Running Exploit As Protected Process Light From Userland

tastypepperoni.medium.com/running-exploit-as-protected-process-ligh-from-userland-f4c7dfe63387

pepperoni August 2, 2022

Overview

This blog reviews the recently patched(Windows 10 21H2 10.0.19044.1826 (24 July 2022 update)) vulnerability in Protected Process Light, which enables us to run any code as the highest level of protection, meaning that the exploit will have full access over any other Protected Process Light and anti-malware services won't be able to monitor it(Since they run with the lower protection of AntiMalware).

Windows 8.1 and all Windows 10 builds, released before the 24 July 2022 update can still be affected by this vulnerability.

The tool(RunAsWinTcb) introduced in this blog is heavily inspired by the project PPLDump, which exploits the same vulnerability and can be used to dump the memory of any Protected Process Light(ex: lsass.exe, if it is configured to run as PPL).

RunAsWinTcb is written entirely in GO and extends the functionality of PPLDump from just dumping a PPL memory to general Code Execution on the signer level of WinTcb-Light. The tool operates entirely in user mode, so there is no need for additional kernel drivers, and the danger of BSOD(Blue Screen Of Death) is eliminated.

It requires Administrator privileges and can be used for persistence and defense-evasion purposes, And the best part: Windows Defender has no problem with it.

What Is The KnownDLLs

KnownDLLs is a directory (\KnownDlls) that contains the "Known DLLs". "Known DLLs" are the most commonly loaded DLLs by Windows processes. "Known DLLs" are pre-loaded in memory and the \KnownDlls directory acts as a cache store for them.

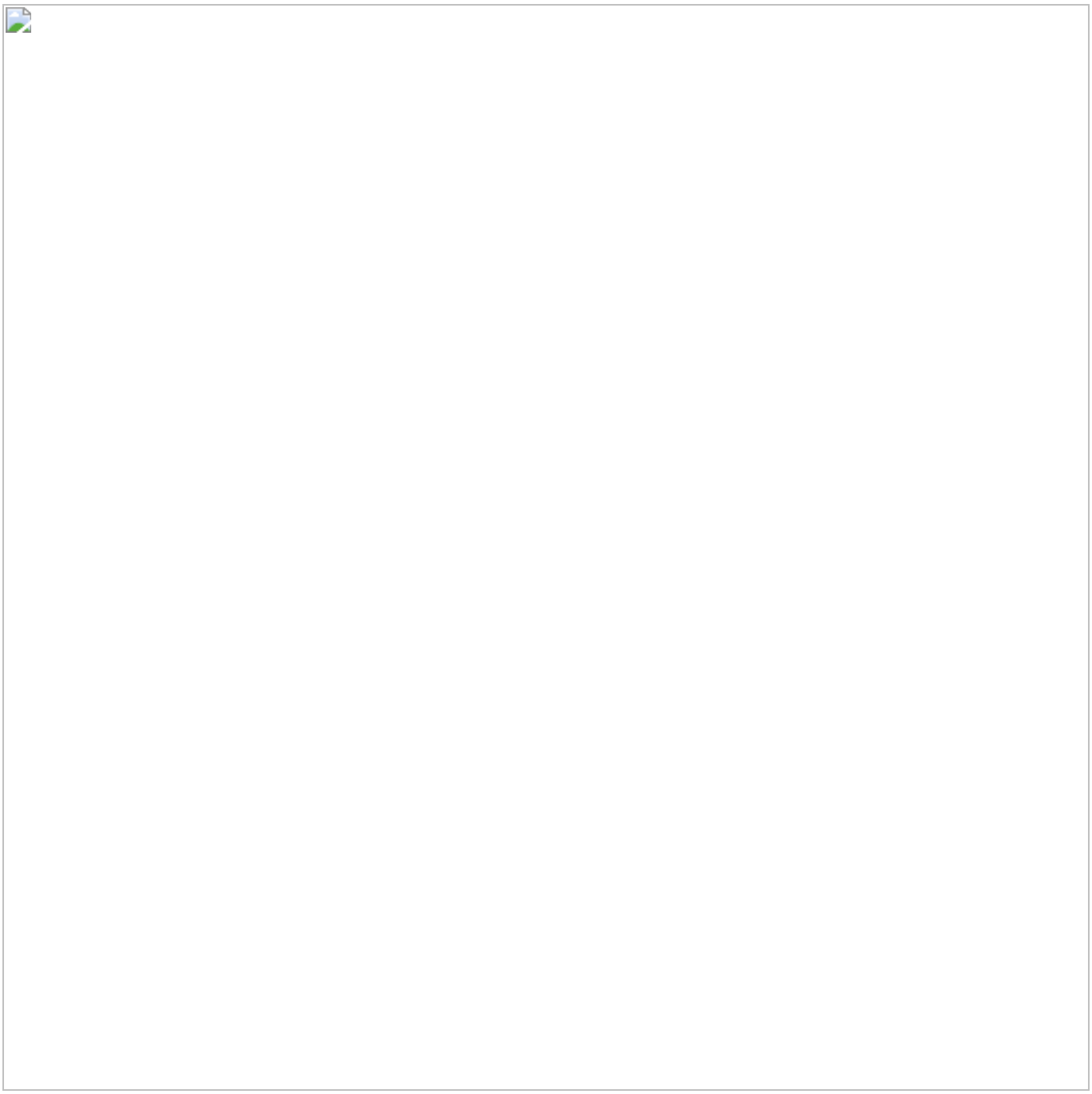
DLLs in \KnownDlls directory are represented in a form of Sections, meaning that they're already mapped into the memory.

The main purpose of \KnownDlls directory is to increase the overall performance of Windows processes.

Windows processes(with exceptions) look for DLLs in \KnownDlls directory first, before starting to search it on the disk. If they're able to find it in \KnownDlls directory, they load it directly from there, meaning that they don't need to map it in memory themselves(since

they’re already mapped).

A full list of the “Known DLLs” can be viewed using WinObj:



List of the “Known DLLs”

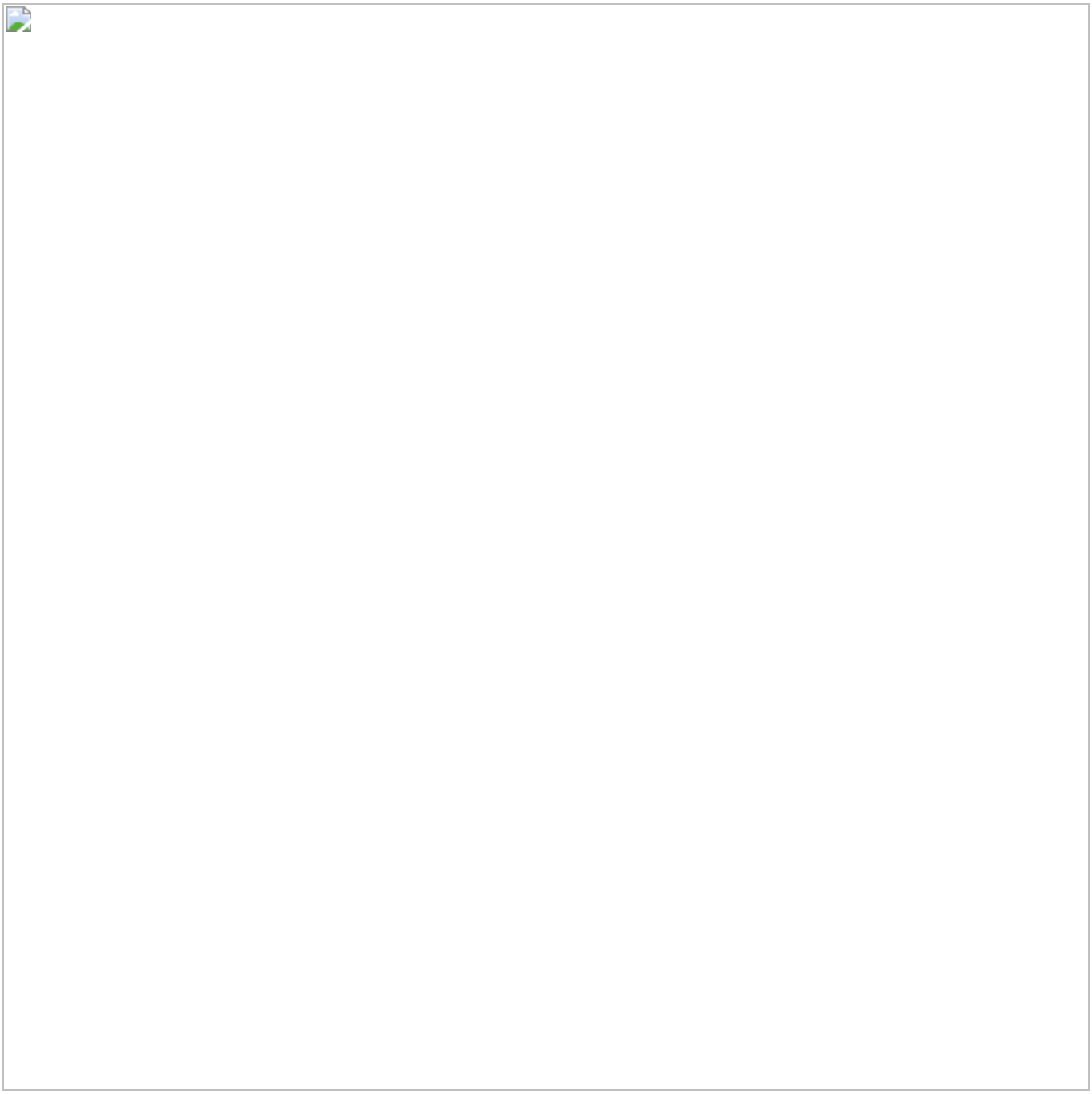
What is Protected Process Light(PPL)

Protected Process Light(PPL) is a security mechanism introduced by Microsoft in Windows 8.1. It ensures that the operating system only loads trusted services and processes by enforcing them to have a valid internal or external signature that meets the Windows

requirements. It also restricts access to processes and is used as a self-defense mechanism by anti-malware and windows native processes.

Key things to note about PPLs to better understand the exploit:

1. PPLs have different levels of protection, which are determined by signer level:

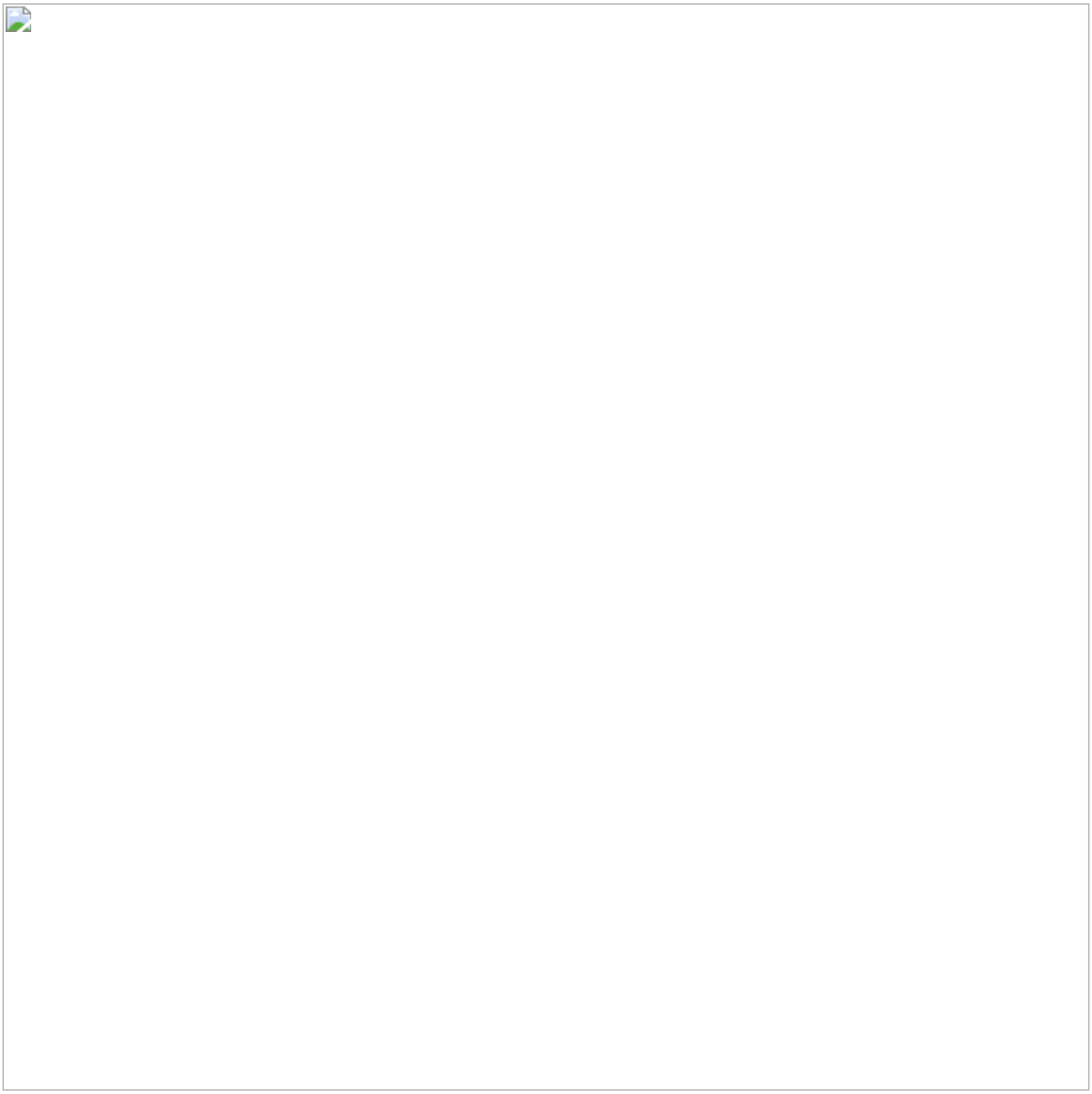


Protection levels of Protected Process Light

Higher or same-value signer processes have access to the same or lower ones, but not vice versa.

