

Windows Exploitation Tricks: Exploiting Arbitrary Object Directory Creation for Local Elevation of Privilege

 googleprojectzero.blogspot.com/2018/08/windows-exploitation-tricks-exploiting.html

Posted by James Forshaw, Project Zero

And we're back again for another blog in my series on Windows Exploitation tricks. This time I'll detail how I was able to exploit [Issue 1550](#) which results in an arbitrary object directory being created by using a useful behavior of the CSRSS privileged process. Once again by detailing how I'd exploit a particular vulnerability I hope that readers get a better understanding of the complexity of the Windows operating system as well as giving Microsoft information on non-memory corruption exploitation techniques so that they can mitigate them in some way.

Quick Overview of the Vulnerability

Object Manager directories are unrelated to normal file directories. The directories are created and manipulated using a separate set of system calls such as `NtCreateDirectoryObject` rather than `NtCreateFile`. Even though they're not file directories they're vulnerable to many of the same classes of issues as you'd find on a file system including privileged creation and symbolic link planting attacks.

[Issue 1550](#) is a vulnerability that allows the creation of a directory inside a user-controllable location while running as `SYSTEM`. The root of the bug is in the creation of [Desktop Bridge](#) applications. The `AppInfo` service, which is responsible for creating the new application, calls the undocumented API `CreateAppContainerToken` to do some internal housekeeping. Unfortunately this API creates object directories under the user's `AppContainerNamedObjects` object directory to support redirecting `BaseNamedObjects` and RPC endpoints by the OS.

As the API is called without impersonating the user (it's normally called in `CreateProcess` where it typically isn't as big an issue) the object directories are created with the identity of the service, which is `SYSTEM`. As the user can write arbitrary objects to their `AppContainerNamedObjects` directory they could drop an object manager symbolic link and redirect the directory creation to almost anywhere in the object manager namespace. As a bonus the directory is created with an explicit security descriptor which allows the user full access, this will become very important for exploitation.

One difficulty in exploiting this vulnerability is that if the object directory isn't created under `AppContainerNamedObjects` because we've redirected its location then the underlying

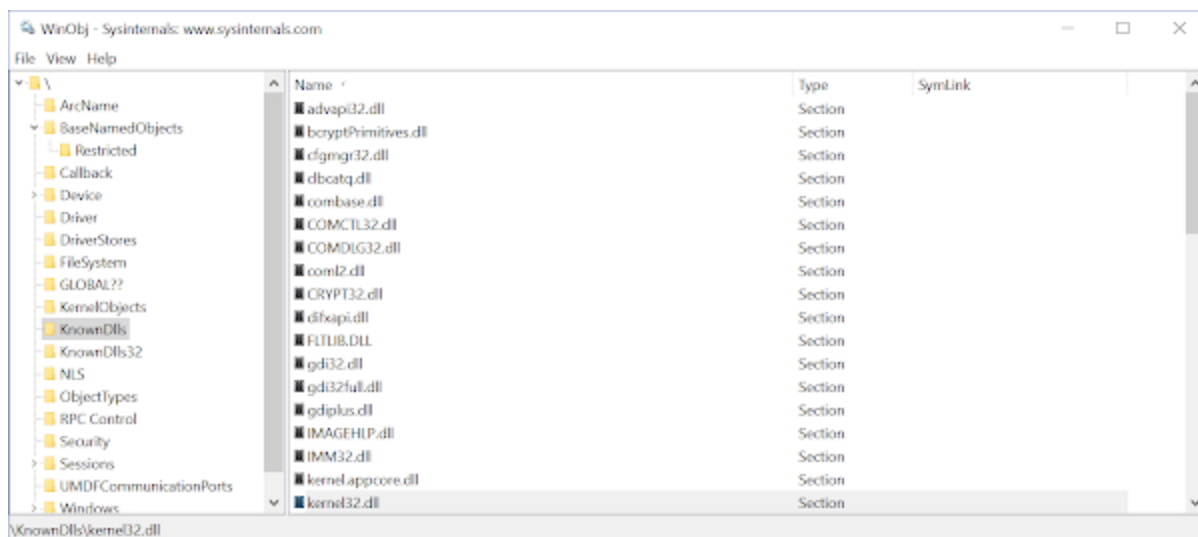
NtCreateLowBoxToken system call which performs the token creation and captures a handle to the directory as part of its operation will fail. The directory will be created but almost immediately deleted again. This behavior is actually due to an earlier issue [I reported](#) which changes the system call's behavior. This is still exploitable by opening a handle to the created directory before it's deleted, and in practice it seems winning this race is reliable as long as your system has multiple processors (which is basically any modern system). With an open handle the directory is kept alive as long as needed for exploitation.

