Windows 10^H^H Symbolic Link Mitigations

googleprojectzero.blogspot.com/2015/08/windows-10hh-symbolic-link-mitigations.html

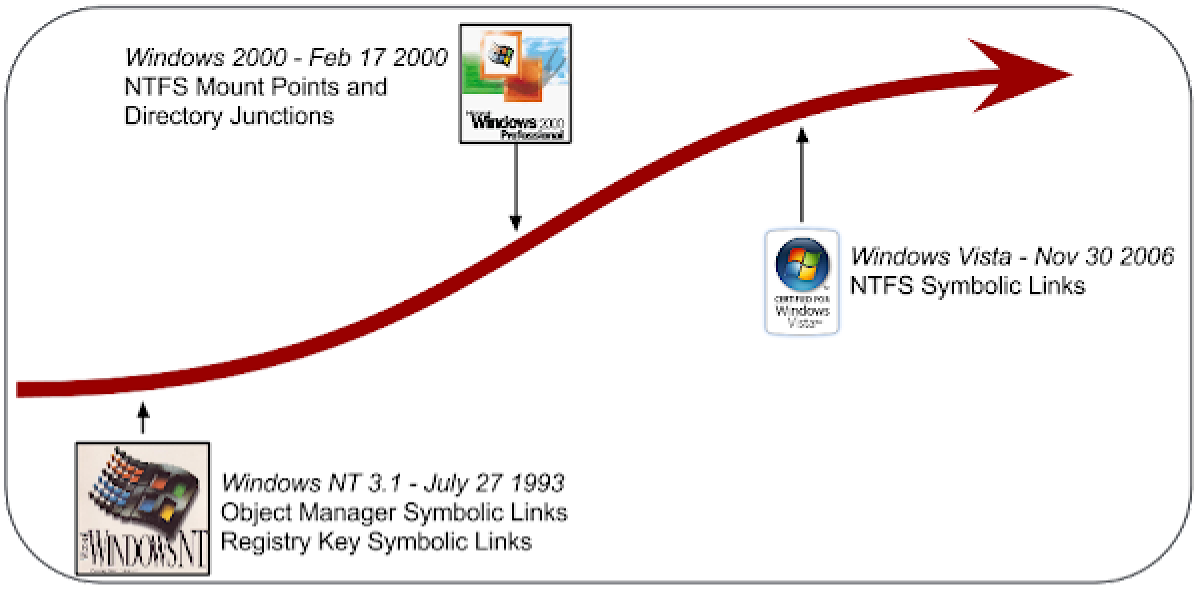
Posted by James Forshaw, abusing symbolic links like it's 1999.

For the past couple of years I've been researching Windows elevation of privilege attacks. This might be escaping sandboxing or gaining system privileges. One of the techniques I've used multiple times is abusing the symbolic link facilities of the Windows operating system to redirect privileged code to create files or registry keys to escape the restrictive execution context. Symbolic links in themselves are not vulnerabilities, instead they're useful primitives for exploiting different classes of vulnerabilities such as resource planting or time-of-check time-of-use.

This blog post contains details of a few changes Microsoft has made to Windows 10, and now back ported (in MS15-090) as far back as Windows Vista which changes who can use certain types of symbolic links. There's not been many mitigations of this type which get back ported to so many older versions of Windows. Therefore I feel this is a good example of a vendor developing mitigations in response to increased attacks using certain techniques which wouldn't have traditionally been considered before for mitigations.

