Injecting Code into Windows Protected Processes using COM - Part 2

googleprojectzero.blogspot.com/2018/11/injecting-code-into-windows-protected.html

Posted by James Forshaw, Project Zero

In my previous blog I discussed a technique which combined numerous issues I've previously reported to Microsoft to inject arbitrary code into a PPL-WindowsTCB process. The techniques presented don't work for exploiting the older, stronger Protected Processes (PP) for a few different reasons. This blog seeks to remedy this omission and provide details of how I was able to also hijack a full PP-WindowsTCB process without requiring administrator privileges. This is mainly an academic exercise, to see whether I can get code executing in a full PP as there's not much more you can do inside a PP over a PPL.

As a quick recap of the previous attack, I was able to identify a process which would run as PPL which also exposed a COM service. Specifically, this was the ".NET Runtime Optimization Service" which ships with the .NET framework and uses PPL at CodeGen level to apply cached signing levels to Ahead-of-Time compiled DLLs to allow them to be used with User-Mode Code Integrity (UMCI). By modifying the COM proxy configuration it was possible to induce a type confusion which allowed me to load an arbitrary DLL by hijacking the KnownDlls configuration. Once running code inside the PPL I could abuse a bug in the cached signing feature to create a DLL signed to load into any PPL and through that escalate to PPL-WindowsTCB level.

Finding a New Target

My first thought to exploit full PP would be to use the additional access we were granted from having code running at PPL-WindowsTCB. You might assume you could abuse the cached signed DLL to bypass security checks to load into a full PP. Unfortunately the kernel's Code Integrity module ignores cached signing levels for full PP. How about KnownDlls in general? If we have administrator privileges and code running in PPL-WindowsTCB we can directly write to the KnownDlls object directory (see another of my blog posts link for why you need to be PPL) and try to get the PP to load an arbitrary DLL. Unfortunately, as I mentioned in the previous blog, this also doesn't work as full PP ignores KnownDlls. Even if it did load KnownDlls I don't want to require administrator privileges to inject code into the process.

I decided that it'd make sense to rerun my PowerShell script from the previous blog to discover which executables will run as full PP and at what level. On Windows 10 1803 there's a significant number of executables which run as PP-Authenticode level, however only four executables would start with a more privileged level as shown in the following table.

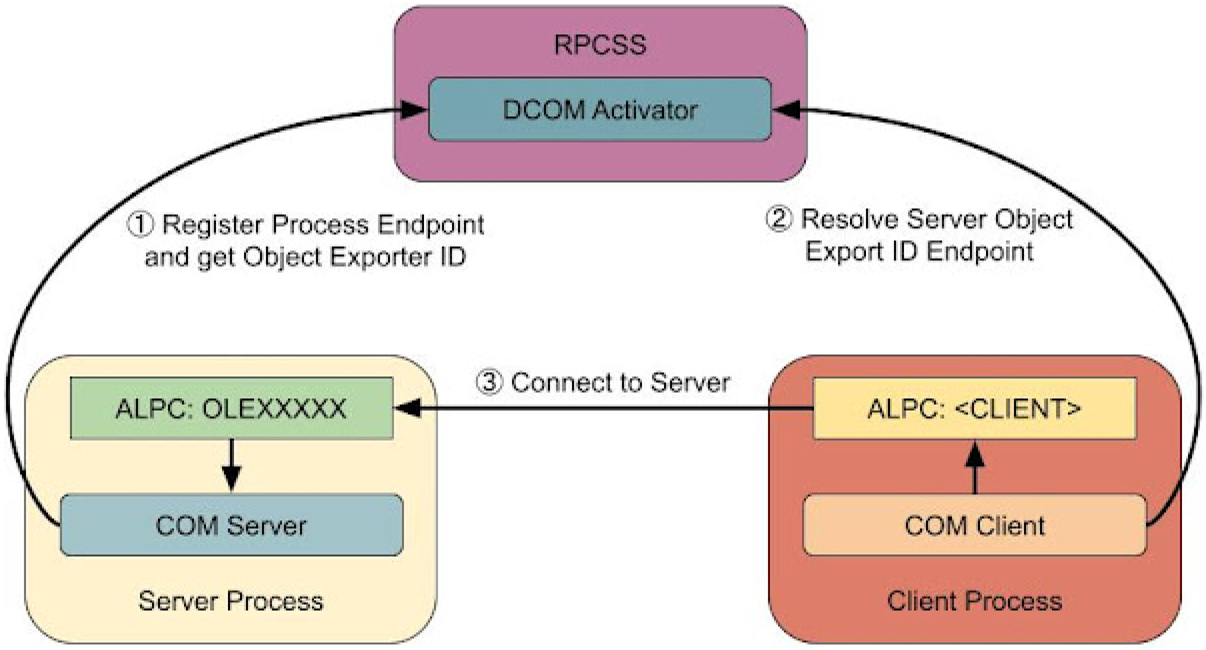
Path	Signing Level
C:\windows\system32\GenValObj.exe	Windows
C:\windows\system32\sppsvc.exe	Windows
C:\windows\system32\WerFaultSecure.exe	WindowsTCB
C:\windows\system32\SgrmBroker.exe	WindowsTCB

As I have no known route from PP-Windows level to PP-WindowsTCB level like I had with PPL, only two of the four executables are of interest, WerFaultSecure.exe and SgrmBroker.exe. I correlated these two executables against known COM service registrations, which turned up no results. That doesn't mean these executables don't expose a COM attack surface, the .NET executable I abused last time also doesn't register its COM service, so I also performed some basic reverse engineering looking for COM usage.