2. PPLs verify the digital signature of a resource(ex: DLL) when the file is mapped, i.e. when a Section is created.

3. Handle of PPLs can be opened by processes with lower privileges, but only with an access level of PROCESS_QUERY_LIMITED_INFORMATION (0x1000)



The handle of PPLs with the access level PROCESS_ALL_ACCESS can be opened with the same or higher-level signer process.

The Plan

If we're able to add an arbitrary entry into \KnownDlls, with the name of a DLL that is loaded by an executable, privileged to run as a PPL, we will be able to execute an executable as PPL and make it load the section value of arbitrary entry from \KnownDlls, instead of the real DLL from the disk, as it originally does(The \KnownDlls directory will be searched first. Basically, we're doing DLL hijacking here).

The section value will contain any custom DLL, potentially, an exploit code.

Since PPLs verify the signature of the DLL when they're mapped in the memory, the digital signature of our custom DLL won't be checked, because they're already mapped in \KnownDLLs directory.

Since our DLL will be loaded in PPL, it will benefit from the same privileges as the host process.

Essentially, we're hijacking a DLL and bypassing the signature verification process. As a result, we get a code execution with a protection level of PPL.

The Exploit

The centerpiece of the exploit is Windows API function: DefineDosDevice.

```
BOOL DefineDosDeviceW(  [in]           DWORD   dwFlags,  [in]           LPCWSTR
lpDeviceName,  [in, optional] LPCWSTR lpTargetPath);
```

As Microsoft describes it, it's used to "Defines, redefines, or deletes MS-DOS device names."

MS-DOS device names

MS-DOS device names are essentially symbolic links in the object manager with a name of the form \DosDevices*DosDeviceName (*ex: the C drive has the name \DosDevices\C:*).* So, this function allows you to map an actual "Device" to a "DOS Device".

This is actually what happens behind the scenes when you plug an external USB device into the computer.

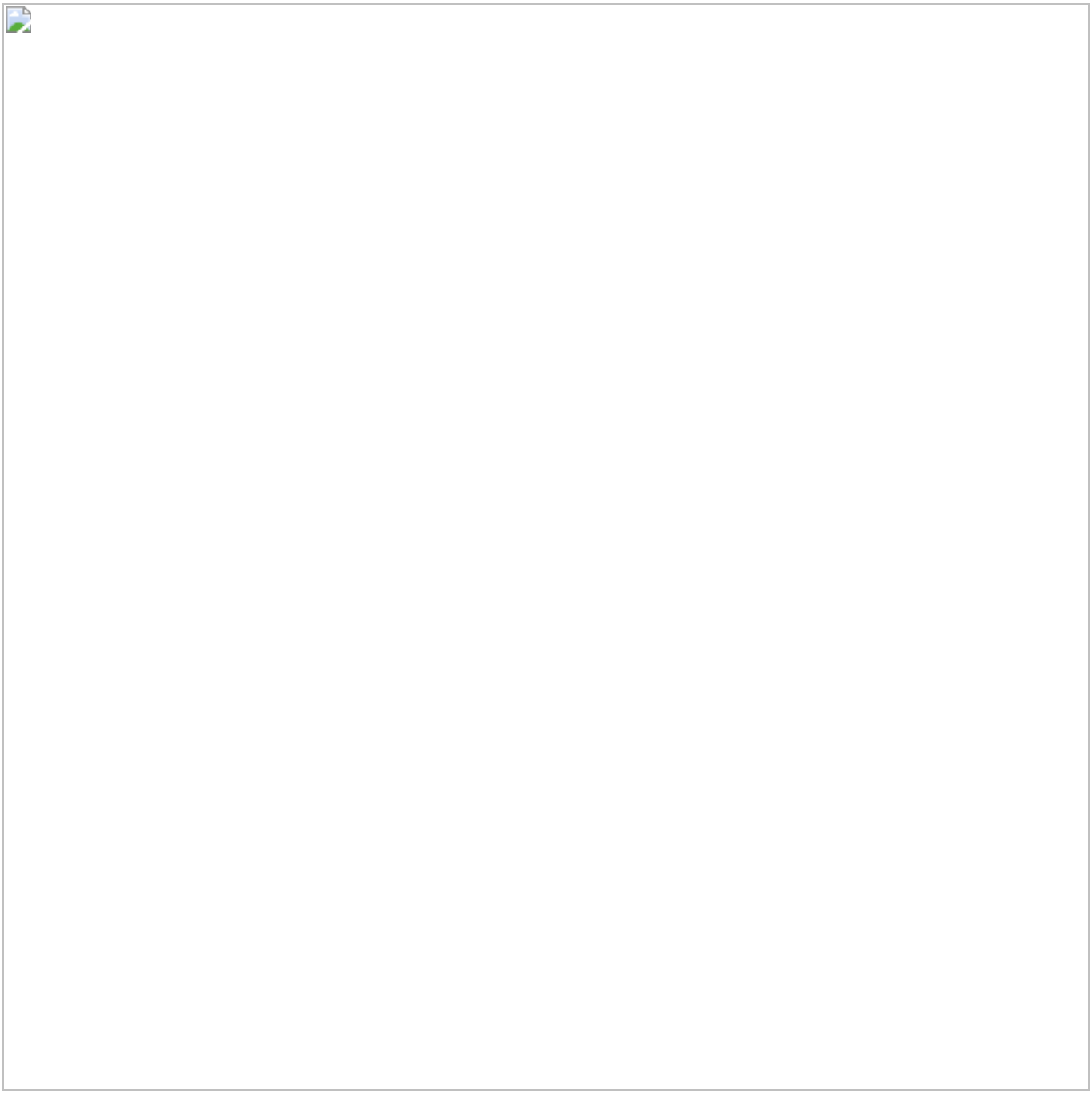
Symbolic link of external USB

It gets dirve letter, which is **"E:"** in this case, as MS-DOS device name and its target path is NT path of the device, **"\Device\HarddiskVolume6"** in this case.

But as Microsoft describes MS-DOS device names, they're represented in the form of **\DosDevices** *,* so it can't be just a single drive letter, "E:".

The explanation for this odd behavior is that the “\DosDevices\” part is implicit. Functions that are used to work with Dos Devices automatically prepend the device name with **“\??\”**. So **“E:”** is not just **“E:”**, its actually **“\??\E:”**

What is **“\??\”** ?, WinObj will answer that question.

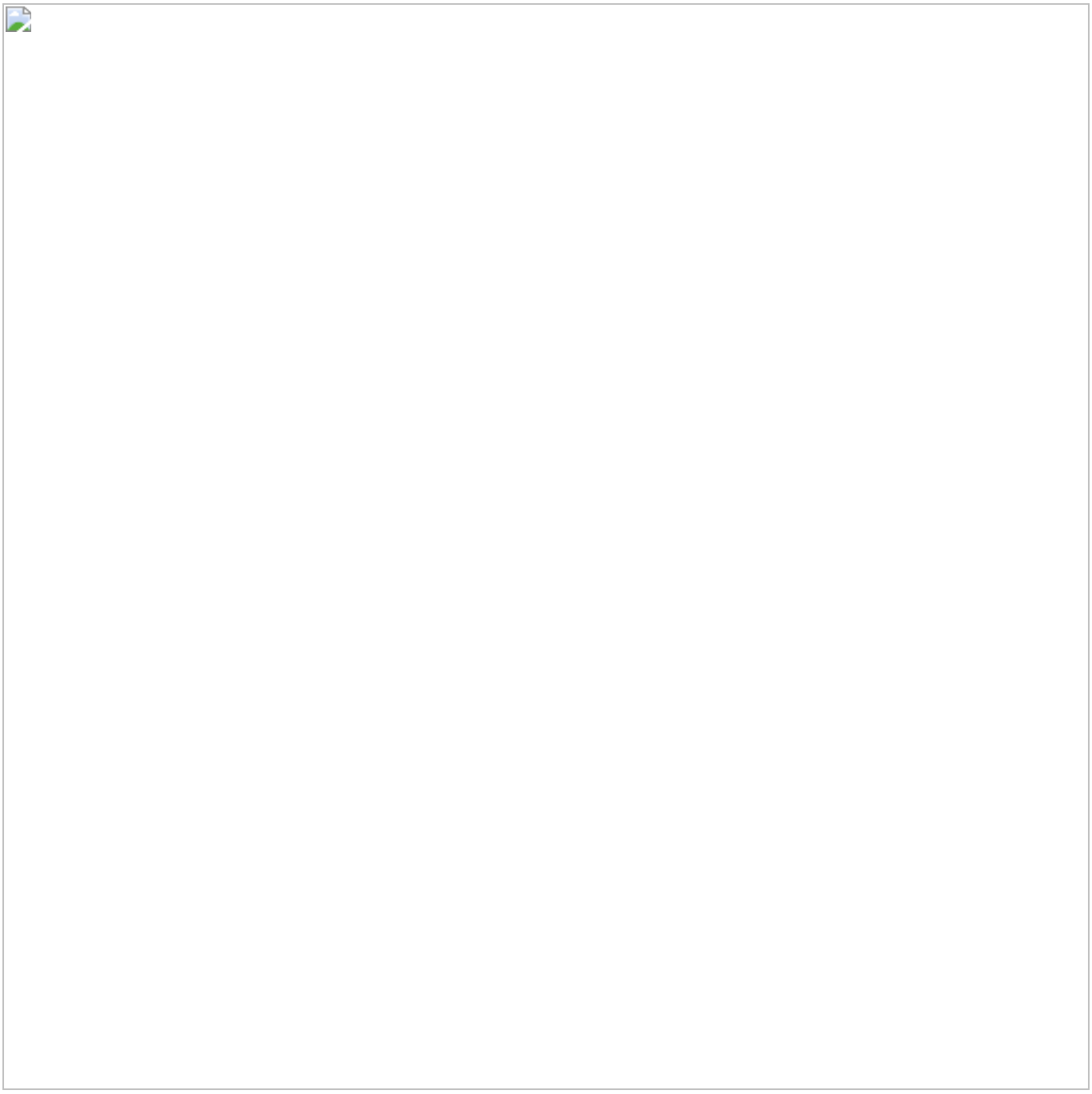


DosDevices Symboli Link

We can see that **“\??”** is the target path of the symbolic link “**DosDevices”**, into the root directory. This means that **“\??”** is translated to **“\DosDevices”**(“\” is added because it’s placed in the root directory)

So **“\??\E:”** is translated to **“\DosDevices\E:”** and we get the version that fits in the description of Microsoft(\DosDevices*DosDeviceName*)

All “translation” steps:

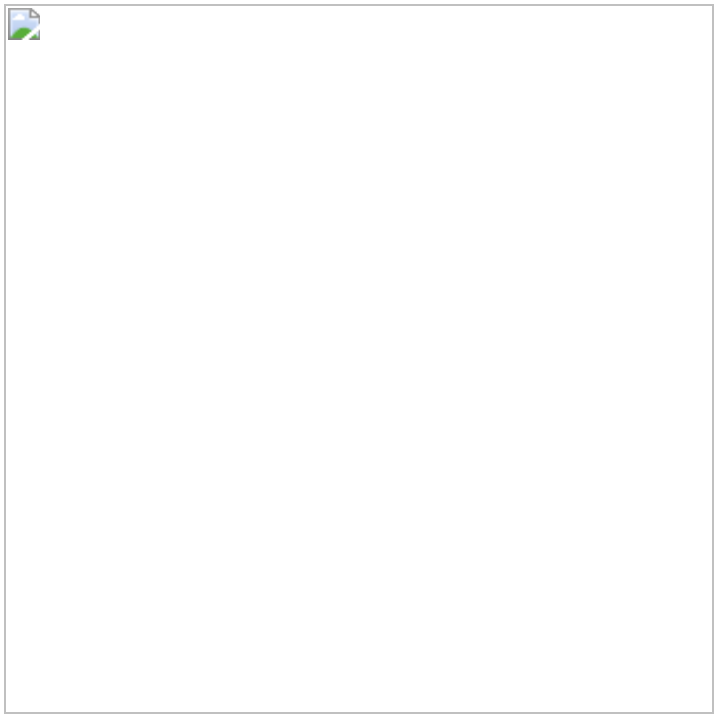


Local DOS Device Directories

Every user has a Local DOS Device directory, which is represented by previously reviewed **“\??\”**.