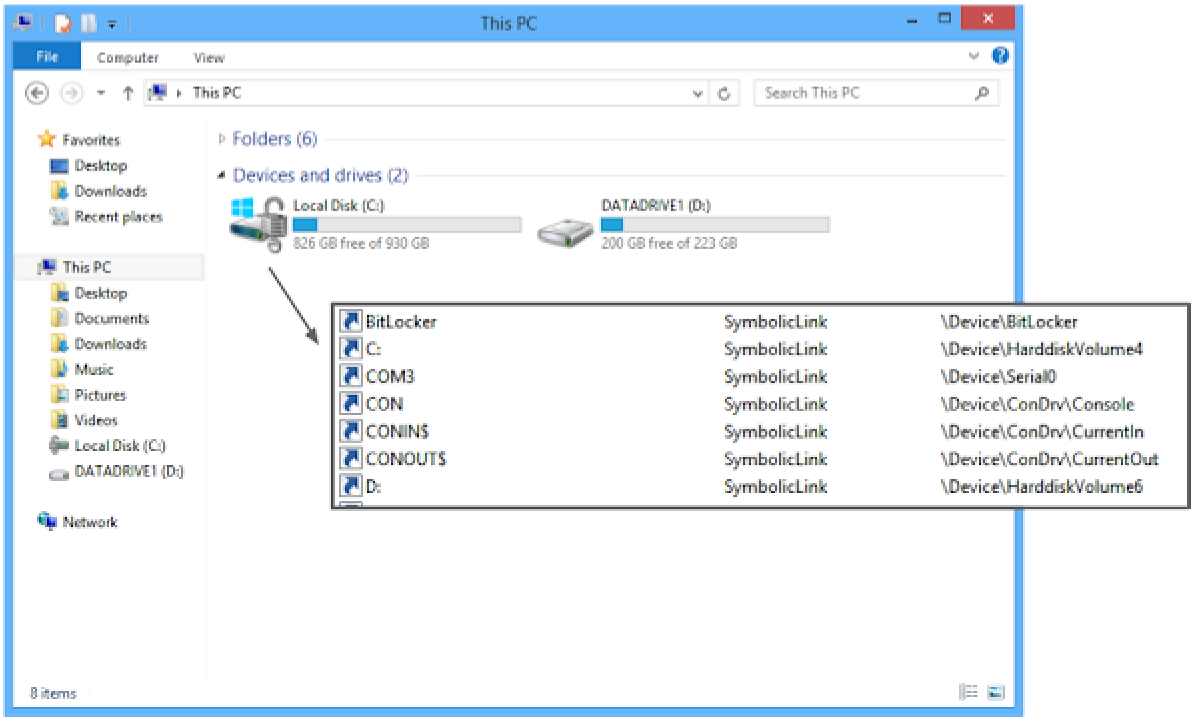
Quick Overview of Windows Symbolic Link Support

If you already know all about Windows Symbolic Link support you can always skip this, or perhaps view my presentation I made at this year's Infiltrate conference about abusing them. If not continue on. There are three types of symbolic links you can access from a low privileged user, Object Manager Symbolic Links, Registry Key Symbolic Links and NTFS Mount Points. There's actually a fourth type, NTFS Symbolic Links, however you can only create this type if you're an administrator making them of little use for privilege escalation. These symbolic link types have been added over the many years of the NT development as shown below.


Windows 2000 - Feb 17 2000
NTFS Mount Points and
Directory Junctions
Windows Vista - Nov 30 2006
NTFS Symbolic Links
Windows NT 3.1 - July 27 1993
Object Manager Symbolic Links
Registry Key Symbolic Links

Object Manager Symbolic Links

From a user perspective Windows supports multiple drives, such as C:. But that's not really what's seen under the hood. Behind the facade of the explorer shell and the Win32 API is another filesystem like structure, the object manager namespace. This is used to hold named resources such as devices and events, but crucially it has support for symbolic links. One of the main users of symbolic links is the aforementioned drive letters. If you look at the namespace using a tool such as WinObj you can find the drive letter symlinks and see them redirect to the real mounted device.