This is the point where the original PoC I sent to MSRC stopped, all the PoC did was create an arbitrary object directory. You can find this PoC attached to the initial bug report in the issue tracker. Now let's get into how we might exploit this vulnerability to go from a normal user account to a privileged SYSTEM account.

Exploitation

The main problem for exploitation is finding a location in which we can create an object directory which can then be leveraged to elevate our privileges. This turns out to be harder than you might think. While almost all Windows applications use object directories under the hood, such as BaseNamedObjects, the applications typically interact with existing directories which the vulnerability can't be used to modify.

An object directory that would be interesting to abuse is KnownDlls (which I mentioned briefly in the previous [blog](#) in this series). This object directory contains a list of named image section objects, of the form NAME.DLL. When an application calls LoadLibrary on a DLL inside the SYSTEM32 directory the loader first checks if an existing image section is present inside the KnownDlls object directory, if the section exists then that will be loaded instead of creating a new section object.



KnownDlls is restricted to only being writable by administrators (not strictly true as we'll see) because if you could drop an arbitrary section object inside this directory you could force a

system service to load the named DLL, for example using the Diagnostics Hub service I described in my last blog post, and it would map the section, not the file on disk. However the vulnerability can't be used to modify the KnownDlls object directory other than adding a new child directory which doesn't help in exploitation. Maybe we can target KnownDlls indirectly by abusing other functionality which our vulnerability can be used with?

Whenever I do research into particular areas of a product I will always note down interesting or unexpected behavior. One example of interesting behavior I discovered when I was researching Windows symbolic links. The Win32 APIs support a function called DefineDosDevice, the purpose of this API is to allow a user to define a new DOS drive letter. The API takes three parameters, a set of flags, the drive prefix (e.g. X:) to create and the target device to map that drive to. The API's primary use is in things like the CMD SUBST command.

On modern versions of Windows this API creates an object manager symbolic link inside the user's own DOS device object directory, a location which can be written to by a normal low privileged user account. However if you look at the implementation of DefineDosDevice you'll find that it's not implemented in the caller's process. Instead the implementation calls an RPC method inside the current session's CSRSS service, specifically the method BaseSrvDefineDosDevice inside BASESRV.DLL. The main reason for calling into a privileged service is it allows a user to create a permanent symbolic link which doesn't get deleted when all handles to the symbolic link object are closed. Normally to create a permanent named kernel object you need the SeCreatePermanentPrivilege privilege, however a normal user does not have that privilege. On the other hand CSRSS does, so by calling into that service we can create the permanent symbolic link.