“\??\” refers to different locations in the Object Manager, depending on the current user’s context.

Concretely, **\??** refers to the full path **\Sessions\0\DosDevices\00000000-XXXXXXXX, where XXXXXXXX** is the user’s login authentication ID.



Sessions DosDevices

But, there’s ONE exception, **“NT AUTHORITY\SYSTEM”.**

For **“nt authority\system”, “\??”** refers to **\GLOBAL??.**

Let’s take a look at the examples for both cases to better understand the working principle of it:

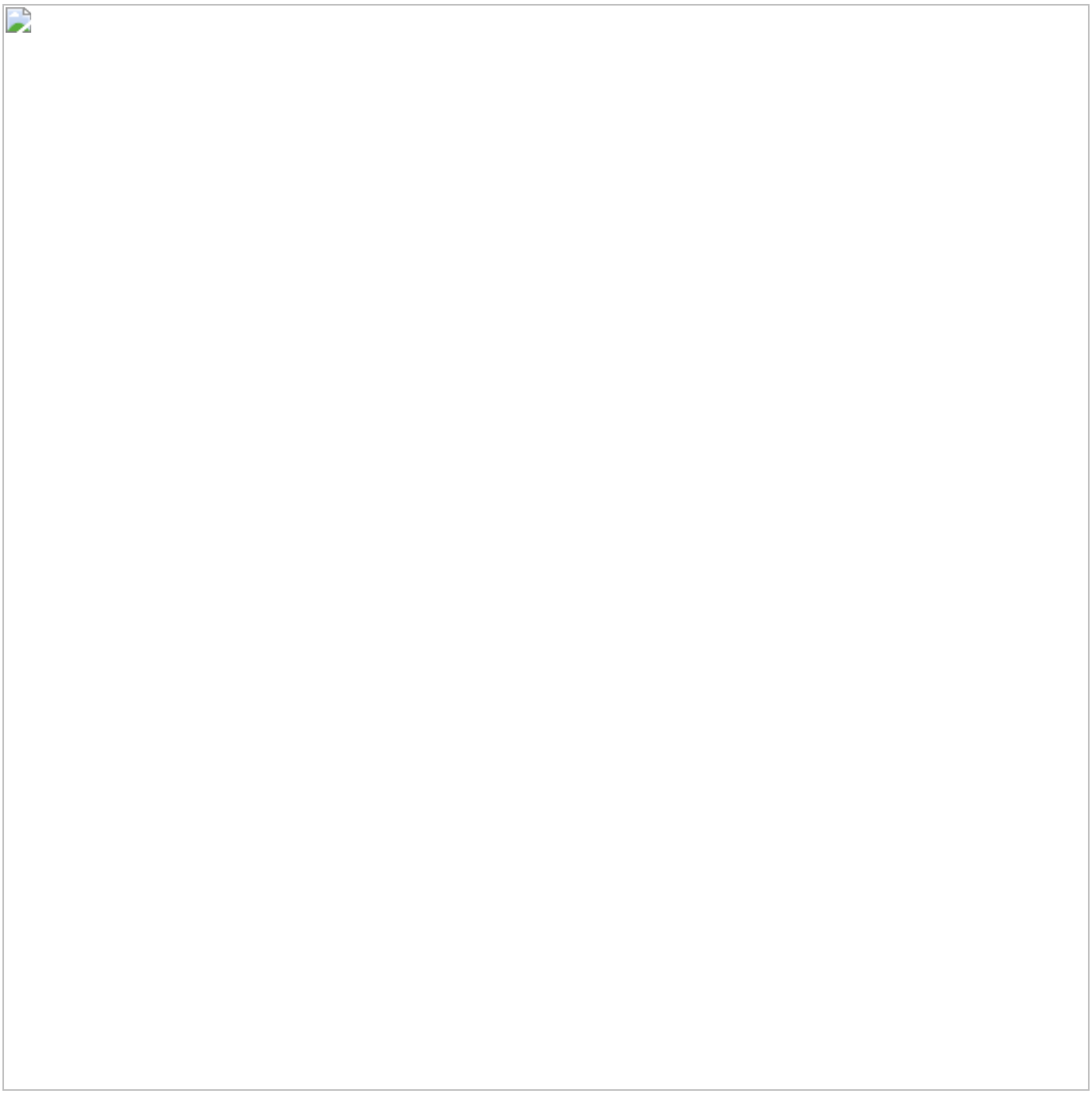
1. When the USB device is plugged into the computer, it is mounted by nt authority\system” user, meaning that its drive letter, for example, will be prepented by and become . After that, itself will be translated to because for “nt authority\system”,

We can verify that in WinObj:

Symbolic link is indeed created in “\GLOBAL??” directory.

2. When the SMB share is mounted, the mapping of the drive letter is done as the logged-on user, meaning that its drive letter, **“F:”** for example, will be prepented by **“\??\”** and become **“\??\F:”**. After that, **“\??\F:”** itself will be translated to **“\Sessions\0\DosDevices\00000000-XXXXXXXX\F:”**, because for logged-on users, **“\??” refers NT path of the current session.**

We can verify it in WinObj:



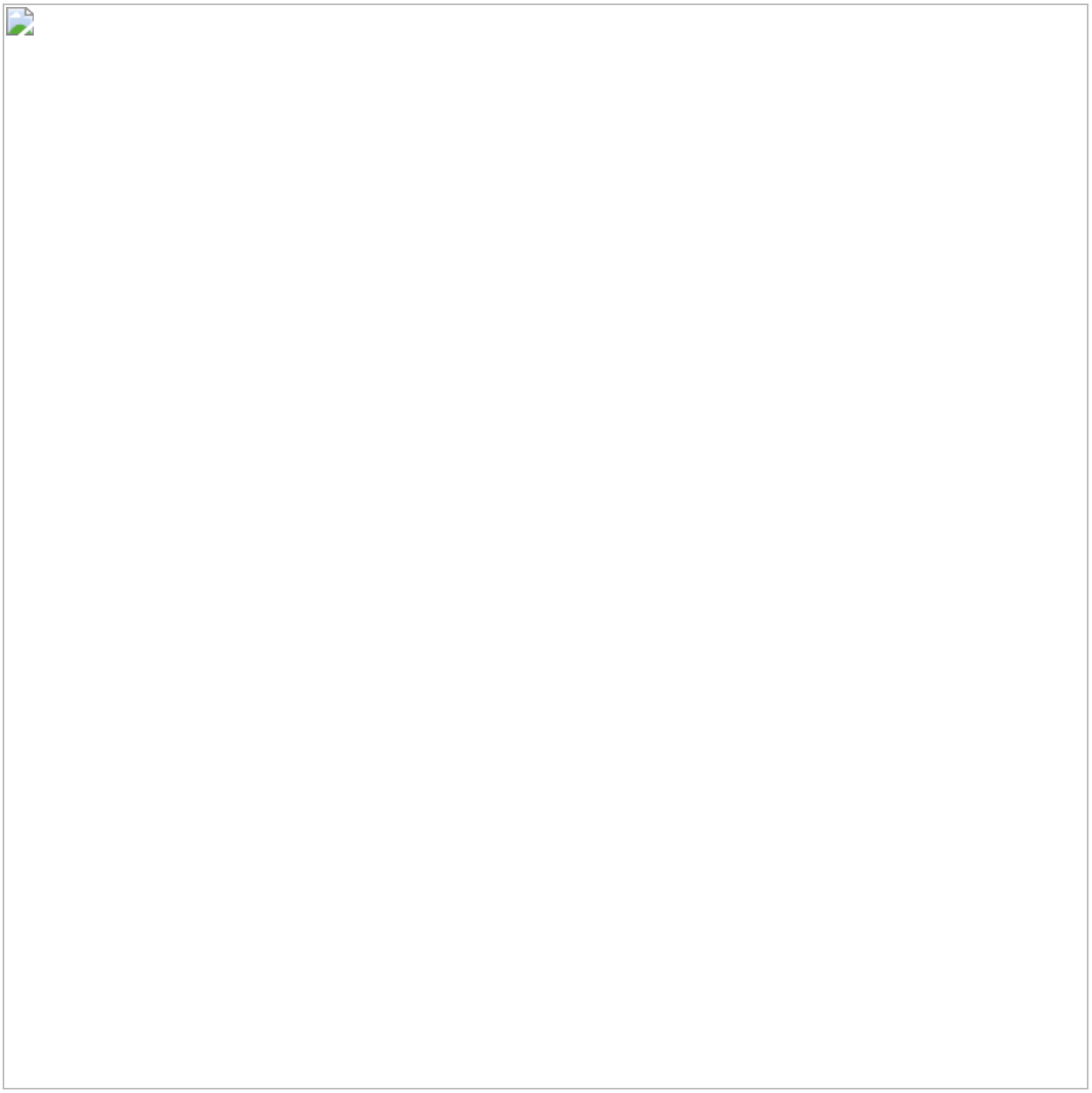
The symbolic link is indeed created in the current logged-on user’s session directory.

Problem In Making Permanent Kernel Object(Not Really)

Low-privileged users can only make Temporary kernel objects, which are removed once all their handles have been closed. To solve this problem, the object must be marked as “Permanent“.

Making objects “Permanent” requires the special privilege, “SeCreatePermanentPrivilege”. So the mapping process should be done by the process, with the “SeCreatePermanentPrivilege” privilege.

As it turns out, “DefineDosDevice” is not implemented in the caller’s process, it’s just a wrapper for an RPC method inside the current session’s CSRSS service, specifically the method *BaseSrvDefineDosDevice* inside BASESRV.DLL.



csrss.exe

csrss.exe is the Protected Process Ligh with a signer type of WinTCB-Light and it has the “SeCreatePermanentPrivilege” privilege enabled by default. It runs as “nt authority\system” user.

RPC(Remote Procedure Call) is a software communication protocol that one program can use to request a service from a program located on another computer on a network without having to understand the network's details. RPC is used to call other processes on the remote systems like a local system.

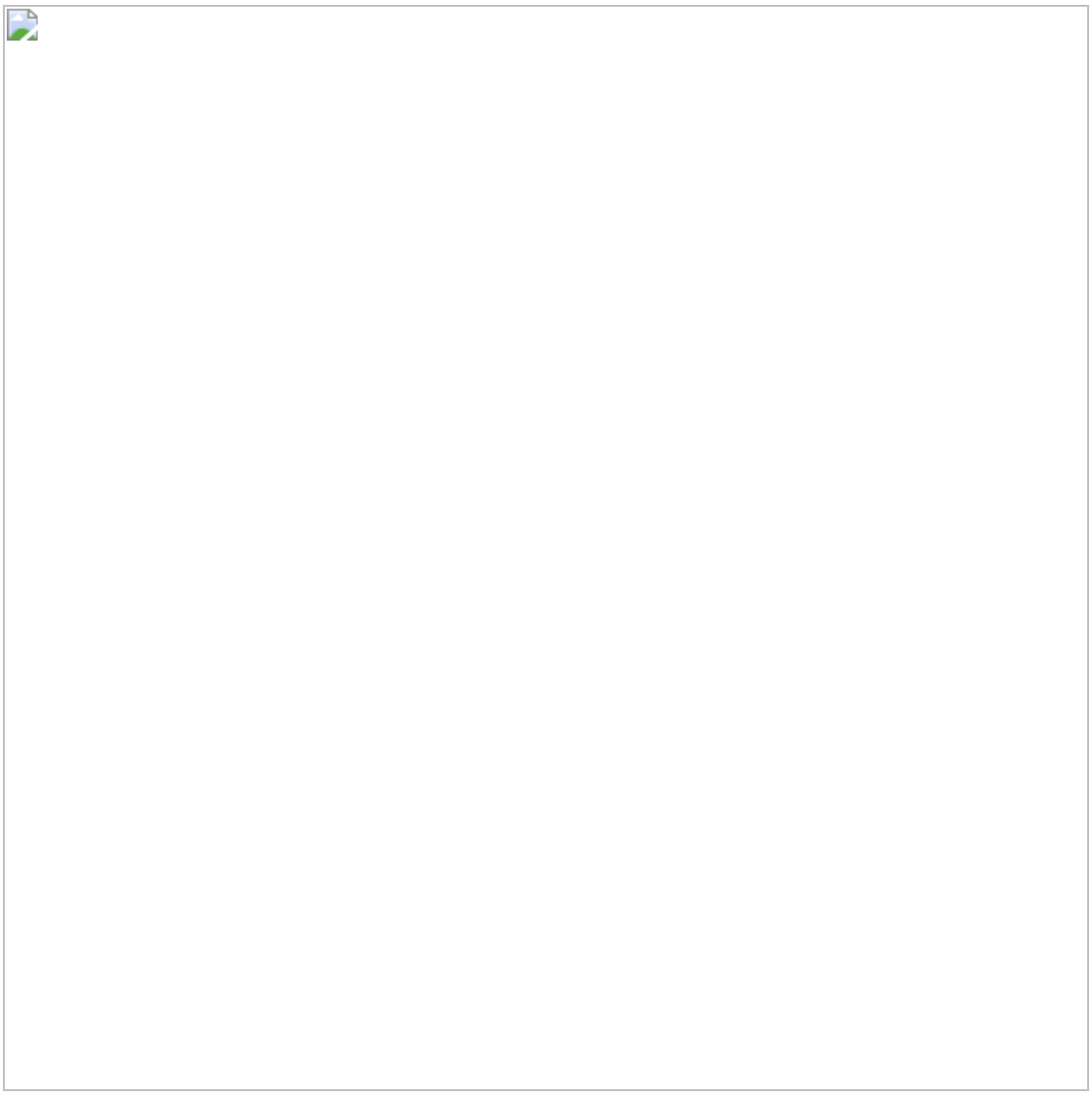
Essentially, when "DefineDosDevice" is called, the RPC method call is made to csrss.exe, and the mapping is done by csrss.exe, instead of the caller. Since csrss.exe has the "SeCreatePermanentPrivilege" privilege, it can mark objects as "Permanent".

