Injecting Code into Windows Protected Processes using COM - Part 1

googleprojectzero.blogspot.com/2018/10/injecting-code-into-windows-protected.html

Posted by James Forshaw, Google Project Zero

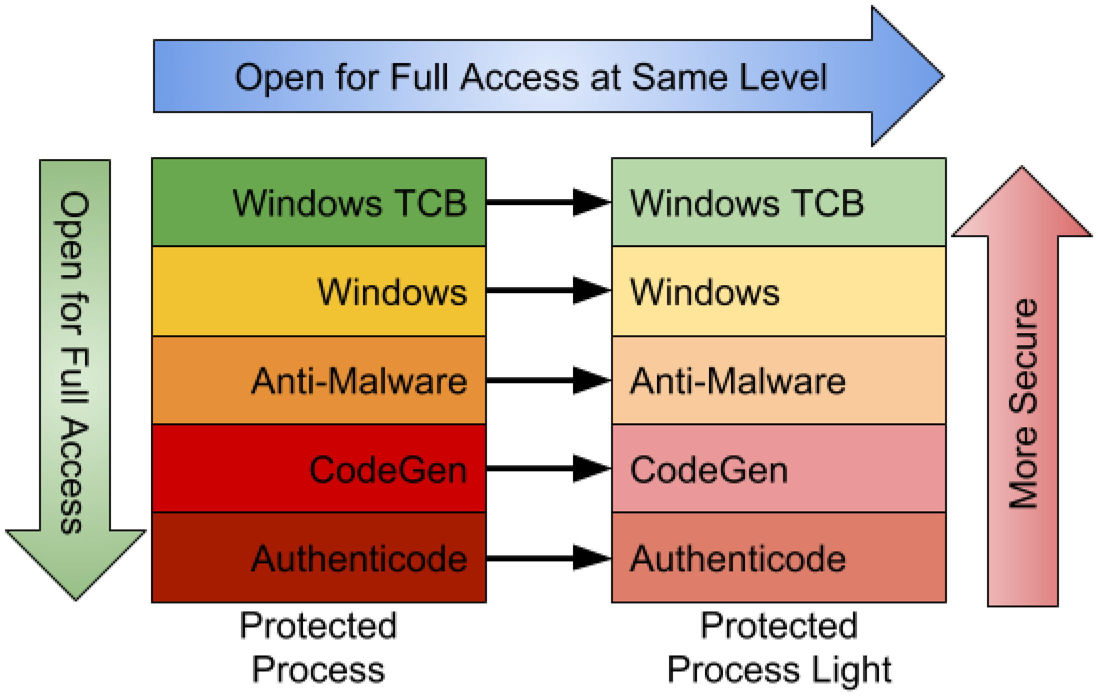
At Recon Montreal 2018 I presented “Unknown Known DLLs and other Code Integrity Trust Violations” with Alex Ionescu. We described the implementation of Microsoft Windows’ Code Integrity mechanisms and how Microsoft implemented Protected Processes (PP). As part of that I demonstrated various ways of bypassing Protected Process Light (PPL), some requiring administrator privileges, others not.

In this blog I’m going to describe the process I went through to discover a way of injecting code into a PPL on Windows 10 1803. As the only issue Microsoft considered to be violating a defended security boundary has now been fixed I can discuss the exploit in more detail.

Background on Windows Protected Processes

The origins of the Windows Protected Process (PP) model stretch back to Vista where it was introduced to protect DRM processes. The protected process model was heavily restricted, limiting loaded DLLs to a subset of code installed with the operating system. Also for an executable to be considered eligible to be started protected it must be signed with a specific Microsoft certificate which is embedded in the binary. One protection that the kernel enforced is that a non-protected process couldn’t open a handle to a protected process with enough rights to inject arbitrary code or read memory.

In Windows 8.1 a new mechanism was introduced, Protected Process Light (PPL), which made the protection more generalized. PPL loosened some of the restrictions on what DLLs were considered valid for loading into a protected process and introduced different signing requirements for the main executable. Another big change was the introduction of a set of signing levels to separate out different types of protected processes. A PPL in one level can open for full access any process at the same signing level or below, with a restricted set of access granted to levels above. These signing levels were extended to the old PP model, a PP at one level can open all PP and PPL at the same signing level or below, however the reverse was not true, a PPL can never open a PP at any signing level for full access. Some of the levels and this relationship are shown below:



Signing levels allow Microsoft to open up protected processes to third-parties, although at the current time the only type of protected process that a third party can create is an Anti-Malware PPL. The Anti-Malware level is special as it allows the third party to add additional permitted signing keys by registering an Early Launch Anti-Malware (ELAM) certificate. There is also Microsoft's TruePlay, which is an Anti-Cheat technology for games which uses components of PPL but it isn't really important for this discussion.

I could spend a lot of this blog post describing how PP and PPL work under the hood, but I recommend reading the blog post series by Alex Ionescu instead (Parts 1, 2 and 3) which will do a better job. While the blog posts are primarily based on Windows 8.1, most of the concepts haven't changed substantially in Windows 10.

I've written about Protected Processes before [link], in the form of the custom implementation by Oracle in their VirtualBox virtualization platform on Windows. The blog showed how I bypassed the process protection using multiple different techniques. What I didn't mention at the time was the first technique I described, injecting JScript code into the process, also worked against Microsoft's PPL implementation. I reported that I could inject arbitrary code into a PPL to Microsoft (see Issue 1336) from an abundance of caution in case Microsoft wanted to fix it. In this case Microsoft decided it wouldn't be fixed as a security bulletin. However Microsoft did fix the issue in the next major release on Windows (version 1803) by adding the following code to CI.DLL, the Kernel's Code Integrity library:

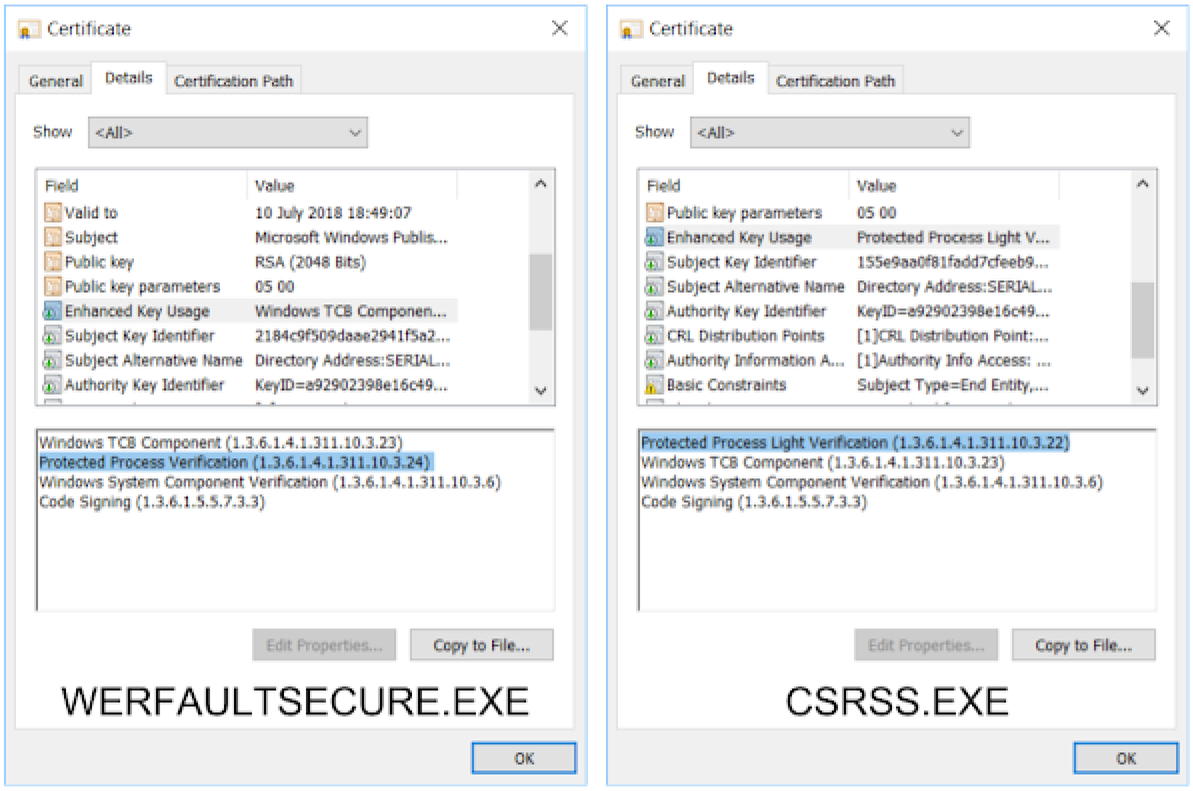
```
UNICODE_STRING g_BlockedDllsForPPL[] = {
 DECLARE_USTR("scrobj.dll"),
 DECLARE_USTR("scrrun.dll"),
 DECLARE_USTR("jscript.dll"),
 DECLARE_USTR("jscript9.dll"),
 DECLARE_USTR("vbscript.dll")
};

NTSTATUS CipMitigatePPLBypassThroughInterpreters(PEPROCESS Process,
                                     LPBYTE Image,
                                     SIZE_T ImageSize) {
 if (!PsIsProtectedProcess(Process))
   return STATUS_SUCCESS;

 UNICODE_STRING OriginalImageName;
 // Get the original filename from the image resources.
 SIPolicyGetOriginalFilenameAndVersionFromImageBase(
    Image, ImageSize, &OriginalImageName);
 for(int i = 0; i < _countof(g_BlockedDllsForPPL); ++i) {
   if (RtlEqualUnicodeString(g_BlockedDllsForPPL[i],
                      &OriginalImageName, TRUE)) {
    return STATUS_DYNAMIC_CODE_BLOCKED;
   }
 }
 return STATUS_SUCCESS;
}
```