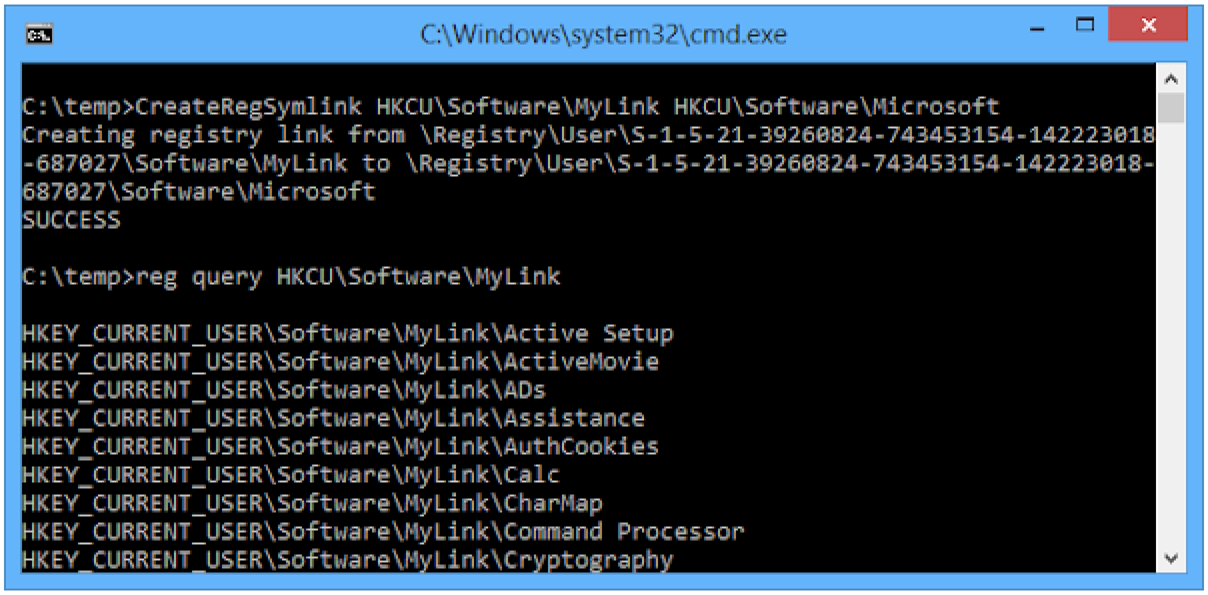


BitLocker	SymbolicLink	\Device\BitLocker
C:	SymbolicLink	\Device\HarddiskVolume4
COM3	SymbolicLink	\Device\Serial0
CON	SymbolicLink	\Device\ConDrv\Console
CONIN$	SymbolicLink	\Device\ConDrv\CurrentIn
CONOUT$	SymbolicLink	\Device\ConDrv\CurrentOut
D:	SymbolicLink	\Device\HarddiskVolume6

While these symbolic links are supposed to be used only for system purposes it's possible to create them as a low privileged user, as long as you have access to the target area of the object manager namespace. While there's a few attacks which can be facilitated with them it's easiest to exploit in file attacks. An simple example is CVE-2015-0055 which was an information disclosure issue in the IE EPM sandbox which abused symbolic links to bypass a security check.

Registry Key Symbolic Links

The Windows Registry is used to store configuration information for the system and isn't something which your normal user of Windows needs to worry about. While reasonably well documented it does have some features which the normal Win32 APIs do not document, one of which is unsurprisingly symbolic links between keys. This is used by the system to map in the current system configuration at boot time (the well known, CurrentControlSet) but it's not really used outside of that.

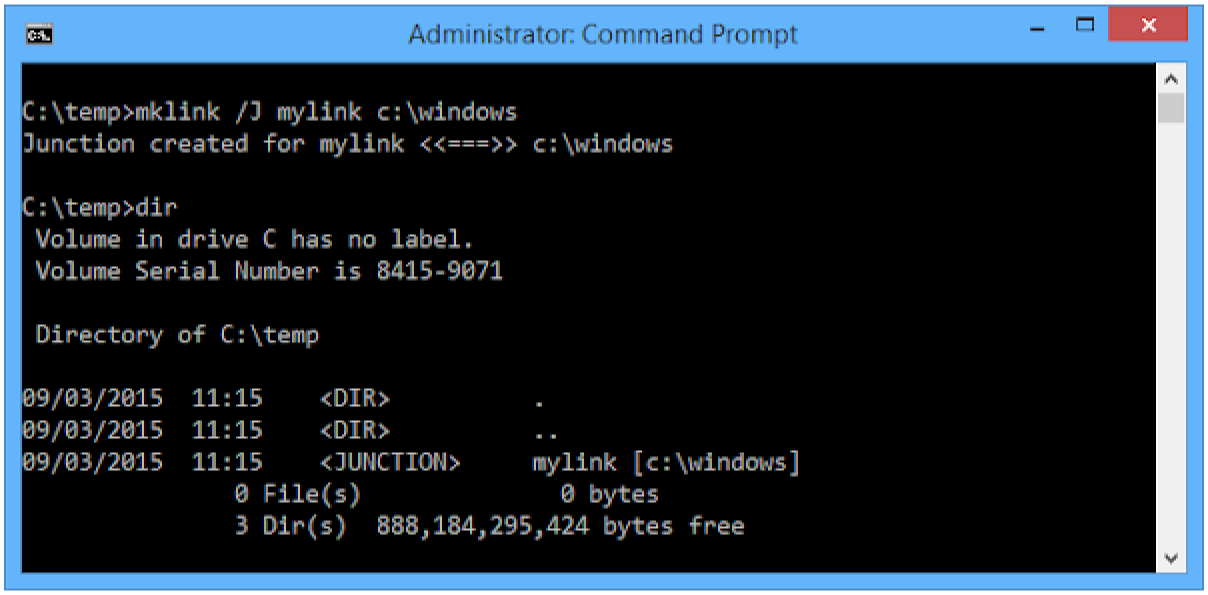


```
C:\temp>CreateRegSymlink HKCU\Software\MyLink HKCU\Software\Microsoft
Creating registry link from \Registry\User\S-1-5-21-39260824-743453154-142223018
-687027\Software\MyLink to \Registry\User\S-1-5-21-39260824-743453154-142223018-
687027\Software\Microsoft
SUCCESS

C:\temp>reg query HKCU\Software\MyLink

HKEY_CURRENT_USER\Software\MyLink\Active Setup
HKEY_CURRENT_USER\Software\MyLink\ActiveMovie
HKEY_CURRENT_USER\Software\MyLink\ADs
HKEY_CURRENT_USER\Software\MyLink\Assistance
HKEY_CURRENT_USER\Software\MyLink\AuthCookies
HKEY_CURRENT_USER\Software\MyLink\Calc
HKEY_CURRENT_USER\Software\MyLink\CharMap
HKEY_CURRENT_USER\Software\MyLink\Command Processor
HKEY_CURRENT_USER\Software\MyLink\Cryptography
```

