Project Zero

News and updates from the Project Zero team at Google

Tuesday, April 21, 2020

You Won't Believe what this One Line Change Did to the Chrome Sandbox

Posted by James Forshaw, Project Zero

The Chromium sandbox on Windows has stood the test of time. It's considered one of the better sandboxing mechanisms deployed at scale without requiring elevated privileges to function. For all the good, it does have its weaknesses. The main one being the sandbox's implementation is reliant on the security of the Windows OS. Changing the behavior of Windows is out of the control of the Chromium development team. If a bug is found in the security enforcement mechanisms of Windows then the sandbox can break.

This blog is about a vulnerability introduced in Windows 10 1903 which broke some of the security assumptions that Chromium relied on to make the sandbox secure. I'll present how I used the bug to develop a chain of execution to escape the sandbox as used for the GPU Process on Chrome/Edge or the default content sandbox in Firefox. The exploitation process is also an interesting insight into the little weaknesses in Windows which in themselves do not cross a security boundary but led to a successful sandbox escape. This vulnerability was fixed in April 2020 as CVE-2020-0981.

Background to the Issue

Search This Blog

Search

Pages

- About Project Zero
- Working at Project Zero
- 0day "In the Wild"
- Vulnerability Disclosure FAQ

Archives

2020

- You Won't Believe what this One Line Change Did to... (Apr)
- TFW you-get-really-excited-you-patch-diffed-a-0day... (Apr)
- Escaping the Chrome Sandbox with RIDL (Feb)
- Mitigations are attack surface, too (Feb)
- A day^W^W Several months in the life of Project Ze... (Feb)
- A day^W^W Several months in the life of Project Ze... (Feb)

Let's have a quick look at how the Chromium sandbox works on Windows before describing the bug itself. The sandbox works on the concept of least privilege by using Restricted Tokens. A Restricted Token is a feature added in Windows 2000 to reduce the access granted to a process through the modification of the Process's Access Token through the following operations:

- Permanently disabling Groups.
- Removing Privileges.
- Adding Restricted SIDs.

Disabling groups removes the Access Token's membership, resulting in disabling access to resources secured by those groups. Removing privileges prevents the process from performing any unnecessary privileged operations. Finally, adding restricted SIDs changes the security access check process. To be granted access to a resource we need to match a security descriptor entry for both a group in our main list as well as the list of Restricted SIDs. If one of the lists of SIDs does not grant access to the resource then access will be denied.

Chromium also uses the Integrity Level (IL) feature added in Vista to further restrict resource access. By setting a low IL we can block write access to higher integrity resources regardless of the result of the access check.

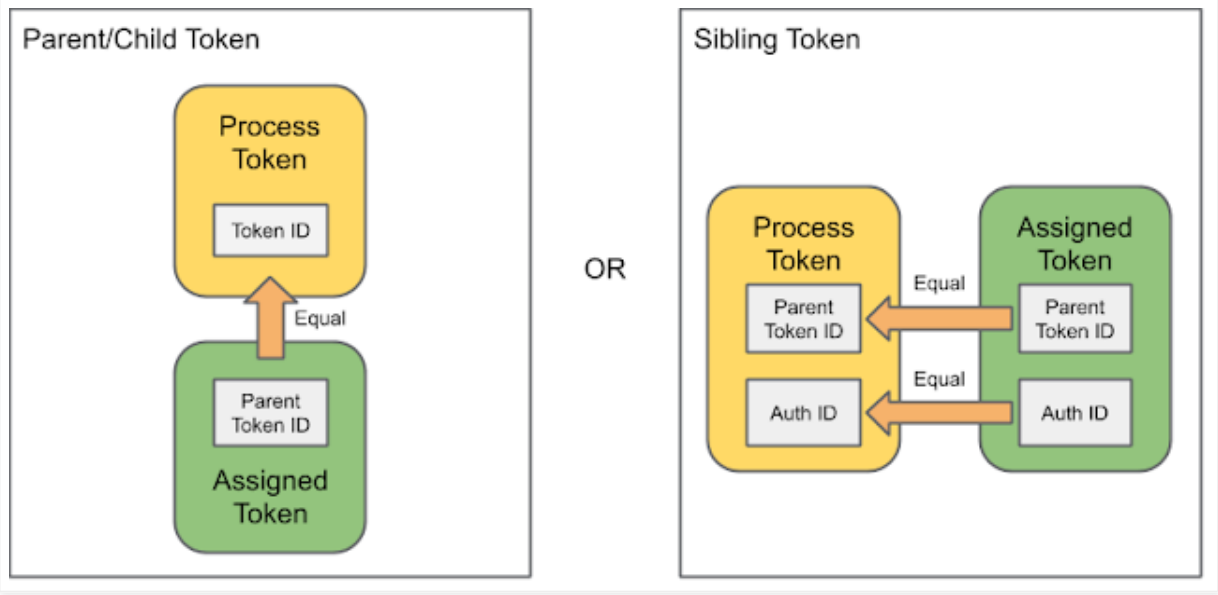
Using Restricted Tokens with IL in this way allows the sandbox to limit what resources a compromised process can access and therefore the impact an RCE can have. It's especially important to block write access as that would typically grant an attacker leverage to compromise other parts of the system by writing files or registry keys.

