Windows Server Containers Are Open, and Here's How You Can Break Out

By Daniel Prizmant

Published: 2020-07-15 · Archived: 2026-04-05 22:05:25 UTC

Executive Summary

In my last post about reverse engineering Windows containers, I outlined the internal implementation of Windows Server Containers. After further investigating Windows Server Containers, I learned that running any code in these containers should be considered as dangerous as running admin on the host. These containers are not designed for sandboxing, and I found that escaping them is easy. Microsoft collaborated with us in fully understanding the security limitations of these containers. The purpose of this post is to raise awareness of the danger of running these containers.

To demonstrate this issue, I will present a container escape technique in Windows containers that I recently discovered. The technique allows processes running inside containers to write files to their host. This could be leveraged to achieve RCE (remote code execution) on the host. In Kubernetes environments, this exploit could be easily leveraged to spread between nodes. In other words, an attacker that successfully breaches a single application instance running inside a Windows Server Container could trivially breach the boundaries of the container and access other applications on the same machine. In the case of Kubernetes, the attacker could even access other machines. This may allow an attacker to gain access to a complete production workload after breaching just one endpoint instance.

This issue may affect users of cloud providers allowing the use of Windows Server Containers, including all of Microsoft's AKS users using Windows. Palo Alto Networks customers are protected from this via Prisma™ Cloud.

Windows Server Containers

As revealed in more depth in my previous post, Microsoft developed two solutions for running Windows-based containers. The first solution is running each container inside a virtual machine (VM) based on HyperV technology. The second option, Windows Server Containers, rely on Windows kernel features, such as Silo objects, to set up containers. The latter solution resembles traditional Linux implementation for containers, i.e. processes that are run on the same kernel with logical mechanisms to isolate each from another.

Some users rely on Windows Server Containers, as opposed to HyperV containers, since running each container inside a VM comes at a performance cost, as documented by Microsoft:

"The additional isolation provided by Hyper-V containers is achieved in large part by a hypervisor layer of isolation between the container and the container host. This affects container density as, unlike Windows Server

Containers, less sharing of system files and binaries can occur, resulting in an overall larger storage and memory footprint. In addition there is the expected additional overhead in some network, storage io, and CPU paths."

My research has led me to believe that the security of Windows Server Containers can be better documented. There are references indicating that the use of HyperV containers is more secure, but I could not find a piece of documentation that clearly mentions that Windows containers are susceptible to a breakout. When we reached out to Microsoft, their guidance was recommending users not run anything in a Windows Server Container that they wouldn't be willing to run as an admin on the host. They also noted:

"Windows Server Containers are meant for enterprise multi-tenancy. They provide a high degree of isolation between workloads, but are not meant to protect against hostile workloads. Hyper-V containers are our solution for hostile multi-tenancy."

In the following sections, I will go through the details of the problem, including kernel internals of Windows symbolic links. Some background in Windows container internals, including Silos, as explained in my previous post, is recommended for better understanding of the proposed technique.

The Escape

Windows symbolic link resolution from inside a container supports the use of an undocumented flag that causes symbolic links to be resolved on the root directory of the host machine. That is, outside the container file system. While container processes should require special privileges to enable that flag, I found a technique to escalate privileges from a default container process that would result in this escape.

In the following sections, I will take you through the journey of how I discovered this technique and elaborate the reasons it was made possible.

Symbolic Links

Symbolic links in Windows aren't well-documented, but they have been around since Windows NT. Windows NT came out with two types of symbolic links: object manager symbolic links and registry key symbolic links. These were not file-related, only an internal implementation of the operating system Microsoft chose to use. Only in Windows 2000 did file system symbolic links come out, and even those weren't file-level symbolic links. They worked only as directory redirection. It was Windows Vista that first featured full file-level symbolic links. In this post, I will only cover object manager symbolic links. The others are outside the scope of this article.

Object Manager Symbolic Links

If you're using Windows at all, you are probably using these without even knowing it. Things like the C drive letter are actually implemented using object manager symbolic links. Under the hood, when one accesses C:\ the object manager redirects the call to the actual mounted device.