The fact that a low privilege process can create these types of symbolic links has been abused before (see MS10-020) which removed the ability to create symbolic links between a untrusted registry hive such as the current user's hive and trusted system hive, but it didn't do anything to block the sandbox case where the symbolic link attacks the same user's hive at different privilege levels. An example of a vulnerability exploitable using this type of link was CVE-2014-6322 which was an issue with the Windows Audio Server which could be exploited from the IE EPM sandbox.

NTFS Mount Points

The final type of symbolic link allows a directory on the NTFS file system to be linked to another directory either on the same volume or on a completely different volume. It can't be used to directly link to a single file (at least without some tricks) but it's still useful. For example if you can find a privilege process which can be used to drop a file in a specified directory the write can be re-directed to somewhere the attacker controls.



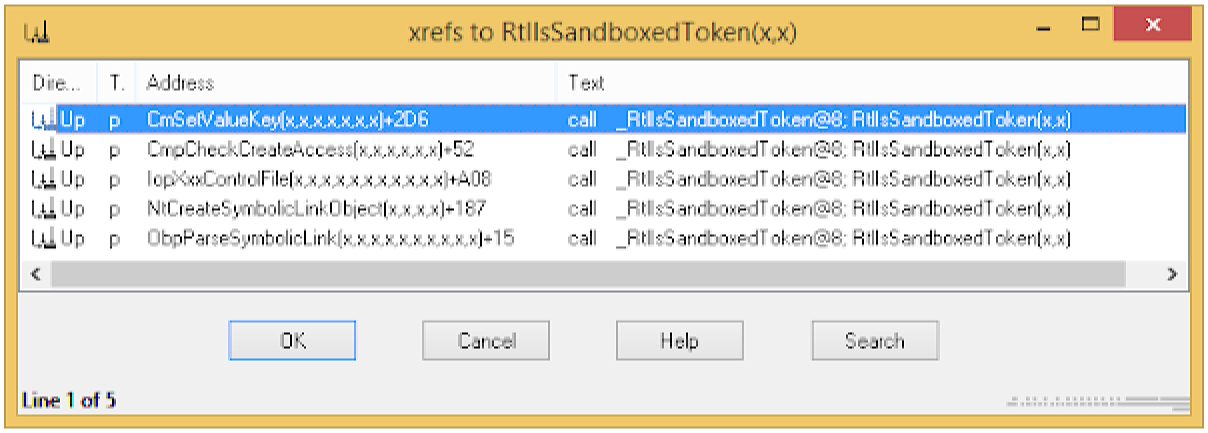
```
C:\temp>mklink /J mylink c:\windows
Junction created for mylink <<===>> c:\windows

C:\temp>dir
 Volume in drive C has no label.
 Volume Serial Number is 8415-9071

 Directory of C:\temp

09/03/2015  11:15    <DIR>          .
09/03/2015  11:15    <DIR>          ..
09/03/2015  11:15    <JUNCTION>     mylink [c:\windows]
               0 File(s)              0 bytes
               3 Dir(s)  888,184,295,424 bytes free
```

The only requirement on creating the mount point is a writable handle to a directory on the filesystem which is usually easy enough to achieve. This is probably the most used type of symbolic link for vulnerability exploitation, at least being used against Chrome, IE, and Adobe Reader sandboxes before now. An example is CVE-2014-0568 which was an issue I found in Adobe Reader which allows you to create an arbitrary file which could be used to escape the sandbox.

Technical Details of Mitigations

Now let’s go into some technical details of the mitigations, what they do and what they don’t. The root of all the mitigations is a new exported kernel function, RtlIsSandboxToken. This function determines if the current caller is considered to be in a sandbox, which in this case means either running at a integrity level below medium integrity or running within an AppContainer. The check is made by capturing the current subject context and calling SeAccessCheck with a Security Descriptor requiring medium integrity level. This shouldn’t be easily bypassable.

