Flare-On 7: Challenge 9 – crackinstaller.exe

Challenge Author: Paul Tarter (@Hefrpidge)

In this challenge players were given a 64-bit Windows executable and a message eluding to the challenge being a crackme that doesn't ask for a password. My intention with this challenge was to combine different tactics and techniques I have encountered while analyzing malicious software. The challenge combines: Global Initializers, COM, Kernel Exploitation, Registry Filtering in the Kernel, and Registry Class Type Strings.

HIGH LEVEL OVERVIEW

This challenge is composed of a single 64-bit Windows executable named `crackstaller.exe` that acts like a dropper that contains three binaries embedded. The dropper has two major paths of execution, the more obvious path starts in main; from main the installer drops and registers a COM server. The second path of execution begins prior to `main` as a CRT initializer where it drops a signed 64-bit driver that contains a Supervisor Mode Execution Prevention (SMEP) bypass. Utilizing the SMEP bypass, the dropper loads an embedded unsigned driver reflectively utilizing the signed driver with SMEP bypass. The unsigned driver registers registry callbacks and filters on registry pre-key-creations. Because the driver is installed prior to `main` executing, it will filter the flag server being registered and as result will add a class type string to a key being registered. The class type string will be the password supplied to the flag server to attain the flag. There are multiple ways to gaining the flag, one way is to place the password in the registry config under password and interact with the COM server appropriately to have it populate the flag value.

COM OVERVIEW

COM is a platform-independent, distributed, object-oriented system for creating binary software components that can interact. COM is not an object-oriented language, it is a standard. COM specifies an object model and programming requirements that enable COM objects (also called COM components, or sometimes simply objects) to interact with other objects. A COM object exposes its features through an interface, which is a collection of member functions. Every interface has its own unique interface identifier, named an IID, which eliminates collisions that could occur with human-readable names. The IID is a globally unique identifier (GUID). One cannot create an instance of a COM interface by itself. Instead, one creates an instance of a class that implements the interface.

All COM interfaces inherit from the `IUnknown` interface. The `IUnknown` interface contains the fundamental COM operations for polymorphism and instance lifetime management. The `IUnknown` interface has three member functions that all COM objects are required to implement:

- `QueryInterface`. Retrieves pointers to the supported interfaces on an object.
- `AddRef`. Increments the reference count for an interface pointer to a COM object. You should call this method whenever you make a copy of an interface pointer.
- `Release` Decrements the reference count for an interface on a COM object.

One of the most important ways for a client to get a pointer to an object is for the client to ask that a server be launched and that an instance of the object provided by the server be created and activated. It is the responsibility of the server to ensure that this happens properly. There are several important parts to this, the server must implement code for a class object through an implementation of either the `IClassFactory` or `IClassFactory2` interface. This challenge dealt with an `IClassFactory`

The server must register its `CLSID` (another GUID) in the system registry on the machine on which it resides. When a client uses a CLSID to request the creation of an object instance, the first step is creation of a class object, an intermediate object that contains an implementation of the methods of the `IClassFactory` interface. While COM provides several instance creation functions, the first step in the implementation of these functions is the creation of a class object.

As a result, all servers must implement the methods of the `IClassFactory` interface, which contains two methods on top of `IUnknown`:

- `CreateInstance`. This method must create an uninitialized instance of the object and return a pointer to a requested interface on the object.
- `LockServer`. This method just increments the reference count on the class object to ensure that the server stays in memory and does not shut down before the client is ready for it to do so.

Utilizing the above information two examples are provided below to show how to instantiate a component object as a client. Figure 1 is a more explicit route with more work on the client's part but provides more flexibility for if multiple objects are required.

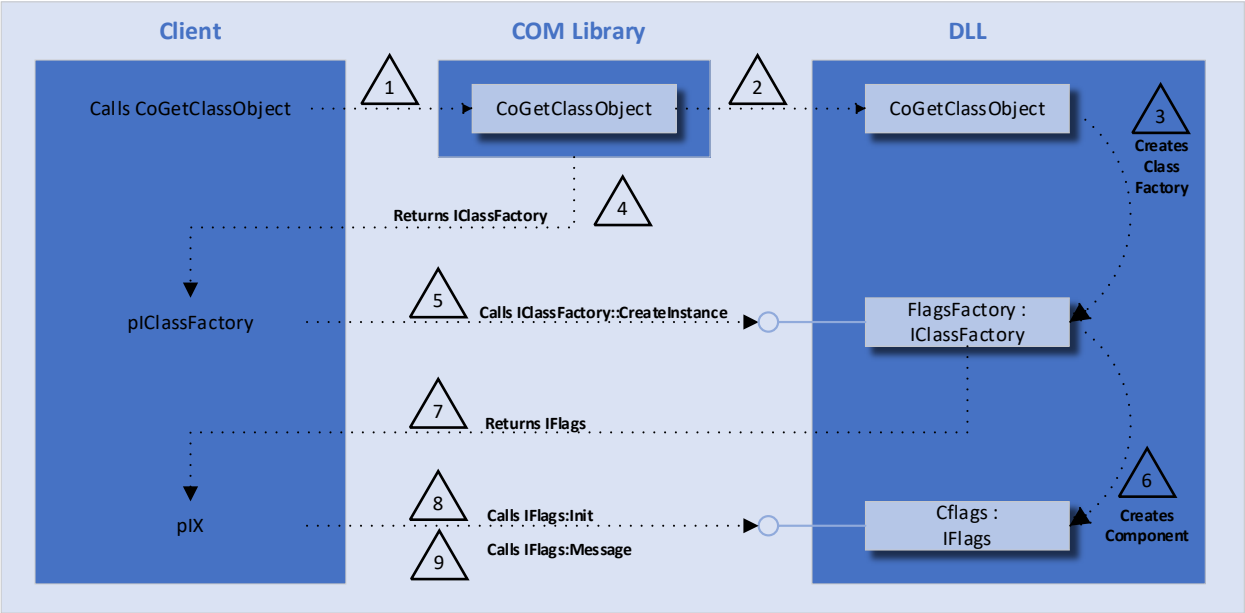


Figure 1: COM client utilizing CoGetClassObject

The second approach shown in Figure 2 for implementing a component object is used when there will only be one instance.

