Weaponizing Privileged File Writes with the USO Service - Part 2/2

itm4n.github.io/usodllloader-part2

August 19, 2019

In the previous post, I showed how the **USO client** could be used to interact with the **USO service** and thus have it load the `windowscoredeviceinfo.dll` DLL on demand with the `StartScan` option. I wasn't totally satisfied with this though. So, I reverse engineered a part of the client and the server in order to **replicate its behavior as a standalone project** that could be reused in future exploits. This is what I'll try to show and explain in this second part.

USO client - Static analysis

Although I also used Ghidra during my research process, I'll stick to IDA in this demonstration for consistency and because of its debugging capabilities.

Before opening `usoclient.exe` in IDA, I downloaded the corresponding PDB file with the following command. Theoritically, IDA will do this automatically but I found that it doesn't always work. The PDB file can then be loaded with `File > Load File > PDB File...`.

`symchk` comes with Windows SDK and is generally located in `C:\Program Files (x86)\Windows Kits\10\Debuggers\x64\`.

```
symchk /s "srv*c:\symbols*https://msdl.microsoft.com/download/symbols"
"c:\windows\system32\usoclient.exe"
```

Note: *PDB stands for "Program Database". Program database (PDB) is a proprietary file format (developed by Microsoft) for storing debugging information about a program (or, commonly, program modules such as a DLL or EXE).* Source: Wikipedia

`usoclient.exe` is now opened in IDA and the symbols are loaded, where do we go from here? Well, here the starting point is quite obvious. We know that the `StartScan` option is a valid "trigger" so, we will naturally look for occurrences of this string in the binary and enumerate all the `Xrefs` to find out where it's used.

```
.rdata:00007FF62D36D908 aStartscan:                             ; DATA XREF: Perform(
.rdata:00007FF62D36D908                                         ; PerformOperationOnS
.rdata:00007FF62D36D908                 unicode 0, <StartScan>,0
.rdata:00007FF62D36D91C                 align 20h
```

xrefs to aStartscan

Direction	Type	Address	Text
Up	o	PerformOperationOnManager(IUpdateMana...	lea rdx, aStartscan ; "StartScan"
Up	o	PerformOperationOnSession(IUsoSessionCo...	lea rdx, aStartscan ; "StartScan"

OK | Cancel | Search | Help

Line 1 of 2

The `StartScan` string is used inside two functions: `PerformOperationOnSession()` and `PerformOperationOnManager()`. Let's check the first one and generate the corresponding pseudocode.

IDA View-A | Pseudocode-A | Strings window | Hex View-1 | Structures

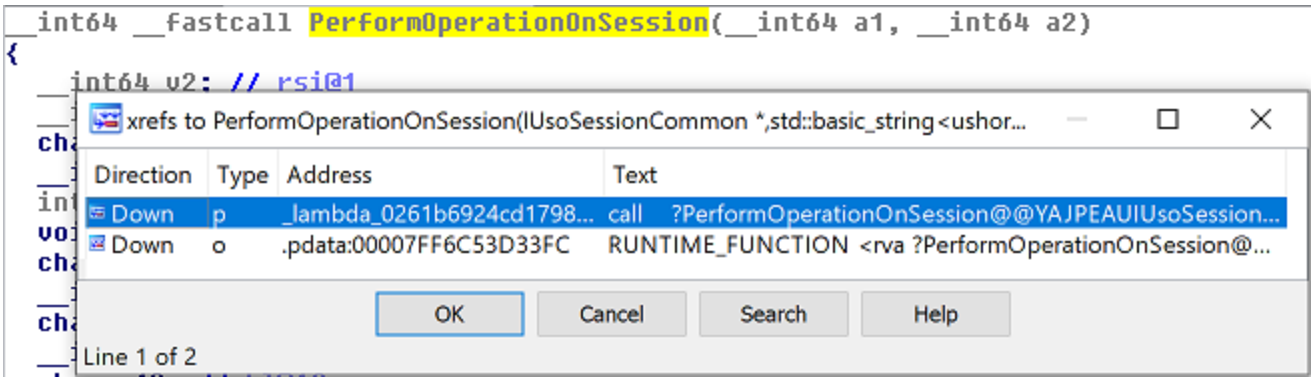
```
60  std::basic_string<unsigned short,std::char_traits<unsigned short>,std::allocator<unsigned sh
61    &v46,
62    L"StartScan");
63  v4 = iequals(v2, &v46);
64  std::basic_string<unsigned short,std::char_traits<unsigned short>,std::allocator<unsigned sh
65  if ( !v4 )
66  {
67    _mm_storeu_si128((__m128i *)&v48, _mm_load_si128((const __m128i *)&_xmm));
68    v46 = 0;
69    std::basic_string<unsigned short,std::char_traits<unsigned short>,std::allocator<unsigned
70      &v46,
71      L"StartDownload");
72    v8 = iequals(v2, &v46);
73    std::basic_string<unsigned short,std::char_traits<unsigned short>,std::allocator<unsigned
74    if ( v8 )
75    {
76      v9 = *(_QWORD *)(*(_QWORD *)v3 + 176i64);
77      v6 = _guard_dispatch_icall_fptr(v3, 0i64);
78      if ( v6 < 0 )
79      {
80        v7 = (void *)85;
81        goto LABEL_59;
82      }
83    }
84    else
85    {
86      _mm_storeu_si128((__m128i *)&v48, _mm_load_si128((const __m128i *)&_xmm));
87      v46 = 0;
88      std::basic_string<unsigned short,std::char_traits<unsigned short>,std::allocator<unsigne
89        &v46,
90        L"StartInstall");
91      v10 = iequals(v2, &v46);
92      std::basic_string<unsigned short,std::char_traits<unsigned short>,std::allocator<unsigne
93      if ( v10 )
```