The fix checks the original file name in the resource section of the image being loaded against a blacklist of 5 DLLs. The blacklist includes DLLs such as JSCRIPT.DLL, which implements the original JScript scripting engine, and SCROBJ.DLL, which implements scriptlet objects. If the kernel detects a PP or PPL loading one of these DLLs the image load is rejected with STATUS_DYNAMIC_CODE_BLOCKED. This kills my exploit, if you modify the resource section of one of the listed DLLs the signature of the image will be invalidated resulting in the image load failing due to a cryptographic hash mismatch. It's actually the same fix that Oracle used to block the attack in VirtualBox, although that was implemented in user-mode.

Finding New Targets

The previous injection technique using script code was a generic technique that worked on any PPL which loaded a COM object. With the technique fixed I decided to go back and look at what executables will load as a PPL to see if they have any obvious vulnerabilities I could exploit to get arbitrary code execution. I could have chosen to go after a full PP, but PPL seemed the easier of the two and I've got to start somewhere. There's so many ways to inject into a PPL if we could just get administrator privileges, the least of which is just loading a kernel driver. For that reason any vulnerability I discover must work from a normal user account. Also I wanted to get the highest signing level I can get, which means PPL at Windows TCB signing level.

The first step was to identify executables which run as a protected process, this gives us the maximum attack surface to analyze for vulnerabilities. Based on the blog posts from Alex it seemed that in order to be loaded as PP or PPL the signing certificate needs a special Object Identifier (OID) in the certificate's Enhanced Key Usage (EKU) extension. There are separate OID for PP and PPL; we can see this below with a comparison between WERFAULTSECURE.EXE, which can run as PP/PPL, and CSRSS.EXE, which can only run as PPL.



WERFAULTSECURE.EXE

Field	Value
Valid to	10 July 2018 18:49:07
Subject	Microsoft Windows Publis...
Public key	RSA (2048 Bits)
Public key parameters	05 00
Enhanced Key Usage	Windows TCB Componen...
Subject Key Identifier	2184c9f509daae2941f5a2...
Subject Alternative Name	Directory Address:SERIAL...
Authority Key Identifier	KeyID=a92902398e16c49...

```
Windows TCB Component (1.3.6.1.4.1.311.10.3.23)
Protected Process Verification (1.3.6.1.4.1.311.10.3.24)
Windows System Component Verification (1.3.6.1.4.1.311.10.3.6)
Code Signing (1.3.6.1.5.5.7.3.3)
```

CSRSS.EXE

Field	Value
Public key parameters	05 00
Enhanced Key Usage	Protected Process Light V...
Subject Key Identifier	155e9aa0f81fadd7cfeeb9...
Subject Alternative Name	Directory Address:SERIAL...
Authority Key Identifier	KeyID=a92902398e16c49...
CRL Distribution Points	[1]CRL Distribution Point:...
Authority Information A...	[1]Authority Info Access: ...
Basic Constraints	Subject Type=End Entity,...

```
Protected Process Light Verification (1.3.6.1.4.1.311.10.3.22)
Windows TCB Component (1.3.6.1.4.1.311.10.3.23)
Windows System Component Verification (1.3.6.1.4.1.311.10.3.6)
Code Signing (1.3.6.1.5.5.7.3.3)
```

I decided to look for executables which have an embedded signature with these EKU OIDs and that'll give me a list of all executables to look for exploitable behavior. I wrote the Get-EmbeddedAuthenticodeSignature cmdlet for my NtObjectManager PowerShell module to extract this information.

At this point I realized there was a problem with the approach of relying on the signing certificate, there's a lot of binaries I expected to be allowed to run as PP or PPL which were missing from the list I generated. As PP was originally designed for DRM there was no obvious executable to handle the Protected Media Path such as AUDIODG.EXE. Also, based on my previous research into Device Guard and Windows 10S, I knew there must be an executable in the .NET framework which could run as PPL to add cached signing level information to NGEN generated binaries (NGEN is an Ahead-of-Time JIT to convert a .NET assembly into native code). The criteria for PP/PPL were more fluid than I expected. Instead of doing static analysis I decided to perform dynamic analysis, just start protected every executable I could enumerate and query the protection level granted. I wrote the following script to test a single executable:

```
Import-Module NtObjectManager

function Test-ProtectedProcess {
  [CmdletBinding()]
  param(
     [Parameter(Mandatory, ValueFromPipelineByPropertyName)]
     [string]$FullName,
     [NtApiDotNet.PsProtectedType]$ProtectedType = 0,
     [NtApiDotNet.PsProtectedSigner]$ProtectedSigner = 0
     )
  BEGIN {
     $config = New-NtProcessConfig abc -ProcessFlags ProtectedProcess `
        -ThreadFlags Suspended -TerminateOnDispose `
        -ProtectedType $ProtectedType `
        -ProtectedSigner $ProtectedSigner
  }

  PROCESS {
     $path = Get-NtFilePath $FullName
     Write-Host $path
     try {
        Use-NtObject($p = New-NtProcess $path -Config $config) {
           $prot = $p.Process.Protection
           $props = @{
              Path=$path;
              Type=$prot.Type;
              Signer=$prot.Signer;
              Level=$prot.Level.ToString("X");
           }
           $obj = New-Object –TypeName PSObject –Prop $props
           Write-Output $obj
        }
     } catch {
     }
  }
```

}

When this script is executed a function is defined, Test-ProtectedProcess. The function takes a path to an executable, starts that executable with a specified protection level and checks whether it was successful. If the ProtectedType and ProtectedSigner parameters are 0 then the kernel decides the “best” process level. This leads to some annoying quirks, for example SVCHOST.EXE is explicitly marked as PPL and will run at PPL-Windows level, however as it’s also a signed OS component the kernel will determine its maximum level is PP-Authenticode. Another interesting quirk is using the native process creation APIs it’s possible to start a DLL as main executable image. As a significant number of system DLLs have embedded Microsoft signatures they can also be started as PP-Authenticode, even though this isn’t necessarily that useful. The list of binaries that will run at PPL is shown below along with their maximum signing level.

