write primitives to write the payload [7]. Then the attacker writes a CTray Object to target memory. The object should contain a pointer 1 pointing to pointer 2 which points to the payload. The attacker then calls `SetWindowLongPtr(<handle to a window of class Shell_TrayWnd>, 0, <ptr to the malicious CTray object>)`. After that, the payload is triggered by `SendNotifyMessageA`.

Detection algorithm:

- From `NtUserMessageCall` syscall used by `SendNotifyMessageA`, we get the handle to the window and the victim PID from the window.
- We search through `NtUserSetWindowLongPtr` historical records to get the target address.
- We check floating code using the target address.

[4] An attacker can use other properties [5]

Unmap + Overwrite In this attack, the attacker performs the following actions. The attacker suspends the target process using `NtSuspendProcess`. Then attacker clones a mapped memory section from the victim (for eg, ntdll.dll) into his own address space. The attacker patches and modifies the clone to insert malicious code. The attacker unmaps the target memory section from the victim and maps his cloned version into the victim using `NtUnmapViewOfSection` and `NtMapViewOfSection` respectively. Then the victim process is resumed using `NtResumeProcess`.

Detection algorithm.

- We save the attacker and victim pids from `NtUnmapViewOfSection` call back
- We use the target address, attacker and victim pids from `NtMapViewOfSection` to check floating code if the attacker and victim pid has a matching pair in `NtUnmapViewOfSection`.

Kernel Control Table An attacker first needs to use `FindWindow` and `OpenWindow` to identify the target window object [17]. An attacker allocates two memory regions in the victim memory. Then one memory region will be written with the payload. The other memory region will be written with a kernel control table (KCT). The `KCT.__fnCOPYData` field contains the address of the payload. After this, the attacker gets the process information structure using `NtQueryInformationProcess` and overwrites KCT pointer in PEB to point to the KCT attacker created. Finally, it uses `SendMessage` to trigger the execution.

Detection algorithm:

- Intercept `NtUserMessageCall` and get handle to the window and attacker pid
- Get target pid using `GetWindowThreadProcessId(handle)`
- Check the history of `WriteProcessMemory` calls with same attacker pid and target pid and check for floating code.

USERDATA This method overwrites a function pointer in the conhost.exe process associated with a console application. The attacker writes malicious code into target using any allocate and write primitive. Then the attacker gets pointer to the user data virtual table using `GetWindowLongPtr(..., GWLP_USERDATA)` Attacker reads the virtual table and gets the pointer to console dispatch table. Attacker copies the console dispatch table and modifies the pointer for GetWindowHandle to point to malicious code. Attacker writes the modified structures back to the target memory. The target payload could be triggered using `SendMessage(..., WM_SETFOCUS, ...)`.

Detection algorithm:

- From `NtUserMessageCall` call back, we get handle to the window and attacker pid
- Get attacker pid using `GetWindowThreadProcessId(handle)`. If the target PID belongs to conhost.exe, proceed further.
- Check the history of `WriteProcessMemory` calls with same attacker pid and target pid and check for floating code.

Ctrl-Inject This method overwrites the function pointer that handles "ctrl-c" input to console applications. The attacker writes the payload using any allocate and write primitive [14]. Then attacker encodes the target address using `RtlEncodeRemotePointer`. Then the encoded target address is written to `kernelbase!SingleHandler` which contains a list of pointers that points to the handling routine for "ctrl-c". Attacker triggers the execution by simulating "ctrl-c" keypress using `SendInput` (for ctrl) and `PostMessageA` (for 'C').

Detection algorithm:

- We check `NtUserPostMessage` to see the key pressed is "c" and check `NtUserSendInput` for LPINPUT (`ki.wVk`) to see if the value is `VK_CONTROL` which indicates a "ctrl" key is sent.
- Get target pid using `GetWindowThreadProcessId(handle)`. If the target PID belongs to a console process, proceed further.
- Check the history of `WriteProcessMemory` calls with same attacker pid and target pid, decode the pointer using `RtlDecodeRemotePointer` and check for floating code.