This seems to be a "*Switch Case Statement*". The input is compared against a list of hardcoded commands: `StartScan`, `StartDownload`, `StartInstall`, etc. If there is a match, an action is taken.

For example, when the `StartScan` option is used, the following code is run.

```
v5 = *(_QWORD *)(*(_QWORD *)v3 +
168i64);
v6 = _guard_dispatch_icall_fptr(v3,
0i64);
if ( v6 >= 0 )
  return 0i64;
```

This code doesn't make much sense. So, I considered it as a dead end for the moment and decided to go up instead by looking for `Xrefs` to this function.



This function is called only once so it's pretty straightforward.

```
473   if ( v39 >= 0 )
474   {
475     v42 = CoSetProxyBlanket(pProxy, 0xFFFFFFFF, 0xFFFFFFFF, (OLECHAR *)0xFFFFFFFF, 0, 3u, 0i64, 0);
476     v5 = v42;
477     if ( v42 >= 0 )
478     {
479       _mm_storeu_si128((__m128i *)&v63, _mm_load_si128((const __m128i *)&_xmm));
480       Dst = 0;
481       std::basic_string<unsigned short,std::char_traits<unsigned short>,std::allocator<unsigned short>>
482         &Dst,
483         (void *)v8);
484       v5 = PerformOperationOnSession((__int64)pProxy, (__int64)&Dst);
485       std::basic_string<unsigned short,std::char_traits<unsigned short>,std::allocator<unsigned short>>
486       if ( v5 >= 0 )
487         goto LABEL_65;
488       v40 = (const char *)(unsigned int)v5;
489       v41 = (void *)407;
490     }
    else
```

I then had a quick look at the pseudocode and I immediately spotted the following calls: `CoInitializeEx()`, `CoCreateInstance()`, `CoSetProxyBlanket()`, etc. Because I already played around with COM (Component Object Model) before, I recognized the sequence of API calls.

Let's take a closer look at the following call.

```
380     pInterface = 0i64;
381     v28 = CoCreateInstance(
382             &GUID_b91d5831_b1bd_4608_8198_d72e155020f7,
383             0i64,
384             4u,
385             &GUID_07f3afac_7c8a_4ce7_a5e0_3d24ee8a77e0,
386             &pInterface);
387     v5 = v28;
```

According to Microsoft documentation, you can call `CoCreateInstance()` to *create a single uninitialized object of the class associated with a specified CLSID* (Source: CoCreateInstance function)

Here is the prototype of the function:

```
HRESULT
CoCreateInstance(
  REFCLSID  rclsid,
  LPUNKNOWN
pUnkOuter,
  DWORD
dwClsContext,
  REFIID    riid,
  LPVOID    *ppv
);
```

- `rclsid` is *the CLSID associated with the data and code that will be used to create the object.*
- `riid` is *a reference to the identifier of the interface to be used to communicate with the object.*

If we apply this to the call in the USO client, it means that the object with the CLSID `b91d5831-b1bd-4608-8198-d72e155020f7` will be created and the interface with the IID `07f3afac-7c8a-4ce7-a5e0-3d24ee8a77e0` will be used to communicate with it.