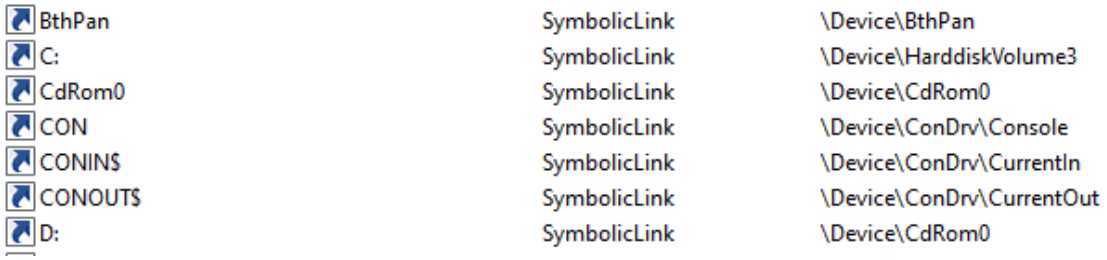

Figure 1. WinObj showing C: is just a symbolic link

The object manager handles not only files, but also registry, semaphores and many more named objects. When a user tries to access C:\secret.txt, the call arrives to the kernel function NtCreateFile with the path \??\C:\secret.txt, which is an NT path that the kernel knows how to work with. The path is modified by user-mode Windows API before the actual system call occurs. The reason for this path conversion is the \??\ part, which points the kernel to the correct directory in the root directory manager. Said directory will hold the target of the C: symbolic link.

Eventually ObpLookupObjectName is called. ObpLookupObjectName's job is to resolve an actual object from a name. This function uses another kernel function, ObpParseSymbolicLinkEx, to parse part of the path, which is a symbolic link to its target.

Every part of the path is checked for being a symbolic link. This check is performed by ObpParseSymbolicLinkEx. The object manager iterates until it finds a leaf node, which is something that cannot be parsed any further by the object manager. If part of the path is a symbolic link, the function returns STATUS_REPARSE or STATUS_REPARSE_OBJECT and changes the relevant part of the path to the target of the symbolic link.

```
1: kd> k
 # Child-SP          RetAddr           Call Site
00 ffff9e01`2af1f5d8 fffff805`4fbed1bf nt!ObpParseSymbolicLinkEx
01 ffff9e01`2af1f5e0 fffff805`4fbeb621 nt!ObpLookupObjectName+0x78f
02 ffff9e01`2af1f7a0 fffff805`4fc30df0 nt!ObOpenObjectByNameEx+0x201
03 ffff9e01`2af1f8e0 fffff805`4fc305b9 nt!IopCreateFile+0x820
04 ffff9e01`2af1f980 fffff805`4f7d2d15 nt!NtCreateFile+0x79
05 ffff9e01`2af1fa10 00007ff8`e79fcb64 nt!KiSystemServiceCopyEnd+0x25
06 00000088`f738fa08 00007ff8`e55345e4 ntdll!NtCreateFile+0x14
07 00000088`f738fa10 00007ff8`e55342d6 KERNELBASE!CreateFileInternal+0x2f4
08 00000088`f738fb80 00007ff6`8ed165c1 KERNELBASE!CreateFileW+0x66
09 00000088`f738fbe0 00007ff6`8ed16ff9 SimpleCreateFile!main+0x61 [C:\Users\Daniel\
```

Figure 2. WinDbg showing the call stack of a CreateFile API

After all of this, our C:\secret.txt was parsed to its actual path, which will look something like \Device\HarddiskVolume3\secret.txt. The \Device\HarddiskVolume3 path will be opened under the root directory object (ObpRootDirectoryObject).

More About the Root Directory Object

The object manager root directory object is like a folder that contains all application-visible named objects (like files, registry keys and more). This mechanism allows applications to create and access these objects among themselves.

The Important Part

When accessing a file from inside a container, everything is parsed under a custom root directory object. When C: is parsed, it will be parsed against a clone C: symbolic link that will point it to a virtual mounted device and not the host's file system.

Symbolic Links and Containers

I decided to follow the lookup process of a symbolic link from inside a container. A process inside a container calls CreateFile with the target file being C:\secret.txt. This path is transferred to \??\C:\secret.txt before getting to the kernel, as I explained earlier. Under the custom root directory object of the container, the system accesses ??, which is a reference to GLOBAL??. The system searches for a symbolic link C: under the GLOBAL?? directory and indeed finds such a symbolic link. At this point, the path is parsed to the target of said symbolic link, which in this case is \Device\VhdHardDisk{a36fab63-7f29-4e03-897e-62a6f003674f}\secret.txt. The kernel now proceeds to open said VhdHardDisk{...} device, but instead of searching this device under the Device folder in the root directory object of the host, it searches this device under the custom root directory object of the container and finds the virtual device of the container's file system.