Path	Signing Level
C:\windows\Microsoft.Net\Framework\v4.0.30319\mscorsvw.exe	CodeGen
C:\windows\Microsoft.Net\Framework64\v4.0.30319\mscorsvw.exe	CodeGen
C:\windows\system32\SecurityHealthService.exe	Windows
C:\windows\system32\svchost.exe	Windows
C:\windows\system32\xbgmsvc.exe	Windows
C:\windows\system32\csrss.exe	Windows TCB
C:\windows\system32\services.exe	Windows TCB
C:\windows\system32\smss.exe	Windows TCB
C:\windows\system32\werfaultsecure.exe	Windows TCB
C:\windows\system32\wininit.exe	Windows TCB

Injecting Arbitrary Code Into NGEN

After carefully reviewing the list of executables which run as PPL I settled on trying to attack the previously mentioned .NET NGEN binary, MSCORSVW.EXE. My rationale for choosing the NGEN binary was:

- Most of the other binaries are service binaries which might need administrator privileges to start correctly.
- The binary is likely to be loading complex functionality such as the .NET framework as well as having multiple COM interactions (my go-to technology for weird behavior).

- In the worst case it might still yield a Device Guard bypass as the reason it runs as PPL is to give it access to the kernel APIs to apply a cached signing level. Any bug in the operation of this binary might be exploitable even if we can't get arbitrary code running in a PPL.

But there is an issue with the NGEN binary, specifically it doesn't meet my own criteria that I get the top signing level, Windows TCB. However, I knew that when Microsoft fixed Issue 1332 they left in a back door where a writable handle could be maintained during the signing process if the calling process is PPL as shown below:

```
NTSTATUS CiSetFileCache(HANDLE Handle, ...) {

 PFILE_OBJECT FileObject;
 ObReferenceObjectByHandle(Handle, &FileObject);

 if (FileObject->SharedWrite ||
    (FileObject->WriteAccess &&
     PsGetProcessProtection().Type != PROTECTED_LIGHT)) {
   return STATUS_SHARING_VIOLATION;
 }

 // Continue setting file cache.
}
```

If I could get code execution inside the NGEN binary I could reuse this backdoor to cache sign an arbitrary file which will load into any PPL. I could then DLL hijack a full PPL-WindowsTCB process to reach my goal.

To begin the investigation we need to determine how to use the MSCORSVW executable. Using MSCORSVW is not documented anywhere by Microsoft, so we'll have to do a bit of digging. First off, this binary is not supposed to be run directly, instead it's invoked by NGEN when creating an NGEN'ed binary. Therefore, we can run the NGEN binary and use a tool such as Process Monitor to capture what command line is being used for the MSCORSVW process. Executing the command:

```
C:\> NGEN install c:\some\binary.dll
```

Results in the following command line being executed:

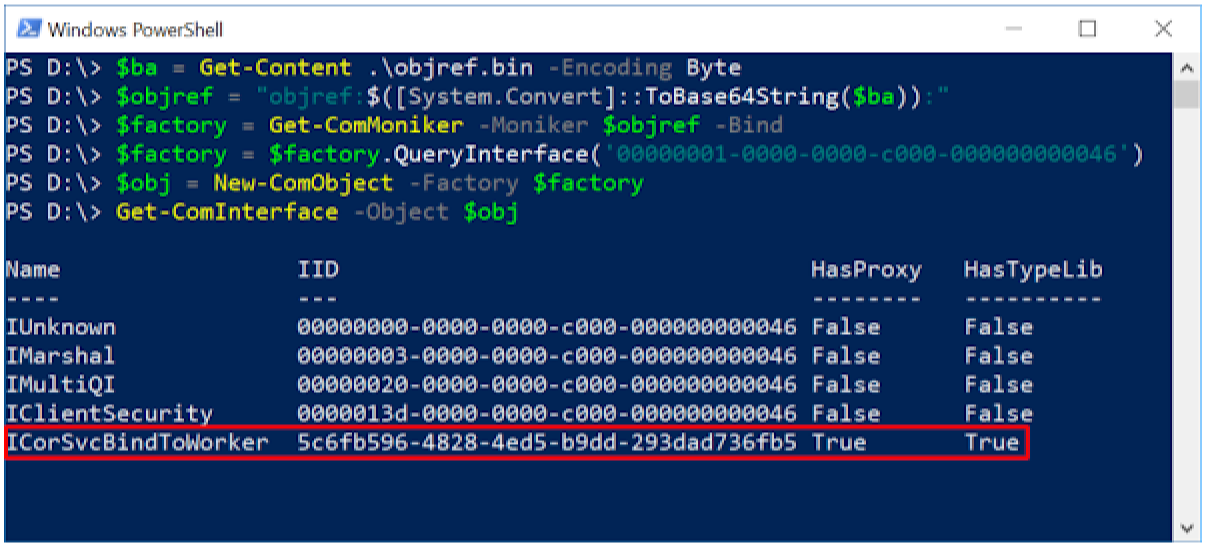
```
MSCORSVW -StartupEvent A -InterruptEvent B -NGENProcess C -Pipe D
```

A, B, C and D are handles which NGEN ensures are inherited into the new process before it starts. As we don't see any of the original NGEN command line parameters it seems likely they're being passed over an IPC mechanism. The "Pipe" parameter gives an indication that named pipes are used for IPC. Digging into the code in MSCORSVW, we find the method NGenWorkerEmbedding, which looks like the following:

```
void NGenWorkerEmbedding(HANDLE hPipe) {
 CoInitializeEx(nullptr, COINIT_APARTMENTTHREADED);
 CorSvcBindToWorkerClassFactory factory;

 // Marshal class factory.
 IStream* pStm;
 CreateStreamOnHGlobal(nullptr, TRUE, &pStm);
 CoMarshalInterface(pStm, &IID_IClassFactory, &factory,

                    MSHCTX_LOCAL, nullptr, MSHLFLAGS_NORMAL);

 // Read marshaled object and write to pipe.
 DWORD length;
 char* buffer = ReadEntireIStream(pStm, &length);
 WriteFile(hPipe, &length, sizeof(length));
 WriteFile(hPipe, buffer, length);
 CloseHandle(hPipe);

 // Set event to synchronize with parent.
 SetEvent(hStartupEvent);

 // Pump message loop to handle COM calls.
 MessageLoop();

 // ...
}
```

This code is not quite what I expected. Rather than using the named pipe for the entire communication channel it's only used to transfer a marshaled COM object back to the calling process. The COM object is a class factory instance, normally you'd register the factory using CoRegisterClassObject but that would make it accessible to all processes at the same security level so instead by using marshaling the connection can be left private only to the NGEN binary which spawned MSCORSVW. A .NET related process using COM gets me interested as I've previously described in another blog post how you can exploit COM objects implemented in .NET. If we're lucky this COM object is implemented in .NET, we can determine if it is implemented in .NET by querying for its interfaces, for example we use the

Get-ComInterface command in my OleViewDotNet PowerShell module as shown in the following screenshot.



```
PS D:\> $ba = Get-Content .\objref.bin -Encoding Byte
PS D:\> $objref = "objref:$([System.Convert]::ToBase64String($ba)):"
PS D:\> $factory = Get-ComMoniker -Moniker $objref -Bind
PS D:\> $factory = $factory.QueryInterface('00000001-0000-0000-c000-000000000046')
PS D:\> $obj = New-ComObject -Factory $factory
PS D:\> Get-ComInterface -Object $obj

Name                 IID                                  HasProxy  HasTypeLib
----                 ---                                  --------  ----------
IUnknown             00000000-0000-0000-c000-000000000046 False     False
IMarshal             00000003-0000-0000-c000-000000000046 False     False
IMultiQI             00000020-0000-0000-c000-000000000046 False     False
IClientSecurity      0000013d-0000-0000-c000-000000000046 False     False
ICorSvcBindToWorker  5c6fb596-4828-4ed5-b9dd-293dad736fb5 True      True
```