Asynchronous Local Inter-Process Communication (ALPC) Callback This attack is limited to processes having ALPC ports. The steps are: Attacker uses any allocation and write primitives to write the payload into target process. Attacker searches for ALPC control data structure which contains a callback structure (TP_CALLBACK_ENVIRON) in the target. Attacker overwrites the call back pointer in this structure to point to the malicious code. Attacker triggers execution by attempting to connect to ALPC ports using `NtConnectPort`.

Detection algorithm:

- From the `NtConnectPort`, we get the attacker pid.
- Then we search `WriteProcessMemory` and `NtMapViewOfSection` calls originating from that attacker pid and check for floating code.

Windows Notfication Facility (WNF) Callback This attack is aimed at processes that have $Shell\_TrayWnd$ window object [9]. The attacker opens the target process and gets a user subscription. Then the attacker uses any allocate and write primitive to write malicious payload into target memory. Attacker overwrites the `WNF_USER_SUBSCRIPTION` callback pointer to point to the payload. Finally, attacker calls `NtUpdateWnfStateData` to trigger the execution.

Detection algorithm:

- From the `NtUpdateWnfStateData` call back, we get attacker pid
- We search historical `WriteProcessMemory` and `NtMapViewOfSection` calls originating from the attacker pid and check for floating code.

4.3 Payload Execution by DLL files

The attacker can also use APIs to load DLLs from the disk. This class includes CreateRemoteThread(LoadLibraryA) and Windows Hook.

CreateRemoteThread with LoadLibrary This attack uses `CreateRemoteThread` to execute `LoadLibraryA` and give the target DLL as the destination path.

Detection algorithm:

- When CreateRemoteThread is called, we dereference the fourth and fifth argument and check if it is calling `LoadLibraryExA` and alert the target dll in the fifth argument as suspecious

Windows Hook This attack gets a handle of the malicious library using `LoadLibraryA` [15]. Then it gets a hook function for `GetMegProc` using `GetProcAddress` so it can use `SetWindowsHookExA` to hook the dll handle to the hook function. Finally, it triggers the execution using `PostThreadMessage`.

Detection algorithm:

- intercept `NtUserSetWindowsHookEx` to get the module handle
- use `NtVirtualQueryMemory` to get the file path for that module and alert the injection attempt

4.4 ROP Gadget Write and Execution

Although many other techniques we have already discussed can also use ROP payload, we will first describe Ghost-Writing [1] since it uniquely avoids using `WriteProcessMemory`. This attack uses a series of `SetThreadContext` calls to write a ROP chain in target process memory. It uses write and loop ROP gadgets to write the rest of the ROP payload in the target memory one by one. The final step involves setting RSP and RIP properly to trigger the ROP chain. This technique does not use `WriteProcessMemory` which is usually monitored by security software. The detection for Ghost-writing also applies to the ROP variations from other techniques.

Detection algorithm

- When Ghost-writing starts to execute the ROP payload, it is detected by the ROP detction primitive.
- From the ROP detection triggered call back, we get the victim pid.
- Then we search the historical record from `NtSuspendThread` to find the attack process that issue `NtSuspendThread` to the victim process.

4.5 Execution without Unique Syscalls

Service control is a payload execution through API technique [8]. It uses allocation and write primitive to write the payload into victim's memory. Then it locates the Internal Dispatch Entry (IDE) and overwrites the ControlHandler pointer with a pointer to the payload, changing the ServiceFlags, writing back to memory and triggering execution via `ControlService`. We have analyzed the `ControlService` API. However, this API does not invoke any unique syscalls. Therefore syscall interception cannot be used to trigger the detection mechanism. To detect this type injection, one either needs to scan the memory region of processes and detect floating code or one needs to relies on ROP detection if the injection uses ROP payload. Fortunately, this PI is limited to service processes as victims. Therefore we scan the memory region for floating code for service processes.

5 Software Architecture and Design

We divide the detection framework into two major components: kernel mode driver and user mode component.

5.1 Kernel Mode Driver

The driver is responsible for hooking the system calls leveraging the ByePG technique. When a system call is called in the kernel, we redirect the execution to a stub inside our driver. Each system call has its own stub in our driver, which extracts the required arguments from the system calls. We also get the process id of the process which called the syscall. We populate a "communication packet" with the information we captured (attacker PID, victim PID, memory address being written to, any other specific information), which is sent to the user-mode component for further processing. We then call the original system call with the same arguments to resume the execution flow.