The SgrmBroker executable doesn't do very much at all, it's a wrapper around an isolated user mode application to implement runtime attestation of the system as part of Windows Defender System Guard and didn't call into any COM APIs. WerFaultSecure also doesn't seem to call into COM, however I already knew that WerFaultSecure can load COM objects, as Alex Ionescu used my original COM scriptlet code execution attack to get PPL-WindowsTCB level though hijacking a COM object load in WerFaultSecure. Even though WerFaultSecure didn't expose a service if it could initialize COM perhaps there was something that I could abuse to get arbitrary code execution? To understand the attack surface of COM we need to understand how COM implements out-of-process COM servers and COM remoting in general.

Digging into COM Remoting Internals

Communication between a COM client and a COM server is over the MSRPC protocol, which is based on the Open Group's DCE/RPC protocol. For local communication the transport used is Advanced Local Procedure Call (ALPC) ports. At a high level communication occurs between a client and server based on the following diagram:



In order for a client to find the location of a server the process registers an ALPC endpoint with the DCOM activator in RPCSS ①. This endpoint is registered alongside the Object Exporter ID (OXID) of the server, which is a 64 bit randomly generated number assigned by RPCSS. When a client wants to connect to a server it must first ask RPCSS to resolve the server's OXID value to an RPC endpoint ②. With the knowledge of the ALPC RPC endpoint the client can connect to the server and call methods on the COM object ③.

The OXID value is discovered either from an out-of-process (OOP) COM activation result or via a marshaled Object Reference (OBJREF) structure. Under the hood the client calls the ResolveOxid method on RPCSS's IObjectExporter RPC interface. The prototype of ResolveOxid is as follows:

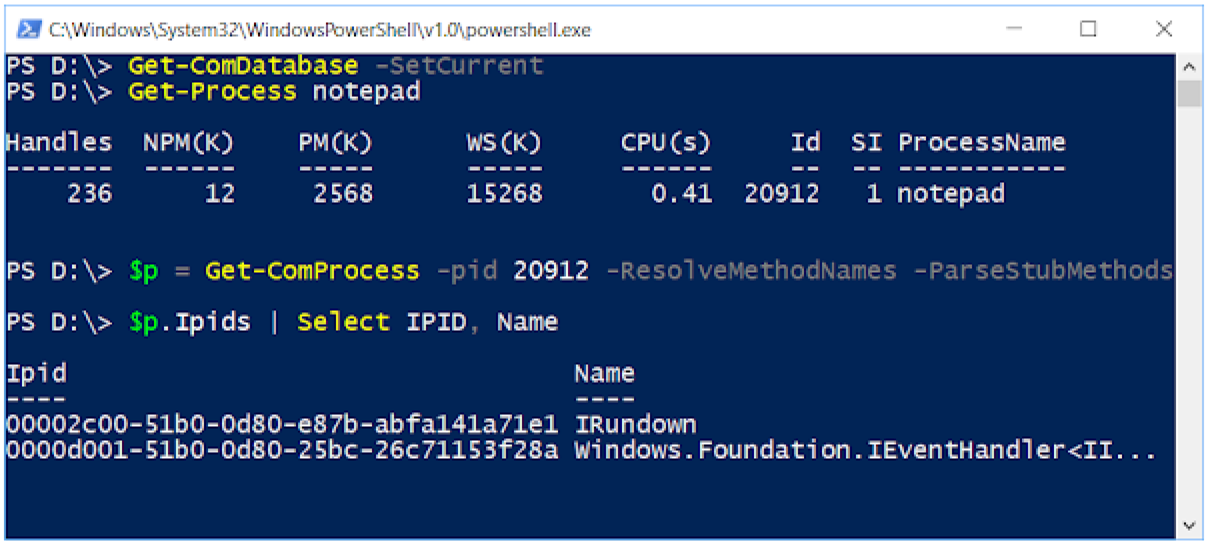
```
interface IObjectExporter {
  // ...
  error_status_t ResolveOxid(
    [in] handle_t hRpc,
    [in] OXID* pOxid,
    [in] unsigned short cRequestedProtseqs,
    [in] unsigned short arRequestedProtseqs[],
    [out, ref] DUALSTRINGARRAY** ppdsaOxidBindings,
    [out, ref] IPID* pipidRemUnknown,
    [out, ref] DWORD* pAuthnHint
);
```

In the prototype we can see the OXID to resolve is being passed in the pOxid parameter and the server returns an array of Dual String Bindings which represent RPC endpoints to connect to for this OXID value. The server also returns two other pieces of information, an

Authentication Level Hint (pAuthnHint) which we can safely ignore and the IPID of the IRemUnknown interface (pipidRemUnknown) which we can't.