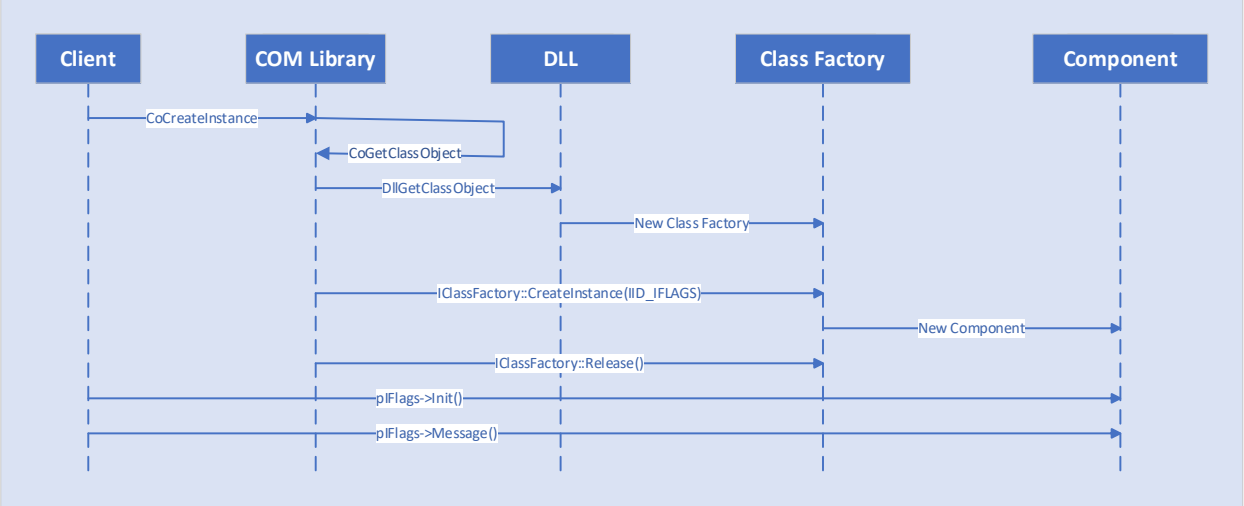


Figure 2: COM client utilizing CoCreateInstance

INITIAL TRIAGE ANALYSIS

Looking at strings for the challenge does show a Base64 alphabet. Also, there are four strings that don't stand out but are near the beginning of a strings listing, "`expa`", "`nd 3`", "`2-by`" and "`te k`", these stand out as signatures to `Salsa20` or `ChaCha20`. Observing the PE header in a tool like CFF Explorer does show a very large `.data` section. Scanning through the `.data` section some reoccurring bytes occur,

specially a large section of 0x08675309, this could indicate an XOR key. There aren't any resources with the binary. `DeviceIoControl` showing up in the imports does raise some suspicion and would be something to investigate.

Running the binary with Procmon there is a point in the output that is easy to miss, a creation of `cfs.dll` followed by `CreateFileMapping`, `QueryStandardInformationFile`, `SetEndOfFileInformation`, and `ReadFile`. While this looks to be part of the standard loading process and the DLL sounds legit enough, the call to `ReadFile` is out of place in this situation. Furthermore, down the execution line there is another call to `CreateFile` immediately followed by a `CloseFile` on the same file. If one is to investigate the parameters for `CreateFile`, shown in Figure 3, it is determined that this is a deletion of the file.

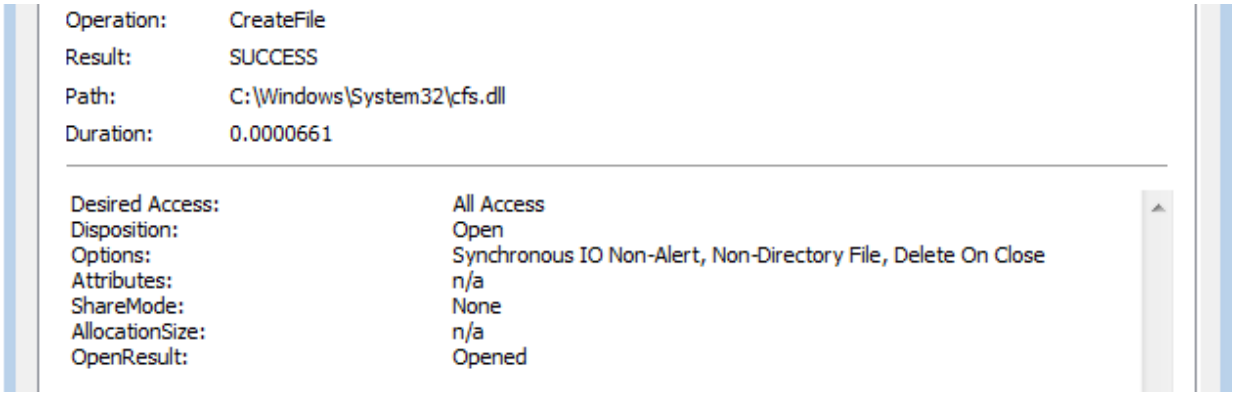


Figure 3 Deleting a file with CreateFile

Further down the execution line there is a much more blatant file dropping that occurs to the path `%LOCALAPPDATA%\Microsoft\Credentials\credHelper.dll`. After this file is written to disk the normal procedure of registering a COM server occurs, evidence is shown in Procmon, as shown in Figure 4 with reg.exe.

```
C:\Users\user>reg query HKCR\CLSID\{CEEACC6E-CCB2-4C4F-BCF6-D2176037A9A7} /s

HKEY_CLASSES_ROOT\CLSID\{CEEACC6E-CCB2-4C4F-BCF6-D2176037A9A7}
    (Default)    REG_SZ    CredHelper

HKEY_CLASSES_ROOT\CLSID\{CEEACC6E-CCB2-4C4F-BCF6-D2176037A9A7}\Config
    Password    REG_SZ
    Flag    REG_SZ

HKEY_CLASSES_ROOT\CLSID\{CEEACC6E-CCB2-4C4F-BCF6-D2176037A9A7}\InProcServer32
    (Default)    REG_SZ    C:\Users\user\AppData\Local\Microsoft\Credentials\credHelper.dll
    ThreadingModel    REG_SZ    Apartment
```

Figure 4: Flags COM server registration

Retrieving the COM server can be done at this point by simply copying `credHelper.dll` from disk. It also appears that this challenge either stores or requires a password in the registry and that a flag is somehow stored in the registry. Both values are currently empty. The challenge message pointed out that this crackme didn't ask directly for a password and that it needed to work on its COMmunication. It appears that this challenge is requiring the password through the registry and interacted with through COM.

RETRIEVING THE COM SERVER

Static analysis of the `crackstaller.exe` is another method of retrieving the COM server and is described below. Further analysis of `crackstaller.exe` will need to be statically analyzed beyond the COM server retrieval.