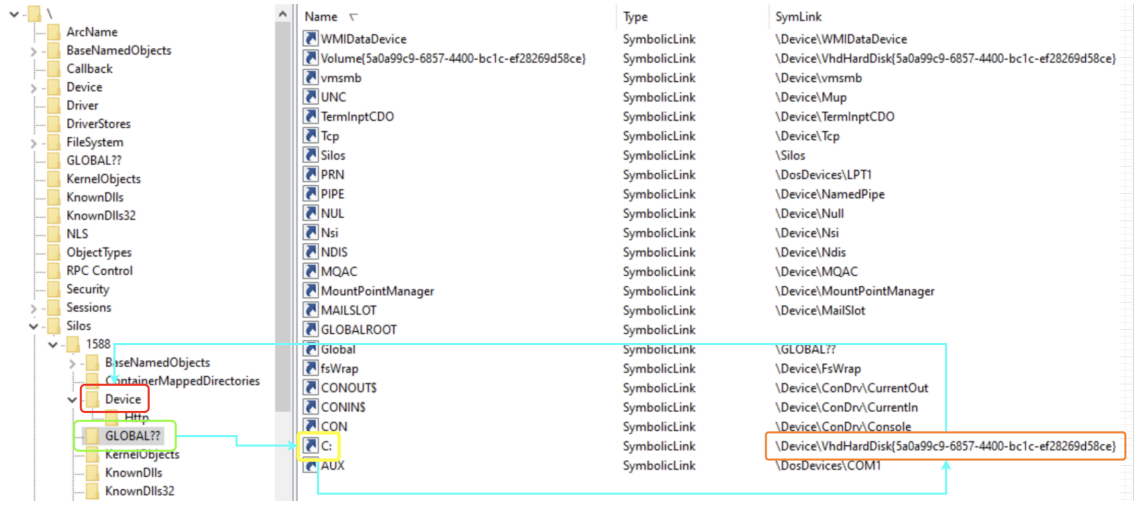

Figure 3. WinObj showing how a path is parsed under the root directory object

But something wasn't right. When I browsed the Device folder under \Silos\1588\ I was expecting to find an object named VhdHardDisk{...} pointing to an actual device, but instead there was a symbolic link with the same name pointing to \Device\VhdHardDisk{...}. What was going on? How does Windows get to the actual device? At this point, I started researching the symbolic link lookup subject until I found a single line in slides from a talk by security researchers Alex Ionescu (CrowdStrike) and James Forshaw (Google Project Zero) at Recon 2018 mentioning there is a flag for a "global" symbolic link. I proceeded to reverse the relevant functions in order to find where this flag might be checked.

I eventually found a branch in ObpLookupObjectName that looked promising:

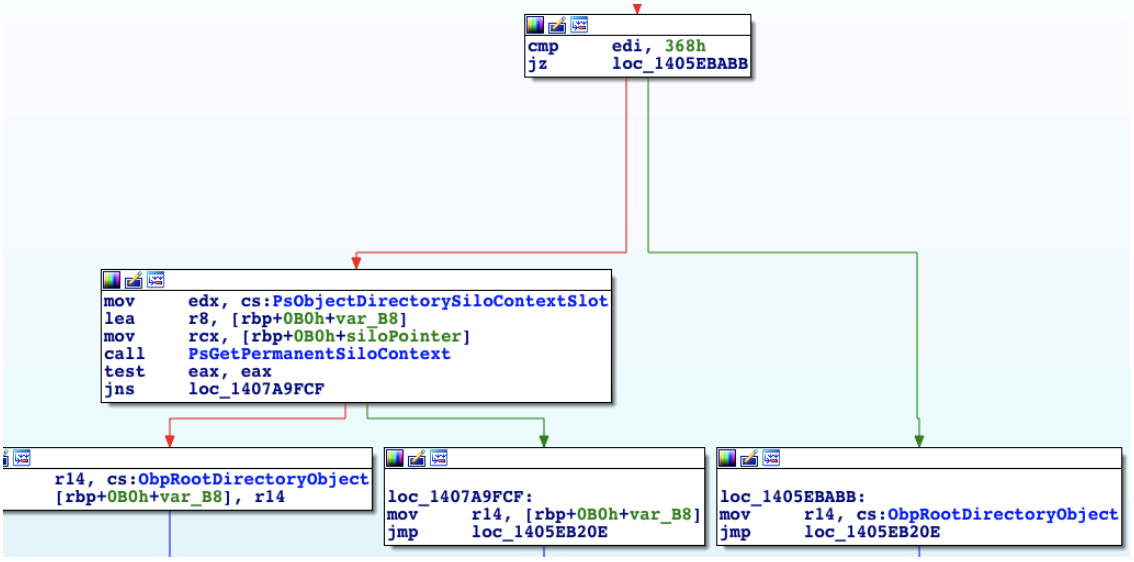

Figure 4. A branch in IDA that looked promising

The register edi holds the return value of ObpParseSymbolicLinkEx, so I searched this value – 368h – and found out it stands for STATUS_REPARSE_GLOBAL. So if ObpParseSymbolicLinkEx returns STATUS_REPARSE_GLOBAL, the object manager opens the file under ObpRootDirectoryObject, which is the regular root directory object, instead of getting the root directory of the Silo.