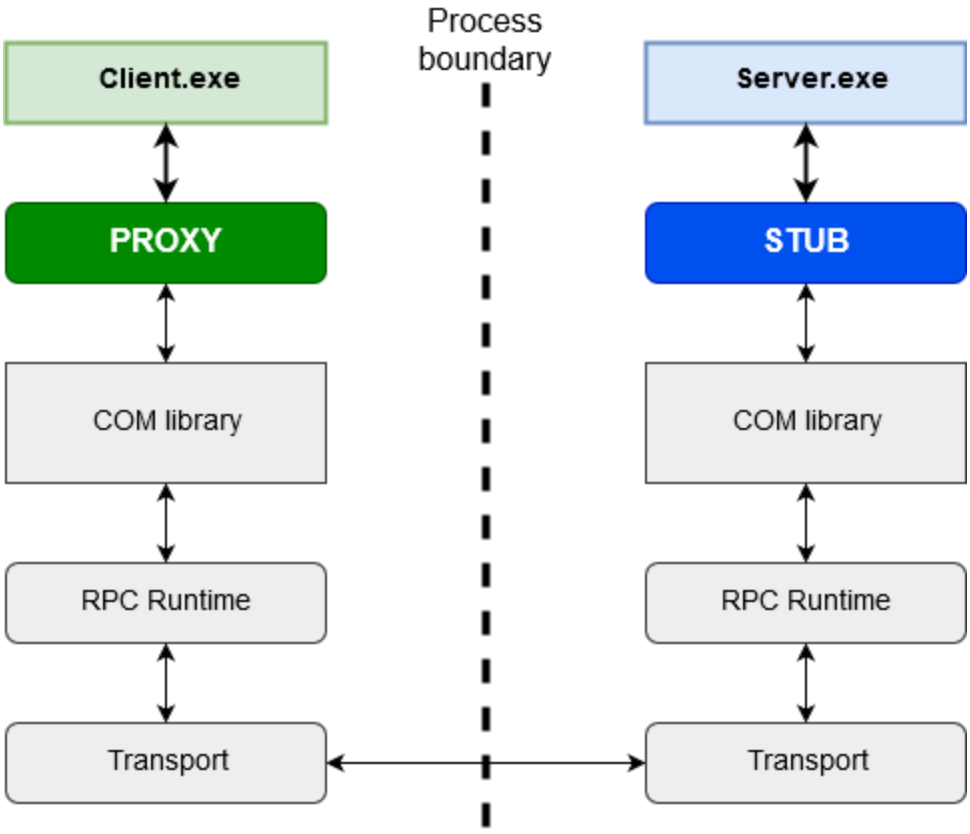
Having read the article Exploiting Arbitrary File Writes for Local Elevation of Privilege by James Forshaw several times, I knew what I had to do next. Thanks to his tool called `OleViewDotNet`, it should be quite easy to **reverse engineer the DCOM object**.

If you're already familiar with this concept, you can skip the next part. For more information: https://docs.microsoft.com/en-us/windows/win32/com/inter-object-communication.

A quick word about (D)COM

As I said earlier, COM stands for *Component Object Model*. It's a standard defined by Microsoft for inter process communications. Since I don't know much about this technology myself, I won't go into the details.

The key point to keep in mind though is how the **communication between a client and a server** is done. It is described on the following diagram. The client's call goes through a ***Proxy*** and then through a *Channel* which is part of the *COM library*. The marshaled call is transmitted to the server's process thanks to the *RPC runtime* and finally, the parameters are unmarshaled by the ***Stub*** before being forwarded to the server.