Any process on Windows can create a new process with a different Token, for example by calling CreateProcessAsUser. What stops a sandboxed process creating a new process using an unrestricted token? Windows and Chromium implement a few security mitigations to make creating a new process outside of the sandbox difficult:

1. The Kernel restricts what Tokens can be assigned by an unprivileged user to a new process.
2. The sandbox restrictions limit the availability of suitable access tokens to use for the new process.
3. Chromium runs a sandboxed process inside a Job object which is inherited by any child processes which has a hard process quota limit of 1.
4. From Windows 10, Chromium uses the Child Process Mitigation Policy to block child process creation. This is applied in addition to the Job object from 3.

All of these mitigations are ultimately relying on Windows to be secure. However by far the most critical is 1. Even if 2 through 4 fail, in theory we shouldn't be able to assign a more privileged access token to the new process. What is the kernel checking when it comes to assigning a new token?

Assuming the calling process doesn't have *SeAssignPrimaryTokenPrivilege* (which we don't) then the new token must meet one of two criteria which are checked in the kernel function *SeIsTokenAssignableToProcess*. The criteria are based on specified values in the kernel's TOKEN object structure as shown in the following diagram



In summary the token must either be:

a. A child of the current process token. Based on the new token's Parent Token ID being equal to the Process Token's ID.

b. A sibling of the current process token. Based on both the Parent Token ID and Authentication ID fields being equal.

There's also additional checks to ensure that the new Token is not an identification level impersonation token (due to this bug I reported) and the IL of the new token must be less than or equal to the current process token. These are equally important, but as we'll see, less useful in practice.

One thing the token assignment does not obviously check is whether the Parent or Child tokens are restricted. If you were in a restricted token sandbox could you get an Unrestricted Token which passes all of the checks and assign it to a child effectively escaping the sandbox? No you can't, the system ensures the Sibling Token check fails when assigning Restricted Tokens and instead ensures the Parent/Child check is the one which will be enforced. If you inspect the kernel function *SepFilterToken*, you'll understand how this is implemented. The following code is executed when copying the existing properties from the parent token to the new restricted token.

```
NewToken->ParentTokenId = OldToken->TokenId;
```

By setting the new Restricted Token's Parent Token ID it ensures that only the process which created the Restricted Token can use it for a child as the Token ID is unique for every instance of a TOKEN object. At the same time by changing the Parent Token ID the sibling check is broken.

However, when I was doing some testing to verify the token assignment behavior on Windows 10 1909 I noticed something odd. No matter what Restricted Token I created I couldn't get the assignment to fail. Looking at *SepFilterToken* again I found the code had changed.

```
NewToken->ParentTokenId = OldToken->ParentTokenId;
```

The kernel code was now just copying the Parent Token ID directly across from the old token. This completely breaks the check, as the new sandboxed process has a token which is considered a sibling of any other token on the desktop.

This one line change could just be sufficient to break out of the Restricted Token sandbox, assuming I could bypass the other 3 child process mitigations already in place. Let's go through the trials and tribulations undertaken to do just that.

Escaping the Sandbox

The final sandbox escape I came up with is quite complicated, it's also not necessarily the optimal approach. However, the complexity of Windows means it can be difficult to find alternative primitives to exploit in our chain.

- Injecting Code into Windows Protected Processes us... (Oct)
- 365 Days Later: Finding and Exploiting Safari Bugs... (Oct)
- A cache invalidation bug in Linux memory managemen... (Sep)
- OATmeal on the Universal Cereal Bus: Exploiting An... (Sep)
- The Problems and Promise of WebAssembly (Aug)
- Windows Exploitation Tricks: Exploiting Arbitrary ... (Aug)
- Adventures in vulnerability reporting (Aug)
- Drawing Outside the Box: Precision Issues in Graph... (Jul)
- Detecting Kernel Memory Disclosure – Whitepaper (Jun)
- Bypassing Mitigations by Attacking JIT Server in M... (May)
- Windows Exploitation Tricks: Exploiting Arbitrary ... (Apr)
- Reading privileged memory with a side-channel (Jan)