What is even more interesting is that the value of lpDeviceName in the "DefineDosDevices" function is not sanitized. This means that we are not bound to provide a drive letter such as E:, we can do a lot more. We will take advantage of this lack of sanitization to trick the csrss.exe into creating an arbitrary symbolic link in an arbitrary location such as \KnownDlls.

DefineDosDevice

As stated in a previous part, "DefineDosDevices" makes an RPC method call to csrss.exe, which itself calls the *BaseSrvDefineDosDevice* function.

Here's a diagram, created by itm4n(author of PPLDump), describing how *BaseSrvDefineDosDevice* works:



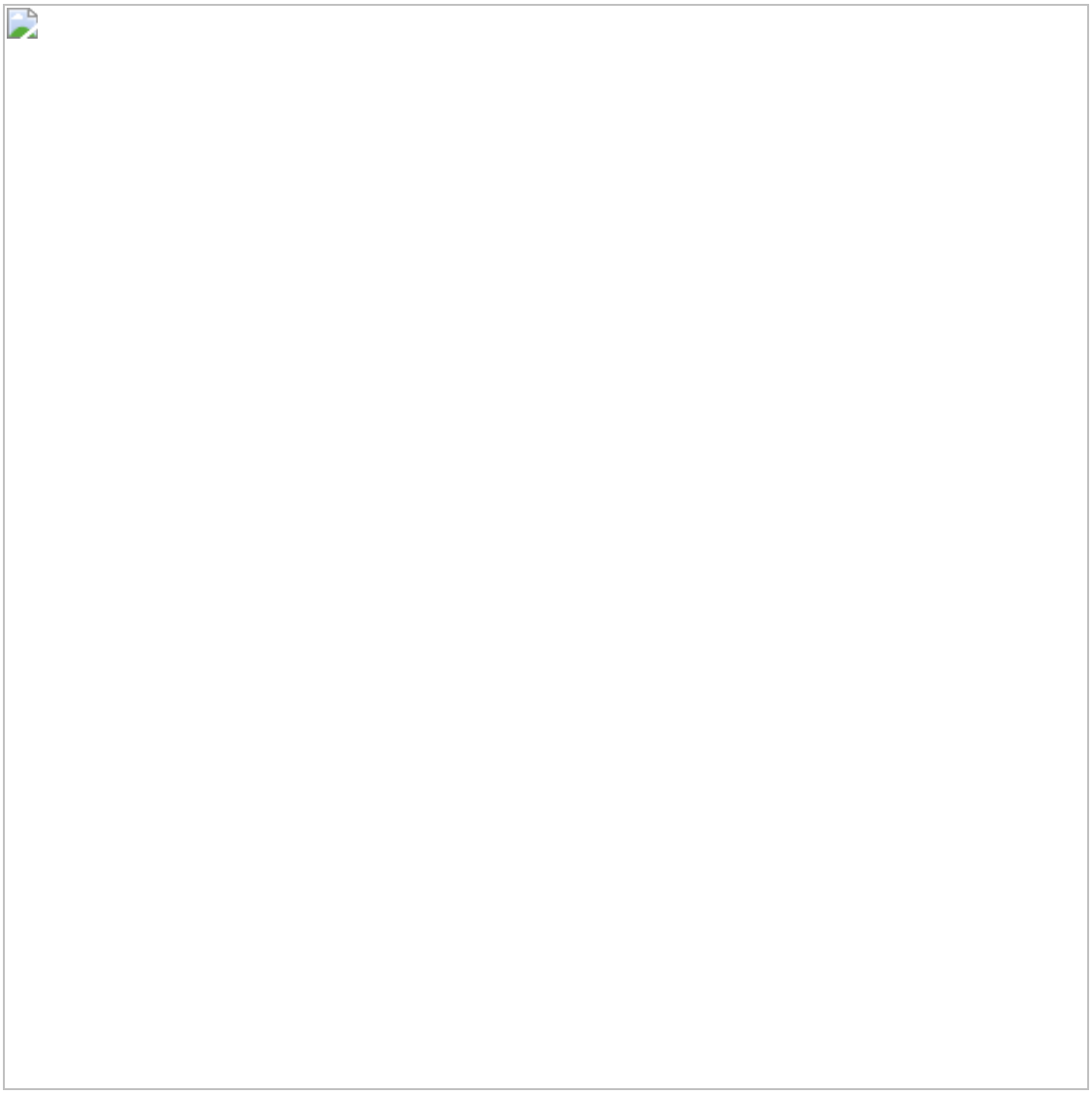
Overview of

Parts highlighted in red is the path we need to make csrss.exe follow.

1. csrss.exe impersonates the RPC client and tries to open \??\DEVICE_NAME. The impersonation step is essential, because, as stated previously, “\??” has different values, based on the user’s context. So csrss.exe needs to impersonate a client, that provided the path, to build the correct path, that it was requested to create. The main purpose of “OpenSymlink” is to delete the symlink if it already exists.
2. If csrss.exe is able to open symlink, it will check if the symbolic link is global. This is simply done by checking if the path of the object starts with “\GLOBAL??\”. If so, impersonation of RPC client(original caller) is disabled and the rest of the instructions, including “NtCreateSymbolicLinkObject()”, are done on the behalf of the user that runs csrss.exe, “nt authority\system”
3. Finally, if “NtCreateSymbolicLinkObject()” succeeds, the symlink will be marked as “Permanent”.

The Vulnerability

If we take a closer look at the same diagram, we can notice that there's a certain path from the "OpenSymlink" step to the "CreateSymlink" step, that includes disabling RPC client impersonation:



If we're able to make csrss.exe follow that path, "OpenSymlink" and "CreateSymlink" instructions will be executed on the behalf of two completely different users(RPC Client and SYSTEM), and, therefore, although the same path is used in both cases, their argument "\?? \DEVICE_NAME" will refer to two completely different locations. (Since "\??" refers to different locations, based on the user's context).

"OpenSymlink" step will always be executed as RPC client, i.e user. But we can make csrss.exe disable impersonation and "CreateSymlink" will be executed as SYSTEM.

The Main Challenge

In order to exploit the behavior of csrss.exe, we need to find the “device name” in “\??\DEVICE_NAME” that:

1. Resolves to a “global object”(starts with “\GLOBAL??\”), while RPC client impersonation is enabled, to make csrss.exe disable impersonation at the proper moment.

2. Resolves to DLL into KnownDLLs directory(ex: \KnownDlls\FOO.dll) while RPC client impersonation is disabled and instructions are executed as SYSTEM.

Solving Challenges

Solving in reverse order:

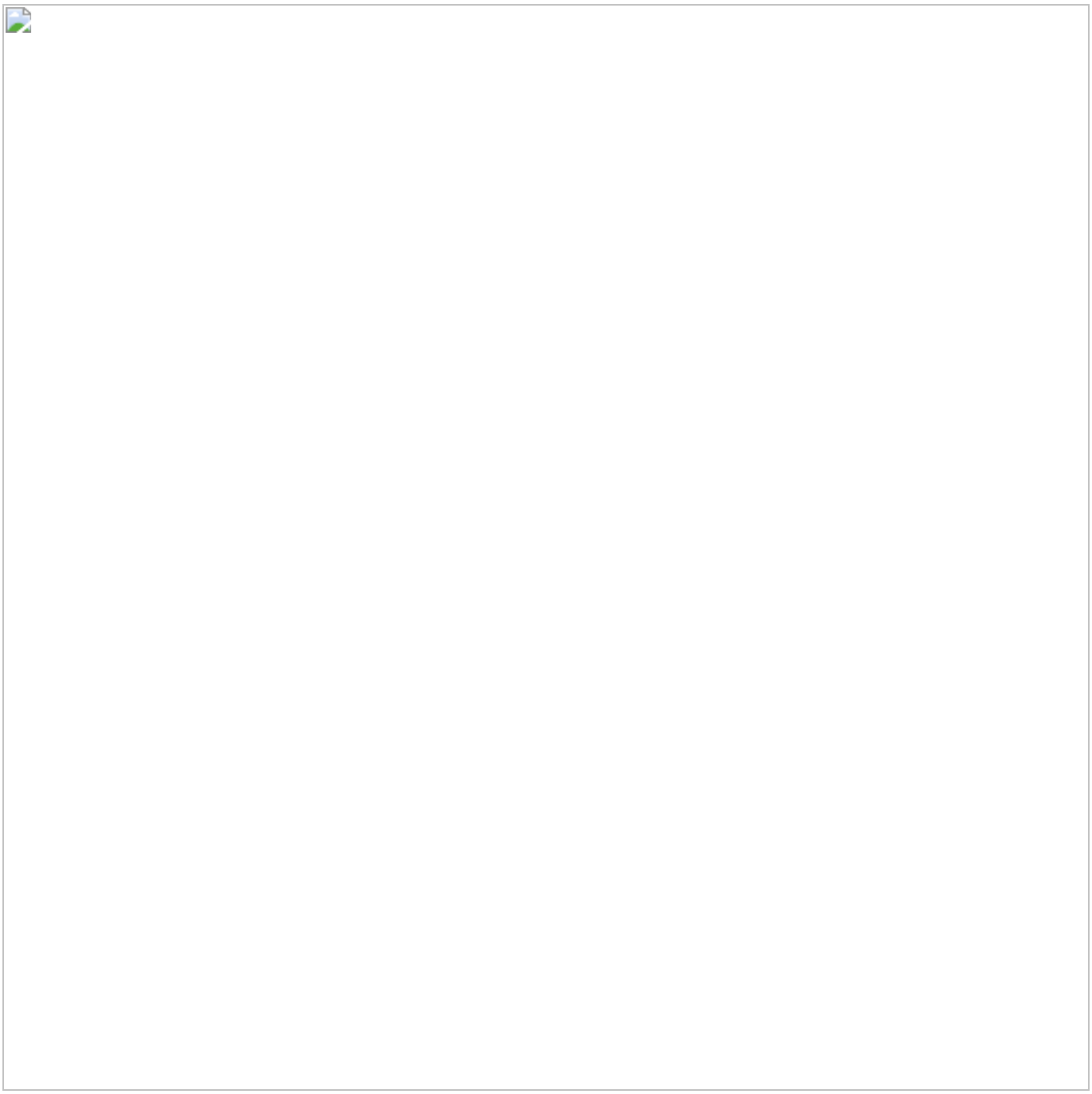
2. In this step, RPC client impersonation is disabled, so the instructions are executed as a “SYSTEM” user.

Main goal: Make “\??\DEVICE_NAME” resolve to “\KnownDlls\FOO.dll”

Local Dos Device directory of SYSTEM is **“\GLOBAL??”**, so **“\??”** will be converted into **“\GLOBAL??”**.