Components of Interprocess Communications

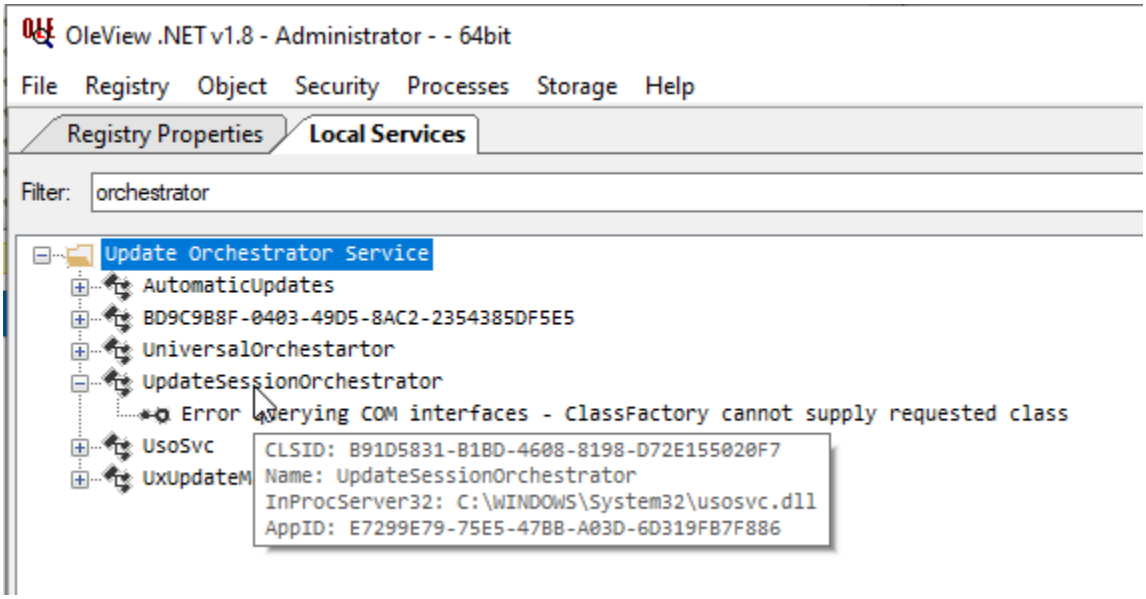
The obvious consequence is that we will find only the proxy definition on client side and, we might miss some key information from the server's side.

Reverse engineering a COM communication (almost) by hand

Let's start the reverse engineering of the COM object. **We already know its CLSID** so, using `OleViewDotNet`, this step should be straigthforward, right?

First we can **enumerate all the objects** exposed by the services running on the host by going to `Registry > Local Services`. Since we also know the name of the service, we can narrow down the list with the keywork `orchestrator`. **This yields a few objects** that we can inspect manually to find the one we are looking for: `UpdateSessionOrchestrator`. The **CLSID** matches the one we saw earlier while reverse engineering the USO client: `b91d5831-b1bd-4608-8198-d72e155020f7`.

The next step would be to expand the corresponding node in order to enumerate all the interfaces of the object. However, in this case it failed with the following error: `Error querying COM interface - ClassFactory cannot supply requested class`.



OK, never mind, we will have to do it manually. From this point on, I went for a dynamic analysis of the client in order to see how the RPC calls worked.

To do so, I used those three tools:

- **IDA** (with debug symbols configured)
- **IDA**'s x86_64 Windows debug server - `C:\Program Files (x86)\IDA 6.8\dbgsrv\win64_remotex64.exe`
- **WinDbg** (with debug symbols configured)

We already know that the `CoCreateInstance()` call is used to instantiate the remote COM object. As a result the variable `pInterface`, as its name implies, holds a pointer to the interface with the IID `07f3afac-7c8a-4ce7-a5e0-3d24ee8a77e0`, which will be used to communicate with the object. My goal now is to understand what happens next. Therefore, I put a breakpoint on the first `_guard_dispatch_icall_fptr` call that comes right after.

```
378     pInterface = 0i64;
379     v26 = CoCreateInstance(
380             &GUID_b91d5831_b1bd_4608_8198_d72e155020f7,
381             0i64,
382             4u,
383             &GUID_07f3afac_7c8a_4ce7_a5e0_3d24ee8a77e0,
384             &pInterface);
385     v5 = v26;
386     if ( v26 < 0 )
387     {
388       wil::details::in1diag3::Return_Hr(
389         retaddr,
390         (void *)0x16F,
391         (unsigned __int64)"onecore\\enduser\\windowsupdate\\muse\\orchestrator\\client\\main.cpp",
392         (const char *)(unsigned int)v26,
393         ppvd);
394       goto LABEL_63;
395     }
396     v27 = *(_QWORD *)(*(_QWORD *)pInterface + 40i64);
397     _guard_dispatch_icall_fptr(pInterface);
398     v28 = **(_QWORD **)pInterface;
399     _guard_dispatch_icall_fptr(pInterface);
```