2017

- aPAColypse now: Exploiting Windows 10 in a Local N... (Dec)
- Over The Air - Vol. 2, Pt. 3: Exploiting The Wi-Fi... (Oct)
- Using Binary Diffing to Discover Windows Kernel Me... (Oct)
- Over The Air - Vol. 2, Pt. 2: Exploiting The Wi-Fi... (Oct)
- Over The Air - Vol. 2, Pt. 1: Exploiting The Wi-Fi... (Sep)
- The Great DOM Fuzz-off of 2017 (Sep)
- Bypassing VirtualBox Process Hardening on Windows (Aug)

Let's start with trying to get a suitable access token to assign to a new process. The token needs to meet some criteria:

1. The Token is a Primary token or convertible to a Primary Token.
2. The Token has an IL equal to the sandbox IL, or is writable so that the IL level can be reduced.
3. The Token meets the sibling token criteria so that it can be assigned.
4. The Token is for the current Console Session.
5. The Token is not sandboxed or is less sandboxed than the current token.

Access Tokens are securable objects therefore if you have sufficient access you can open a handle to a Token. However, Access Tokens are not referred to by a name, instead to open a Token you need to have access to either a Process or an Impersonating Thread. We can use my NtObjectManager PowerShell module to find accessible tokens using the *Get-AccessibleToken* command.

```
PS> $ps = Get-NtProcess -Name "chrome.exe" `
                -FilterScript { $_.IsSandboxToken } `
                -IgnoreDeadProcess
PS> $ts = Get-AccessibleToken -Processes $ps -CurrentSession `
                              -AccessRights Duplicate
PS> $ts.Count
101
```

This script gets a handle to every sandboxed Chrome process running on my machine (obviously start Chrome first), then uses the access token from each process to determine what other tokens we can open for *TOKEN_DUPLICATE* access. The reason for checking for *TOKEN_DUPLICATE* to use as the token in a new process is that we need to make a copy of the token as two processes can't use the same access token object. The access check takes into account whether the calling process would have *PROCESS_QUERY_LIMITED_INFORMATION* access to the target process which is a prerequisite for opening the Token. We've got a fair number of results, over 100 entries.

However this number is deceiving, for a start, some of the Tokens we can access will almost certainly be sandboxed more than the current token is sandboxed. Really we want only accessible tokens which are unsandboxed. Secondly, while there's a lot of accessible tokens, that's likely an artifact of a small number of processes being able to access a large number of tokens. We'll filter it down to just the command lines of the Chrome processes which can access non-sandboxed tokens.

- Windows Exploitation Tricks: Arbitrary Directory C... (Aug)
- Trust Issues: Exploiting TrustZone TEEs (Jul)
- Exploiting the Linux kernel via packet sockets (May)
- Exploiting .NET Managed DCOM (Apr)
- Exception-oriented exploitation on iOS (Apr)
- Over The Air: Exploiting Broadcom's Wi-Fi Stack (P... (Apr)
- Notes on Windows Uniscribe Fuzzing (Apr)
- Pandavirtualization: Exploiting the Xen hypervisor... (Apr)
- Over The Air: Exploiting Broadcom's Wi-Fi Stack (P... (Apr)
- Project Zero Prize Conclusion (Mar)
- Attacking the Windows NVIDIA Driver (Feb)
- Lifting the (Hyper) Visor: Bypassing Samsung's Rea... (Feb)

2016

- Chrome OS exploit: one byte overflow and symlinks (Dec)
- BitUnmap: Attacking Android Ashmem (Dec)
- Breaking the Chain (Nov)
- task_t considered harmful (Oct)
- Announcing the Project Zero Prize (Sep)
- Return to libstagefright: exploiting libutils on A... (Sep)
- A Shadow of our Former Self (Aug)
- A year of Windows kernel font fuzzing #2: the tech... (Jul)
- How to Compromise the Enterprise Endpoint (Jun)

```
PS> $ts | ? Sandbox -ne $true | `
    Sort {$_.TokenInfo.ProcessCommandLine} -Unique | `
    Select {$_.TokenInfo.ProcessId},
{$_.TokenInfo.ProcessCommandLine}

ProcessId ProcessCommandLine
--------- ---------------------------------
     6840 chrome.exe --type=gpu-process ...
    13920 chrome.exe --type=utility --service-sandbox-type=audio
...
```

Out of all the potential Chrome processes only the GPU process and the Audio utility process have access to non-sandbox tokens. This shouldn't come as a massive surprise. The renderer processes are significantly more locked down than either the GPU or Audio sandboxes due to the limitations of calling into system services for those processes to function. This does mean that the likelihood of an RCE to sandbox escape is much reduced, as most RCE occur in rendering HTML/JS content. That said GPU bugs do exist, for example this bug is one used by Lokihardt at Pwn2Own 2016.