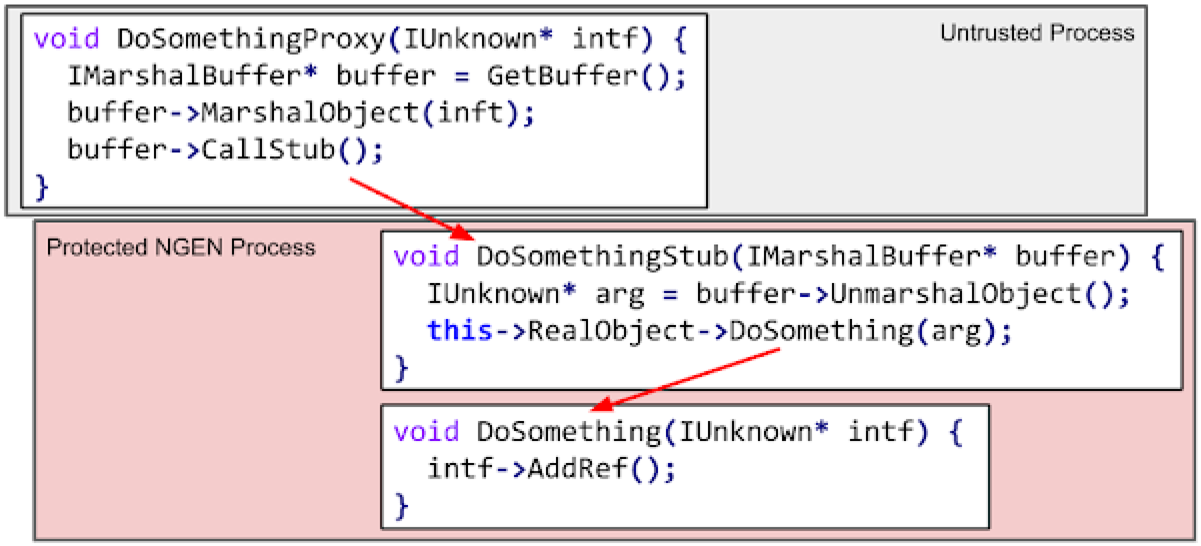
We're out of luck, this object is not implemented in .NET, as you'd at least expect to see an instance of the _Object interface. There's only one interface implemented, ICorSvcBindToWorker so let's dig into that interface to see if there's anything we can exploit.

Something caught my eye, in the screenshot there's a HasTypeLib column, for ICorSvcBindToWorker we see that the column is set to True. What HasTypeLib indicates is rather than the interface's proxy code being implemented using an predefined NDR byte stream it's generated on the fly from a type library. I've abused this auto-generating proxy mechanism before to elevate to SYSTEM, reported as issue 1112. In the issue I used some interesting behavior of the system's Running Object Table (ROT) to force a type confusion in a system COM service. While Microsoft has fixed the issue for User to SYSTEM there's nothing stopping us using the type confusion trick to exploit the MSCORSVW process running as PPL at the same privilege level and get arbitrary code execution. Another advantage of using a type library is a normal proxy would be loaded as a DLL which means that it must meet the PPL signing level requirements; however a type library is just data so can be loaded into a PPL without any signing level violations.

How does the type confusion work? Looking at the ICorSvcBindToWorker interface from the type library:

```
interface ICorSvcBindToWorker : IUnknown {
  HRESULT BindToRuntimeWorker(
          [in] BSTR pRuntimeVersion,
          [in] unsigned long ParentProcessID,
          [in] BSTR pInterruptEventName,
          [in] ICorSvcLogger* pCorSvcLogger,
        [out] ICorSvcWorker** pCorSvcWorker);
};
```

The single BindToRuntimeWorker takes 5 parameters, 4 are inbound and 1 is outbound. When trying to access the method over DCOM from our untrusted process the system will automatically generate the proxy and stub for the call. This will include marshaling COM interface parameters into a buffer, sending the buffer to the remote process and then unmarshaling to a pointer before calling the real function. For example imagine a simpler function, DoSomething which takes a single IUnknown pointer. The marshaling process looks like the following:



```
void DoSomethingProxy(IUnknown* intf) {
  IMarshalBuffer* buffer = GetBuffer();
  buffer->MarshalObject(inft);
  buffer->CallStub();
}
```

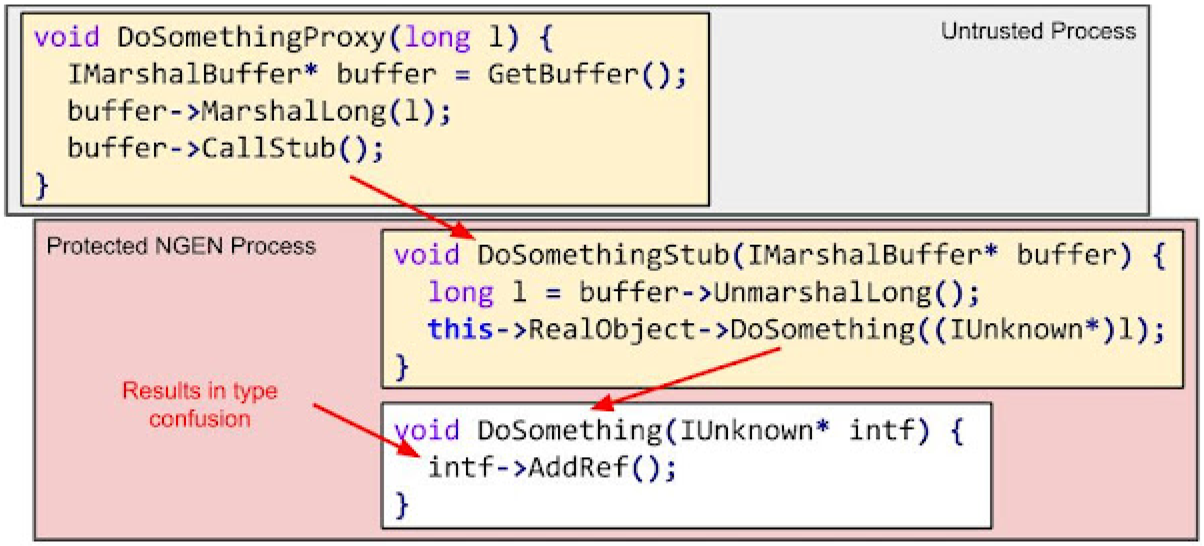
```
void DoSomethingStub(IMarshalBuffer* buffer) {
  IUnknown* arg = buffer->UnmarshalObject();
  this->RealObject->DoSomething(arg);
}
```

```
void DoSomething(IUnknown* intf) {
  intf->AddRef();
}
```

The operation of the method call is as follow:

1. The untrusted process calls DoSomething on the interface which is actually a pointer to DoSomethingProxy which was auto-generated from the type library passing an IUnknown pointer parameter.
2. DoSomethingProxy marshals the IUnknown pointer parameter into the buffer and calls over RPC to the Stub in the protected process.
3. The COM runtime calls the DoSomethingStub method to handle the call. This method will unmarshal the interface pointer from the buffer. Note that this pointer is not the original pointer from step 1, it's likely to be a new proxy which calls back to the untrusted process.
4. The stub invokes the real implemented method inside the server, passing the unmarshaled interface pointer.
5. DoSomething uses the interface pointer, for example by calling AddRef on it via the object's VTable.

How would we exploit this? All we need to do is modify the type library so that instead of passing an interface pointer we pass almost anything else. While the type library file is in a system location which we can't modify we can just replace the registration for it in the current user's registry hive, or use the same ROT trick from before issue 1112. For example if we modifying the type library to pass an integer instead of an interface pointer we get the following:



```
void DoSomethingProxy(long l) {
  IMarshalBuffer* buffer = GetBuffer();
  buffer->MarshalLong(l);
  buffer->CallStub();
}
```

```
void DoSomethingStub(IMarshalBuffer* buffer) {
  long l = buffer->UnmarshalLong();
  this->RealObject->DoSomething((IUnknown*)l);
}
```

```
void DoSomething(IUnknown* intf) {
  intf->AddRef();
}
```

The operation of the marshal now changes as follows:

1. The untrusted process calls DoSomething on the interface which is actually a pointer to DoSomethingProxy which was auto-generated from the type library passing an arbitrary integer parameter.
2. DoSomethingProxy marshals the integer parameter into the buffer and calls over RPC to the Stub in the protected process.
3. The COM runtime calls the DoSomethingStub method to handle the call. This method will unmarshal the integer from the buffer.
4. The stub invokes the real implement method inside the server, passing the integer as the parameter. However DoSomething hasn't changed, it's still the same method which accepts an interface pointer. As the COM runtime has no more type information at this point the integer is type confused with the interface pointer.
5. DoSomething uses the interface pointer, for example by calling AddRef on it via the object's VTable. As this pointer is completely under control of the untrusted process this likely results in arbitrary code execution.

