An introduction to privileged file operation abuse on Windows

offsec.almond.consulting/intro-to-file-operation-abuse-on-Windows.html

Published on Wed 20 March 2019 (2019-03-20T17:00:00+01:00) by @clavoillotte
Edited on Sat 05 October 2019 (2019-10-05T18:00:00+02:00)

TL;DR This is a (bit long) introduction on how to abuse file operations performed by privileged processes on Windows for local privilege escalation (user to admin/system), and a presentation of available techniques, tools and procedures to exploit these types of bugs.

Privileged file operation bugs

Processes running with high privileges perform operations on files like all processes do. But when a highly privileged process accesses user-controlled files or directories without enough precautions, this can become a security vulnerability, as there is potential to abuse the operations performed by that privileged process to make it do something it is not supposed to. This is true for many cases of privileged access to user-controlled resources, files are just an easy target.

Examples well-known to pentesters include user-writable service executables and DLL planting: if you have write access to a file that the privileged service will execute, or to a directory that it will look for DLLs in, you can get your payload executed in this privileged process. This is a well-known vulnerability and, aside from the occasional configuration mistake, one most privileged software now defends against (hopefully).

However, the potential abuse of other filesystem operations does not seem as well-known, yet are just as dangerous: if you can make a privileged process create, copy, move or delete arbitrary files for you, chances are that sweet `SYSTEM` shell is not far away.

Also, because these are logical vulnerabilities, they are usually very stable (no memory corruption involved), often survive code refactoring (as long as the file manipulation logic does not change), and are exploited exactly the same way regardless of the processor architecture. These characteristics make them very valuable for attackers.