Let's focus on escaping the GPU process sandbox. As I don't have a GPU RCE to hand I'll just inject a DLL into the process to run the escape. That's not as simple as it sounds, once the GPU process has started the process is locked down to only loading Microsoft signed DLLs. I use a trick with *KnownDlls* to load the DLL into memory (see this blog post for full details).

In order to escape the sandbox we need to do is the following:

1. Open an unrestricted token.
2. Duplicate token to create a new Primary Token and make the token writable.
3. Drop the IL of the token to match the current token (for GPU this is Low IL)
4. Call *CreateProcessAsUser* with the new token.
5. Escape Low IL sandbox.

Even for step 1 we've got a problem. The simplest way of getting an unrestricted token would be to open the token for the parent process which is the main Chrome browser process. However, if you look through the list of tokens the GPU process can access you'll find that the main Chrome browser process is not included. Why is that? This is intentional, as I realized after reporting this bug in the kernel that a GPU process sandbox could open the browser process' token. With this token it's possible to create a new restricted token

which would pass the sibling check to create a new process with much more access and escape the sandbox. To mitigate this I modified the access for the process token to block lower IL processes from opening the token for *TOKEN_DUPLICATE* access. See HardenTokenIntegerityLevelPolicy. Prior to this fix you didn't need a bug in the kernel to escape the Chrome GPU sandbox, at least to a normal Low IL token.

Therefore the easy route is not available to us, however we should be able to trivially enumerate processes and find one which meets our criteria. We can do this by using the *NtGetNextProcess* system call as I described in a previous blog post (on a topic we'll come back to later). We open all processes for *PROCESS_QUERY_LIMITED_INFORMATION* access, then open the token for *TOKEN_DUPLICATE* and *TOKEN_QUERY* access. We can then inspect the token to ensure it's unrestricted before proceeding to step 2.

To duplicate the token we call DuplicateTokenEx and request a primary token passing *TOKEN_ALL_ACCESS* as the desired access. But there's a new problem, when we try and lower the IL we get *ERROR_ACCESS_DENIED* from SetTokenInformation. This is due to a sandbox mitigation Microsoft added to Windows 10 and back ported to all supported OS's (including Windows 7). The following code is a snippet from *NtDuplicateToken* where the mitigation has been introduced.

```
ObReferenceObjectByHandle(TokenHandle, TOKEN_DUPLICATE,
    SeTokenObjectType, &Token, &Info);
DWORD RealDesiredAccess = 0;
if (DesiredAccess) {
    SeCaptureSubjectContext(&Subject);
    if (RtlIsSandboxedToken(Subject.PrimaryToken)
     && RtlIsSandboxedToken(Subject.ClientToken)) {
        BOOLEAN IsRestricted;
        SepNewTokenAsRestrictedAsProcessToken(Token,
            Subject.PrimaryToken, &IsRestricted);
        if (Token == Subject.PrimaryToken || IsRestricted)
            RealDesiredAccess = DesiredAccess;
        else
            RealDesiredAccess = DesiredAccess
                & (Info.GrantedAccess | TOKEN_READ |
TOKEN_EXECUTE);
    }
} else {
    RealDesiredAccess = Info.GrantedAccess;
}
```

```
SepDuplicateToken(Token, &DuplicatedToken, ...)
ObInsertObject(DuplicatedToken, RealDesiredAccess, &Handle);
```

When you duplicate a token the kernel checks if the caller is sandboxed. If sandboxed the kernel then checks if the token to be duplicated is less restricted than the caller. If it's less restricted then the code limits the desired access to *TOKEN_READ* and *TOKEN_EXECUTE*. This means that if we request a write access such as *TOKEN_ADJUST_DEFAULT* it'll be removed on the handle returned to us from the duplication call. In turn this will prevent us reducing the IL so that it can be assigned to a new process.

This would seem to end our exploit chain. If we can't write to the token, we can't reduce the token's IL, which prevents us from assigning it. But the implementation has a tiny flaw, the duplicate operation continues to complete and returns a handle just with limited access rights. When you create a new token object the default security grants the caller full access to the Token object. This means once you get back a handle to the new Token you can call the normal DuplicateHandle API to convert it to a fully writable handle. It's unclear if this was intentional or not, although it should be noted that the similar check in *CreateRestrictedToken* returns an error if the new token isn't as restricted. Whatever the case we can abuse this misfeature to get an writable unrestricted token to assign to the new process with the correct IL.