An IPID is a GUID value called the Interface Process ID. This represents the unique identifier for a COM interface inside the server, and it's needed to communicate with the correct COM object as it allows the single RPC endpoint to multiplex multiple interfaces over one connection. The IRemUnknown interface is a default COM interface every COM server must implement as it's used to query for new IPIDs on an existing object (using RemQueryInterface) and maintain the remote object's reference count (through RemAddRef and RemRelease methods). As this interface must always exist regardless of whether an actual COM server is exported and the IPID can be discovered through resolving the server's OXID, I wondered what other methods the interface supported in case there was anything I could leverage to get code execution.

The COM runtime code maintains a database of all IPIDs as it needs to lookup the server object when it receives a request for calling a method. If we know the structure of this database we could discover where the IRemUnknown interface is implemented, parse its methods and find out what other features it supports. Fortunately I've done the work of reverse engineering the database format in my OleViewDotNet tool, specifically the command Get-ComProcess in the PowerShell module. If we run the command against a process which uses COM, but doesn't actually implement a COM server (such as notepad) we can try and identify the correct IPID.


C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe

```
PS D:\> Get-ComDatabase -SetCurrent
PS D:\> Get-Process notepad

Handles  NPM(K)    PM(K)      WS(K)     CPU(s)     Id  SI ProcessName
-------  ------    -----      -----     ------     --  -- -----------
    236      12     2568      15268       0.41  20912   1 notepad


PS D:\> $p = Get-ComProcess -pid 20912 -ResolveMethodNames -ParseStubMethods

PS D:\> $p.Ipids | Select IPID, Name

Ipid                                 Name
----                                 ----
00002c00-51b0-0d80-e87b-abfa141a71e1 IRundown
0000d001-51b0-0d80-25bc-26c71153f28a Windows.Foundation.IEventHandler<II...
```

In this example screenshot there's actually two IPIDs exported, IRundown and a Windows.Foundation interface. The Windows.Foundation interface we can safely ignore, but IRundown looks more interesting. In fact if you perform the same check on any COM process you'll discover they also have IRundown interfaces exported. Are we not expecting an IRemUnknown interface though? If we pass the ResolveMethodNames and ParseStubMethods parameters to Get-ComProcess, the command will try and parse method parameters for the interface and lookup names based on public symbols. With the parsed interface data we can pass the IPID object to the Format-ComProxy command to get a basic text representation of the IRundown interface. After cleanup the IRundown interface looks like the following:

```
[uuid("00000134-0000-0000-c000-000000000046")]
interface IRundown : IUnknown {
   HRESULT RemQueryInterface(...);
   HRESULT RemAddRef(...);
   HRESULT RemRelease(...);
   HRESULT RemQueryInterface2(...);
   HRESULT RemChangeRef(...);
   HRESULT DoCallback([in] struct XAptCallback* pCallbackData);
   HRESULT DoNonreentrantCallback([in] struct XAptCallback* pCallbackData);
   HRESULT AcknowledgeMarshalingSets(...);
   HRESULT GetInterfaceNameFromIPID(...);
   HRESULT RundownOid(...);
}
```

This interface is a superset of IRemUnknown, it implements the methods such as RemQueryInterface and then adds some more additional methods for good measure. What really interested me was the DoCallback and DoNonreentrantCallback methods, they sound like they might execute a “callback” of some sort. Perhaps we can abuse these methods? Let’s look at the implementation of DoCallback based on a bit of RE (DoNonreentrantCallback just delegates to DoCallback internally so we don’t need to treat it specially):

```
struct XAptCallback {
 void* pfnCallback;
 void* pParam;
 void* pServerCtx;
 void* pUnk;
 void* iid;
 int   iMethod;
 GUID  guidProcessSecret;
};

HRESULT CRemoteUnknown::DoCallback(XAptCallback *pCallbackData) {
 CProcessSecret::GetProcessSecret(&pguidProcessSecret);
 if (!memcmp(&pguidProcessSecret,
            &pCallbackData->guidProcessSecret, sizeof(GUID))) {
   if (pCallbackData->pServerCtx == GetCurrentContext()) {
     return pCallbackData->pfnCallback(pCallbackData->pParam);
```

```
  } else {
    return SwitchForCallback(
              pCallbackData->pServerCtx,
              pCallbackData->pfnCallback,
              pCallbackData->pParam);
  }
 }
 return E_INVALIDARG;
}
```

This method is very interesting, it takes a structure containing a pointer to a method to call and an arbitrary parameter and executes the pointer. The only restrictions on calling the arbitrary method is you must know ahead of time a randomly generated GUID value, the process secret, and the address of a server context. The checking of a per-process random value is a common security pattern in COM APIs and is typically used to restrict functionality to only in-process callers. I abused something similar in the Free-Threaded Marshaler way back in 2014.