Our starting template: **“\??\DEVICE_NAME”** becomes “**\GLOBAL??\DEVICE_NAME”**

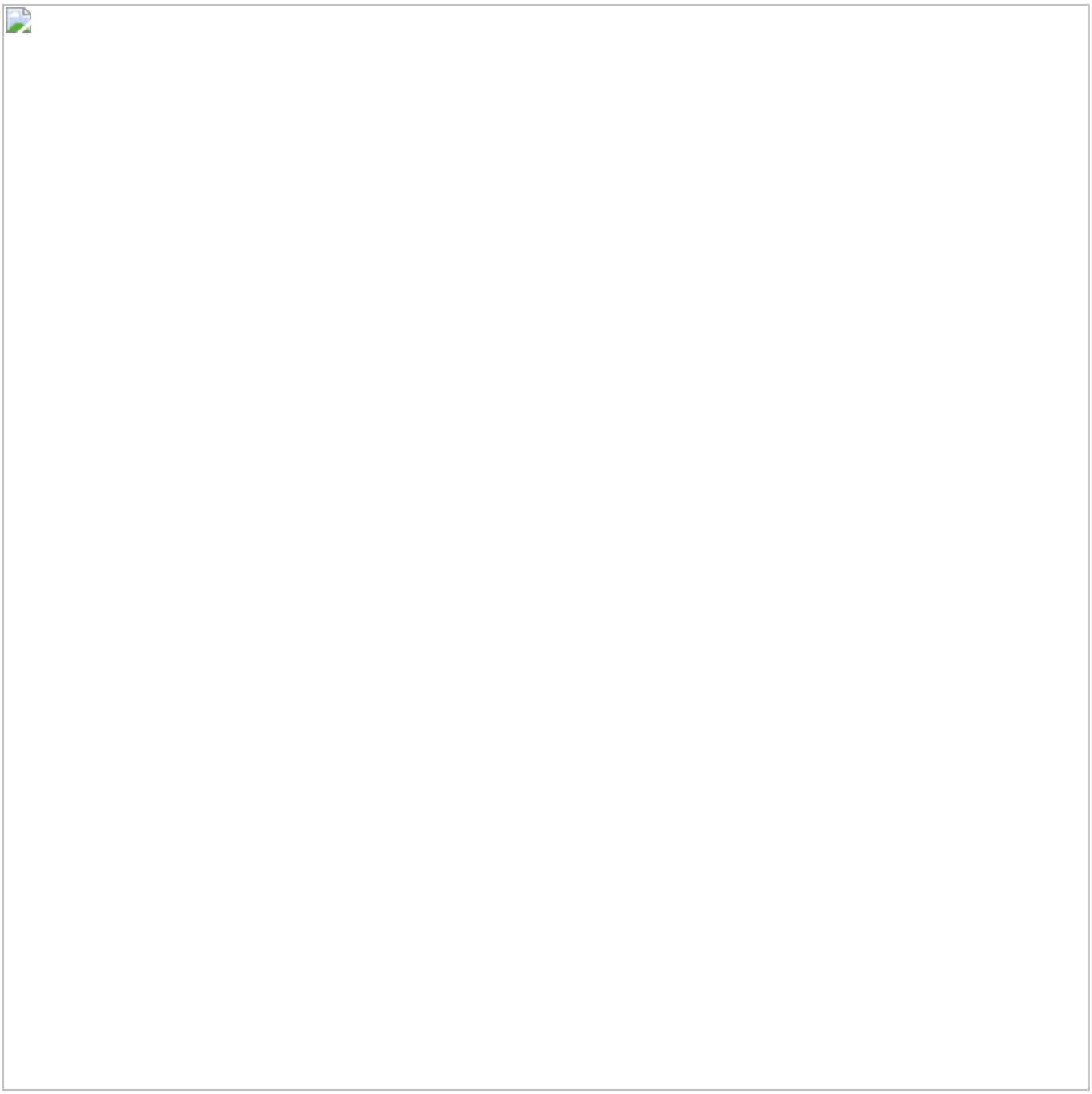
If we check the **“\GLOBAL??”** directory in WinObj, we can notice an object called **“GLOBALROOT”**, which is a symlink that points to an empty path:



GLOBALROOT

This means that **“\Global??\GLOBALROOT”** will resolve to **“\”**, the root path of Object Manager.

Implementing it into our template of **“\??\DEVICE_NAME”**, we get:



So, finally, **“\??\GLOBALROOT\KnownDll\FOO.dll”** resolves to **“\KnownDlls\FOO.dll”** while executing as “SYSTEM”.

Now we know that we should supply “GLOBALROOT\KnownDlls\FOO.dll” as the “device name” for DefineDosDevice(“\??\ will be prepended automatically).

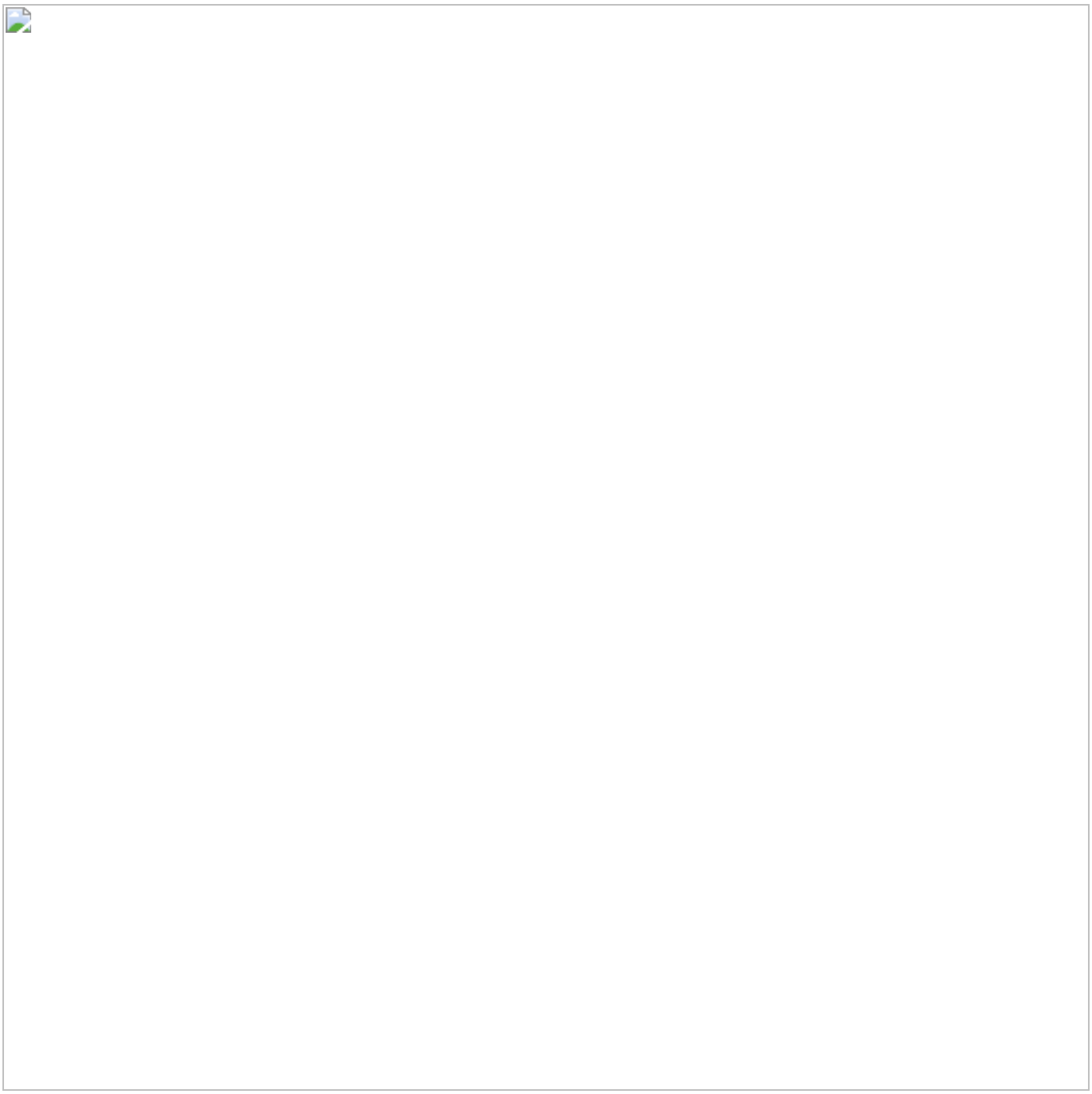
1. In this step, RPC client impersonation is still enabled, so the instructions are executed on the behalf of RPC client.

In the second challenge, we've built the path for the "SYSTEM" user, since both steps use the same path, we need to work with **"\??\GLOBALROOT\KnownDlls\FOO.DLL"** in this step too.

Main goal: Transform "\??\GLOBALROOT\KnownDlls\FOO.DLL" into "\GLOBAL??\KnownDlls\FOO.dll", for it to be considered as a "global object"(starts with "\GLOBAL??\"), make csrss.exe disable impersonation and pass execution to SYSTEM.

We know that since RPC client impersonation is enabled, "\??" will be resolved to the Local Dos Device directory of the RPC client(original caller of DefineDosDevice).

So the conversion process is done so:



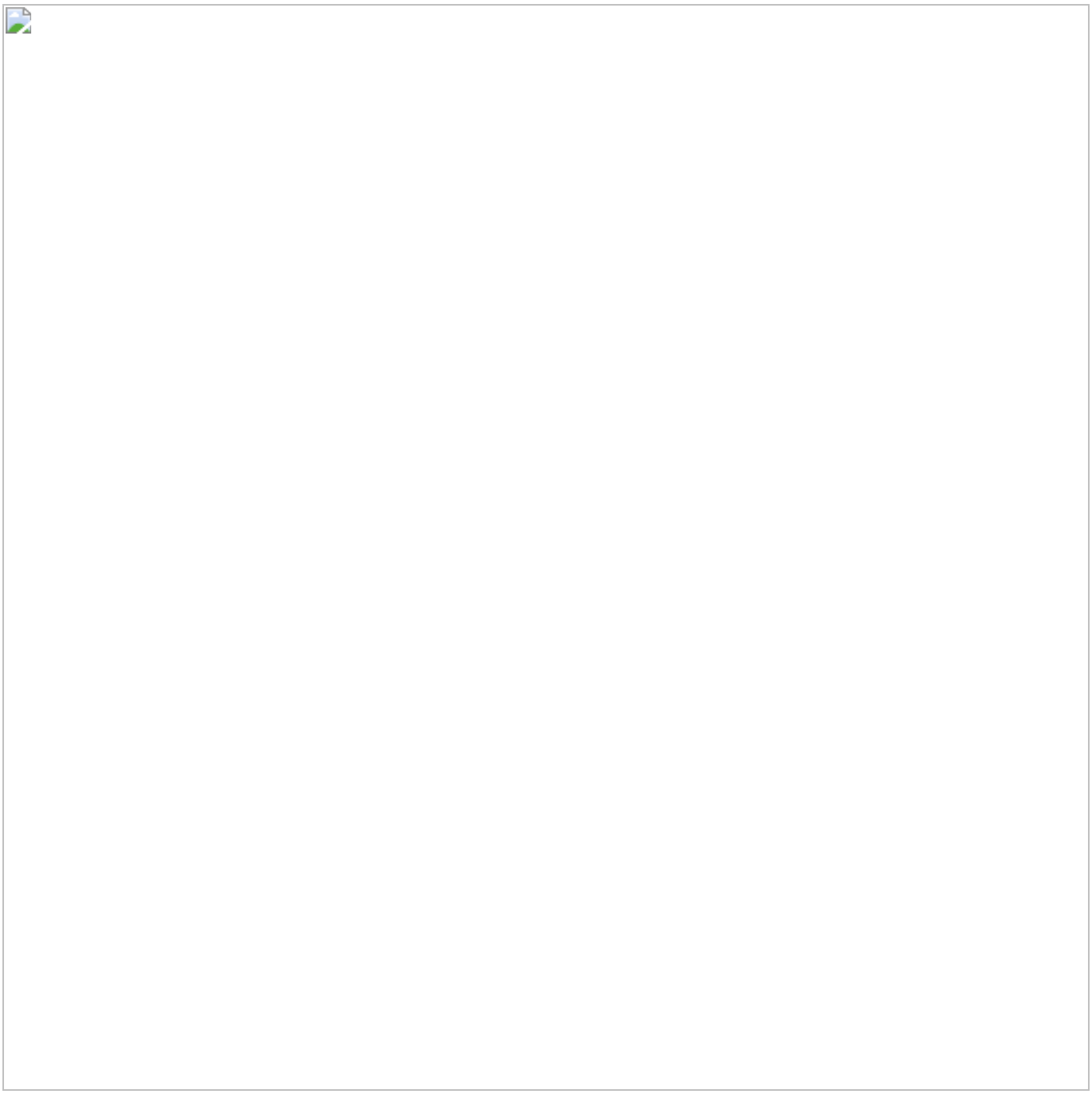
Objects in our Local Dos Device directory are controlled by us, meaning that we can create symbolic links in it.

So what we can do is just create a symbolic link **“\??\GLOBALROOT”** that points to **“\GLOBAL??”**.