Now that we can get an unrestricted token we can call *CreateProcessAsUser* to create our new process. But not so fast, as the GPU process is still running in a restrictive Job object which prevents creating new processes. I detailed how Job objects prevent new process creation in my "In-Console-Able" blog post almost 5 years ago. Can we not use the same bug in the Console Driver to escape the Job object? On Windows 8.1 you probably can (although I'll admit I've not tested), however on Windows 10 there's two things which prevent us from using it:

1. Microsoft changed Job objects to support an auxiliary process counter. If you have *SeTcbPrivilege* you can pass a flag to *NtCreateUserProcess* to create a new process still inside the Job which doesn't count towards the process count. This is used by the Console Driver to remove the requirement to escape the Job. As we don't have *SeTcbPrivilege* in the sandbox we can't use this feature.
2. Microsoft added a new flag to Tokens which prevent them being used for a new process. This flag is set by Chrome on all sandboxed processes to restrict new child processes. Even without '1' the flag would block abusing the Console Driver to spawn a new process.

The combination of these two features blocks spawning a new process outside of the current Job by abusing the Console Driver. We need to come up with another way of escaping both the Job object restriction and to also circumvent the child process restriction flag.

- pwn4fun Spring 2014 - Safari - Part II (Nov)
- Project Zero Patch Tuesday roundup, November 2014 (Nov)
- Did the "Man With No Name" Feel Insecure? (Oct)
- More Mac OS X and iPhone sandbox escapes and kerne... (Oct)
- Exploiting CVE-2014-0556 in Flash (Sep)
- The poisoned NUL byte, 2014 edition (Aug)
- What does a pointer look like, anyway? (Aug)
- Mac OS X and iPhone sandbox escapes (Jul)
- pwn4fun Spring 2014 - Safari - Part I (Jul)
- Announcing Project Zero (Jul)

The Job object is inherited from parent to child, therefore if we could find a process outside of a Job object which the GPU process can control we can use that process as a new parent and escape the Job. Unfortunately, at least by default, if you check what processes the GPU process can access it can only open itself.

```
PS> Get-AccessibleProcess -ProcessIds 6804 -AccessRights GenericAll
`
             | Select-Object ProcessId, Name
ProcessId Name
--------- ----
     6804 chrome.exe
```

Opening itself isn't going to be very useful, and we can't rely on getting lucky with a process which just happens to be running at the time which is both accessible and not running a Job. We need to make our own luck.

One thing I noticed is that there's a small race condition setting up a new Chrome sandbox process. The process is first created, then the Job object is applied. If we could get the Chrome browser to spawn a new GPU process we could use it as a parent before the Job object is applied. The handling of the GPU process even supports respawning the process if it crashes. However I couldn't find a way of getting a new GPU process to spawn without also causing the current one to terminate so it wasn't possible to have code running long enough to exploit the race.

Instead I decided to concentrate on finding a RPC service which would create a new process outside of the Job. There's quite a few RPC services where process creation is the main goal, and others where process creation is a side effect. For example I already documented the Secondary Logon service in a previous blog post where the entire purpose of the RPC service is to spawn new processes.

There is a slight flaw in this idea though, specifically the child process mitigation flag in the token is inherited across impersonation boundaries. As it's common to use the impersonated token as the basis for the new process any new process will be blocked. However, we have an unrestricted token that does not have the flag set. We can use the unrestricted token to create a restricted token we can impersonate during a RPC call and we can bypass the child process mitigation flag.

I tried to list what known services could be used in this way, which I've put together in the following table:

Service	Is Accessible	Can Escape Job
Secondary Logon Service	No	No
WMI Win32_Process	No	Yes
User Account Control (UAC)	Yes	No
Background Intelligent Transfer Service (BITS)	No	Yes
DCOM Activator	Yes	Yes

The table is not exhaustive and there's likely to be other RPC services which would allow processes to be created. As we can see in the table, well known RPC services which spawn processes such as Secondary Logon, WMI and BITS are not accessible from our sandbox level. The UAC service is accessible and as I described in a previous blog post there exists a way of abusing the service to run arbitrary privileged code by abusing debug objects. Unfortunately when a new UAC process is created the service sets the parent process to the caller process. As the Job object is inherited the new process will be blocked.

The last service in the list is the DCOM Activator. This is the system service which is responsible for starting out-of-process COM servers and is accessible from our sandbox level. It also starts all COM servers as children of the service process which means the Job object is not inherited. Seems ideal, however there is a slight issue, in order for the DCOM Activator to be useful we need an out-of-process COM server that the sandbox can create. This object must meet a set of criteria:

1. The Launch Security for the server grants local activation to the sandbox.
2. The server must not run as Interactive User (which would spawn out of the sandbox) or inside a service process.
3. The server executable must be accessible to the restricted token.