What is the purpose of DoCallback? The COM runtime creates a new IRundown interface for every COM apartment that's initialized. This is actually important as calling methods between apartments, say calling a STA object from a MTA, you need to call the appropriate IRemUnknown methods in the correct apartment. Therefore while the developers were there they added a few more methods which would be useful for calling between apartments, including a general "call anything you like" method. This is used by the internals of the COM runtime and is exposed indirectly through methods such as CoCreateObjectInContext. To prevent the DoCallback method being abused OOP the per-process secret is checked which should limit it to only in-process callers, unless an external process can read the secret from memory.

Abusing DoCallback

We have a primitive to execute arbitrary code within any process which has initialized COM by invoking the DoCallback method, which should include a PP. In order to successfully call arbitrary code we need to know four pieces of information:

1. The ALPC port that the COM process is listening on.
2. The IPID of the IRundown interface.
3. The initialized process secret value.
4. The address of a valid context, ideally the same value that GetCurrentContext returns to call on the same RPC thread.

Getting the ALPC port and the IPID is easy, if the process exposes a COM server, as both will be provided during OXID resolving. Unfortunately WerFaultSecure doesn't expose a COM object we can create so that angle wouldn't be open to us, leaving us with a problem we need to solve. Extracting the process secret and context value requires reading the contents of process memory. This is another problem, one of the intentional security features of PP is preventing a non-PP process from reading memory from a PP process. How are we going to solve these two problems?

Talking this through with Alex at Recon we came up with a possible attack if you have administrator access. Even being an administrator doesn't allow you to read memory directly from a PP process. We could have loaded a driver, but that would break PP entirely, so we considered how to do it without needing kernel code execution.

First and easiest, the ALPC port and IPID can be extracted from RPCSS. The RPCSS service does not run protected (even PPL) so this is possible to do without any clever tricks other than knowing where the values are stored in memory. For the context pointer, we should be able to brute force the location as there's likely to be only a narrow range of memory locations to test, made slightly easier if we use the 32 bit version of WerFaultSecure.

Extracting the secret is somewhat harder. The secret is initialized in writable memory and therefore ends up in the process' working set once it's modified. As the page isn't locked it will be eligible for paging if the memory conditions are right. Therefore if we could force the page containing the secret to be paged to disk we could read it even though it came from a PP process. As an administrator, we can perform the following to steal the secret:

1. Ensure the secret is initialized and the page is modified.
2. Force the process to trim its working set, this should ensure the modified page containing the secret ends up paged to disk (eventually).
3. Create a kernel memory crash dump file using the NtSystemDebugControl system call. The crash dump can be created by an administrator without kernel debugging being enabled and will contain all live memory in the kernel. Note this doesn't actually crash the system.
4. Parse the crash dump for the Page Table Entry of the page containing the secret value. The PTE should disclose where in the paging file on disk the paged data is located.
5. Open the volume containing the paging file for read access, parse the NTFS structures to find the paging file and then find the paged data and extract the secret.

After coming up with this attack it seemed far too much like hard work and needed administrator privileges which I wanted to avoid. I needed to come up with an alternative solution.

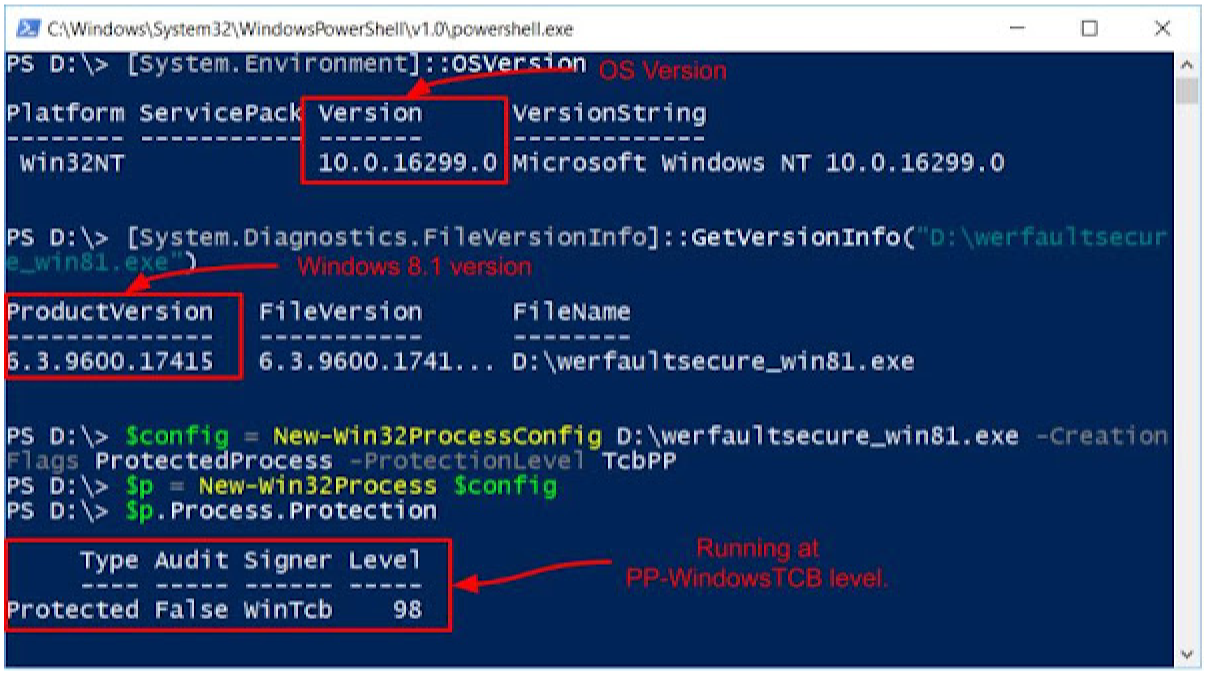
Using WerFaultSecure for its Original Purpose