Here is what happens right before the call:

1. The `RCX` register holds the location of the interface's pointer (i.e. `pInterface`).
2. The value pointed to by `RCX` is loaded into `RAX` - i.e. `RAX` = `pInterface`.
3. The value that was stored in `RSI` is copied to `RDX` - We don't know what it is yet.
4. The value pointed to by `RAX+0x28` is loaded into `RAX` - i.e. `ProxyVTable[5]` as we will see.

```
.text:00007FF605AE6833 loc_7FF605AE6833:                       ; CODE XREF: _lambda_0261b6924cd17987a6b
.text:00007FF605AE6833 mov     rcx, [rsp+218h+pInterface]      ; _QWORD
.text:00007FF605AE6838 mov     rax, [rcx]
.text:00007FF605AE683B mov     rdx, rsi                        ; _QWORD
.text:00007FF605AE683E mov     rax, [rax+28h]
.text:00007FF605AE6842
.text:00007FF605AE6842 loc_7FF605AE6842:                       ; DATA XREF: .rdata:stru_7FF605AEF640↓o
RIP .text:00007FF605AE6842 call    cs:__guard_dispatch_icall_fptr
```

```
RAX 00007FF8E48F7E80 ↳ usoapi.dll:usoapi_DllCanUnloadNow+66D0
RBX 0000000000000000 ↳
RCX 000002344FA53D68 ↳ debug033:000002344FA53D68
RDX 000002344FA39450 ↳ debug033:000002344FA39450
RSI 000002344FA39450 ↳ debug033:000002344FA39450
RDI 0000000000000000 ↳
RBP 000000F3CFB6F590 ↳ Stack[000017F8]:000000F3CFB6F590
RSP 000000F3CFB6F490 ↳ Stack[000017F8]:000000F3CFB6F490
RIP 00007FF605AE6842 ↳ _lambda_0261b6924cd17987a6b78c10ba58966a
```

The value of `RCX` is `0x000002344FA53D68`. Let's see what we can find at this address with WinDbg.

```
0:000> dqs 0x00002344FA53D68 L1
00000234`4fa53d68  00007ff8`e48fd560 usoapi!IUpdateSessionOrchestratorProxyVtbl+0x10
```

We find the start address of the Proxy VTable of the UpdateSessionOrchestrator's interface. We can then enumerate all the pointers listed in the VTable.

```
0:000> dqs 0x00007ff8e48fd560 LB
00007ff8`e48fd560  00007ff8`e48f8040 usoapi!IUnknown_QueryInterface_Proxy
00007ff8`e48fd568  00007ff8`e48f7d90 usoapi!IUnknown_AddRef_Proxy
00007ff8`e48fd570  00007ff8`e48f7ed0 usoapi!IUnknown_Release_Proxy
00007ff8`e48fd578  00007ff8`e48f7dc0 usoapi!ObjectStublessClient3
00007ff8`e48fd580  00007ff8`e48f8090 usoapi!ObjectStublessClient4
00007ff8`e48fd588  00007ff8`e48f7e80 usoapi!ObjectStublessClient5
00007ff8`e48fd590  00007ff8`e48f7ef0 usoapi!ObjectStublessClient6
00007ff8`e48fd598  00007ff8`e48f7e60 usoapi!ObjectStublessClient7
00007ff8`e48fd5a0  00007ff8`e49068b0 usoapi!IID_IMoUsoUpdate
00007ff8`e48fd5a8  00007ff8`e48fefb0 usoapi!CAutomaticUpdates::`vftable'+0x3b0
00007ff8`e48fd5b0  00000000`00000019
```

The first three functions are `QueryInterface`, `AddRef` and `Release`. These are the functions that a COM interface inherits from `IUnknown`. Then, there are 5 other functions but we don't know their names.