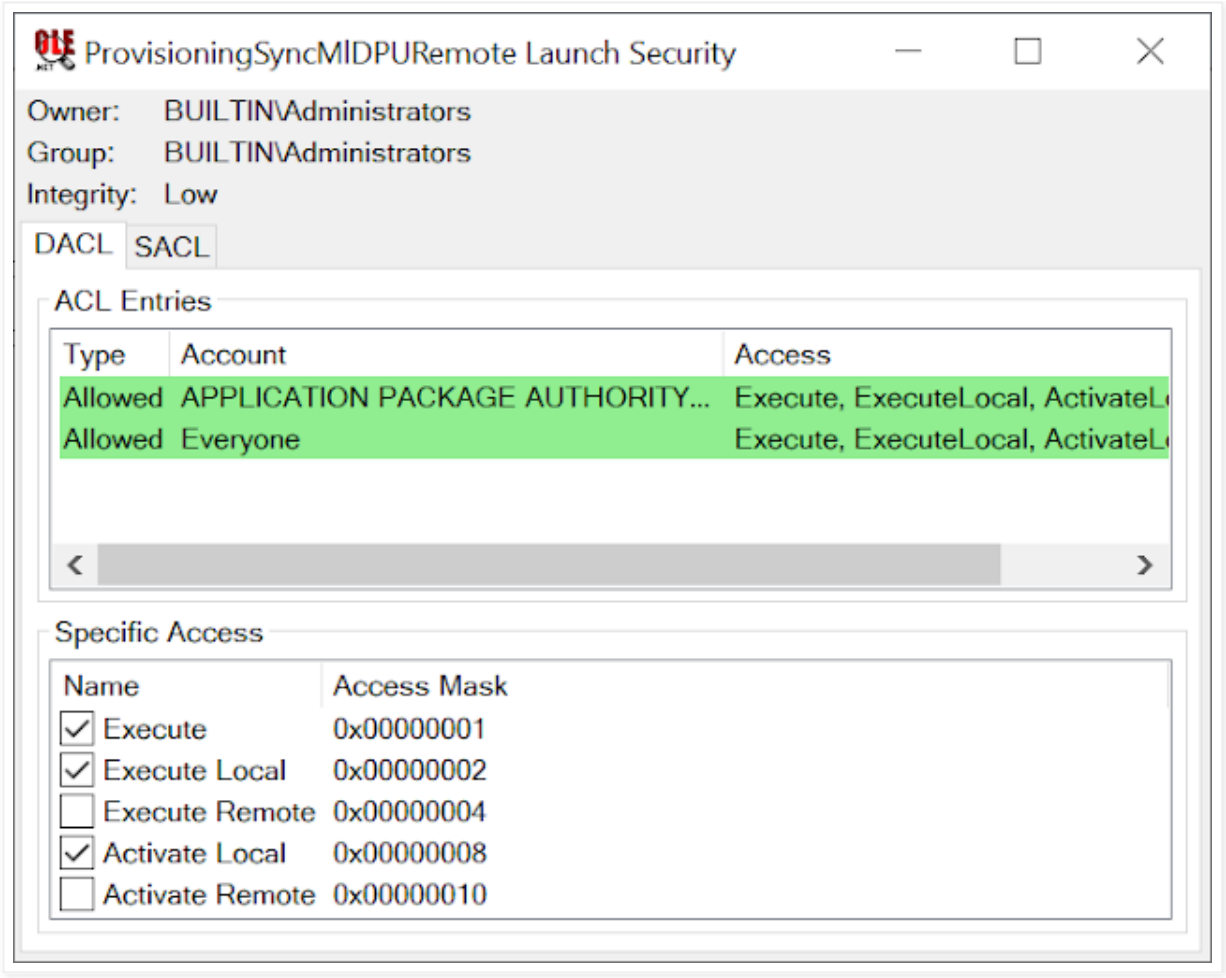
We don't have to worry about criteria 3, the GPU process can access system executables so we'll stick to pre-installed COM servers. It also doesn't matter if we can't access the COM server after creation, all we need is the rights to start the COM server process outside of the Job and then we can hijack it. We can find accessible COM servers using OleViewDotNet and the *Select-ComAccess* command.

```
PS> Get-ComDatabase -SetCurrent
PS> Get-ComClass -ServerType LocalServer32 | `
      Where-Object RunAs -eq "" | `
```

```
      Where-Object {$_.AppIdEntry.ServiceName -eq ""} | `
      Select-ComAccess -ProcessId 6804 `
           -LaunchAccess ActivateLocal -Access 0 | `
      Select-Object Clsid, DefaultServerName

Clsid                                DefaultServerName
-----                                -----------------
3d5fea35-6973-4d5b-9937-dd8e53482a56 coredpussvr.exe
417976b7-917d-4f1e-8f14-c18fccb0b3a8 coredpussvr.exe
46cb32fa-b5ca-8a3a-62ca-a7023c0496c5 ieframe.dll
4b360c3c-d284-4384-abcc-ef133e1445da ieframe.dll
5bbd58bb-993e-4c17-8af6-3af8e908fca8 ieproxy.dll
d63c23c5-53e6-48d5-adda-a385b6bb9c7b ieframe.dll
```

On a default installation of Windows 10 we have 6 candidates. Note that the last 4 are all in DLLs, however these classes are registered to run inside a DLL Surrogate so can still be used out-of-process. I decided to go for the servers in COREDPUSSVR because it's a unique executable rather than the generic DLLHOST so makes it easier to find. The Launch Security for this COM server grants *Everyone* and all AppContainer packages local activation permission as shown below:



As an aside, even though there are two classes registered for COREDPUSSVR, only the one starting with *417976b7* is actually registered by the executable. Creating the other class will start the server executable, however the class creation will hang waiting for a class which will never come.