At this point you might assume that all the mitigations will do is call the method when creating a symbolic link and refuse to create them, but due to application compatibility it isn’t that simple. It’s easy to find the interactions by looking at the reference to the function in IDA or similar. I’ll briefly describe how each one is applied and compromises being made.



Dire...	T.	Address	Text
Up	p	CmSetValueKey(x,x,x,x,x,x,x)+2D6	call _RtlIsSandboxedToken@8; RtlIsSandboxedToken(x,x)
Up	p	CmpCheckCreateAccess(x,x,x,x,x,x)+52	call _RtlIsSandboxedToken@8; RtlIsSandboxedToken(x,x)
Up	p	IopXxxControlFile(x,x,x,x,x,x,x,x,x,x,x)+A08	call _RtlIsSandboxedToken@8; RtlIsSandboxedToken(x,x)
Up	p	NtCreateSymbolicLinkObject(x,x,x,x)+187	call _RtlIsSandboxedToken@8; RtlIsSandboxedToken(x,x)
Up	p	ObpParseSymbolicLink(x,x,x,x,x,x,x,x,x,x)+15	call _RtlIsSandboxedToken@8; RtlIsSandboxedToken(x,x)

Line 1 of 5

Registry Key Symbolic Link Mitigation (CVE-2015-2429)

The simplest mitigation implementation is for registry keys. Effectively a sandboxed process is not allowed to ever create a registry key symbolic link. This is implemented by calling RtlIsSandboxToken function when creating a new key (you need to specific a special flag when creating a key symbolic link). It's also called when setting the SymbolicLinkValue value which contains the link target. This second check is necessary to prevent modifying existing symbolic links, although it would be unlikely to be something found on a real system.

Object Manager Symbolic Link Mitigation (CVE-2015-2428)

If an application tries to create an object manager symbolic link from a sandbox process it will still seem to work, however if you look at where the check is called you'll find it doing something interesting. When the symbolic link is created the RtlIsSandboxToken function is called but the kernel doesn't immediately return an error. Instead it uses it to set a flag inside the symbolic link kernel object which indicates to the object manager a sandboxed process has created this link.

This flag is then used in the ObpParseSymbolicLink function which is called when the object manager is resolving the target of a symbolic link. The RtlIsSandboxToken is called again, if the current caller is not in a sandbox but the creator was in a sandbox then the kernel will return an error and not resolve the symbolic link, effective making the link useless for a sandboxed to unsandboxed elevation.

The behaviour is likely for applications running in AppContainers which might need to create symbolic links, but only other sandbox processes would need to follow the links. It's quite a pragmatic way of mitigating the issue without breaking application compatibility. However it does bring an additional performance cost, every time a symbolic link is resolved (such as the C: drive) an access check must be performed, presumably this has been measured to have a negligible impact.

NTFS Mount Point Mitigation (CVE-2015-2430)

The final mitigation is for NTFS mount points. In early technical previews of Windows 10 (I first spotted the change in 10130) the check was in the NTFS driver itself and explicitly blocked the creation of mount points from a sandboxed process. Again for presumably application compatibility reasons this restriction has been relaxed in the final release and the back ported mitigations.

Instead of completely blocking creation the kernel function IopXxxControlFile has been modified so whenever it sees the FSCTL_SET_REPARSE_POINT file system control code being passed to a driver with a mount point reparse tag it tries to verify if the sandboxed caller has write access to the target directory. If access is not granted, or the directory doesn’t exist then setting the mount point fails. This ensures that in the the majority of situations the sandboxed application couldn’t elevate privileges, as it could already write to the directory already. There’s obviously a theoretical issue in that the target could later be deleted and replaced by something important for a higher privileged process but that’s not very likely to occur in a practical, reliable exploit.

Conclusions

These targeted mitigations gives a clear indication that bug hunting and disclosing the details of how to exploit certain types of vulnerabilities can lead into mitigation development, even if they’re not traditional memory corruption bugs. While I didn’t have a hand in the actual development of the mitigation It’s likely my research was partially responsible for Microsoft acting to develop them. It’s very interesting that 3 different approaches ended up being taken, reflecting the potential application compatibility issues which might arise.

Excluding any bypasses which might come to light these should make entire classes of resource planting bugs unexploitable from a compromised sandboxed process and would make things like time-of-check time-of-use harder to exploit. Also it shows the level of effort that implementing mitigations without breaking backwards compatibility requires. The fact that these only target sandboxes and not system level escalation is particularly telling in this regard.