The Problem

At this point, I was certain I understood this behavior. I thought that creating a global symbolic link requires some special permission only system processes have. At the creation time of the container, the creating process has these special permissions and can create global symbolic links for the container to use, but no process inside the container can do that. The creating process controls what the global symbolic link points to and uses it only to access some special devices like the VhdHardDisk, so there is no real problem. It turned out, that was only partially true.

The Real Problem

I started searching for the value 368h that represents STATUS_REPARSE_GLOBAL in kernel code. After some work with IDA and WinDbg I ended up in the function ObpParseSymbolicLinkEx, which led me to find the relevant flag in the symbolic link object is at offset 28h (Object + 0x28). I placed a breakpoint in NtCreateSymbolicLinkObject, which is the function that creates a new symbolic link, and proceeded to create a new container using Docker. This raised many breaks for every creation of a new symbolic link for the container. This led me to the creation of the actual \Silos\1588\Device\VhdHardDisk{a36fab63-7f29-4e03-897e-62a6f003674f} object.

A reminder: This was the symbolic link object that behaved like a global symbolic link. I ended up putting an access breakpoint on the symbolic link object at offset 28h. Success! Right after the creation of the symbolic link, another function tried to modify the memory where I placed the breakpoint. The function was NtSetInformationSymbolicLink. This function seemed to get a handle to a symbolic link, open the relevant object and change things in it.

Luckily, this also got a wrapper function with the same name in ntdll, so we can easily call it from user mode. I reverse engineered this function and found a part of the code that checks for Tcb privilege in it. Tcb stands for Trusted Computing Base and its privileges description is, “Act as part of the operating system.”

I reversed ObpParseSymbolicLinkEx just enough to understand under what conditions it returns STATUS_REPARSE_GLOBAL as well as the exact parameters NtSetInformationSymbolicLink requires in order to change a symbolic link to make it global. These parameters are deliberately omitted from this post to make it harder for attackers to create an exploit.

Exploitation Plan

Knowing that I may be able to enable this global flag with Tcb privileges, and that it may allow for a container escape, I came up with the following plan to escape a container’s file system:

1. Create a symbolic link for the host’s C: drive.
2. Gain Tcb privileges.
3. Make said symbolic link global.
4. Access files on the host’s file system.

The only part missing from my plan was how to accomplish step two. We don’t have Tcb privileges in the container, do we? Well, our container processes do not have Tcb privileges by default. However, there is a special process in Windows containers called CExecSvc. This process is in charge of many aspects of the container execution, including communication between the host and the container. It also has Tcb privileges, so if a container process could execute code through CExecSvc, it would run with Tcb privileges, and the plan could unfold.