In order to find more information about the VTable, we have to inspect the server. We know the name of the COM object - `UpdateSessionOrchestrator` - and we know the name of the service - `USOsvc`. So, theoritically, we should find all the information we need in `usosvc.dll`.

```
.rdata:00000001800582F8 dq offset UpdateSessionOrchestrator::QueryInterface(void)
.rdata:0000000180058300 dq offset UpdateSessionOrchestrator::AddRef(void)
.rdata:0000000180058308 dq offset UpdateSessionOrchestrator::Release(void)
.rdata:0000000180058310 dq offset
UpdateSessionOrchestrator::CreateUpdateSession(tagUpdateSessionType,_GUID const
&,void * *)
.rdata:0000000180058318 dq offset
UpdateSessionOrchestrator::GetCurrentActiveUpdateSessions(IUsoSessionCollection * *)
.rdata:0000000180058320 dq offset UpdateSessionOrchestrator::LogTaskRunning(ushort
const *)
.rdata:0000000180058328 dq offset
UpdateSessionOrchestrator::CreateUxUpdateManager(IUxUpdateManager * *)
.rdata:0000000180058330 dq offset
UpdateSessionOrchestrator::CreateUniversalOrchestrator(IUniversalOrchestrator * *)
```

Nice! Here is the complete VTable. We can see that the function at offset 5 is `UpdateSessionOrchestrator::LogTaskRunning(ushort const *)`.

Finally, the value of RDX is `0x000002344FA39450`. Let's check what we can find at this address as well, with IDA this time:

```
000002344FA39450  53 00 74 00 61 00 72 00  74 00 53 00 63 00 61 00  S.t.a.r.t.S.c.a.
000002344FA39460  6E 00 00 00 AB AB AB AB  AB AB AB AB AB AB AB AB  n...¯¯¯¯¯¯¯¯¯¯¯¯
```

It's just a pointer to the null terminated unicode string `L"StartScan"`.

All this information can be summarized as follows.

```
RAX = VTable[5] = `UpdateSessionOrchestrator::LogTaskRunning(ushort const *)`
RCX = argv[0]   = `UpdateSessionOrchestrator pInterface`
RDX = argv[1]   = L"StartScan"
```

If we consider the x86_64 calling convention of Windows, this can be represented by the following pseudocode.

```
pInterface->LogTaskRunning(L"StartScan");
```

The same process can be applied to the next call.

```
378     pInterface = 0i64;
379     v26 = CoCreateInstance(
380             &GUID_b91d5831_b1bd_4608_8198_d72e155020f7,
381             0i64,
382             4u,
383             &GUID_07f3afac_7c8a_4ce7_a5e0_3d24ee8a77e0,
384             &pInterface);
385     v5 = v26;
386     if ( v26 < 0 )
387     {
388       wil::details::in1diag3::Return_Hr(
389         retaddr,
390         (void *)0x16F,
391         (unsigned __int64)"onecore\\enduser\\windowsupdate\\muse\\orchestrator\\client\\main.cpp",
392         (const char *)(unsigned int)v26,
393         ppvd);
394       goto LABEL_63;
395     }
396     v27 = *(_QWORD *)(*(_QWORD *)pInterface + 40i64);
397     _guard_dispatch_icall_fptr(pInterface);
398     v28 = **(_QWORD **)pInterface;
399     _guard_dispatch_icall_fptr(pInterface);
```

This would yield the following:

```
RAX = VTable[0] = `UpdateSessionOrchestrator::QueryInterface()`
RCX = argv[0]   = `UpdateSessionOrchestrator pInterface`
RDX = argv[1]   = `*GUID(c57692f8-8f5f-47cb-9381-34329b40285a)`
R8  = argv[2]   = Output pointer location
```