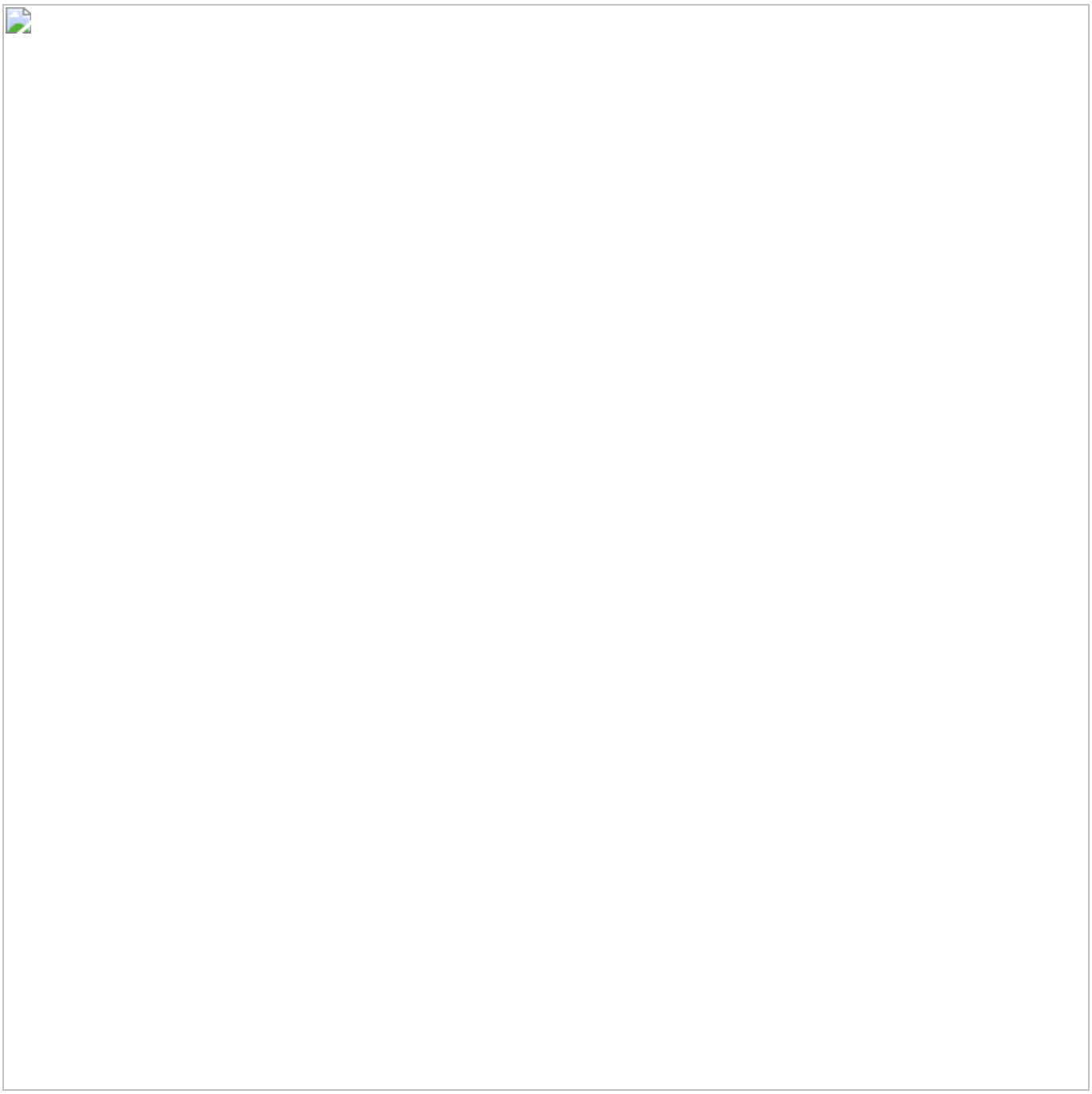
“\??” in **“\??\GLOBALROOT”** will be translated into our Local Dos Device directory and a symlink with the name of “GLOBALROOT” will be created in our Local Dos Device and will point to **“\GLOBAL??”**

Here’s what happens from the point of view of csrss.exe in this step:

1. It receives the path:

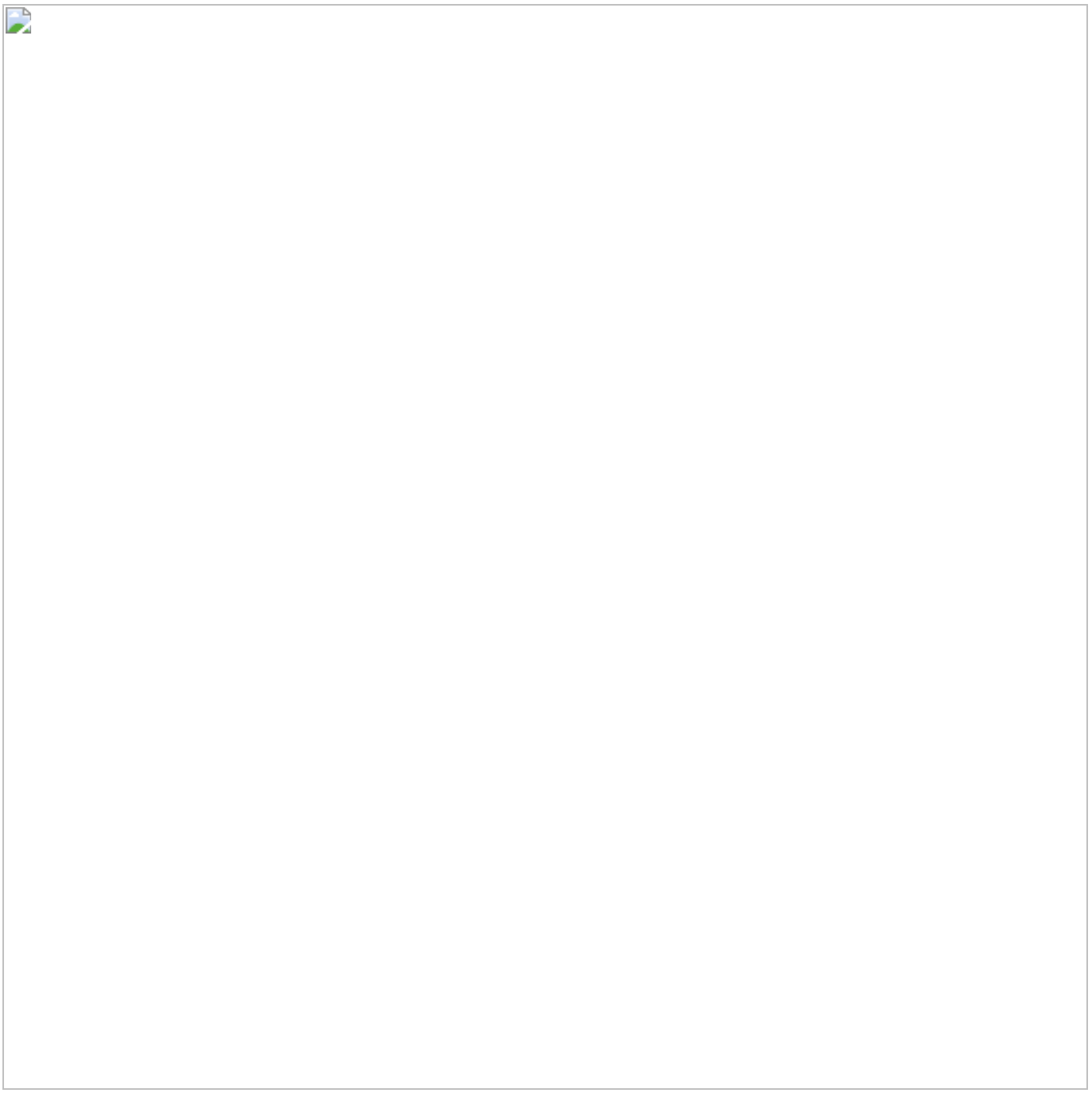


2. Translates “\??” to RPC client’s Local Dos Device directory:

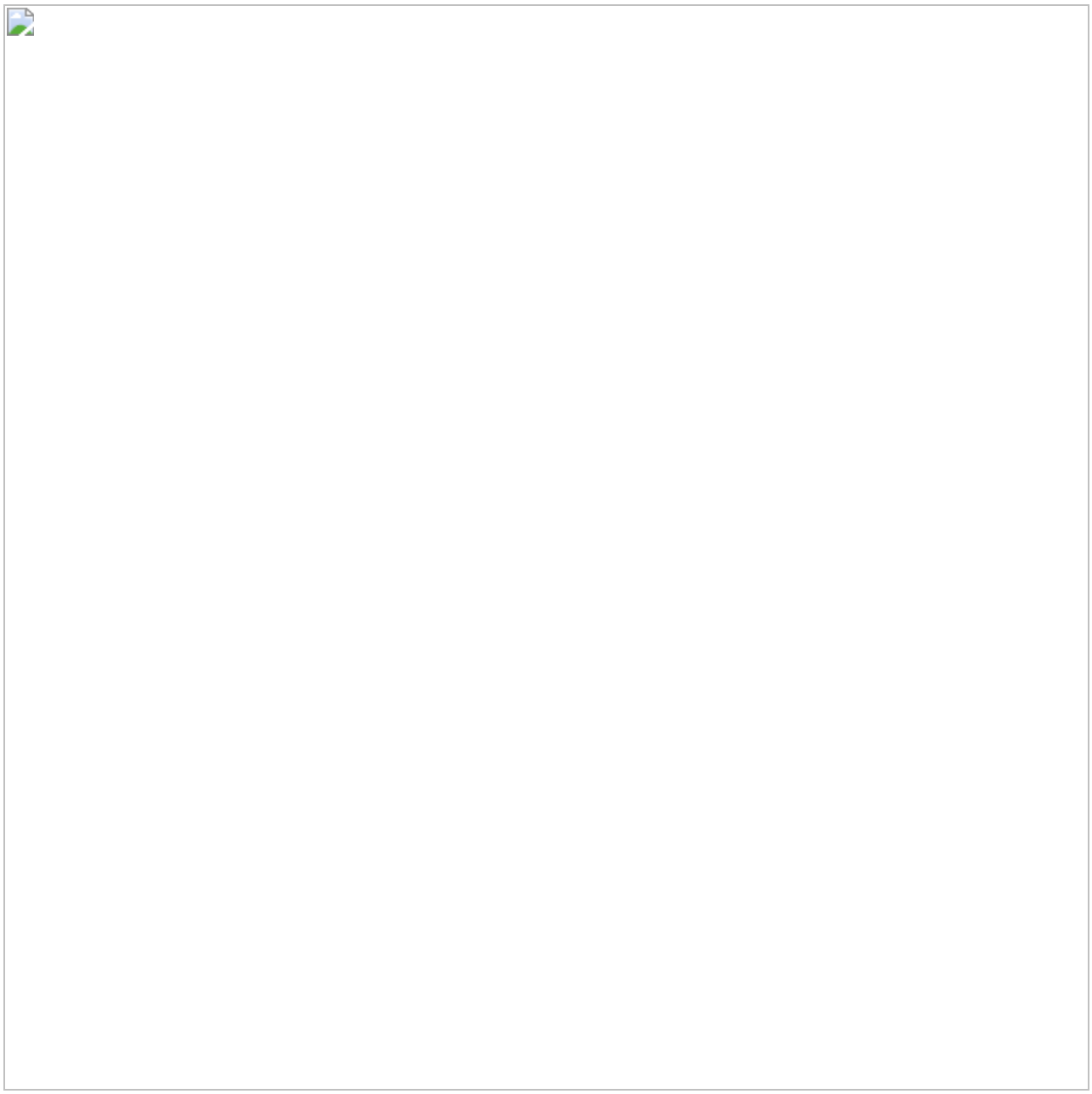


3. Uses our reccenlty created symlink called GLOBALROOT in our Local Dos Device directory.

So this:



After resolving symlink, becomes this:



4. Sees that it starts with “\GLOBAL??”, considers it as a “global object”, disables RPC client impersonation, and passes the rest of the execution to “nt authority\system”

One Last Thing

In order to make csrss.exe take the path, where the object is checked for being a “global object”, the object must first exist. Otherwise, the “OpenSymlink” step will just fail. So we need to make sure **“\GLOBAL??\KnownDlls\FOO.dll”** exists, before calling DefineDosDevices.

There’s a small issue here, objects and directories in “\GLOBAL??\” can only be created by the SYSTEM user. We can’t be running as SYSTEM, because the trick used against csrss.exe will fail. “OpenSymlink” and “CreateSymlink” will both be run as the same,

SYSTEM user(RPC client will be SYSTEM, csrss.exe is also SYSTEM).

This is not a big deal. We will need to just temporarily elevate to SYSTEM, create “\GLOBAL??\KnownDlls\” directory, and create a dummy symbolic link with the name of FOO.dll(It does not matter where that symlinks will point to, as its only purpose is to just exists for “OpenSymlink” step done by csrss.exe to be successful). Then just revert back to the current user and call DefineDosDevice as that user.