The ability to create a permanent symbolic link is certainly interesting, but if we were limited to only creating drive letters in the user's DOS devices directory it wouldn't be especially useful. I also noticed that the implementation never verified that the lpDeviceName parameter is a drive letter. For example you could specify a name of "GLOBALROOT\RPC Control\ABC" and it would actually create a symbolic link outside of the user's DosDevices directory, specifically in this case the path "\RPC Control\ABC". This is because the implementation prepends the DosDevice prefix "\??" to the device name and passes it to NtCreateSymbolicLink. The kernel would follow the full path, finding GLOBALROOT which is a special symbolic link to return to the root and then follow the path to creating the arbitrary object. It was unclear if this was intentional behavior so I looked in more depth at the implementation in CSRSS, which is shown in abbreviated form below.

```
NTSTATUS BaseSrvDefineDosDevice(DWORD dwFlags,  
                               LPCWSTR lpDeviceName,  
                               LPCWSTR lpTargetPath) {  
    WCHAR device_name[];
```

```

snwprintf_s(device_name, L"\\?\\?\\%s", lpDeviceName);
UNICODE_STRING device_name_ustr;
OBJECT_ATTRIBUTES objattr;
RtlInitUnicodeString(&device_name_ustr, device_name);
InitializeObjectAttributes(&objattr, &device_name_ustr,

                        OBJ_CASE_INSENSITIVE);

BOOLEAN enable_impersonation = TRUE;
CsrImpersonateClient();
HANDLE handle;
NTSTATUS status = NtOpenSymbolicLinkObject(&handle, DELETE, &objattr);①
CsrRevertToSelf();

if (NT_SUCCESS(status)) {
    BOOLEAN is_global = FALSE;

    // Check if we opened a global symbolic link.
    IsGlobalSymbolicLink(handle, &is_global); ②
    if (is_global) {
        enable_impersonation = FALSE; ③
        snwprintf_s(device_name, L"\\GLOBAL??\\%s", lpDeviceName);
        RtlInitUnicodeString(&device_name_ustr, device_name);
    }

    // Delete the existing symbolic link.
    NtMakeTemporaryObject(handle);
    NtClose(handle);
}

if (enable_impersonation) { ④
    CsrRevertToSelf();
}

// Create the symbolic link.
UNICODE_STRING target_name_ustr;
RtlInitUnicodeString(&target_name_ustr, lpTargetPath);

status = NtCreateSymbolicLinkObject(&handle, MAXIMUM_ALLOWED,

                        objattr, target_name_ustr); ⑤

if (enable_impersonation) { ⑥
    CsrRevertToSelf();
}

```

```

if (NT_SUCCESS(status)) {
    status = NtMakePermanentObject(handle); ⑦
    NtClose(handle);
}
return status;
}

```

We can see the first thing the code does is build the device name path then try and open the symbolic link object for DELETE access ①. This is because the API supports redefining an existing symbolic link, so it must first try to delete the old link. If we follow the default path where the link doesn't exist we'll see the code impersonates the caller (the low privileged user in this case) ④ then creates the symbolic link object ⑤, reverts the impersonation ⑥ and makes the object permanent ⑦ before returning the status of the operation. Nothing too surprising, we can understand why we can create arbitrary symbolic links because all the code does is prefix the passed device name with “\??”. As the code impersonates the caller when doing any significant operation we can only create the link in a location that the user could already write to.

What's more interesting is the middle conditional, where the target symbolic link is opened for DELETE access, which is needed to call NtMakeTemporaryObject. The opened handle is passed to another function ②, IsGlobalSymbolicLink, and based on the result of that function a flag disabling impersonation is set and the device name is recreated again with the global DOS device location \GLOBAL?? as the prefix ③. What is IsGlobalSymbolicLink doing? Again we can just RE the function and check.

```

void IsGlobalSymbolicLink(HANDLE handle, BOOLEAN* is_global) {
    BYTE buffer[0x1000];
    NtQueryObject(handle, ObjectNameInformation, buffer, sizeof(buffer));
    UNICODE_STRING prefix;
    RtlInitUnicodeString(&prefix, L"\\GLOBAL??\\");
    // Check if object name starts with \GLOBAL??
    *is_global = RtlPrefixUnicodeString(&prefix, (PUNICODE_STRING)buffer);
}

```

The code checks if the opened object's name starts with \GLOBAL??\. If so it sets the is_global flag to TRUE. This results in the flag enabling impersonation being cleared and the device name being rewritten. What this means is that if the caller has DELETE access to a symbolic link inside the global DOS device directory then the symbolic link will be recreated without any impersonation, which means it will be created as the SYSTEM user. This in itself doesn't sound especially interesting as by default only an administrator could open one of the global symbolic links for DELETE access. However, what if we could create a child directory underneath the global DOS device directory which could be written to by a low privileged