`Crackstaller.exe` installs a COM server through a standard COM method, `DllRegisterServer.` This function is one of four expected functions exported by a COM server: `DllRegisterServer, DllUnregisterServer, DllCanUnloadNow, DllGetClassObject.`

`main` is straightforward without much obfuscation. It begins immediately allocating `0x0001A600` bytes and then multibyte-xor-decoding a blob of data stored in the .data section with the key, previously mentioned, 0x8675309, as seen in Figure 5.

```
.rdata:0000000140016828 com_dll_xor_key db 8, 67h, 53h, 9
```

Figure 5: COM Server XOR key

Next `ShGetKnownFolderPath` is used to attain the path to install the COM server. The GUID, `{F1B32785-6FBA-4FCF-9D55-7B8E7F157091}` resolves to `FOLDERID_LocalAppData`, the default path is: `%LOCALAPPDATA%` (`%USERPROFILE%\AppData\Local`).

```
.text:000000014000218D                  lea     rcx, guid_FOLDERID_LocalAppData ; rfid
.text:0000000140002194                  xor     edx, edx        ; dwFlags
.text:0000000140002196                  mov     ebx, r14d
.text:0000000140002199                  call    cs:SHGetKnownFolderPath
.rdata:000000014000F340 guid_FOLDERID_LocalAppData dd 0F1B32785h          ; Data1
.rdata:000000014000F340                                                   ; DATA XREF: main+69to
.rdata:000000014000F340                                                   ; main+17Fto
.rdata:000000014000F340                 dw 6FBAh                          ; Data2
.rdata:000000014000F340                 dw 4FCFh                          ; Data3
.rdata:000000014000F340                 db 9Dh, 55h, 7Bh, 8Eh, 7Fh, 15h, 70h, 91h; Data4
```

Figure 6: FOLDERID_LocalAppData

The path to write the COM server to is multibyte-xor-encoded with a key that is seen throughout the rest of the challenge and used with different wrappers to handle multibyte, wide characters and `UNICODE_STRING`. The XOR key is shown in Figure 7:

```
.rdata:0000000140016820 common_xor_key  db 3Ch, 67h, 7Eh, 7Bh, 3Ch, 69h, 74h, 0
```

Figure 7: Challenge common XOR key

Once the COM server and installation path are created, and the server is written to disk, the final task of installing the server is performed by calling `DllRegisterServer`. This is performed by multibyte-xor-decoding the export function name, then dynamically loading the DLL and lastly calling the export as shown in Figure 8.

```
.text:0000000140002323                  lea     rcx, [rbp+780h+LibFileName] ; lpLibFileName
.text:000000014000232A                  call    cs:LoadLibraryW
.text:0000000140002330                  mov     rbx, rax
.text:0000000140002333                  test    rax, rax
.text:0000000140002336                  jz      short loc_140002357
.text:0000000140002338                  mov     edx, 12h
.text:000000014000233D                  lea     rcx, str_DllRegisterServer
.text:0000000140002344                  call    multi_byte_xor_decode
.text:0000000140002349                  mov     rdx, rax        ; lpProcName
.text:000000014000234C                  mov     rcx, rbx        ; hModule
.text:000000014000234F                  call    cs:GetProcAddress
.text:0000000140002355                  call    rax
```

Figure 8: DllRegisterServer

The four main approaches to retrieving the server for analysis are by: decoding in the IDB and dumping, dumping and decoding, dumping the decoded server while debugging or the simplest form, retrieve the server from its installed location: `%LOCALAPPDATA%\Microsoft\Credentials\credHelper.dll`

This location for `credhelper.dll` was chosen due to the numerous times I have seen malware drop to a location that is very similar to this. The dropping technique is common and the fake credential folders with a DLL name to support can be seen often.

REVERSING THE COM SERVER

With the basic overview of COM provided above one can reverse and even write a simple client to interact with the COM server that is written to disk. As expected, the COM server exports the four functions: `DllCanUnloadNow`, `DllRegisterServer`, `DllUnregisterServer` and `DllGetClassObject`. The first three are simple and don't provide us with anything new. When reversing COM servers, *DllRegisterServer* and `DllUnregisterServer` are helpful for determining the installation, `DllCanUnloadNow` will usually just have some locking or reference counting but could have other helpful information. To learn more about the interfaces this server offers, one should look to `DllGetClassObject`.

`DllGetClassObject` has the following function prototype:

```
1. HRESULT DllGetClassObject(
2.     const IID *const rclsid,
3.     const IID *const riid,
4.     LPVOID *ppv);
```

COM servers can implement multiple classes, it is standard for this function to do a check on `rclsid`. This is also helpful in searching the registry for a registered COM server by the `rclsid` that are compared against. The following code shown in Figure 9 is performed within `credHelper.dll`.

```
text:0000000180001031                mov     rax, qword ptr cs:rguid.Data1
text:0000000180001038                mov     rbx, r8
text:000000018000103B                mov     rsi, rdx
text:000000018000103E                cmp     rax, [rcx]
text:0000000180001041                jnz     short loc_180001091
text:0000000180001043                mov     rax, qword ptr cs:rguid.Data4
text:000000018000104A                cmp     rax, [rcx+8]
```

Figure 9: rclsid comparison

Memory is then allocated for an object, an `IClassFactory`. The `ClassFactory` object, as with all COM objects, are built upon `IUnknown`. The structures shown in Figure 10 can be added in IDA to aid with reversing. Creating custom structures on top of IUnknown can be helpful as you build a custom interface.

```
00000000 IClassFactoryVtbl struc ; (sizeof=0x28, align=0x8, copyof_60)
00000000 QueryInterface  dq ?                    ; offset
00000008 AddRef          dq ?                    ; offset
00000010 Release         dq ?                    ; offset
00000018 CreateInstance  dq ?                    ; offset
00000020 LockServer      dq ?                    ; offset
00000028 IClassFactoryVtbl ends
00000028
00000000 ; ---------------------------------------------------------------------------
00000000
00000000 IUnknownVtbl    struc ; (sizeof=0x18, align=0x8, copyof_65)
00000000 QueryInterface  dq ?                    ; offset
00000008 AddRef          dq ?                    ; offset
00000010 Release         dq ?                    ; offset
00000018 IUnknownVtbl    ends
00000018
00000000 ; ---------------------------------------------------------------------------
00000000
00000000 IUnknown        struc ; (sizeof=0x8, align=0x8, copyof_64)
00000000 lpVtbl          dq ?                    ; offset
00000008 IUnknown        ends
00000008
00000000 ; ---------------------------------------------------------------------------
00000000
00000000 IClassFactory   struc ; (sizeof=0x8, align=0x8, copyof_59)
00000000 lpVtbl          dq ?                    ; offset
00000008 IClassFactory   ends
```

Figure 10: IUnknown and IClassFactory structures

The vtable at `0x1800178C0`, seen din Figure 11, is an `IClassFactoryVtbl` and the following variable in the memory allocated is used for reference counting.

```
.rdata:00000001800178C0 IClassFactory_vtbl IClassFactoryVtbl <offset factory_QueryInterface, \
.rdata:00000001800178C0                                       ; DATA XREF: DllGetClassObject+3E↑o
.rdata:00000001800178C0                                       ; factory_Release+D↑o
.rdata:00000001800178C0                                     offset factory_AddRef, offset factory_Release, \
.rdata:00000001800178C0                                     offset factory_CreateInstance, \
.rdata:00000001800178C0                                     offset factory_LockServer>
```

Figure 11: Credhelper's IFactory vtable

The function `factory_QueryInterface` is called directly instead of through the vtable. Analyzing the function will verify that an `IClassFactory` is expected. The only two `IIDs` that are accepted are: `IUnknown` and `IClassFactory`. To find the actual classes that are cared about in this challenge one must look into `IFactoryClass::CreateInstance`.