By changing the type of parameter from an interface pointer to an integer we induce a type confusion which allows us to get an arbitrary pointer dereferenced, resulting in arbitrary code execution. We could even simplify the attack by adding to the type library the following structure:

```
struct FakeObject {
  BSTR FakeVTable;
};
```

If we pass a pointer to a FakeObject instead of the interface pointer the auto-generated proxy will marshal the structure and its BSTR, recreating it on the other side in the stub. As a BSTR is a counted string it can contain NULLs so this will create a pointer to an object, which contains a pointer to an arbitrary byte array which can act as a VTable. Place known function pointers in that BSTR and you can easily redirect execution without having to guess the location of a suitable VTable buffer.

To fully exploit this we'd need to call a suitable method, probably running a ROP chain and we might also have to bypass CFG. That all sounds too much like hard work, so instead I'll take a different approach to get arbitrary code running in the PPL binary, by abusing KnownDlls.

KnownDlls and Protected Processes.

In my previous blog post I described a technique to elevate privileges from an arbitrary object directory creation vulnerability to SYSTEM by adding an entry into the KnownDlls directory and getting an arbitrary DLL loaded into a privileged process. I noted that this was also an administrator to PPL code injection as PPL will also load DLLs from the system's KnownDlls location. As the code signing check is performed during section creation not section mapping as long as you can place an entry into KnownDlls you can load anything into a PPL even unsigned code.

This doesn't immediately seem that useful, we can't write to KnownDlls without being an administrator, and even then without some clever tricks. However it's worth looking at how a Known DLL is loaded to get an understanding on how it can be abused. Inside NTDLL's loader (LDR) code is the following function to determine if there's a preexisting Known DLL.

```
NTSTATUS LdrpFindKnownDll(PUNICODE_STRING DllName, HANDLE *SectionHandle)
{
 // If KnownDll directory handle not open then return error.
 if (!LdrpKnownDllDirectoryHandle)
   return STATUS_DLL_NOT_FOUND;

 OBJECT_ATTRIBUTES ObjectAttributes;
 InitializeObjectAttributes(&ObjectAttributes,
   &DllName,
   OBJ_CASE_INSENSITIVE,
   LdrpKnownDllDirectoryHandle,
   nullptr);
 return NtOpenSection(SectionHandle,
                  SECTION_ALL_ACCESS,
                  &ObjectAttributes);
}
```

The LdrpFindKnownDll function calls NtOpenSection to open the named section object for the Known DLL. It doesn't open an absolute path, instead it uses the feature of the native system calls to specify a root directory for the object name lookup in the OBJECT_ATTRIBUTES structure. This root directory comes from the global variable LdrpKnownDllDirectoryHandle. Implementing the call this way allows the loader to only specify the filename (e.g. EXAMPLE.DLL) and not have to reconstruct the absolute path as the lookup with be relative to an existing directory. Chasing references to LdrpKnownDllDirectoryHandle we can find it's initialized in LdrpInitializeProcess as follows:

```
NTSTATUS LdrpInitializeProcess() {
 // ...
 PPEB peb = // ...
 // If a full protected process don't use KnownDlls.
 if (peb->IsProtectedProcess && !peb->IsProtectedProcessLight) {
   LdrpKnownDllDirectoryHandle = nullptr;
 } else {
   OBJECT_ATTRIBUTES ObjectAttributes;
   UNICODE_STRING DirName;
   RtlInitUnicodeString(&DirName, L"\\KnownDlls");
   InitializeObjectAttributes(&ObjectAttributes,
                         &DirName,
                         OBJ_CASE_INSENSITIVE,
                         nullptr, nullptr);
   // Open KnownDlls directory.
   NtOpenDirectoryObject(&LdrpKnownDllDirectoryHandle,
                    DIRECTORY_QUERY | DIRECTORY_TRAVERSE,
                    &ObjectAttributes);
}
```

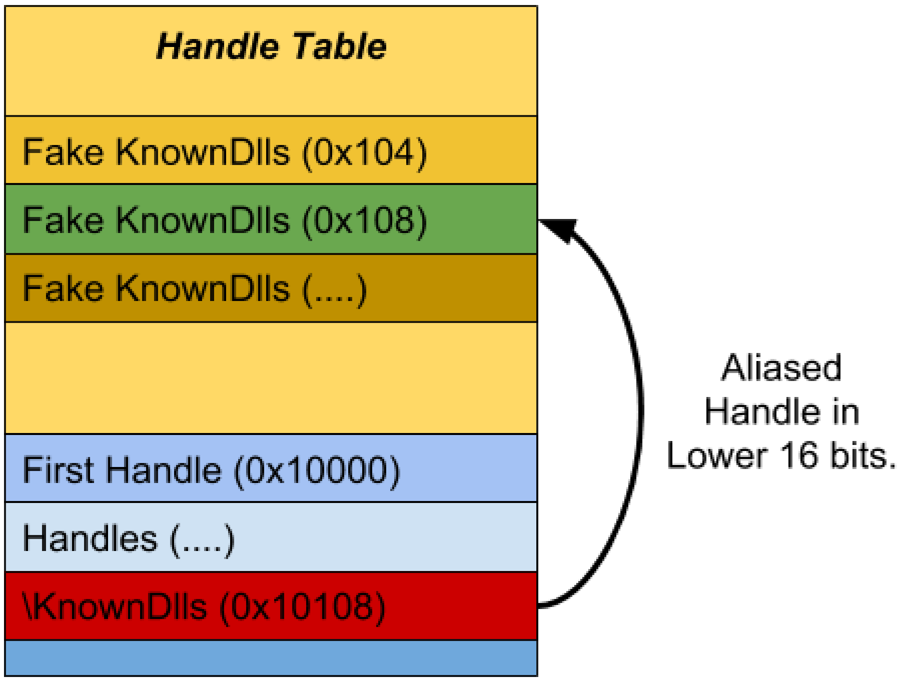
This code shouldn't be that unexpected, the implementation calls NtOpenDirectoryObject, passing the absolute path to the KnownDlls directory as the object name. The opened handle is stored in the LdrpKnownDllDirectoryHandle global variable for later use. It's worth noting that this code checks the PEB to determine if the current process is a full protected process. Support for loading Known DLLs is disabled in full protected process mode, which is why even with administrator privileges and the clever trick I outlined in the last blog post we could only compromise PPL, not PP.

How does this knowledge help us? We can use our COM type confusion trick to write values into arbitrary memory locations instead of trying to hijack code execution resulting in a data only attack. As we can inherit any handles we like into the new PPL process we can setup an object directory with a named section, then use the type confusion to change the value of LdrpKnownDllDirectoryHandle to the value of the inherited handle. If we induce a DLL load from System32 with a known name the LDR will check our fake directory for the named section and map our unsigned code into memory, even calling DllMain for us. No need for injecting threads, ROP or bypassing CFG.

All we need is a suitable primitive to write an arbitrary value, unfortunately while I could find methods which would cause an arbitrary write I couldn't sufficiently control the value being written. In the end I used the following interface and method which was implemented on the object returned by ICorSvcBindToWorker::BindToRuntimeWorker.

```
interface ICorSvcPooledWorker : IUnknown {
  HRESULT CanReuseProcess(
        [in] OptimizationScenario scenario,
        [in] ICorSvcLogger* pCorSvcLogger,
        [out] long* pCanContinue);
};
```

In the implementation of CanReuseProcess the target value of pCanContinue is always initialized to 0. Therefore by replacing the [out] long* in the type library definition with [in] long we can get 0 written to any memory location we specify. By prefilling the lower 16 bits of the new process' handle table with handles to a fake KnownDlls directory we can be sure of an alias between the real KnownDlls which will be opened once the process starts and our fake ones by just modifying the top 16 bits of the handle to 0. This is shown in the following diagram:

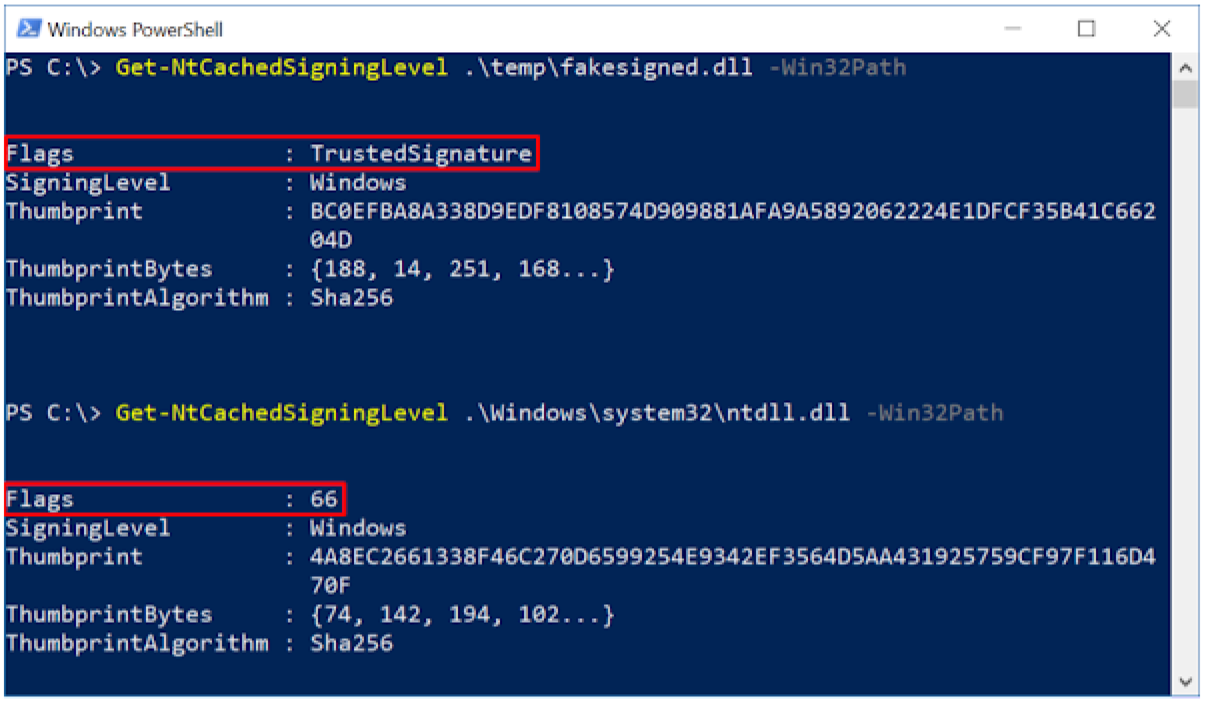


Once we’ve overwritten the top 16 bits with 0 (the write is 32 bits but handles are 64 bits in 64 bit mode, so we won’t overwrite anything important) LdrpKnownDllDirectoryHandle now points to one of our fake KnownDlls handles. We can then easily induce a DLL load by sending a custom marshaled object to the same method and we’ll get arbitrary code execution inside the PPL.

Elevating to PPL-Windows TCB

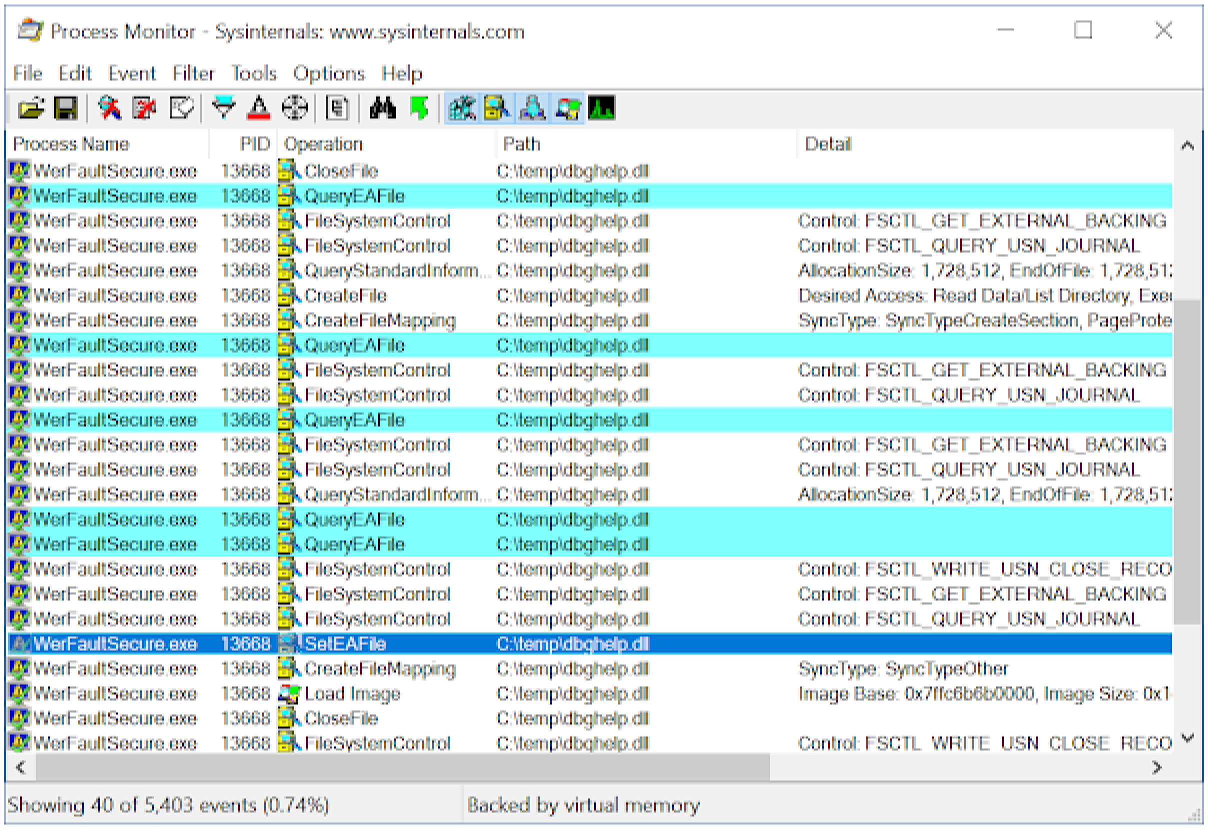
We can’t stop here, attacking MSCORSVW only gets us PPL at the CodeGen signing level, not Windows TCB. Knowing that generating a fake cached signed DLL should run in a PPL as well as Microsoft leaving a backdoor for PPL processes at any signing level I converted my C# code from Issue 1332 to C++ to generate a fake cached signed DLL. By abusing a DLL hijack in WERFAULTSECURE.EXE which will run as PPL Windows TCB we should get code execution at the desired signing level. This worked on Windows 10 1709 and earlier, however it didn’t work on 1803. Clearly Microsoft had changed the behavior of cached signing level in some way, perhaps they’d removed its trust in PPL entirely. That seemed unlikely as it would have a negative performance impact.

After discussing this a bit with Alex Ionescu I decided to put together a quick parser with information from Alex for the cached signing data on a file. This is exposed in NtObjectManager as the Get-NtCachedSigningLevel command. I ran this command against a fake signed binary and a system binary which was also cached signed and immediately noticed a difference:



```
PS C:\> Get-NtCachedSigningLevel .\temp\fakesigned.dll -Win32Path


Flags               : TrustedSignature
SigningLevel        : Windows
Thumbprint          : BC0EFBA8A338D9EDF8108574D909881AFA9A5892062224E1DFCF35B41C662
                      04D
ThumbprintBytes     : {188, 14, 251, 168...}
ThumbprintAlgorithm : Sha256



PS C:\> Get-NtCachedSigningLevel .\Windows\system32\ntdll.dll -Win32Path


Flags               : 66
SigningLevel        : Windows
Thumbprint          : 4A8EC2661338F46C270D6599254E9342EF3564D5AA431925759CF97F116D4
                      70F
ThumbprintBytes     : {74, 142, 194, 102...}
ThumbprintAlgorithm : Sha256
```

For the fake signed file the Flags are set to TrustedSignature (0x02), however for the system binary PowerShell couldn't decode the enumeration and so just outputs the integer value of 66 which is 0x42 in hex. The value 0x40 was an extra flag on top of the original trusted signature flag. It seemed likely that without this flag set the DLL wouldn't be loaded into a PPL process. Something must be setting this flag so I decided to check what happened if I loaded a valid cached signed DLL without the extra flag into a PPL process. Monitoring it in Process Monitor I got my answer:



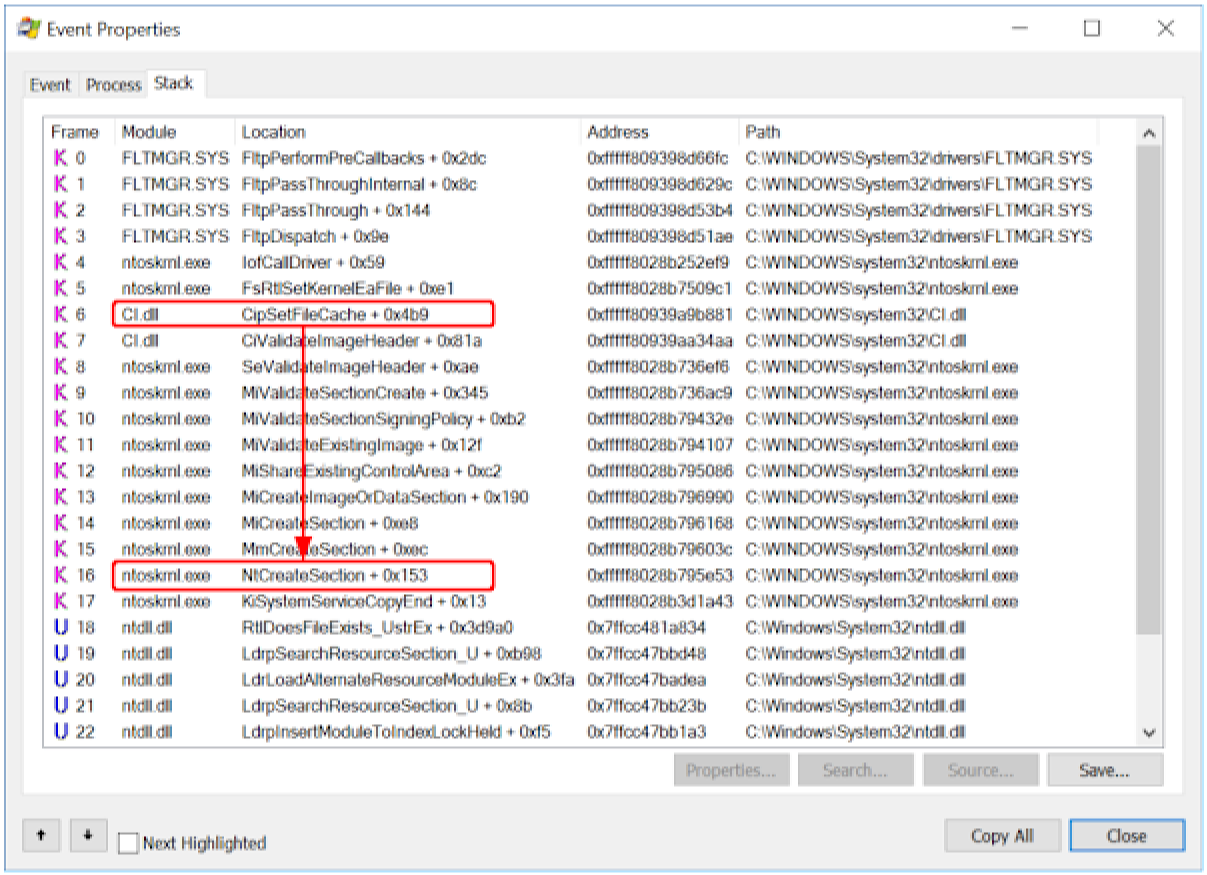
Process Monitor - Sysinternals: www.sysinternals.com

File Edit Event Filter Tools Options Help

Process Name	PID	Operation	Path	Detail
WerFaultSecure.exe	13668	CloseFile	C:\temp\dbghelp.dll	
WerFaultSecure.exe	13668	QueryEAFile	C:\temp\dbghelp.dll	
WerFaultSecure.exe	13668	FileSystemControl	C:\temp\dbghelp.dll	Control: FSCTL_GET_EXTERNAL_BACKING
WerFaultSecure.exe	13668	FileSystemControl	C:\temp\dbghelp.dll	Control: FSCTL_QUERY_USN_JOURNAL
WerFaultSecure.exe	13668	QueryStandardInform...	C:\temp\dbghelp.dll	AllocationSize: 1,728,512, EndOfFile: 1,728,51
WerFaultSecure.exe	13668	CreateFile	C:\temp\dbghelp.dll	Desired Access: Read Data/List Directory, Exe
WerFaultSecure.exe	13668	CreateFileMapping	C:\temp\dbghelp.dll	SyncType: SyncTypeCreateSection, PageProte
WerFaultSecure.exe	13668	QueryEAFile	C:\temp\dbghelp.dll	
WerFaultSecure.exe	13668	FileSystemControl	C:\temp\dbghelp.dll	Control: FSCTL_GET_EXTERNAL_BACKING
WerFaultSecure.exe	13668	FileSystemControl	C:\temp\dbghelp.dll	Control: FSCTL_QUERY_USN_JOURNAL
WerFaultSecure.exe	13668	QueryEAFile	C:\temp\dbghelp.dll	
WerFaultSecure.exe	13668	FileSystemControl	C:\temp\dbghelp.dll	Control: FSCTL_GET_EXTERNAL_BACKING
WerFaultSecure.exe	13668	FileSystemControl	C:\temp\dbghelp.dll	Control: FSCTL_QUERY_USN_JOURNAL
WerFaultSecure.exe	13668	QueryStandardInform...	C:\temp\dbghelp.dll	AllocationSize: 1,728,512, EndOfFile: 1,728,51
WerFaultSecure.exe	13668	QueryEAFile	C:\temp\dbghelp.dll	
WerFaultSecure.exe	13668	QueryEAFile	C:\temp\dbghelp.dll	
WerFaultSecure.exe	13668	FileSystemControl	C:\temp\dbghelp.dll	Control: FSCTL_WRITE_USN_CLOSE_RECO
WerFaultSecure.exe	13668	FileSystemControl	C:\temp\dbghelp.dll	Control: FSCTL_GET_EXTERNAL_BACKING
WerFaultSecure.exe	13668	FileSystemControl	C:\temp\dbghelp.dll	Control: FSCTL_QUERY_USN_JOURNAL
WerFaultSecure.exe	13668	SetEAFile	C:\temp\dbghelp.dll	
WerFaultSecure.exe	13668	CreateFileMapping	C:\temp\dbghelp.dll	SyncType: SyncTypeOther
WerFaultSecure.exe	13668	Load Image	C:\temp\dbghelp.dll	Image Base: 0x7ffc6b6b0000, Image Size: 0x1
WerFaultSecure.exe	13668	CloseFile	C:\temp\dbghelp.dll	
WerFaultSecure.exe	13668	FileSystemControl	C:\temp\dbghelp.dll	Control: FSCTL_WRITE_USN_CLOSE_RECO

Showing 40 of 5,403 events (0.74%) Backed by virtual memory

The Process Monitor trace shows that first the kernel queries for the Extended Attributes (EA) from the DLL. The cached signing level data is stored in the file's EA so this is almost certainly an indication of the cached signing level being read. In the full trace artifacts of checking the full signature are shown such as enumerating catalog files, I've removed those artifacts from the screenshot for brevity. Finally the EA is set, if I check the cached signing level of the file it now includes the extra flag. So setting the cached signing level is done automatically, the question is how? By pulling up the stack trace we can see how it happens:



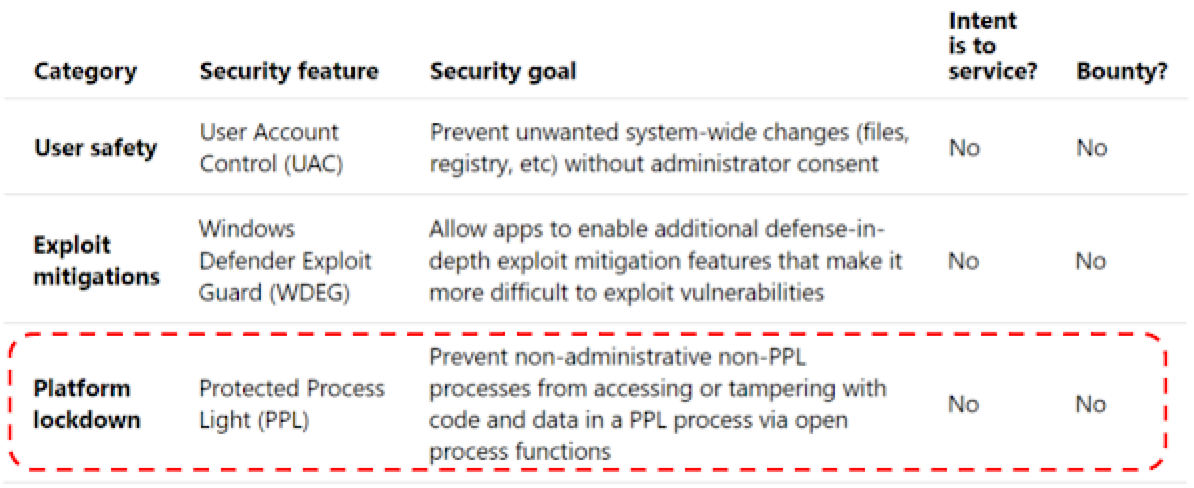
Frame	Module	Location	Address	Path
K 0	FLTMGR.SYS	FltpPerformPreCallbacks + 0x2dc	0xfffff809398d66fc	C:\WINDOWS\System32\drivers\FLTMGR.SYS
K 1	FLTMGR.SYS	FltpPassThroughInternal + 0x8c	0xfffff809398d629c	C:\WINDOWS\System32\drivers\FLTMGR.SYS
K 2	FLTMGR.SYS	FltpPassThrough + 0x144	0xfffff809398d53b4	C:\WINDOWS\System32\drivers\FLTMGR.SYS
K 3	FLTMGR.SYS	FltpDispatch + 0x9e	0xfffff809398d51ae	C:\WINDOWS\System32\drivers\FLTMGR.SYS
K 4	ntoskrnl.exe	IofCallDriver + 0x59	0xfffff8028b252ef9	C:\WINDOWS\system32\ntoskrnl.exe
K 5	ntoskrnl.exe	FsRtlSetKernelEaFile + 0xe1	0xfffff8028b7509c1	C:\WINDOWS\system32\ntoskrnl.exe
K 6	CI.dll	CipSetFileCache + 0x4b9	0xfffff80939a9b881	C:\WINDOWS\system32\CI.dll
K 7	CI.dll	CiValidateImageHeader + 0x81a	0xfffff80939aa34aa	C:\WINDOWS\system32\CI.dll
K 8	ntoskrnl.exe	SeValidateImageHeader + 0xae	0xfffff8028b736ef6	C:\WINDOWS\system32\ntoskrnl.exe
K 9	ntoskrnl.exe	MiValidateSectionCreate + 0x345	0xfffff8028b736ac9	C:\WINDOWS\system32\ntoskrnl.exe
K 10	ntoskrnl.exe	MiValidateSectionSigningPolicy + 0xb2	0xfffff8028b79432e	C:\WINDOWS\system32\ntoskrnl.exe
K 11	ntoskrnl.exe	MiValidateExistingImage + 0x12f	0xfffff8028b794107	C:\WINDOWS\system32\ntoskrnl.exe
K 12	ntoskrnl.exe	MiShareExistingControlArea + 0xc2	0xfffff8028b795086	C:\WINDOWS\system32\ntoskrnl.exe
K 13	ntoskrnl.exe	MiCreateImageOrDataSection + 0x190	0xfffff8028b796990	C:\WINDOWS\system32\ntoskrnl.exe
K 14	ntoskrnl.exe	MiCreateSection + 0xe8	0xfffff8028b796168	C:\WINDOWS\system32\ntoskrnl.exe
K 15	ntoskrnl.exe	MmCreateSection + 0xec	0xfffff8028b79603c	C:\WINDOWS\system32\ntoskrnl.exe
K 16	ntoskrnl.exe	NtCreateSection + 0x153	0xfffff8028b795e53	C:\WINDOWS\system32\ntoskrnl.exe
K 17	ntoskrnl.exe	KiSystemServiceCopyEnd + 0x13	0xfffff8028b3d1a43	C:\WINDOWS\system32\ntoskrnl.exe
U 18	ntdll.dll	RtlDoesFileExists_UstrEx + 0x3d9a0	0x7ffcc481a834	C:\Windows\System32\ntdll.dll
U 19	ntdll.dll	LdrpSearchResourceSection_U + 0xb98	0x7ffcc47bbd48	C:\Windows\System32\ntdll.dll
U 20	ntdll.dll	LdrLoadAlternateResourceModuleEx + 0x3fa	0x7ffcc47badea	C:\Windows\System32\ntdll.dll
U 21	ntdll.dll	LdrpSearchResourceSection_U + 0x8b	0x7ffcc47bb23b	C:\Windows\System32\ntdll.dll
U 22	ntdll.dll	LdrpInsertModuleToIndexLockHeld + 0xf5	0x7ffcc47bb1a3	C:\Windows\System32\ntdll.dll

Looking at the middle of the stack trace we can see the call to CipSetFileCache originates from the call to NtCreateSection. The kernel is automatically caching the signature when it makes sense to do so, e.g. in a PPL so that subsequent image mapping don't need to recheck the signature. It's possible to map an image section from a file with write access so we can reuse the same attack from Issue 1332 and replace the call to NtSetCachedSigningLevel with NtCreateSection and we can fake sign any DLL. It turned out that the call to set the file cache happened after the write check introducted to fix Issue 1332 and so it was possible to use this to bypass Device Guard again. For that reason I reported the bypass as Issue 1597 which was fixed in September 2018 as CVE-2018-8449. However, as with Issue 1332 the back door for PPL is still in place so even though the fix eliminated the Device Guard bypass it can still be used to get us from PPL-CodeGen to PPL-WindowsTCB.

Conclusions

This blog showed how I was able to inject arbitrary code into a PPL without requiring administrator privileges. What could you do with this new found power? Actually not a great deal as a normal user but there are some parts of the OS, such as the Windows Store which rely on PPL to secure files and resources which you can't modify as a normal user. If you elevate to administrator and then inject into a PPL you'll get many more things to attack such as CSRSS (through which you can certainly get kernel code execution) or attack Windows Defender which runs as PPL Anti-Malware. Over time I'm sure the majority of the use cases for PPL will be replaced with Virtual Secure Mode (VSM) and Isolated User Mode (IUM) applications which have greater security guarantees and are also considered security boundaries that Microsoft will defend and fix.

Did I report these issues to Microsoft? Microsoft has made it clear that they will not fix issues only affecting PP and PPL in a security bulletin. Without a security bulletin the researcher receives no acknowledgement for the find, such as a CVE. The issue will not be fixed in current versions of Windows although it might be fixed in the next major version. Previously confirming Microsoft's policy on fixing a particular security issue was based on precedent, however they've recently published a list of Windows technologies that will or will not be fixed in the Windows Security Service Criteria which, as shown below for Protected Process Light, Microsoft will not fix or pay a bounty for issues relating to the feature. Therefore, from now on I will not be engaging Microsoft if I discover issues which I believe to only affect PP or PPL.



Category	Security feature	Security goal	Intent is to service?	Bounty?
User safety	User Account Control (UAC)	Prevent unwanted system-wide changes (files, registry, etc) without administrator consent	No	No
Exploit mitigations	Windows Defender Exploit Guard (WDEG)	Allow apps to enable additional defense-in-depth exploit mitigation features that make it more difficult to exploit vulnerabilities	No	No
Platform lockdown	Protected Process Light (PPL)	Prevent non-administrative non-PPL processes from accessing or tampering with code and data in a PPL process via open process functions	No	No

The one bug I reported to Microsoft was only fixed because it could be used to bypass Device Guard. When you think about it, only fixing for Device Guard is somewhat odd. I can still bypass Device Guard by injecting into a PPL and setting a cached signing level, and yet Microsoft won't fix PPL issues but will fix Device Guard issues. Much as the Windows Security Service Criteria document really helps to clarify what Microsoft will and won't fix it's still somewhat arbitrary. A secure feature is rarely secure in isolation, the feature is almost certainly secure because other features enable it to be so.

In part 2 of this blog we'll go into how I was also able to break into Full PP-WindowsTCB processes using another interesting feature of COM.