To start the server you call CoCreateInstance while impersonating the child process mitigation flag-free restricted token. You need to pass the *CLSCTX_ENABLE_CLOAKING* as well to activate the server using the impersonation token, the default would use the process token which has the child process mitigation flag set and so would block process creation. Doing this, you'll find an instance of COREDPUSSVR running at

the same sandbox level however outside of the Job object and without the child process mitigation. Success?

Not so fast. Normally the default security of a new process is based on the default DACL inside the access token used to create it. Unfortunately for some unclear reason the DCOM activator sets an explicit DACL on the process which only grants access to the user, SYSTEM and the current logon SID. This doesn't allow the GPU process to open the new COM server process, even though it's running at effectively the same security level. So close and yet so far. I tried a few approaches to get code executed inside the COM server such as Windows Hooks however nothing obvious worked.

Fortunately, the default DACL is still used for any threads created after process startup. We can open one of those threads for full access and change the thread context to redirect execution using SetThreadContext. We'll need to brute force the thread IDs of these new threads as further sandbox mitigations block us from using CreateToolhelp32Snapshot to enumerate processes we can't open directly, and the *NtGetNextThread* API requires the parent process handle which we also don't have.

Abusing threads is painful especially as we can't write anything into the process directly, but it at least works. Where to redirect execution to? I decided for simplicity to call WinExec, which will spawn a new process and only requires a command line to execute. The new process will have the security based on the default DACL and so we can open it. I could have chosen something else like *LoadLibrary* to load a DLL in-process. However when messing with thread contexts there's a chance of crashing the process. I felt it was best just avoid that by escaping the process as quickly as possible.

What to use as the command line for *WinExec*? We can't directly write or allocate memory in the COM server process but we can easily repurpose existing strings in the binary to execute. To avoid having to find string addresses or deal with ASLR I just chose to use the PE signature at the start of DLL which gives us the string "PE". When passed to *WinExec* the current PATH environment variable will be used to find the executable to start. We can set PATH to anything we like in the COM server as the DCOM activator will use the caller's environment when starting a process at the same security level. The only thing we need to do is find a directory we can write to, and this time we can use *Get-AccessibleFile* to find a candidate as shown.

```
PS> Get-AccessibleFile -Win32Path "C:\" -Recurse -ProcessIds 6804 `
     -DirectoryAccessRights AddFile -CheckMode DirectoriesOnly `
     -FormatWin32Path | Select-Object Name
Name
----
```

```
C:\ProgramData\Microsoft\DeviceSync
```