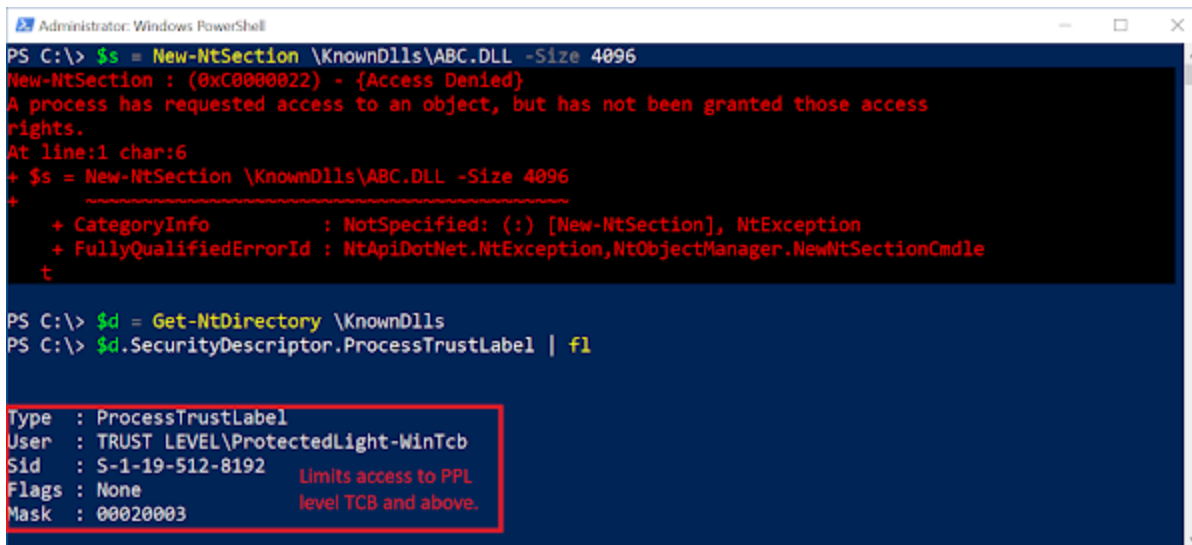
user? Any symbolic link in that directory could be opened for DELETE access as the low privileged user could specify any access they liked, the code would flag the link as being global, when in fact that's not really the case, disable impersonation and recreate it as SYSTEM. And guess what, we have a vulnerability which would allow us to create an arbitrary object directory under the global DOS device directory.

Again this might not be very exploitable if it wasn't for the rewriting of the path. We can abuse the fact that the path "\\??\\ABC" isn't the same as "\\GLOBAL??\\ABC" to construct a mechanism to create an arbitrary symbolic link anywhere in the object manager namespace as SYSTEM. How does this help us? If you write a symbolic link to KnownDlls then it will be followed by the kernel when opening a section requested by DLL loader. Therefore even though we can't directly create a new section object inside KnownDlls, we can create a symbolic link which points outside that directory to a place that the low-privileged user can create the section object. We can now abuse the hijack to load an arbitrary DLL into memory inside a privileged process and privilege elevation is achieved.