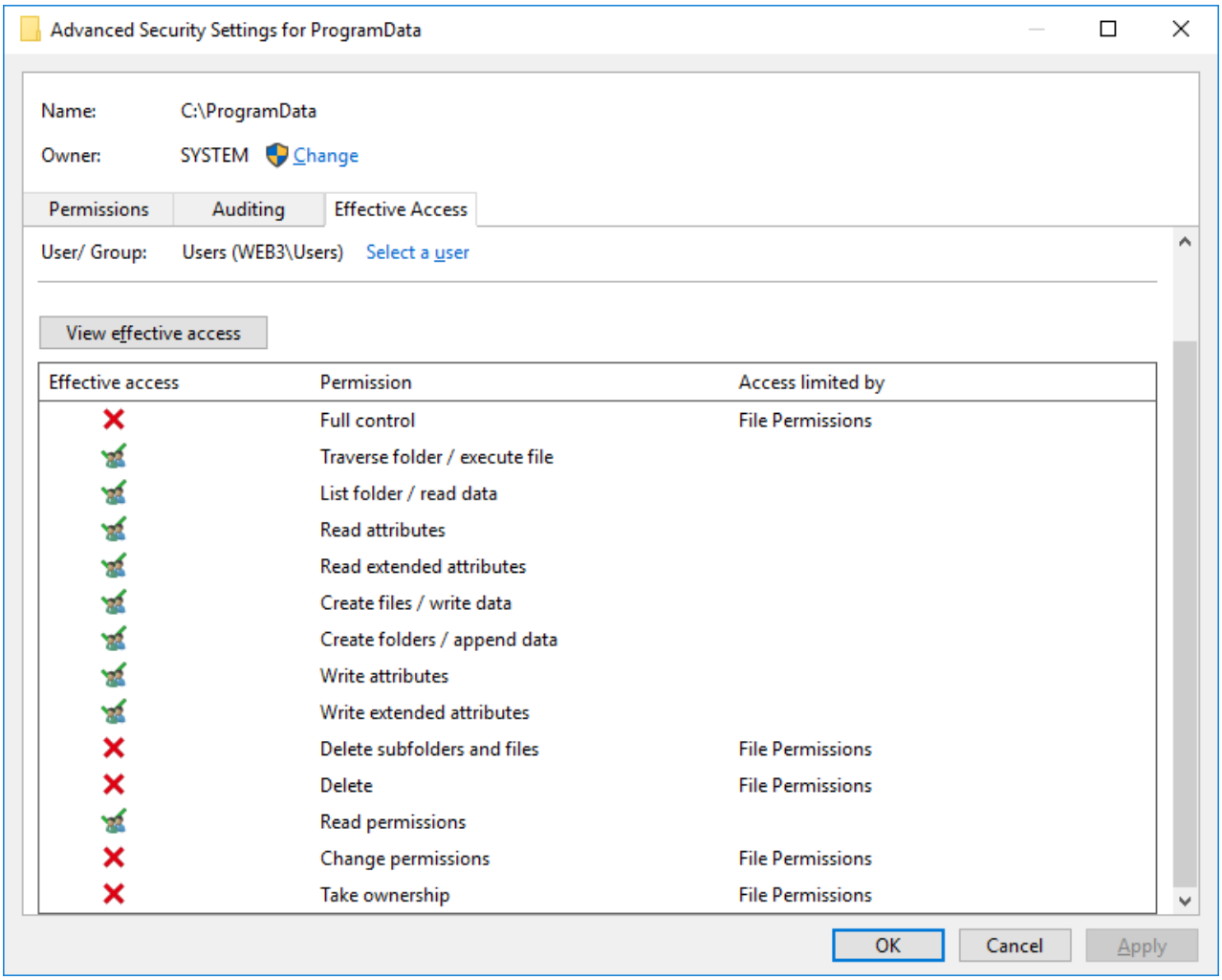
Finding (some of) the bugs

User-writable locations

While most privileged programs will not manipulate some unprivileged user's files directly (with some exceptions like AV), many will perform operations on files that may be located somewhere a user can fiddle with. Interesting locations that unprivileged users have some form of write access to includes:

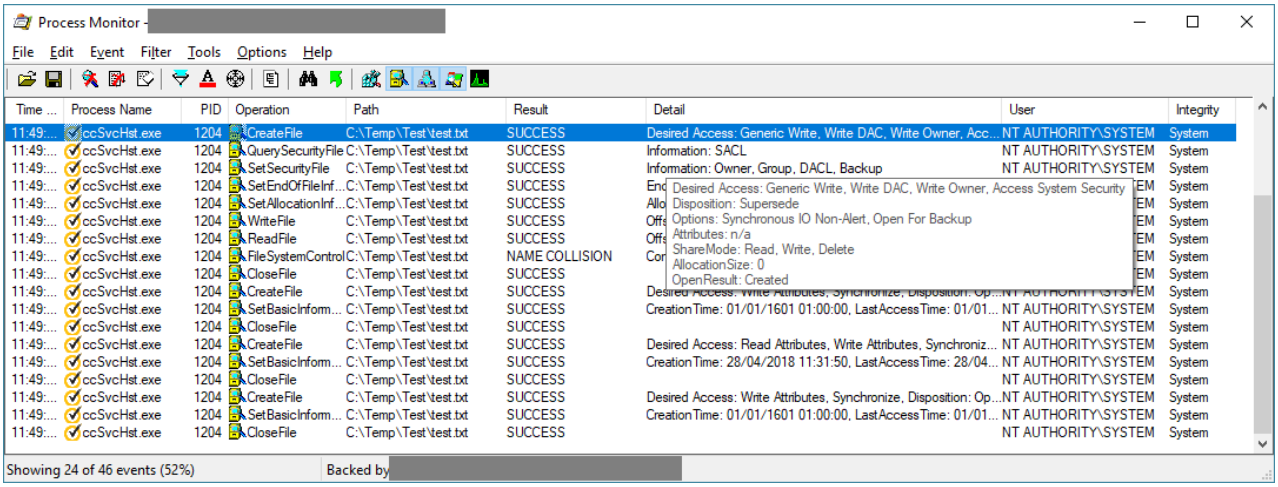
- The user's own files & directories, including its `AppData` and `Temp` folders, that some privileged process may use if you're lucky or running an AV
- The Public user's files & directories: idem
- Directories created in `C:\` with default ACL: by default, directories created at the root of partitions do have a permissive ACL that allows write access for users
- Subdirectories of `C:\ProgramData` with default ACL: by default, users can create files and directories, but not modify existing ones. This is often the first place to look at.
- Subdirectories of `C:\Windows\Temp`: by default, users can create files and directories, but not modify existing ones and read files / access directories created by other users. Interesting to look at for installers and other privileged software & scripts that run punctually and do not check for preexisting files & directories

You can check file permissions using tools & commands such as SysInternals' AccessChk, `icacls` or PowerShell's `Get-Acl`, or simply explorer's Security tab: the Advanced form has an Effective Access tab that allows to list the accesses that a specific account or group has over that file/directory (like AccessChk does on the command line). The following screenshot shows the (default) access rights granted to the Users group on the `C:\ProgramData` directory:



Looking for privileged file operations

To find instances of file operations performed by privileged processes, we can simply use SysInternals' ProcMon, filter file event for the processes of interest. When we see it accessing user-controllable files & directories, we can check whether the process uses impersonation to do so (mentioned in the details when used). And sometimes, it doesn't:



This will of course only give us the operations performed by the process on its own (e.g. when it starts), to find more we will have to look at operations that can be performed by an unprivileged user, either directly through the UI if it has one, or indirectly through COM, LPC, network interfaces, and other exposed attack surface. Some of these may require reverse engineering the product (cool examples here, here and here), for this introduction we're just going to go for the low-hanging fruits.

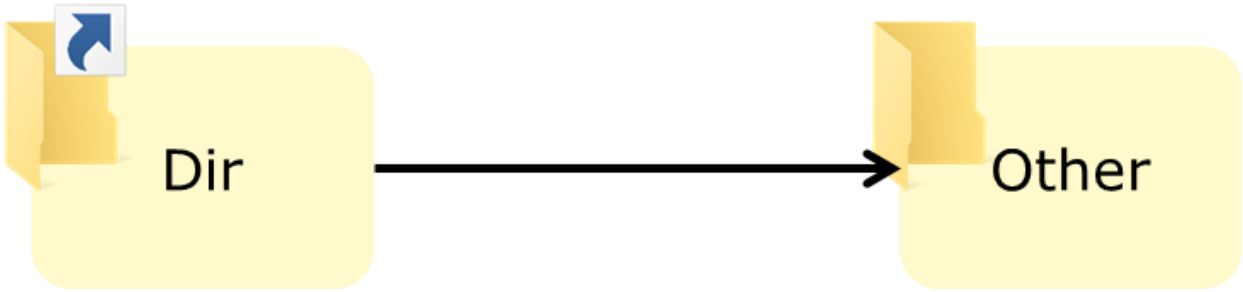
Exploitation techniques & tools

Once we find some file operations performed on user/user-controllable files & directories, we will need a way to hijack these operations to do something interesting.

Thankfully, James Forshaw (@tiraniddo) did all the heavy lifting with his seminal work on the NTFS filesystem and Windows internals, which he published in numerous articles (quick selection: symlinks, hardlinks, NTPathConversion, DirCreate2FileRead, FileWrite2EoP, AccessModeMismatch) and presented at conferences such as Infiltrate and SyScan. He came up with several techniques to abuse Windows filesystem and path resolution features (roughly summarized below) and implemented them in the open-source symboliclink-testing-tools toolkit and NtApiDotNet library. His techniques and toolkits opened the doors for many testers to go hunt for this type of bugs (myself included) by making these bugs possible – and even easy – to exploit, effectively turning them into the new privesc low-hanging fruits.

NTFS junctions

Junctions are an NTFS feature that allows directories to be set as mount points for a filesystem, like a mount point in Unix, but can also be set up to resolve to another directory (on the same or another filesystem). For our purposes, we can think of them as a sort of directory-only symbolic links.



The interesting thing is, the path resolution will transparently follow junction in most cases (unless a parameter is explicitly set to prevent this), so in the setup above, a program that tries to open `C:\Dir\file.txt` will in fact open `C:\Other\file.txt` as the IO manager will follow the junction.

Junctions can be created by unprivileged users. They work across volumes, so you can "redirect" `C:\Dir` to `D:\OtherDir` as well. An existing directory can be turned into a junction if you have write access to it, but it must be empty.

NTFS junctions are implemented with reparse points and, while built-in tools won't let you do it, they can be made to resolve to arbitrary paths with an implementation that sets custom reparse points. The `CreateMountPoint` tool (from symboliclink-testing-tools) allows you to do just that. For regular junctions, you can also use `mklink` and PowerShell's `New-Item` with the `-Type Junction` parameter.

Hard links

Unprivileged users can also create hard links which, like their Unix counterpart, will serve as an additional path to an existing file. It does not work on directories, or across volumes (wouldn't really make sense for hard links).