Within `CreateInstance` the memory allocation is going to start with a vtable that starts with `IUnknown`. This vtable is stored at `0x180017908`.

```
.rdata:0000000180017908 flags_vtable    dq offset flags_QueryInterface
.rdata:0000000180017908                                 ; DATA XREF: flags_Release+D↑o
.rdata:0000000180017908                                 ; factory_CreateInstance+35↑o
.rdata:0000000180017910                 dq offset factory_AddRef
.rdata:0000000180017918                 dq offset flags_Release
.rdata:0000000180017920                 dq offset sub_18000153C
.rdata:0000000180017928                 dq offset sub_1800016D8
```

Figure 12: Flags Interface

The two functions after `IUnknown` in Figure 12 are interfaces functions. To determine the IID for this interface one can look to `flags_QueryInterface`. Observing the data as a GUID with `alt-q` in IDA will allow the GUID to be much easier to identify.

```
.text:00000001800014D5                  mov     rax, qword ptr cs:IID_Flags.Data1
.text:00000001800014DC                  cmp     rax, [rdx]
.text:00000001800014DF                  jnz     short loc_1800014FB
.text:00000001800014E1                  mov     rax, qword ptr cs:IID_Flags.Data4
.text:00000001800014E8                  cmp     rax, [rdx+8]
.text:00000001800014EC                  jnz     short loc_1800014FB
```

```
.rdata:00000001800178F8 IID_Flags       dd 0E27297B0h           ; Data1
.rdata:00000001800178F8                                         ; DATA XREF: flags_QueryInterface:loc_1800014D5↑r
.rdata:00000001800178F8                                         ; flags_QueryInterface+2D↑r
.rdata:00000001800178F8                 dw 1E98h                ; Data2
.rdata:00000001800178F8                 dw 4033h                ; Data3
.rdata:00000001800178F8                 db 0B3h, 89h, 24h, 0ECh, 0A2h, 46h, 0, 2Ah; Data4
```

Figure 13: IID_FLAGS

At this point the CLSID and IID is known for what will be call `CFlags` and `IFlags`:

- `CFlags: 0xceeacc6e, 0xccb2, 0x4c4f, 0xbc, 0xf6, 0xd2, 0x17, 0x60, 0x37, 0xa9, 0xa7`
- `IFlags: 0xe27297b0, 0x1e98, 0x4033, 0xb3, 0x89, 0x24, 0xec, 0xa2, 0x46, 0x00, 0x2a`

Finally, one can investigate the last two functions of the interface and determine how this interface works. The first function at `0x18000153C` will retrieve the password stored in the registry at:

`HKEY_CLASSES_ROOT\CLSID\{CEEACC6E-CCB2-4C4F-BCF6-D2176037A9A7}\Config\Password`

Following this is a tight loop over 256 bytes that matches RC4. The function takes one parameter (other than the `this` pointer). The one parameter is initialized with an RC4 context. The context follows a more standard RC4 context.

```
1.   typedef struct {
2.           unsigned char x, y , m[256];
```

```
3.  } rc4_ctx
```

As to writing a client for this, the structure can be considered opaque, as long as enough the structure is large enough that the program doesn't crash the initialization will work properly. At this point the first Interface function can be labeled `IFlags::Init`.

The second interface function at `0x1800016D8` uses the same RC4 context and decrypts a blob of data that is stored at `0x18001A9F0` and stores the Result in the registry at:

```
HKEY_CLASSES_ROOT\CLSID\{CEEACC6E-CCB2-4C4F-BCF6-D2176037A9A7}\Config\Flag
```

This is the desired flag, but leaves the question, "What is the password???".

CREATING A COM CLIENT

Most likely people solving this challenge will recognize the RC4 algorithm and not worry about creating a COM client for this challenge. If one so desires, the end of this write-up provides an example client to retrieve the flag. The sample code follows the two COM diagrams provided above on how COM interfaces work.

CAPCOM.SYS

Now that there is obviously more to this challenge than a COM server one can return to looking at clues given during initial triage. One option is to look at the cross references to `DeviceIOControl`. The IOCTL for the device being controlled in this case is `0xAA013044`. A quick search of this value will lead to many articles about a driver named `capcom.sys` and how it allows execution out of an untrusted application's memory space by disabling Supervisor Mode Execution Prevention (SMEP). At one point a buffer in user space could be passed to a kernel driver and be executed at a privilege level 0. Intel allocated a bit (0x14) into in the CR4 register that if set, the current privilege level (CPL) cannot be less than the requested privilege level (RPL). `Capcom.sys` will set this bit to zero and run the code being passed to it. One can always rely on `MmGetSystemRoutineAddress` to be the first parameter to the code being executed; this is something `capcom.sys` ensures. There appears to be a fair amount of code leading up to the call to `DeviceIoControl`, to understand the context of the input buffer for the call it is helpful to trace the function call stack. Following the cross references, one will dead end at a function that is within a table shown in Figure 14.

CPP GLOBAL INITIALIZERS

```
.rdata:000000014000F2B0 ; const _PVFV First
.rdata:000000014000F2B0 First           dq 0                    ; DATA XREF: __scrt_common_main_seh(void)+75↑o
.rdata:000000014000F2B8                 dq offset ?pre_cpp_initialization@@YAXXZ ; pre_cpp_initialization(void)
.rdata:000000014000F2C0                 dq offset sub_140001000
```

Figure 14: Global initializer table

This table is passed into a function, `initterm`, that is cross referenced by `__scrt_common_main_seh`. This function is one of many ways that code can be executed prior to main. This is an example of a global initializer, MSDN states:

By default, the linker includes the CRT library, which provides its own startup code. This startup code initializes the CRT library, calls global initializers, and then calls the user-provided `main` function for console applications.

Consider The following code

```
1.    int func(void)
2.    {
3.        return 3;
4.    }
5.
6.    int gi = func();
7.
8.    int main()
9.    {
10.     return gi;
11.   }
```

According to the C/C++ standard, func() must be called before main() is executed. (Microsoft, 2016)

This form of pre-main execution is one that can be easily overlooked, it doesn't have a break in debuggers and isn't listed as an export in IDA. The function at `0x14000100` will be labeled `PreMain`.