Up to this point I've been discussing WerFaultSecure as a process that can be abused to get arbitrary code running inside a PP/PPL. I've not really described why the process can run at the maximum PP/PPL levels. WerFaultSecure is used by the Windows Error Reporting service to create crash dumps from protected processes. Therefore it needs to run at elevated PP levels to ensure it can dump any possible user-mode PP. Why can we not just get WerFaultSecure to create a crash dump of itself, which would leak the contents of process memory and allow us to extract any information we require?

The reason we can't use WerFaultSecure is it encrypts the contents of the crash dump before writing it to disk. The encryption is done in a way to only allow Microsoft to decrypt the crash dump, using asymmetric encryption to protect a random session key which can be provided to the Microsoft WER web service. Outside of a weakness in Microsoft's implementation or a new cryptographic attack against the primitives being used getting the encrypted data seems like a non-starter.

However, it wasn't always this way. In 2014 Alex presented at NoSuchCon about PPL and discussed a bug he'd discovered in how WerFaultSecure created encrypted dump files. It used a two step process, first it wrote out the crash dump unencrypted, then it encrypted the crash dump. Perhaps you can spot the flaw? It was possible to steal the unencrypted crash dump. Due to the way WerFaultSecure was called it accepted two file handles, one for the unencrypted dump and one for the encrypted dump. By calling WerFaultSecure directly the unencrypted dump would never be deleted which means that you don't even need to race the encryption process.

There's one problem with this, it was fixed in 2015 in MS15-006. After that fix WerFaultSecure encrypted the crash dump directly, it never ends up on disk unencrypted at any point. But that got me thinking, while they might have fixed the bug going forward what prevents us from taking the old vulnerable version of WerFaultSecure from Windows 8.1 and executing it on Windows 10? I downloaded the ISO for Windows 8.1 from Microsoft's website (link), extracted the binary and tested it, with predictable results:


C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
OS Version
Windows 8.1 version
Running at PP-WindowsTCB level.

```
PS D:\> [System.Environment]::OSVersion

Platform ServicePack Version      VersionString
-------- ----------- -------      -------------
 Win32NT             10.0.16299.0 Microsoft Windows NT 10.0.16299.0


PS D:\> [System.Diagnostics.FileVersionInfo]::GetVersionInfo("D:\werfaultsecur
e_win81.exe")

ProductVersion   FileVersion      FileName
--------------   -----------      --------
6.3.9600.17415   6.3.9600.1741... D:\werfaultsecure_win81.exe


PS D:\> $config = New-Win32ProcessConfig D:\werfaultsecure_win81.exe -Creation
Flags ProtectedProcess -ProtectionLevel TcbPP
PS D:\> $p = New-Win32Process $config
PS D:\> $p.Process.Protection

     Type Audit Signer Level
     ---- ----- ------ -----
Protected False WinTcb    98
```

We can take the vulnerable version of WerFaultSecure from Windows 8.1 and it will run quite happily on Windows 10 at PP-WindowsTCB level. Why? It's unclear, but due to the way PP is secured all the trust is based on the signed executable. As the signature of the executable is still valid the OS just trusts it can be run at the requested protection level. Presumably there must be some way that Microsoft can block specific executables, although at least they can't just revoke their own signing certificates. Perhaps OS binaries should have an EKU in the certificate which indicates what version they're designed to run on? After all Microsoft already added a new EKU when moving from Windows 8 to 8.1 to block downgrade attacks to bypass WinRT UMCI signing so generalizing might make some sense, especially for certain PP levels.