Pulling this all together we can exploit our vulnerability using the following steps:

1. Use the vulnerability to create the directory "\\GLOBAL??\\KnownDlls"
2. Create a symbolic link inside the new directory with the name of the DLL to hijack, such as TAPI32.DLL. The target of this link doesn't matter.
3. Inside the user's DOS device directory create a new symbolic link called "GLOBALROOT" pointing to "\\GLOBAL??". This will override the real GLOBALROOT symbolic link object when a caller accesses it via the user's DOS device directory.
4. Call DefineDosDevice specifying a device name of "GLOBALROOT\\KnownDlls\\TAPI32.DLL" and a target path of a location that the user can create section objects inside. This will result in the following operations:
 1. CSRSS opens the symbolic link "\\??\\GLOBALROOT\\KnownDlls\\TAPI32.DLL" which results in opening "\\GLOBAL??\\KnownDlls\\TAPI32.DLL". As this is controlled by the user the open succeeds, and the link is considered global which disables impersonation.
 2. CSRSS rewrites the path to "\\GLOBAL??\\GLOBALROOT\\KnownDlls\\TAPI32.DLL" then calls NtCreateSymbolicLinkObject without impersonation. This results in following the real GLOBALROOT link, which results in creating the symbolic link "\\KnownDlls\\TAPI32.DLL" with an arbitrary target path.
5. Create the image section object at the target location for an arbitrary DLL, then force it to be loaded into a privileged service such as the Diagnostics Hub by getting the service to call LoadLibrary with a path to TAPI32.DLL.
6. Privilege escalation is achieved.

Abusing the DefineDosDevice API actually has a second use, it's an Administrator to Protected Process Light (PPL) bypass. PPL processes still use KnownDlls, so if you can add a new entry you can inject code into the protected process. To prevent that attack vector Windows marks the KnownDlls directory with a Process Trust Label which blocks all but the highest level level PPL process from writing to it, as shown below.



```
Administrator: Windows PowerShell
PS C:\> $s = New-NtSection \KnownDlls\ABC.DLL -Size 4096
New-NtSection : (0xC0000022) - {Access Denied}
A process has requested access to an object, but has not been granted those access
rights.
At line:1 char:6
+ $s = New-NtSection \KnownDlls\ABC.DLL -Size 4096
+ ~~~~~
+ CategoryInfo          : NotSpecified: (:) [New-NtSection], NtException
+ FullyQualifiedErrorId : NtApiDotNet.NtException,NtObjectManager.NewNtSectionCmdle
t

PS C:\> $d = Get-NtDirectory \KnownDlls
PS C:\> $d.SecurityDescriptor.ProcessTrustLabel | fl

Type : ProcessTrustLabel
User  : TRUST_LEVEL\ProtectedLight-WinTcb
Sid   : S-1-19-512-8192      Limits access to PPL
Flags : None                level TCB and above.
Mask  : 00020003
```

How does our exploit work then? CSRSS actually runs as the highest level PPL so is allowed to write to the KnownDlls directory. Once the impersonation is dropped the identity of the process is used which will allow full access.

If you want to test this exploit I've attached the new PoC to the issue tracker [here](#).

Wrapping Up

You might wonder at this point if I reported the behavior of DefineDosDevice to MSRC? I didn't, mainly because it's not in itself a vulnerability. Even in the case of Administrator to PPL, MSRC do not consider that a serviceable security boundary ([example](#)). Of course the Windows developers might choose to try and change this behavior in the future, assuming it doesn't cause a major regression in compatibility. This function has been around since the early days of Windows and the current behavior since at least Windows XP so there's probably something which relies on it. By describing this exploit in detail, I want to give MS as much information as necessary to address the exploitation technique in the future.

I did report the vulnerability to MSRC and it was fixed in the June 2018 patches. How did Microsoft fix the vulnerability? The developers added a new API, CreateAppContainerTokenForUser which impersonates the token during creation of the new AppContainer token. By impersonating during token creation the code ensures that all objects are created only with the privileges of the user. As it's a new API existing code would

have to be changed to use it, therefore there's a chance you could still find code which uses the old `CreateAppContainerToken` in a vulnerable pattern.

Exploiting vulnerabilities on any platform sometimes requires pretty in-depth knowledge about how different components interact. In this case while the initial vulnerability was clearly a security issue, it's not clear how you could proceed to full exploitation. It's always worth keeping a log of interesting behavior which you encounter during reverse engineering as even if something is not a security bug itself, it might be useful to exploit another vulnerability.