PREMAIN

The first function to get called is a standard dynamic loader of libraries and exports using multibyte-xor-encoded strings. This function can be found at `0x140001CD8`. The function uses the same common xor key that was used previously meaning it could have been cross-referenced to code prior to main. Although it can be observed, the wrapper functions do take an extra parameter that appears to be a junk variable. The following functions are resolved as global variables:

```
CreateService       StartService        OpenService         OpenSCManager
ControlService      CreateFile          CloseServiceHandle  DeleteService
```

BINARY DECRYPTION AND DECOMPRESSION

There are two binaries that are stored in the `.data` section in a encrypted and compressed format. They both are decrypted with the same password; this routine can be found at `0x140002370`. First a 256-bit key is created by creating a SHA256 digest of a hard-coded password:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/=
```

This password can be recognized as a Base64 alphabet but is never used for such purposes. The alphabet creates a SHA256 digest to be used as an encryption key:

```
e173d43de98094098259467ff632b4fc61496af96f3a354a006360d246e8166f
```

The encryption library used for the binaries is ChaCha20, a successor to Salsa20. The two algorithms are very similar and difficult to identify while reverse engineering. The designer of both crypto-algorithms, Daniel Bernstein has this to say,

ChaCha8 is a 256-bit stream cipher based on the 8-round cipher Salsa20/8. The changes from Salsa20/8 to ChaCha8 are designed to improve diffusion per round, conjecturally increasing resistance to cryptanalysis, while preserving—and often improving—time per round. ChaCha12 and ChaCha20 are analogous modifications of the 12-round and 20-round ciphers Salsa20/12 and Salsa20/20. (Bernstein, 2008)

Both ChaCha20 and Salsa20 have a distinct string signature, "`expand 32-byte k`", found in the init function, `0x140001010`. ChaCha20 does use a 64-bit nonce and is found to be a parameter to the setup function. The nonce in this case is all null bytes.

Once the data has been decrypted it is in a compressed state. From a static analysis perspective, it is difficult to determine the function's purpose at *0x140001610.* A dynamic approach to solving this portion is much more beneficial if one wants to determine the decompression routine used. Alternatively, one can dynamically debug passed the decompression function and dump the binary from memory. The decompression algorithm used is LZNT1, which is the same used for `RtlCompressBuffer`, but instead is using an external library.

The two binaries being decrypted and decompressed are: `capcom.sys` and a no name driver (`driver.sys`). Below is a portion of *capcom.sys* in a compressed stated. Enough of the data can be seen to know that this is a PE file.

```
00000000: 0E BA 00 4D 5A 90 00 03  00 00 00 02 04 00 30 FF  ...MZ.........0.
00000010: FF 00 00 B8 00 29 03 00  40 00 1F 00 C0 00 0C 0E  .....)..@.......
00000020: 1F 00 BA 0E 00 B4 09 CD  21 B8 00 01 4C CD 21 54  ........!...L.!T
00000030: 68 69 73 00 20 70 72 6F  67 72 61 6D 00 20 63 61  his. program. ca
00000040: 6E 6E 6F 74 20 00 62 65  20 72 75 6E 20 69 00 6E  nnot .be run i.n
00000050: 20 44 4F 53 20 6D 6F 80  64 65 2E 0D 0D 0A 24 04   DOS mo.de....$.
00000060: 86 00 ED F4 14 DD A9 95  7A 8E 11 07 03 7B 8E AE  ........z....{..
00000070: 00 07 DF 08 01 44 8E AA  02 07 14 8E A8 02 07 02  .....D..........
00000080: 61 02 07 52 69 63 68 01  27 05 83 50 00 45 00 00  a..Rich.'..P.E..
00000090: 64 86 05 00 15 08 14 CD  57 05 13 F0 00 22 00 00  d.......W....".."
000000A0: 0B 02 08 00 80 06 00 00  34 80 02 03 13 3C 00 0B  ........4....<..
000000B0: 02 E8 00 01 C5 02 0F 80  04 03 05 00 02 86 01 02  ................
000000C0: 00 52 0C 83 13 1B EA 85  14 04 83 0B 10 0D 85 2E  .R..............
000000D0: 10 8C 07 0A 05 00 80 0A  00 84 00 28 09 92 09 00  ...........(....
```

One program that I have found to be very helpful in determining compression algorithm is QuickBMS (https://aluigi.altervista.org/quickbms.htm) show in Figure 15. Using `quickbms.exe`, `comtype_scan2.bms` and `comtype_scan2.bat` many compression algorithms will be tried and output to a file. It is helpful if one knows the decompressed size and possibly the decompressed data. Both of these variables are available with this sample and therefore this approach can be used nicely. The `decrypt_decompress` function prototype is as follows as well as the driver size information

```
char* decrypt_decompress(char* data, size_t compressed_size, size_t uncompressed_size)
```

Driver Name	Compressed Size In Bytes	Uncompressed Size in Bytes
capcom.sys	8,069	10,576
driver.sys	8,877	22,528

Table 1: Driver compression statistics

Figure 15: QuickBMS

Given all this information the following snippet of code can be used to decrypt and decompress driver binaries stored in *crackstaller.exe.*

```
1.  def decrypt_decompress(data):
2.      import hashlib
3.      import lznt1
4.      from Crypto.Cipher import ChaCha20
5.
6.      nonce = b'\x00' * 8
7.      h = hashlib.sha256()
8.      h.update(b'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/=')
9.      key = h.digest()
10.     cipher = ChaCha20.new(key=key, nonce=nonce)
11.     return lznt1.decompress(cipher.decrypt(data))
```

LOADING AND UNLOADING CAPCOM

`Capcom.sys` is written to disk as `C:\windows\system32\cfs.sys`, but as seen in Procmon earlier it was not as obvious that a file was being written. This is due to the use of creating the file as a memory map and then copying the driver to the memory mapped region. This is performed in function `0x140002ED8`. After writing `capcom.sys` to disk the driver is loaded using the Service Control Manager APIs that were loaded earlier. The loading process can be found in function `0x140001FB4`. The loading function will also open the device and return it. This handle will be used for communicating with the device for sending an IOCTL.

There are three major functions within the scope left with `0x140002530`, they are:

- Deleting capcom service after exploit: `0x140001EB4`
- Delete capcom file (*cfs.sys)* With the `FILE_FLAG_DELETE_ON_CLOSE` being passed into CreateFile
- Load `driver.sys` using capcom: `0x140002C44`

LOADING DRIVER.SYS THROUGH CAPCOM.SYS

The term exploit is commonly used along with `capcom.sys`, although it is noted that this is a legitimately signed driver with the full intention of its primary and only capability, to disable SMEP and run code from userland. Code being passed to `capcom.sys` will be created more like shellcode carried by an exploit. `capcom.sys` will take the buffer passed into `DeviceIoControl` in the input buffer and execute it with a privilege level of 0. `capcom.sys` will execute what is at the address of the input buffer minus eight bytes.

A typical input buffer for a `capcom.sys` is:

```
1.  typedef struct _CAPCOM_EXPOIT_IO
2.  {
3.      PVOID sc_addr;
4.      SHELLCODE sc;
5.  }CAPCOM_EXPOIT_IO, * PCAPCOM_EXPOIT_IO;
```

The flow of function `0x140002C44`, `capcom_exploit`, begins by retrieving another multibyte-xor-encoded string, `DriverBootstrap`. The single function, `0x140002E84`, is called multiple times following, this function is a Relative Address to Pointer (`rva_to_ptr`) address conversion function for PE binaries. Because the PE is being dealt with in its disk format the addresses must be converted to a raw address pointer. `capcom_exploit` parses the driver PE to find its `IMAGE_EXPORT_DIRECTORY`. The exports for the driver are parsed searching for a string comparison match of `DriverBootstrap`. Once the export is found, `capcom_exploit` saves the offset to this function, note this is not the address that you would retrieve with `GetProcAddress`. Next some global variables are set as seen in Table 2.

Size in Bytes	Address	Description
4	0x14003638B	DriverBootstrap offset
8	0x14003637B	Driver Address
4	0x140036385	Driver Size

Table 2: Global variables

Lastly, prior to calling `DeviceIoControl`, the `CAPCOM_EXPLOIT_IO` structure is populated. The `sc` field is assigned with the global address of `0x140036338`. The previous table values were markups within this global shellcode. Knowing that this is shellcode one can analyze the following address as code shown in Figure 16.

Figure 16: capcom shellcode

The shellcode is executed first at PASSIVE_LEVEL with interruption in a disabled state. The shellcode first enables interruptions so that Windows can page-in the kernel payload even if it is paged-out, and the payload can call kernel APIs. Then the shellcode sets parameters to the calling function being transferred to at `0x140036355`, `PreBootStrap`, `0x140002A10` with prototype:

```
1. void PreBootStrap (
2.     MmGetSystemRoutineAddress_t* MmGetSystemRoutineAddress,
3.     void * driver_address,
4.     unsigned long driver_size
5.     unsigned long DriverBootstrap_offset);
```

Once sending an IOCTL to capcom the code that is in the address space of `crackstaller.exe` will be executed at privilege level 0. The shellcode is a small buffer that was run in the address space of `capcom.sys`, but when the jump to `PreBootStrap` occurred, the address space returned to `crackstaller.exe`, which is still within the same IDB.

Some may have taken a quick look over `PreBootStrap` and quickly written it of as creating a context structure to pass to the driver that is passed with `PsCreateSystemThread`. While this is true, there is one other bit of detail that can end up causing problems later.

PREBOOTSTRAP

The reflective loader prefers to resolve function address over using `MmGetSystemRoutineAddress`, it does this by implementing `MmGetSystemRoutineAddress` and comparing pre-determined hash values of the names it is looking for. There is a small chicken before the egg problem that arises, to implement `MmGetSystemRoutineAddress`, one must know the kernel base address. While there are some methods to do this without makings system calls, I found stable code to be my best reliable option and therefore chose to use `ZwQuerySystemInformation` and look through the `SystemModuleInformation` list and find the hash of `ntoskrnl.exe`. Certain API's needed to be resolved with `MmGetSystemRoutineAddress` to find the kernel base address. These strings were multibyte-xor encoded. The hash function that was used when finding `ntoskrnl` base address, is the same used when resolving the rest of the functions later. The function is very similar to that used in Metasploit. Metasploit creates a hash by rotating the hash right by 13 bits each time, `GetHash` in `crackstaller.exe` rotates right 14. Using the following code, one can create hashes of all their desired strings. A simple script could be used to parse the exports of `ntoskrnl.exe` and create hashes, or, one could create a script that reverses the process and run it within IDA to fix up labels. The provided script assumes strings are in a lower case format. Table 3 Displays the hashes used for this challenge.

```python
def ror(dword, bits):
    return (dword >> bits | dword << (32 - bits)) & 0xFFFFFFFF

def GetHash(function, bits=14):
    function_hash = 0
    for c in str(function + b'\x00'):
        function_hash = ror(function_hash, bits)
        function_hash += ord(c)
    return function_hash & 0xFFFFFFFF
```

Hash	String
0x490A231A	ExAllocatePoolWithTag
0x34262863	ExFreePoolWithTag
0x01128974	IoCreateDriver
0xE2A9259B	RtlImageNtHeader
0xCF424038	RtlImageDirectoryEntryToData
0xCE968D51	RtlQueryModuleInformation
0xB40D00D9	PsCreateSystemThread
0xA95BE347	ZwClose
0xC036346A	Ntoskrnl.exe

Table 3: Hash Table

prior to calling PsCreateSystemThread a context structure is created, it can be helpful to create a context structure in IDA; it will be passed to the `driver.sys` with `PsCreateSystemThread` so it will need to be created there too.

```
00000000 DRIVER_CONTEXT  struc ; (sizeof=0x40, mappedto_64)
00000000 ExAllocatePoolWithTag dq ?
00000008 ExFreePoolWithTag dq ?
00000010 RtlImageNtHeader dq ?
00000018 RtlImageDirectoryEntryToData dq ?
00000020 RtlQueryModuleInformation dq ?
00000028 IoCreateDriver  dq ?
00000030 raw_driver      dq ?
00000038 raw_driver_size dd ?
0000003C bootstrap_offset dd ?
00000040 DRIVER_CONTEXT  ends
```

Figure 17: DRIVER_CONTEXT Struct

Lastly before calling `PsCreateSystemThread` there is a scan through the driver doing a search and replace of the value `0xDC16F3C3B57323` to `0x41424143414242` (`ABACABB`). This value ends up being the password used in decrypting the password to be used to retrieve the flag. This value was chosen because Capcom, a big gaming company, didn't have any popular cheat codes from back in the day. The Konami code is probably the most popular cheat code, but doesn't translate well to a short hex value, so I went with the blood code from the original Mortal Combat on Sega Genesis.

Both `PreBootstrap` and `DriverBootstrap` could have been combined into one function that all occurred in `PreBootstrap`. The intention was to separate functionality between two separate binaries, making it more needed to reverse both. They both share data needed by each other, specifically the password gets patched in one binary that affects the other.

REFLECTIVE LOADER

Execution is finally happening out of the driver codebase and is also in kernel address space. `PsCreateSystemThread` has the driver context prepared above passed into it. Immediately upon entry to `DriverBootstrap` a new context is created and values from the context passed in are copied over as seen in Figure 17.