5.2 User Mode Program

This component receives the communication packet from the driver which contains PIDs of suspected attacker and victim, the memory address being modified/accessed, the syscall name and other syscall specific information. We use the memory address to determine floating code, function pointer overwrites, etc, based on algorithm 1.

5.3 Communication Between User and Kernel Mode

In Windows driver programming, user mode must always initiate the communication to the driver in order to get something back from kernel mode. However, we need our detection framework to be real-time and hooking of system calls is being done in kernel mode. Hence we need a way to send data back to user mode without user mode explicitly requesting for data during every syscall. To do this, we use a method called Inverted call model [2]. Without going into implementation details, we send multiple IRPs with a custom IOCTL code from user mode. These requests are stored in a queue both in user mode and kernel mode. When the kernel mode driver encounters a syscall, it extracts relevant information, pulls an IRP from its queue and populates it with the information it wants to send to user mode. Then the IRP is marked as completed and removed from the queue. The user mode program which polls its queue for completed IRP packets will see the completed request and extract data for further processing. The queue size is controlled by the user mode program.

6 Conclusion

In this paper, we create comprehensive API and behavior analyses for PIs and proposed a PI detection technique. Our detection algorithm leverages four detection primitives and they can detect all the PI techniques in [15] and their mix and match variants. We implemented the detection algorithm on Windows 10 x64 and show that they are practical.

References

1. A paradox: Writing to another process without openning it nor actually writing to it, 2007. http://blog.txipinet.com/2007/04/05/69-a-paradox-writing-to-another-process-without-openning-it-nor-actually-writing-to-it/.
2. The inverted call model in kmdf, 2013. https://www.osr.com/nt-insider/2013-issue1/inverted-call-model-kmdf/.
3. Atombombing: Brand new code injection for windows, 2016. https://blog.ensilo.com/atombombing-brand-new-code-injection-for-windows.
4. Sigdrop: Signature-based rop detection using hardware performance counters, 2016. https://arxiv.org/pdf/1609.02667.pdf.
5. Propagate - a new code injection trick, 2017. http://www.hexacorn.com/blog/2017/10/26/propagate-a-new-code-injection-trick/.
6. Windows process injection: Consolewindowclass, 2018. https://modexp.wordpress.com/2018/09/12/process-injection-user-data/.
7. Windows process injection: Extra window bytes, 2018. https://modexp.wordpress.com/2018/08/26/process-injection-ctray/.
8. Windows process injection: Service control handler, 2018. https://modexp.wordpress.com/2018/08/30/windows-process-injection-control-handler/.
9. Windows process injection: Windows notification facility, 2018. https://modexp.wordpress.com/2019/06/15/4083/.
10. Byepg, 2019. https://github.com/can1357/ByePg.
11. Windows process injection: Print spooler, 2019. https://modexp.wordpress.com/2019/03/07/process-injection-print-spooler/.
12. Pavel Asinovsky. Diving into zberp's unconventional process injection technique, 2016. https://securityintelligence.com/diving-into-zberps-unconventional-process-injection-technique/.
13. Stephen Fewer. Reflectivedllinjection, 2013. https://github.com/stephenfewer/ReflectiveDLLInjection.
14. Rotem Kerner. Ctrl-inject, 2018. https://blog.ensilo.com/ctrl-inject.
15. Itzik Kotler and Amit Klein. Process Injection Techniques - Gotta Catch Them All. Blackhat USA Briefings, 2019. https://www.blackhat.com/us-19/briefings/schedule/#process-injection-techniques---gotta-catch-them-all-16010.

16. Microsoft. About Event Tracing. Windows Dev Center, 2018. https://docs.microsoft.com/en-us/windows/win32/etw/about-event-tracing.
17. Office 365 Threat Research and Microsoft Defender ATP Research Team. Finfisher exposed: A researcher's tale of defeating traps, tricks, and complex virtual machines, 2018. https://www.microsoft.com/security/blog/2018/03/01/finfisher-exposed-a-researchers-tale-of-defeating-traps-tricks-and-complex-virtual-machines/.
18. Marcos Oviedo. Memhunter - Automated hunting of memory resident malware at scale. Defcon Demo Labs, 2019. https://github.com/marcosd4h/memhunter.