Choosing PPL and DLL To Hijack

Our goal is to execute code inside a PPL, ideally with the signer level of WinTcb(as it’s the highest signer level). On Windows 10, there are 4 binaries that can be executed with such level of protection: wininit.exe, services.exe, smss.exe, and csrss.exe.

services.exe is the perfect fit for our purposes.

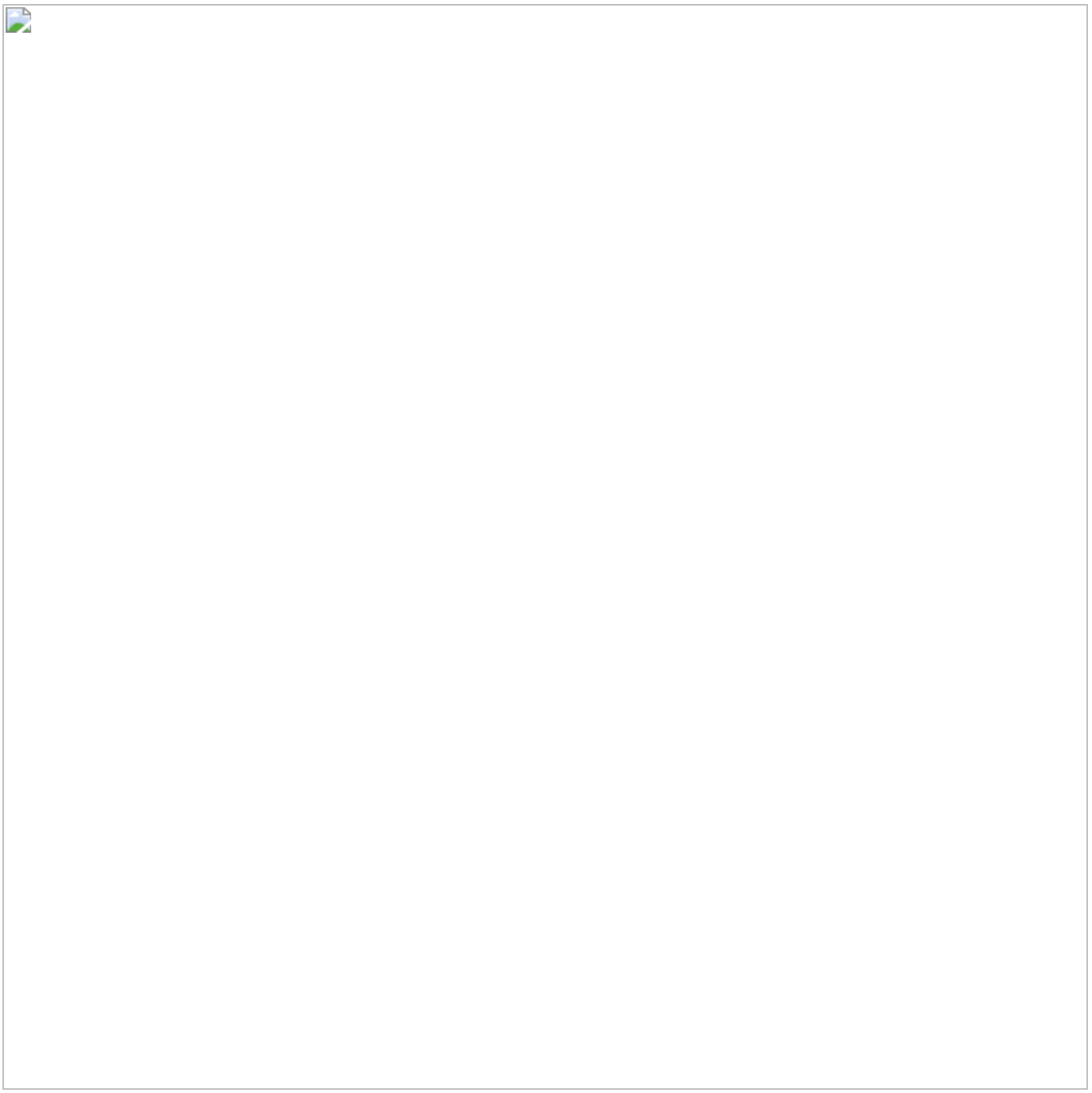
In Windows 10, services.exe loads “EventAggregation.dll”.

In Windows 8.1, services.exe loads “SspiCli.dll”

Criteria For The Exploit DLL

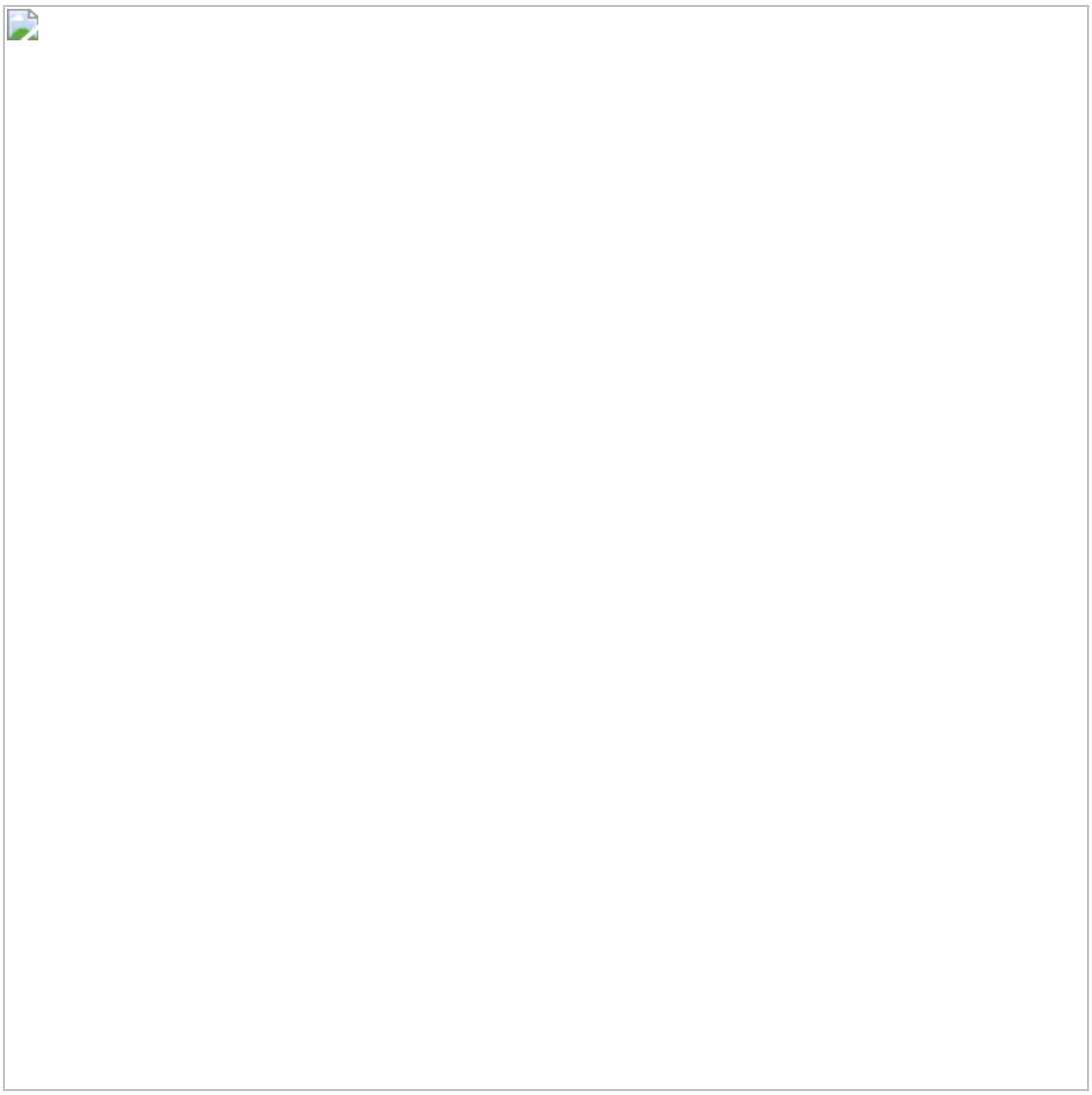
The exploit DLL should export the same functions as the hijacked DLL. Otherwise, it won’t be loaded.

In the case of “EventAggregation.dll”, these functions are:



EventAggregation.dll exports

In the case of “SspiCli.dll”, these functions are:



SspiCli.dll exports

The Full Exploit

1. Elevate to “nt authority\system” to get the privilege of creating objects and directories in “‘\GLOBAL??\”
2. Create the object directory in “\GLOBAL??\” called KnownDlls
3. Create a dummy symbolic link in the directory, created in step 2, with the name of EventAggregation.dll. The target path of the symlink can be anything.
4. Drop the system privileges and revert back to Administrator
5. Create a symbolic link in our Local Dos Device directory called “GLOBALROOT” and point it to “\GLOBAL??”
6. Call DefineDosDevices with the value “GLOBALROOT\KnownDlls\EventAggregation.dll” and the target path of any path, controlled by us. For example: “\KernelObjects\EventAggregation.dll”

Here’s what happens in csrss.exe after step 6:

1. It receives the “GLOBALROOT\KnownDlls\EventAggregation.dll” and prepends it with “\??”

Result: “\??\GLOBALROOT\KnownDlls\EventAggregation.dll”

2. Impersonates RPC client.

3. Converts “\??\” to the Local Dos Device directory of the RPC client.

Result:

\Sessions\0\DosDevices\00000000-XXXXXXXX\GLOBALROOT\KnownDlls\EventAggregation.dll

4. Resolves “GLOBALROOT” to the target path “\GLOBAL??”(because of the symlink, created in step 5 in the exploitation steps)

Result:

\GLOBAL??\KnownDlls\EventAggregation.dll

5. Succesfully opens it(since it was created by us in step 3 of exploitation steps)

6. Verifies that it starts with “\GLOBAL??” and considers it as a “global object”

7. Disables RPC client impersonation and passes execution to “nt authority\system”

8. Deletes the symlink

9. Creates symlink with the same path, passed to it in the 1st step.

“GLOBALROOT\KnownDlls\EventAggregation.dll”

10. Prepends it with “\??”

Result: “\??\GLOBALROOT\KnownDlls\EventAggregation.dll”

11. Converts “\??\” to the Local Dos Device directory of “nt authority\system”

Result: \GLOBAL??\GLOBALROOT\KnownDlls\EventAggregation.dll

12. Resolves “GLOBALROOT” to the target path “\”(pre-defined in Object Manager)

Result: “\KnownDlls\EventAggregation.dll”

13. Creates symbolic link “\KnownDlls\EventAggregation.dll” with the target path of “\KernelObjects\EventAggregation.dll” and marks it as permanent.

(At this point, we're able to create symlinks in "\KnownDlls\" directory with an arbitrary target path.)

Continuing the exploit:

7. Map the exploit DLL(create section) in the target path of the symbolic link, created in step 6 of the exploitation steps.("\KernelObjects\EventAggregation.dll")

8. Run services.exe as PPL

services.exe will:

1. Try to load EventAggregation.dll and look for it in the "\KnownDlls\" folder first.
2. Discovers "\KnownDlls\EventAggregation.dll" symbolic link(created by us in step 6 of exploitation steps) with an arbitrary target path, controlled by us.
3. Resolves "\KnownDlls\EventAggregation.dll" to "\KernelObjects\EventAggregation.dll"
4. Discovers a section of a DLL in "\KernelObjects\EventAggregation.dll"(section is created in step 7 of exploitation steps)
5. Since it is already mapped in memory, it will not check its digital signature of it and load it.

At this point, our exploit DLL is running with WinTcb rights.

RunAsWinTcb

As discussed above, the exploit contains two separate files.

1. An executable that is used to create a symlink in "\KnownDlls\" directory
2. DLL that will be loaded by services.exe

The Github repository of RunAsWinTcb contains the source code and already-built version of both.