```
00000000 LoaderFuncs     struc ; (sizeof=0x30, mappedto_25)
00000000                                       ; XREF: DriverBootstrap/r
00000000 ExAllocatePoolWithTag dq ?
00000008 ExFreePoolWithTag dq ?
00000010 IoCreateDriver  dq ?
00000018 RtlImageNtHeader dq ?
00000020 RtlImageDirectoryEntryToData dq ?
00000028 RtlQueryModuleInformation dq ?
00000030 LoaderFuncs     ends
```

Figure 18: Loader function structure

The code of `DriverBootstrap` follows the same logic of other reflective loaders

1. Allocate Memory from `nt->OptionalHeader.SizeOfImage`
2. Copy headers from raw driver to loaded driver
3. Load each `IMAGE_`*`SECTION_HEADER`* with the count from `nt->FileHeader.NumberOfSections`
4. Fixup relocations
5. Fixup imports table
6. Call `IoCreateDriver` with `DriverEntry` found at `nt->OptionalHeader.AddressOfEntryPoint`

DRIVER ENTRY

Because this driver was loaded with `IoCreateDriver`, an undocumented API call, it comes with some caveats: the driver doesn't have a `RegistryPath`, and many parameters in the `DriverObject` will not be populated, The only way to unload the driver is to call `DriverUnload` from code as the SCM didn't load

it, it isn't registered so it can't be unloaded as such. Once again it is observed the use of the same multibyte-xor-key being used, the new addition to this is a wrapper that populates a `UNICODE_STRING` on the stack with the resulting string.

`DriverEntry` performs the following:

1. Resolves the UNICODE_STRING, `360000`, which is the Altitude used when registering a Registry Callback
2. Assigns the `DriverUnload` function to `DriverObject->DriverUnload`
3. Creates an unnamed device with memory allocated for an extension, which is to be be populated during the execution of the driver as seen in Figure 18
4. Initializes a mutex and a notification event and stores them in the device extension
5. Registers a registry callback routine with `CmRegisterCallbackEx` *(`0x140004570`)*

```
00000000 DeviceContext   struc ; (sizeof=0x59, mappedto_37)
00000000 registry_count  dd ?
00000004 reference_count dd ?
00000008 lock            FAST_MUTEX ?
00000040 event_unload    KEVENT ?
00000058 unloading       db ?
00000059 DeviceContext   ends
```

Figure 19: Device Extension

REGISTRY CALLBACK

Beginning in Windows Vista a driver can call `CmRegisterCallbackEx` to register a Registry Callback, which is called every time a thread performs an operation on the registry. (Microsoft, 2018) The callback routine has the following prototype:

```
1. EX_CALLBACK_FUNCTION ExCallbackFunction;
2.
3. NTSTATUS ExCallbackFunction(
4.   PVOID CallbackContext,
5.   PVOID Argument1,
6.   PVOID Argument2
7. );
```

The `CallbackContext` is specified when registering, in the case of this driver, the `DriverObject` is the context. From the driver object one can access the `DeviceObject` and ultimately the `DeviceExtension`. As stated before, this SCM won't unload this driver so when the purpose of this driver has been performed it will unload itself. The `DriverUnload` function is stored within the `DriverObject`, lastly to create a system thread the `DriverObject` itself needs to be known.

Argument1 contains a REG_NOTIFY_CLASS enumeration value, this specifies to the callback routine what type of operation is being filtered currently. Argument2 contains a pointer to a structure that contains information on the registry operation which is dependent on Argument1.

RegistryCallback for this driver seems lengthy as most kernel code tends to be. It initially does a comparison of Argument1 against RegNtPreCreateKeyEx. Next some synchronization happens to ensure the driver isn't unloading. Because this function is dealing with RegNtPreCreateKeyEx the corresponding structure for Argument2 is REG_CREATE_KEY_INFORMATION_V1. The function will go through a fair amount of work ensuring that it has the full key correctly and then does a string comparison against

```
{CEEACC6E-CCB2-4C4F-BCF6-D2176037A9A7}\Config
```

Next SHA256 code as analyzed in crackstaller.exe appears, this time the password is as shown in Figure 19.

```
.data:000000014000608C chacha20_password db 23h, 73h, 0B5h, 0C3h, 0F3h, 16h, 0DCh, 0
```

Figure 20: ChaCha20 password

Now if one is observant enough this will look familiar to the data that was patched during the loader. If not, then the key for decrypting the flag will be incorrect. If one is debugging this dynamically the password will show correctly. When patching the binary don't let endianness screw things up, it should appear as.

```
{0x42, 0x42, 0x41, 0x43, 0x41, 0x42, 0x41}
```

Dependent on the version of IDA Pro the function for chacha20_decrypt will be mislabeled to wcstombs_l at address 0x1400049A4. It is possible to run this driver in x64dbg as a user executable and upon driver entry set EIP to 0x1400048D8. Some massaging needs to take place to set up some values, but decryption can be done without being in kernel and completely bypass ever having to do kernel debugging. This is because the SHA256 routines and chacha20 routines neither have any kernel API calls. Because of this one will find anti-debugging scattered throughout them, some calls to KeDelayExecutionThread, also the decryption will allocate memory to decrypt, then copy to the return buffer. All these measures can still be bypassed in a user debugger but make it more a pain.

When a registry filtering driver's RegistryCallback routine receives a pre-notification, the routine can handle the registry operation itself and then return STATUS_CALLBACK_BYPASS. When the registry receives STATUS_CALLBACK_BYPASS from the driver, it just returns STATUS_SUCCESS to the calling thread and does not process the operation. The driver preempts the registry operation and must completely handle it, and the driver must be careful to return valid output values in the REG_XXX_KEY_INFORMATION structure. (Microsoft, 2017)

This driver changes the return status to STATUS_CALLBACK_BYPASS, when it has found a match to the key it is searching for and creates the same key, with one addition. It stores the decrypted password in a registry class string. When creating a key with the API RegCreateKeyEx one of the parameters is lpClass; "The user-defined class type of this key. This parameter may be ignored. This parameter can be NULL." (Microsoft, 2018).

This parameter has never really been used and doesn't have any great documentation on what it is. Data can be stored in it and it will not be displayed with tools such as regedit, reg.exe, PowerShell and even Python. In python pywin32::win32api.RegQueryInfoKey doesn't return class strings. I have ran into one case where malware stored their configuration's encryption key within a class type string, and from that this challenge received influence. Also Microsoft stores their 16 byte boot key, SysKey, an encryption key used to unlock the SAM database across four separate keys as class type strings:

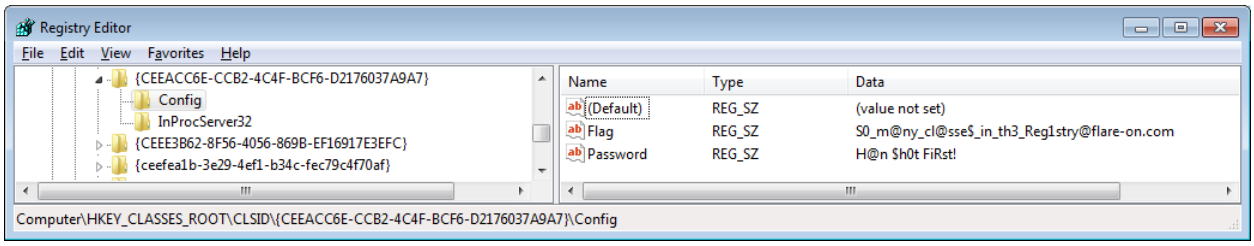
```
SYSTEM\CurrentControlSet\Control\Lsa\{JD,Skew1,GBG,Data}
```

The correct way to go about retrieving the password to decrypt the flag is to write the code with C.