After a little bit of RE and reference to Alex's presentation I was able to work out the various parameters I needed to be passed to the WerFaultSecure process to perform a dump of a PP:

Parameter	Description
/h	Enable secure dump mode.
/pid {pid}	Specify the Process ID to dump.
/tid {tid}	Specify the Thread ID in the process to dump.
/file {handle}	Specify a handle to a writable file for the unencrypted crash dump
/encfile {handle}	Specify a handle to a writable file for the encrypted crash dump
/cancel {handle}	Specify a handle to an event to indicate the dump should be cancelled

/type {flags}	Specify MIMDUMPTYPE flags for call to MiniDumpWriteDump

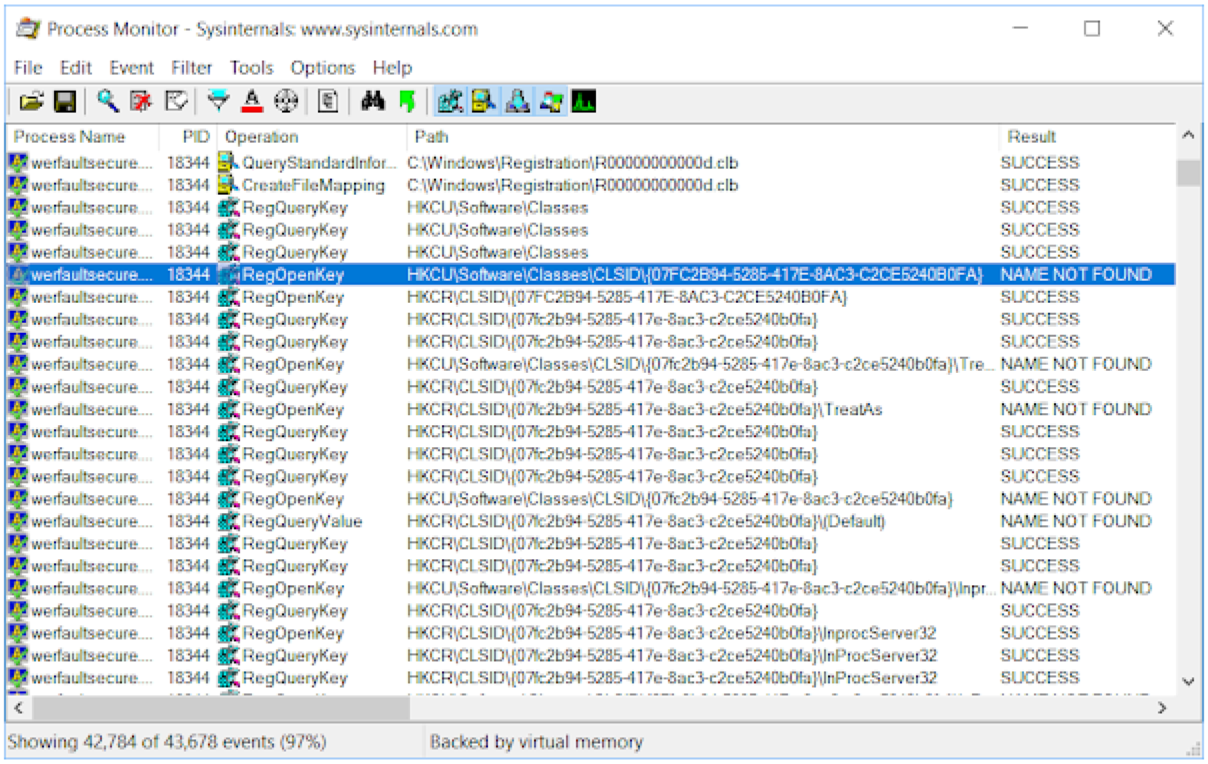
This gives us everything we need to complete the exploit. We don't need administrator privileges to start the old version of WerFaultSecure as PP-WindowsTCB. We can get it to dump another copy of WerFaultSecure with COM initialized and use the crash dump to extract all the information we need including the ALPC Port and IPID needed to communicate. We don't need to write our own crash dump parser as the Debug Engine API which comes installed with Windows can be used. Once we've extracted all the information we need we can call DoCallback and invoke arbitrary code.

Putting it All Together

There's still two things we need to complete the exploit, how to get WerFaultSecure to start up COM and what we can call to get completely arbitrary code running inside the PP-WindowsTCB process.

Let's tackle the first part, how to get COM started. As I mentioned earlier, WerFaultSecure doesn't directly call any COM methods, but Alex had clearly used it before so to save time I just asked him. The trick was to get WerFaultSecure to dump an AppContainer process, this results in a call to the method CCrashReport::ExemptFromPlmHandling inside the FaultRep DLL resulting in the loading of CLSID {07FC2B94-5285-417E-8AC3-C2CE5240B0FA}, which resolves to an undocumented COM object. All that matters is this allows WerFaultSecure to initialize COM.

Unfortunately I've not been entirely truthful during my description of how COM remoting is setup. Just loading a COM object is not always sufficient to initialize the IRundown interface or the RPC endpoint. This makes sense, if all COM calls are to code within the same apartment then why bother to initialize the entire remoting code for COM. In this case even though we can make WerFaultSecure load a COM object it doesn't meet the conditions to setup remoting. What can we do to convince the COM runtime that we'd really like it to initialize? One possibility is to change the COM registration from an in-process class to an OOP class. As shown in the screenshot below the COM registration is being queried first from HKEY_CURRENT_USER which means we can hijack it without needing administrator privileges.