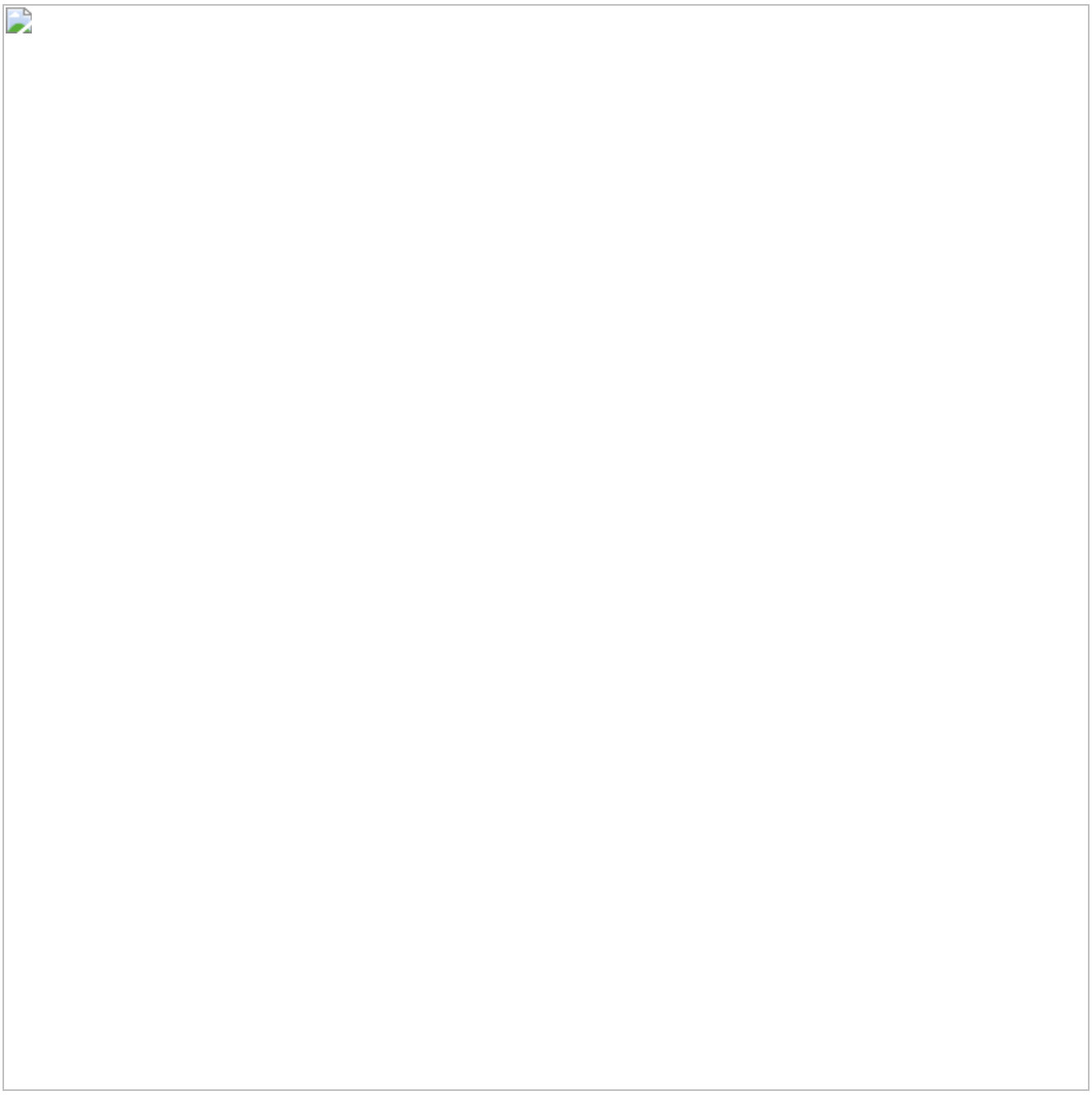
DLL is just a simple Proof-of-Concept DLL that writes the path of the executable, it is loaded into(services.exe) to the file "poc.txt", and signals RunAsWinTcb back, notifying that it has been loaded successfully. After that, it does some clean-up.

POC DLL can be used as a starting point for a real exploit.

RunAsWinTcb works with both, Administrator and SYSTEM level privileges. (If executed as SYSTEM, it impersonates Local Service to make the csrss.exe trick work)

The source code and already-build parts of RunAsWinTcb can be found here.

Demonstration

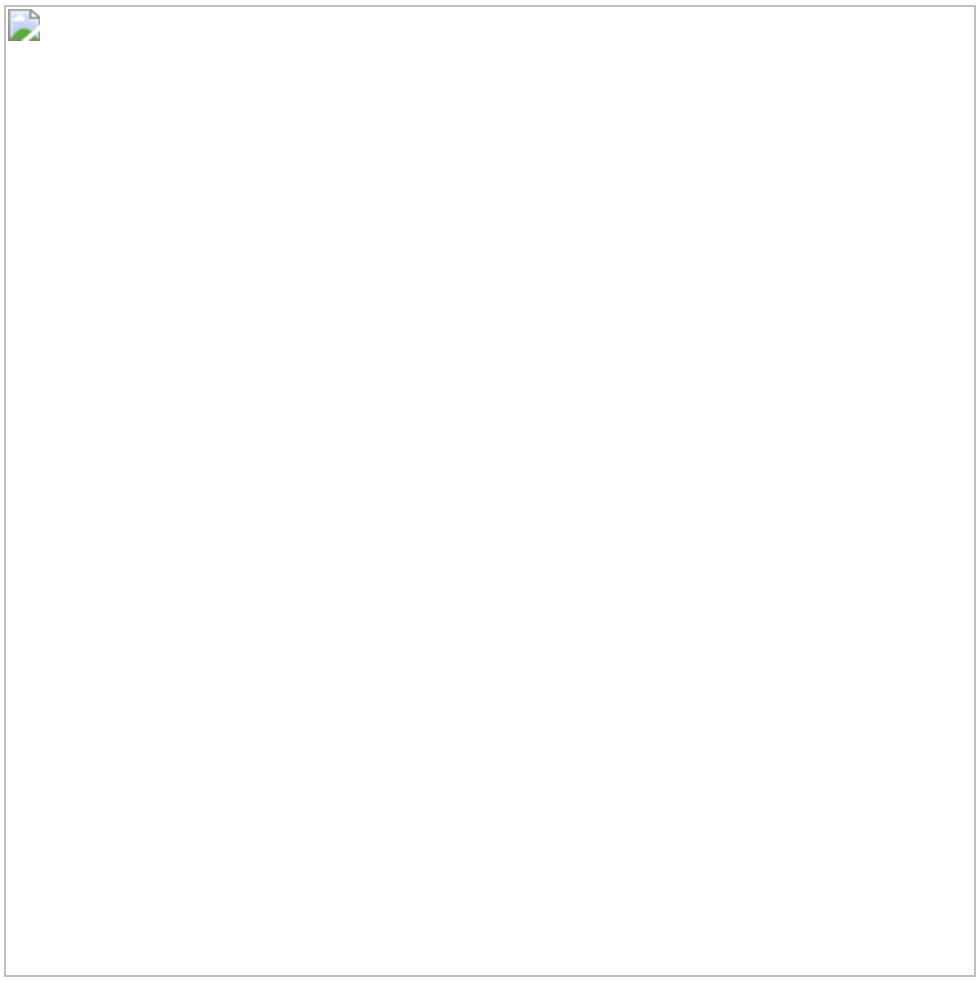


Basic Demonstration

Want to see something cooler?

Let’s kill the Windows Defender service(“MsMpEng.exe”).

Inspecting the “MsMpEng.exe” in Process Explorer shows that it is indeed a Protected Process Light, with a signer type of AntiMalware-Light.



Since we're running with a higher level of protection(WinTcb-Light, the highest in this hierarchy), we should be able to open a "PROCESS_ALL_ACESS" handle to it, allowing us to terminate a process.

Just modify a DLL:

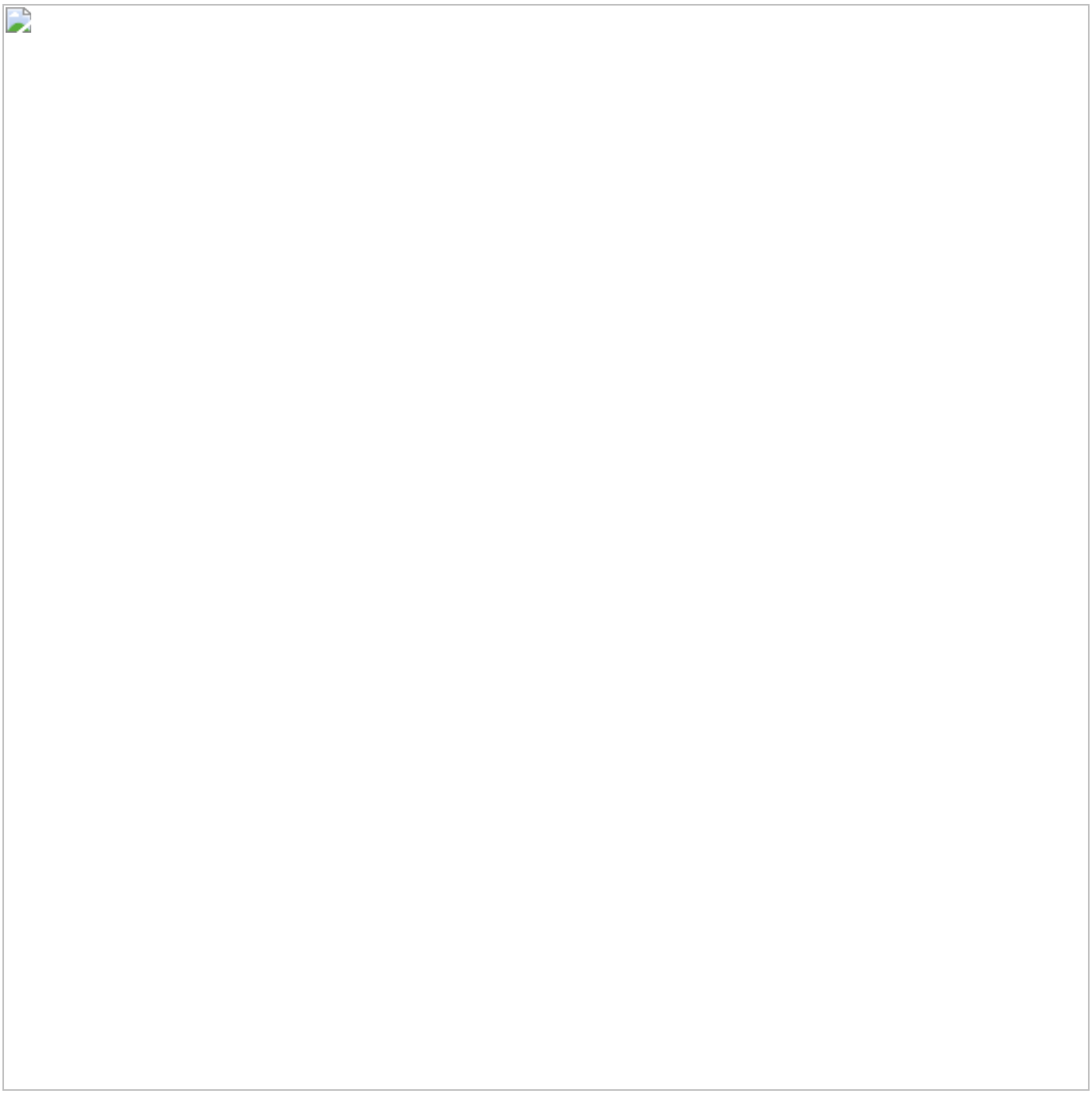
Add function to find the process ID by its name:

```
DWORD FindProcessId(const std::wstring& processName){    PROCESSENTRY32 processInfo;
processInfo.dwSize = sizeof(processInfo);HANDLE processesSnapshot =
CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, NULL);    if (processesSnapshot ==
INVALID_HANDLE_VALUE) {        return 0;    }Process32First(processesSnapshot,
&processInfo);    if (!processName.compare(processInfo.szExeFile))    {
CloseHandle(processesSnapshot);        return processInfo.th32ProcessID;    }while
(Process32Next(processesSnapshot, &processInfo))    {        if
(!processName.compare(processInfo.szExeFile))        {
CloseHandle(processesSnapshot);            return processInfo.th32ProcessID;        }
}CloseHandle(processesSnapshot);    return 0;}
```

Use it to find the PID of the Defender service, open the "PROCESS_ALL_ACCESS" handle to it, and call TerminateProcess():

```
auto processId = FindProcessId(L"MsMpEng.exe");HANDLE hDefender =
OpenProcess(PROCESS_ALL_ACCESS, false, processId);TerminateProcess(hDefender,
1);CloseHandle(hDefender);
```

Fire up RunAsWinTcb and watch the defender cease to exist:



Killing Defender Process

The Fix

In Windows 10 21H2 10.0.19044.1826 (24 July 2022 update), this vulnerability was patched.

Starting from that update, PPLs will no longer use the KnownDLLs directory in the search order of DLLs.

This means that the only way for PPLs to load DLLs is to load them from the disk and map them in memory themselves, meaning that their digital signature will be verified before loading them into the process.

Credits

@itm4n — Author of PPLDump

@tiraniddo — Windows Exploitation Tricks: Exploiting Arbitrary Object Directory Creation for Local Elevation of Privilege
https://googleprojectzero.blogspot.com/2018/08/windows-exploitation-tricks-exploiting.html

Special thanks to @_hillu for supporting me in developing RunAsWinTcb.