Here also, built-in tools won't let you create a hard link to a file you don't have write access to, but the actual system call lets you do it with a file opened for reading. Use the `CreateHardLink` tool from symboliclink-testing-tools (or this PowerShell script by Ruben Boonen) to create hard links to file you don't have write access to. Beware, you won't be able to delete the created link if you don't have write access to the file (the same way you can't delete the file using its original path).

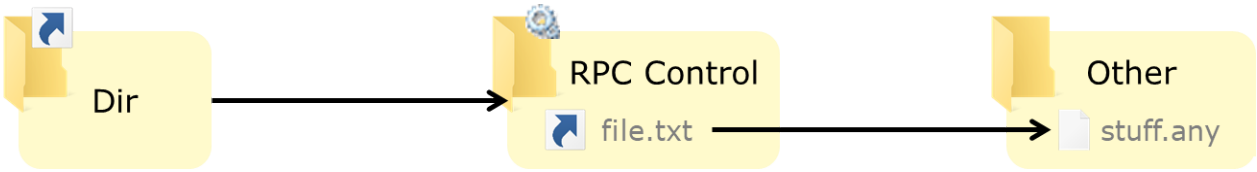
Update: This technique is being mitigated in an upcoming version of Windows 10.

Object Manager symbolic links

While NTFS does offer filesystem symbolic links, on Windows unprivileged users can't create symlinks on the filesystem: it requires SeCreateSymbolicLinkPrivilege, which by default is only granted to Administrators.

Unprivileged users can, however, create symbolic links in Windows' Object Manager which, as its name suggests, manages objects such as processes, sections and files. The Object Manager uses symlinks, e.g. for drive letters and named pipes that are associated with the corresponding devices. Users can create object symlinks in writable object directories such as `\RPC CONTROL\`, and these symbolic links can point to arbitrary paths – including paths on the filesystem – whether that path currently exists or not.

Object symlinks are especially interesting when combined with an NTFS junction. Indeed, as an unprivileged user we can chain a mount point that resolves to the `\RPC Control\` directory with an object manager symlink in that directory:



This gives us something that behaves somewhat like a filesystem symlink: in the image above, `C:\Dir\file.txt` resolves as `C:\Other\stuff.any`. This of course is not an exact equivalent, but it is enough to abuse programs in many cases.

You can use `CreateMountPoint` and `CreateDosDeviceSymlink` to do these steps separately, but the `CreateSymlink` tool implements this technique in one handy command.

Opportunistic locks

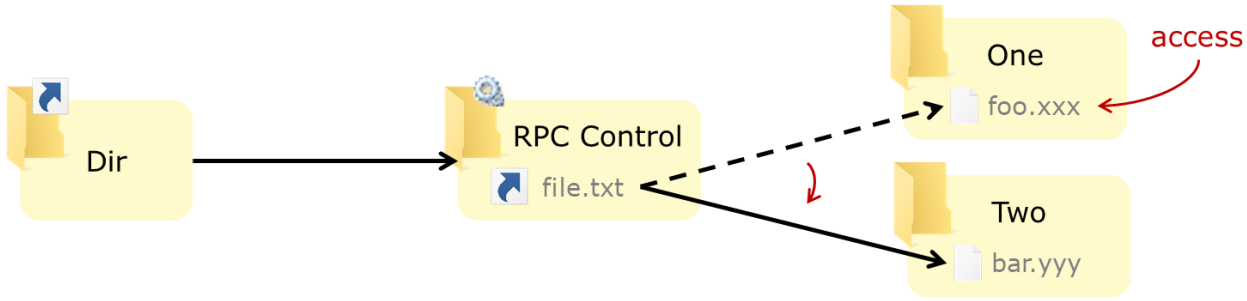
An opportunistic lock (oplock) is a lock that can be placed on a file to be informed when other processes want access to that file – while delaying access from these processes so that the locking process can leave the file in a proper state before lifting the lock. Initially designed for caching client-server file access through SMB, oplocks can be placed locally by invoking a specific control code on the file handle.

This is useful to exploit TOCTOU bugs, as you can easily win a race against a process by locking a file or directory it tries to open. Of course, it does come with some limitations: you can't granularly “let through” just one access (all pending access will occur once the lock if lifted), and it does not work with all types of access, but it is usually very effective.

The `SetOpLock` tool lets you create these and block access to a file or directory until you press enter to release the lock. It lets you choose between read, write and exclusive oplocks.