Process Monitor - Sysinternals: www.sysinternals.com
File Edit Event Filter Tools Options Help

Process Name	PID	Operation	Path	Result
werfaultsecure...	18344	QueryStandardInfor...	C:\Windows\Registration\R00000000000d.clb	SUCCESS
werfaultsecure...	18344	CreateFileMapping	C:\Windows\Registration\R00000000000d.clb	SUCCESS
werfaultsecure...	18344	RegQueryKey	HKCU\Software\Classes	SUCCESS
werfaultsecure...	18344	RegQueryKey	HKCU\Software\Classes	SUCCESS
werfaultsecure...	18344	RegQueryKey	HKCU\Software\Classes	SUCCESS
werfaultsecure...	18344	RegOpenKey	HKCU\Software\Classes\CLSID\{07FC2B94-5285-417E-8AC3-C2CE5240B0FA}	NAME NOT FOUND
werfaultsecure...	18344	RegOpenKey	HKCR\CLSID\{07FC2B94-5285-417E-8AC3-C2CE5240B0FA}	SUCCESS
werfaultsecure...	18344	RegQueryKey	HKCR\CLSID\{07fc2b94-5285-417e-8ac3-c2ce5240b0fa}	SUCCESS
werfaultsecure...	18344	RegQueryKey	HKCR\CLSID\{07fc2b94-5285-417e-8ac3-c2ce5240b0fa}	SUCCESS
werfaultsecure...	18344	RegOpenKey	HKCU\Software\Classes\CLSID\{07fc2b94-5285-417e-8ac3-c2ce5240b0fa}\Tre...	NAME NOT FOUND
werfaultsecure...	18344	RegQueryKey	HKCR\CLSID\{07fc2b94-5285-417e-8ac3-c2ce5240b0fa}	SUCCESS
werfaultsecure...	18344	RegOpenKey	HKCR\CLSID\{07fc2b94-5285-417e-8ac3-c2ce5240b0fa}\TreatAs	NAME NOT FOUND
werfaultsecure...	18344	RegQueryKey	HKCR\CLSID\{07fc2b94-5285-417e-8ac3-c2ce5240b0fa}	SUCCESS
werfaultsecure...	18344	RegQueryKey	HKCR\CLSID\{07fc2b94-5285-417e-8ac3-c2ce5240b0fa}	SUCCESS
werfaultsecure...	18344	RegQueryKey	HKCR\CLSID\{07fc2b94-5285-417e-8ac3-c2ce5240b0fa}	SUCCESS
werfaultsecure...	18344	RegOpenKey	HKCU\Software\Classes\CLSID\{07fc2b94-5285-417e-8ac3-c2ce5240b0fa}	NAME NOT FOUND
werfaultsecure...	18344	RegQueryValue	HKCR\CLSID\{07fc2b94-5285-417e-8ac3-c2ce5240b0fa}\(Default)	NAME NOT FOUND
werfaultsecure...	18344	RegQueryKey	HKCR\CLSID\{07fc2b94-5285-417e-8ac3-c2ce5240b0fa}	SUCCESS
werfaultsecure...	18344	RegQueryKey	HKCR\CLSID\{07fc2b94-5285-417e-8ac3-c2ce5240b0fa}	SUCCESS
werfaultsecure...	18344	RegOpenKey	HKCU\Software\Classes\CLSID\{07fc2b94-5285-417e-8ac3-c2ce5240b0fa}\Inpr...	NAME NOT FOUND
werfaultsecure...	18344	RegQueryKey	HKCR\CLSID\{07fc2b94-5285-417e-8ac3-c2ce5240b0fa}	SUCCESS
werfaultsecure...	18344	RegOpenKey	HKCR\CLSID\{07fc2b94-5285-417e-8ac3-c2ce5240b0fa}\InprocServer32	SUCCESS
werfaultsecure...	18344	RegQueryKey	HKCR\CLSID\{07fc2b94-5285-417e-8ac3-c2ce5240b0fa}\InProcServer32	SUCCESS
werfaultsecure...	18344	RegQueryKey	HKCR\CLSID\{07fc2b94-5285-417e-8ac3-c2ce5240b0fa}\InProcServer32	SUCCESS

Showing 42,784 of 43,678 events (97%)   Backed by virtual memory

Unfortunately looking at the code this won't work, a cut down version is shown below:

```
HRESULT CCrashReport::ExemptFromPlmHandling(DWORD dwProcessId) {
 CoInitializeEx(NULL, COINIT_APARTMENTTHREADED);
 IOSTaskCompletion* inf;
 HRESULT hr = CoCreateInstance(CLSID_OSTaskCompletion,
    NULL, CLSCTX_INPROC_SERVER, IID_PPV_ARGS(&inf));
 if (SUCCEEDED(hr)) {
   // Open process and disable PLM handling.
 }
}
```

The code passes the flag, CLSCTX_INPROC_SERVER to CoCreateInstance. This flag limits the lookup code in the COM runtime to only look for in-process class registrations. Even if we replace the registration with one for an OOP class the COM runtime would just ignore it. Fortunately there's another way, the code is initializing the current thread's COM apartment as a STA using the COINIT_APARTMENTTHREADED flag with CoInitializeEx. Looking at the registration of the COM object its threading model is set to "Both". What this means in practice is the object supports being called directly from either a STA or a MTA.