Here, the returned value is `NULL` so, all the code after the `if` statement would be ignored.

```
378     pInterface = 0i64;
379     v26 = CoCreateInstance(
380             &GUID_b91d5831_b1bd_4608_8198_d72e155020f7,
381             0i64,
382             4u,
383             &GUID_07f3afac_7c8a_4ce7_a5e0_3d24ee8a77e0,
384             &pInterface);
385     v5 = v26;
386     if ( v26 < 0 )
387     {
388       wil::details::in1diag3::Return_Hr(
389         retaddr,
390         (void *)0x16F,
391         (unsigned __int64)"onecore\\enduser\\windowsupdate\\muse\\orchestrator\\client\\main.cpp",
392         (const char *)(unsigned int)v26,
393         ppvd);
394       goto LABEL_63;
395     }
396     v27 = *(_QWORD *)(*(_QWORD *)pInterface + 40i64);
397     _guard_dispatch_icall_fptr(pInterface);
398     v28 = **(_QWORD **)pInterface;
399     _guard_dispatch_icall_fptr(pInterface);
```

Therefore, we can skip it and jump right here:

```
469     v37 = _guard_dispatch_icall_fptr(pInterface);
470     v5 = v37;
471     if ( v37 >= 0 )
472     {
473       v40 = CoSetProxyBlanket(pProxy, 0xFFFFFFFF, 0xFFFFFFFF, (OLECHAR *)0xFFFFFFFF, 0, 3u, 0i64, 0);
474       v5 = v40;
475       if ( v40 >= 0 )
476       {
477         _mm_storeu_si128((__m128i *)&v61, _mm_load_si128((const __m128i *)&_xmm));
478         Dst = 0;
479         std::basic_string<unsigned short,std::char_traits<unsigned short>,std::allocator<unsigned short>>::assig
480           &Dst,
481           (void *)v8);
482         v5 = PerformOperationOnSession((__int64)pProxy, (__int64)&Dst);
483         std::basic_string<unsigned short,std::char_traits<unsigned short>,std::allocator<unsigned short>>::_Tidy
484         if ( v5 >= 0 )
```

Nice! Were are getting closer to the target `PerformOperationOnSession()` call.

With the same reverse engineering process, we find the following.

```
RAX = VTable[3] =
`UpdateSessionOrchestrator::CreateUpdateSession(tagUpdateSessionType,_GUID const
&,void * *)`
RCX = argv[0]   = `UpdateSessionOrchestrator pInterface`
RDX = argv[1]   = 1
R8  = argv[2]   = `*GUID(fccc288d-b47e-41fa-970c-935ec952f4a4)`
R9  = argv[3]   = `void **param_3 (usoapi!IUsoSessionCommonProxyVtbl+0x10)` -->
IUsoSessionCommon pProxy
```

Here, we can see that another interface is involded: `IUsoSessionCommon`. It's identified by the IID `fccc288d-b47e-41fa-970c-935ec952f4a4` and its VTable has 68 entries so I won't list all the functions here.

Next there is a `CoSetProxyBlanket()` call. This is a standard WinApi function that is used to *set the authentication information that will be used to make calls on the specified proxy* (Source: CoSetProxyBlanket function).

```
473     v40 = CoSetProxyBlanket(pProxy, 0xFFFFFFFF, 0xFFFFFFFF, (OLECHAR *)0xFFFFFFFF, 0, 3u, 0i64, 0);
```

If we translate all the hexadecimal values back to Win32 constants, this yields the following API call.

```
IUsoSessionCommonPtr usoSessionCommon;
CoSetProxyBlanket(usoSessionCommon, RPC_C_AUTHN_DEFAULT, RPC_C_AUTHZ_DEFAULT,
COLE_DEFAULT_PRINCIPAL, RPC_C_AUTHN_LEVEL_DEFAULT, RPC_C_IMP_LEVEL_IMPERSONATE,
nullptr, NULL);
```

Now, we can enter the `PerformOperationOnSession()` function and, we are back to the piece of code that didn't make sense before. However, thanks to the reverse engineering process we just went through, this is now getting clearer. This is a simple call on the `IUsoSessionCommon` proxy. We just need to determine **which function** is called and with **which parameters**.

```
350  v5 = *(_QWORD *)(*(_QWORD *)v3 + 168i64);
351  v6 = _guard_dispatch_icall_fptr(v3);
352  if ( v6 >= 0 )
353    return 0i64;
```

```
.text:00007FF605AE57FE mov     rax, [rdi]
.text:00007FF605AE5801 xor     r8d, r8d
.text:00007FF605AE5804 xor     edx, edx                        ; _QWORD
.text:00007FF605AE5806 lea     r9, aScantriggeruso             ; "ScanTriggerUsoClient"
.text:00007FF605AE580D mov     rcx, rdi                        ; _QWORD
.text:00007FF605AE5810 mov     rax, [rax+0A8h]
RIP .text:00007FF605AE5817 call    cs:__guard_dispatch_icall_fptr
```