```
1.  void PrintPassword()
2.  {
3.              HKEY hkey = NULL;
4.      CHAR password[MAX_PATH] = {0};
5.      DWORD buf_size = MAX_PATH;
6.
7.      if (ERROR_SUCCESS != (result = RegOpenKeyA(
8.          HKEY_CLASSES_ROOT,
9.          "CLSID\\{CEEACC6E-CCB2-4C4F-BCF6-D2176037A9A7}\\Config,
10.       &hkey))) return;
11.
12.   if (ERROR_SUCCESS != (result = RegQueryInfoKeyA(
13.       hkey,
14.       password,
15.       &buf_size,
16.       NULL,
17.       NULL, NULL, NULL, NULL,
18.       NULL, NULL, NULL, NULL))) return;
19.
20.   if (0 == buf_size) return;
21.   printf("password: %s\n", password);
22.   return;
23. }
24.
```

How to solve

1. Execute *crackstaller.exe*
2. Retrieve the password by reading the class type string in
    - CLSID\{CEEACC6E-CCB2-4C4F-BCF6-D2176037A9A7}\Config
3. Place the password in
    - *CLSID\{CEEACC6E-CCB2-4C4F-BCF6-D2176037A9A7}\Config\Password*
4. Use a COM client to call Flags::Init and then Flags::Message
5. Retrieve the flag from
    - *CLSID\{CEEACC6E-CCB2-4C4F-BCF6-D2176037A9A7}\Config\Flag*



Flags COM Client

```
1.  #include <stdio.h>
2.  #include <initguid.h>
3.  #include <objbase.h>
4.
5.  typedef struct {
6.      unsigned char x, y, m[256];
7.  } rc4_ctx;
8.
9.  // {CEEACC6E-CCB2-4C4F-BCF6-D2176037A9A7}
10. DEFINE_GUID(CLSID_FlagsServer,
11.     0xceeacc6e, 0xccb2, 0x4c4f, 0xbc, 0xf6, 0xd2, 0x17, 0x60, 0x37, 0xa9, 0xa7);
12.
13. // {E27297B0-1E98-4033-B389-24ECA246002A}
14. DEFINE_GUID(IID_IFlags,
15.     0xe27297b0, 0x1e98, 0x4033, 0xb3, 0x89, 0x24, 0xec, 0xa2, 0x46, 0x0, 0x2a);
16.
17. DECLARE_INTERFACE_(IFlags, IUnknown)
18. {
19.     STDMETHOD(Init)(rc4_ctx* ctx) PURE;
20.     STDMETHOD(Message)(rc4_ctx * ctx) PURE;
21. };
22.
23. HRESULT FlagViaCoGetClassObject()
24. {
25.     IClassFactory* pCf = NULL;
26.     IUnknown* pUnk = NULL;
27.     IFlags* pFlags = NULL;
28.     HRESULT hr;
29.     rc4_ctx ctx;
30.
31.     hr = CoInitialize(NULL);
32.     if (FAILED(hr))
33.     {
34.         printf("CoInitialize Failed!\n");
35.         goto end;
36.     }
37.
38.     hr = CoGetClassObject(
39.            CLSID_FlagsServer,
40.            CLSCTX_INPROC_SERVER,
41.            NULL,
42.            IID_IClassFactory,
43.            (void**)&pCf);
44.     if (FAILED(hr))
45.     {
46.         printf("CoGetClassObject Failed: %X\n", hr);
47.         goto end;
48.     }
49.
50.     hr = pCf->CreateInstance(NULL, IID_IUnknown, (void**)&pUnk);
51.     if (FAILED(hr))
52.     {
53.         printf("CreateInstance Failed: %X\n", hr);
54.         goto end;
55.     }
56.
```

```
57.     hr = pUnk->QueryInterface(IID_IFlags, (void**)&pFlags);
58.     if (FAILED(hr))
59.     {
60.     printf("QueryInterface Failed: %X\n", hr);
61.         goto end;
62.     }
63.
64.     hr = pFlags->Init(&ctx);
65.     if (FAILED(hr))
66.     {
67.         printf("Init failed: 0x%08X\n", hr);
68.         goto end;
69.     }
70.
71.     hr = pFlags->Message(&ctx);
72.     if (FAILED(hr))
73.     {
74.         printf("Message Failed: 0x%08X\n", hr);
75.         goto end;
76.     }
77.
78. end:
79.     if (pFlags)pFlags->Release();
80.     if (pUnk) pUnk->Release();
81.     if (pCf) pCf->Release();
82.
83.     CoUninitialize();
84.     return hr;
85. }
86.
87. HRESULT FlagViaCoCreateInstance()
88. {
89.     IFlags* pFlags = NULL;
90.     HRESULT hr;
91.     rc4_ctx ctx;
92.
93.     hr = CoInitialize(NULL);
94.     if (FAILED(hr))
95.     {
96.         printf("CoInitialize Failed!\n");
97.         goto end;
98.     }
99.
100.    hr = CoCreateInstance(
101.            CLSID_FlagsServer,
102.            NULL,
103.            CLSCTX_INPROC_SERVER,
104.            IID_IFlags,
105.            (void**)&pFlags);
106.    if (FAILED(hr))
107.    {
108.        printf("CoCreateInstance Failed!\n");
109.        goto end;
110.    }
111.
112.    hr = pFlags->Init(&ctx);
113.    if (FAILED(hr))
114.    {
115.        printf("Init failed: 0x%08X\n", hr);
116.        goto end;
```

```
117.    }
118.
119.    hr = pFlags->Message(&ctx);
120.    if (FAILED(hr))
121.    {
122.        printf("Message Failed: 0x%08X\n", hr);
123.        goto end;
124.    }
125.
126.end:
127.    if (pFlags)pFlags->Release();
128.    CoUninitialize();
129.    return hr;
130.}
131.
132.int main()
133.{
134.    HRESULT hr;
135.    //hr = FlagViaCoGetClassObject();
136.    hr = FlagViaCoCreateInstance();
137.    return hr;
138.}
```

References

Bernstein, D. J. (2008). *ChaCha, a variant of Salsa20.* Chicago: The University of Illinois at Chicago.

Microsoft. (2016, 04 11). *CRT Initialization*. Retrieved from MSDN: https://docs.microsoft.com/en-us/cpp/c-runtime-library/crt-initialization?view=vs-2019

Microsoft. (2017, 06 16). *Handling Notifications*. Retrieved from MSDN: https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/handling-notifications

Microsoft. (2018, 04 30). *CmRegisterCallbackEx function*. Retrieved from MSDN: https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-cmregistercallbackex

Microsoft. (2018, 05 31). *Component Object Model (COM)*. Retrieved from MSDN: https://docs.microsoft.com/en-us/windows/win32/com/component-object-model--com--portal

Microsoft. (2018, 12 05). *RegCreateKeyExA*. Retrieved from MSDN: https://docs.microsoft.com/en-us/windows/win32/api/winreg/nf-winreg-regcreatekeyexa