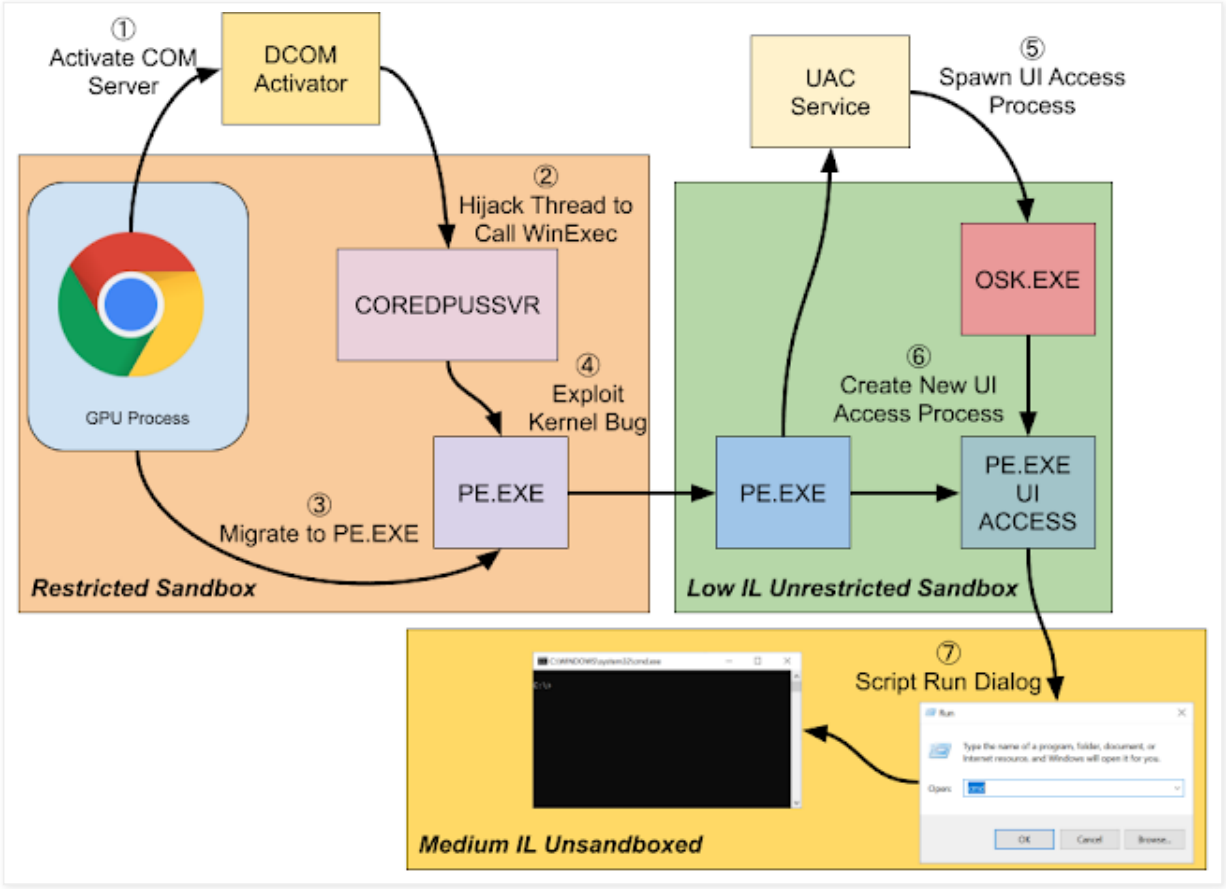
By setting the PATH Environment variable to contain the *DeviceSync* path and copying an executable named PE.exe to that directory we can set the thread context and spawn a new process which is out of the Job object and can be opened by the GPU process.

We can now exploit the kernel bug and call *CreateProcessAsUser* from the new process with the unrestricted token running at Low IL. This removes all sandboxing except for Low IL. The final step is breaking out of Low IL.Again there's probably plenty of ways of doing this but I decided to abuse the UAC service. I could have abused the Debug Object bug documented in my previous blog, however I decided to abuse a different "feature" of UAC. By abusing the same Token access we're abusing in the chain to open the unrestricted token we can get UI Access permissions. This allows us to automate privileged UI (such as the Explorer Run Dialog) to execute arbitrary code outside of the Low IL sandbox. It's not necessarily efficient but it's more photogenic. Full documentation for this attack is available in another blog post.

The final chain is as follows:

1. Open an unrestricted token.
    a. Brute force finding the process until a suitable process token is found.
2. Duplicate token to create a new Primary Token and make the token writable.
    a. Duplicate Token as Read Only
    b. Duplicate Handle to get back Write Access
3. Drop the IL of the token to match the current token.
4. Call *CreateProcessAsUser* with the new token.
    a. Create a new restricted token to remove the child process mitigation flag.
    b. Set the environment block's PATH to contain the *DeviceSync* folder and drop the PE.exe file.
    c. Impersonate restricted token and create OOP COM server.
    d. Brute force thread IDs of the COM server process.
    e. Modify thread context to call WinExec passing the address of a known PE signature in memory.
    f. Wait for the PE process to be created.
5. Escape Low IL sandbox.
    a. Spawn a copy of the On-Screen Keyboard and open its token.
    b. Create a new process with UI Access permission based on the opened token.
    c. Automate the Run dialog to escape the Low IL sandbox.

Or in diagrammatic form:



Wrapping Up

I hope this gives an insight into how such a small change in the Windows kernel can have a disproportionate impact on the security of a sandbox environment. It also demonstrates the value of exploit mitigations around sandbox behaviors. At numerous points the easy path to exploitation was shut down due to the mitigations.

It'd be interesting to read the post-mortem on how the vulnerability was introduced. I find it likely that someone was updating the code and thought that this was a mistake and so "fixed" it. Perhaps there was no comment indicating its purpose, or just the security critical nature of the single line was lost in the mists of time. Whatever the case it should now be fixed, which indicates it wasn't an intentional change.

Due to "features" in the OS, there's usually some way around these mitigations to achieve your goal even if it takes a lot of effort to discover. These features are not in themselves security issues but are useful for building chains.

Posted by Tim at 11:25 AM

No comments:

Post a Comment

Enter your comment...

Comment as: Google Accoun

Publish Preview

Home Older Post

Subscribe to: Post Comments (Atom)