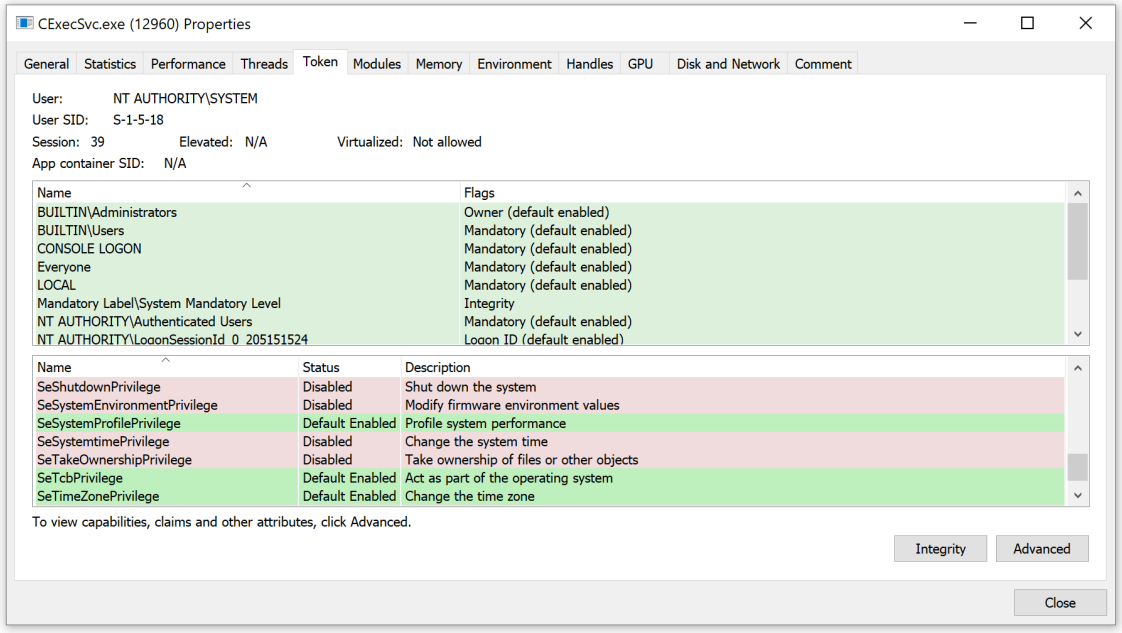

Figure 5. ProcessHacker showing CExecSvc has SeTcbPrivilege

Execution

I chose to do a simple DLL injection to CExecSvc, which included the attack logic. This worked well, and I was immediately able to gain access to the host's file system. Because CExecSvc is a system process, I gained full, unbounded access to the entire host file system, exactly as any other system process has.

Azure Kubernetes Service (AKS)

Azure Kubernetes Service (AKS) is a managed container orchestration service, based on the open-source Kubernetes system, which is available on Microsoft Azure Public Cloud. An organization can use AKS to deploy, scale and manage Docker containers and container-based applications across a cluster of container hosts.

AKS uses Windows Server Containers for each pod, meaning every single Kubernetes cluster that has a Windows node is vulnerable to this escape.

Not only that, but once an attacker gains access to one of the Windows nodes, it is easy to spread to the rest of the cluster.

The following image shows that the Windows node has everything we need in order to control the rest of the cluster. This displays the situation after we managed to access the host (in this case, the node) from the container (in this case, the pod).

```
azureuser@aksnpwin000000 C:\k>dir
 Volume in drive C is Windows
 Volume Serial Number is 764B-ED9D

 Directory of C:\k

03/17/2020  08:24 AM    <DIR>          .
03/17/2020  08:24 AM    <DIR>          ..
03/12/2020  08:42 AM             3,673 azure-vnet-ipam.json
03/15/2020  12:54 PM               255 azure-vnet-telemetry.log
03/15/2020  12:54 PM         5,244,122 azure-vnet-telemetry.log.1
03/10/2020  11:01 AM         5,243,655 azure-vnet-telemetry.log.2
03/12/2020  08:42 AM             2,456 azure-vnet.json
03/12/2020  08:42 AM           167,893 azure-vnet.log
03/05/2020  07:58 AM               809 azure.json
03/05/2020  07:59 AM    <DIR>          azurecni
03/12/2020  08:42 AM               645 azuremetadata.json
03/05/2020  07:58 AM             1,722 ca.crt
03/05/2020  07:58 AM    <DIR>          cni
03/05/2020  07:58 AM             9,506 config
03/05/2020  07:59 AM    <DIR>          debug
12/11/2019  12:04 PM            14,733 hns.psm1
12/11/2019  12:04 PM        40,086,016 kube-proxy.exe
12/11/2019  12:04 PM        43,478,016 kubectl.exe
```

Figure 6. Everything we need inside the Windows node

From here, one can just use *kubectl* to control the rest of the cluster.

```
azureuser@aksnpwin000000 C:\k>kubectl get node --kubeconfig=config
NAME                                STATUS   ROLES   AGE   VERSION
aks-nodepool1-34420028-vmss000000   Ready    agent   12d   v1.15.7
aks-nodepool1-34420028-vmss000001   Ready    agent   12d   v1.15.7
aksnpwin000000                      Ready    agent   12d   v1.15.7
```

Figure 7. Using kubectl from **inside** the node

Conclusion

In this post, I have demonstrated a complete technique to escalate privileges and escape Windows Server Containers. Users should follow Microsoft's guidance recommending not to run Windows Server Containers and strictly use Hyper-V containers for anything that relies on containerization as a security boundary. Any process running in Windows Server Containers should be assumed to be with the same privileges as admin on the host. In case you are running applications in Windows Server Containers that need to be secured, we recommend moving these applications to Hyper-V containers.

I would like to thank Alex Ionescu and James Forshaw for advising me with this research.

Palo Alto Networks Prisma™ Cloud protects customers from having their containers compromised. Prisma Cloud Compute also provides a compliance feature called Trusted Images that allows restricting users to run only known and signed images. By using this feature, customers can further reduce the attack surface by preventing execution of malicious images.