However, if the threading model was instead set to "Free" then the object only supports direct calls from an MTA, which means the COM runtime will have to enable remoting, create the object in an MTA (using something similar to DoCallback) then marshal calls to that object from the original apartment. Once COM starts remoting it initializes all remote features including IRundown. As we can hijack the server registration we can just change the threading model, this will cause WerFaultSecure to start COM remoting which we can now exploit.

What about the second part, what can we call inside the process to execute arbitrary code? Anything we call using DoCallback must meet the following criteria, to avoid undefined behavior:

1. Only takes one pointer sized parameter.
2. Only the lower 32 bits of the call are returned as the HRESULT if we need it.
3. The callsite is guarded by CFG so it must be something which is a valid indirect call target.

As WerFaultSecure isn't doing anything special then at a minimum any DLL exported function should be a valid indirect call target. LoadLibrary clearly meets our criteria as it takes a single parameter which is a pointer to the DLL path and we don't really care about the return value so the truncation isn't important. We can't just load any DLL as it must be correctly signed, but what about hijacking KnownDlls?

Wait, didn't I say that PP can't load from KnownDlls? Yes they can't but only because the value of the LdrpKnownDllDirectoryHandle global variable is always set to NULL during process initialization. When the DLL loader checks for the presence of a known DLL if the handle is NULL the check returns immediately. However if the handle has a value it will do the normal check and just like in PPL no additional security checks are performed if the process maps an image from an existing section object. Therefore if we can modify the LdrpKnownDllDirectoryHandle global variable to point to a directory object inherited into the PP we can get it to load an arbitrary DLL.

The final piece of the puzzle is finding an exported function which we can call to write an arbitrary value into the global variable. This turns out to be harder than expected. The ideal function would be one which takes a single pointer value argument and writes to that location with no other side effects. After a number of false starts (including trying to use gets) I settled on the pair, SetProcessDefaultLayout and GetProcessDefaultLayout in USER32. The set function takes a single value which is a set of flags and stores it in a global location (actually in the kernel, but good enough). The get method will then write that value to an arbitrary pointer. This isn't perfect as the values we can set and therefore write are limited to the numbers 0-7, however by offsetting the pointer in the get calls we can write a value of the form 0x0?0?0?0? where the ? can be any value between 0 and 7. As the value just has to refer to the handle inside a process under our control we can easily craft the handle to meet these strict requirements.

Wrapping Up

In conclusion to get arbitrary code execution inside a PP-WindowsTCB without administrator privileges process we can do the following:

1. Create a fake KnownDlls directory, duplicating the handle until it meets a pattern suitable for writing through Get/SetProcessDefaultLayout. Mark the handle as inheritable.
2. Create the COM object hijack for CLSID {07FC2B94-5285-417E-8AC3-C2CE5240B0FA} with the ThreadingModel set to “Free”.
3. Start Windows 10 WerFaultSecure at PP-WindowsTCB level and request a crash dump from an AppContainer process. During process creation the fake KnownDlls must be added to ensure it’s inherited into the new process.
4. Wait until COM has initialized then use Windows 8.1 WerFaultSecure to dump the process memory of the target.
5. Parse the crash dump to discover the process secret, context pointer and IPID for IRundown.
6. Connect to the IRundown interface and use DoCallback with Get/SetProcessDefaultLayout to modify the LdrpKnownDllDirectoryHandle global variable to the handle value created in 1.
7. Call DoCallback again to call LoadLibrary with a name to load from our fake KnownDlls.

This process works on all supported versions of Windows 10 including 1809. It’s worth noting that invoking DoCallback can be used with any process where you can read the contents of memory and the process has initialized COM remoting. For example, if you had an arbitrary memory disclosure vulnerability in a privileged COM service you could use this attack to convert the arbitrary read into arbitrary execute. As I don’t tend to look for memory corruption/memory disclosure vulnerabilities perhaps this behavior is of more use to others.

That concludes my series of attacking Windows protected processes. I think it demonstrates that preventing a user from attacking processes which share resources, such as registry and files is ultimately doomed to fail. This is probably why Microsoft do not support PP/PPL as a security boundary. Isolated User Mode seems a much stronger primitive, although that does come with additional resource requirements which PP/PPL doesn’t for the most part. I wouldn’t be surprised if newer versions of Windows 10, by which I mean after version 1809, will try to mitigate these attacks in some way, but you’ll almost certainly be able to find a bypass.