With this final breakpoint, the function's offset and the parameters can be easily determined.

```
RAX = VTable[21] = combase_NdrProxyForwardingFunction21
RCX = argv[0]    = IUsoSessionCommon pProxy
RDX = argv[1]    = 0
R8  = argv[2]    = 0
R9  = argv[3]    = L"ScanTriggerUsoClient"
```

This would be equivalent to the following pseudocode.

```
pProxy->Proc21(0, 0,
L"ScanTriggerUsoClient");
```

If all the pieces are put together, the "StartScan" action in the USO client can be summarized with the following simplified code.

```
HRESULT hResult;
// Initialize the COM library
hResult = CoInitializeEx(0, COINIT_MULTITHREADED);
// Create the remote UpdateSessionOrchestrator object
GUID CLSID_UpdateSessionOrchestrator = { 0xb91d5831, 0xb1bd, 0x4608, { 0x81,
0x98, 0xd7, 0x2e, 0x15, 0x50, 0x20, 0xf7 } };
IUpdateSessionOrchestratorPtr updateSessionOrchestrator;
hResult = CoCreateInstance(CLSID_UpdateSessionOrchestrator, nullptr,
CLSCTX_LOCAL_SERVER, IID_PPV_ARGS(&updateSessionOrchestrator));
// Invoke LogTaskRunning()
updateSessionOrchestrator->LogTaskRunning(L"StartScan");
// Create an update session
IUsoSessionCommonPtr usoSessionCommon;
GUID IID_IUsoSessionCommon = { 0xfccc288d, 0xb47e, 0x41fa, { 0x97, 0x0c, 0x93,
0x5e, 0xc9, 0x52, 0xf4, 0xa4 } };
updateSessionOrchestrator->CreateUpdateSession(1, &IID_IUsoSessionCommon,
&usoSessionCommon);
// Set the authentication information
CoSetProxyBlanket(usoSessionCommon, RPC_C_AUTHN_DEFAULT, RPC_C_AUTHZ_DEFAULT,
COLE_DEFAULT_PRINCIPAL, RPC_C_AUTHN_LEVEL_DEFAULT, RPC_C_IMP_LEVEL_IMPERSONATE,
nullptr, NULL);
// Trigger the "StartScan" action
usoSessionCommon->Proc21(0, 0, L"ScanTriggerUsoClient")
// Close the COM library
CoUninitialize();
```

Conclusion

Knowing how the USO client works and how it can trigger privileged actions, it is now possible to replicate this behavior as a standalone application: UsoDllLoader. Of course, the transition from this reverse engineering process to the actual C++ code requires a bit more work but it's not the most interesting part so I skipped it. The only thing that I should mention though is that the DiagHub PoC did help a lot.

Regarding the reverse engineering part, I have to say that it wasn't too difficult because the COM client already exists and is provided with Windows by default. OleViewDotNet did help a lot in the end as well. It was able to generate the code for the second interface (UsoSessionCommon) - you know, the one with 68 functions!

Well, that wraps it up for this post. I hope you enjoyed it.

Links & Resources

- Microsoft Documentation - CoCreateInstance
  https://docs.microsoft.com/en-us/windows/win32/api/combaseapi/nf-combaseapi-cocreateinstance
- Microsoft Documentation - Inter-Object Communications
  https://docs.microsoft.com/en-us/windows/win32/com/inter-object-communication
- Microosft Documentation - x64 calling convention
  https://docs.microsoft.com/en-us/cpp/build/x64-calling-convention?view=vs-2019
- Windows Exploitation Tricks: Exploiting Arbitrary File Writes for Local Elevation of Privilege
  https://googleprojectzero.blogspot.com/2018/04/windows-exploitation-tricks-exploiting.html