Again, James combined this technique with the previous ones to create a powerful primitive that eases the exploitation of some TOCTOU bugs: by setting up a pseudo-symlink (as before) and placing an oplock on the final file (target of the symlink), we can

change the symlink when the target file is opened (even if the target file is locked, the symlink is not) and make it point to another target file:



On the setup shown above, the first access on the file `C:\Dir\file.txt` will open `C:\One\foo.xxx`, and the second access will open `C:\Two\bar.yyy`.

The `BaitAndSwitch` tool implements this technique with exclusive oplocks, if you need read or write locks you can use `SetOpLock` and `CreateSymlink`.

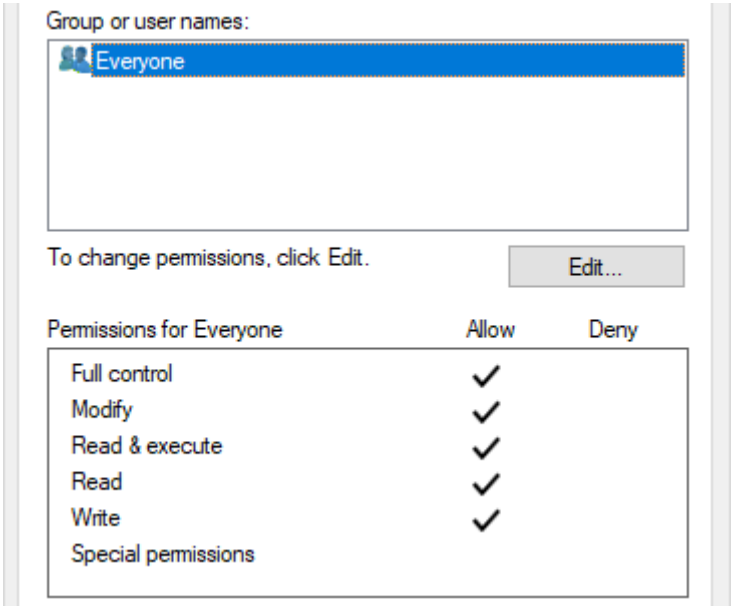
Exploitation strategies

A classic example

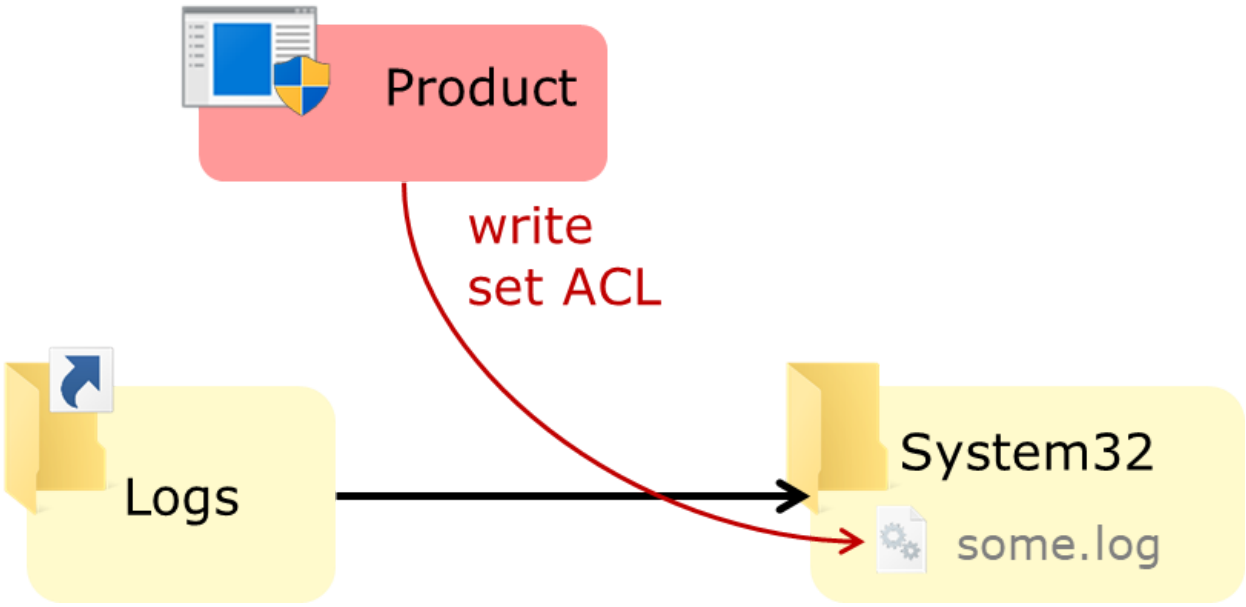
Let's consider the following behavior for Product X:

- Creates log files in `C:\ProgramData\Product\Logs` (directory with default/inherited access rights)
- Logs files are created/written to by both privileged (system) and unprivileged (user) processes
- The process that creates the log file sets an explicit ACL so that everybody can write to the file (and rotate it if needed):

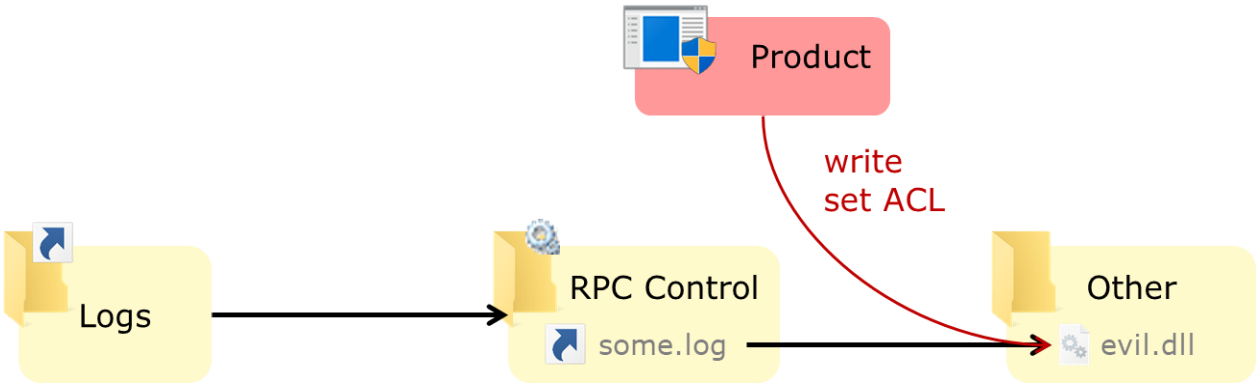
This results in a vulnerability that can be exploited to create arbitrary files with arbitrary content.



If we remove existing log files, and turn the Logs directory into a junction to `C:\Windows\System32` (thanks to the access rights inherited from `C:\ProgramData`), the privileged processes of Product X will create their logs in the `System32` directory:



We can also use the symlink technique to divert a specific log file (e.g. `some.log`) to create an arbitrary file with an attacker-chosen name, e.g. a DLL in the program's directory:



Because the privileged process also sets a permissive ACL on the log file, we can also change the content of the file to our liking.

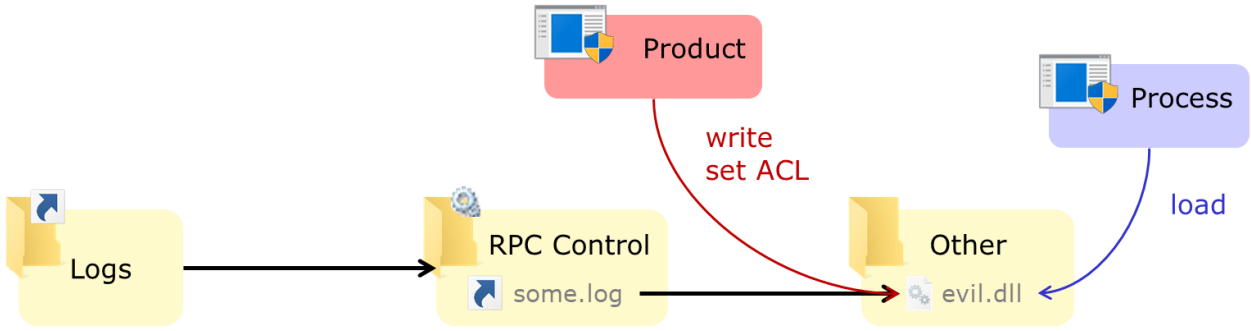
It is a bug that has been found on several products, presumably because it is the simple implementation of a common need (log files writable by all components – user and system components, common logging code for all components). We saw several instances of this over the last year of so:

- In Cylance by Ryan Hanson
- In Symantec / Altiris agent by Ben Turner
- In McAfee Endpoint Security (patched)
- In NVIDIA GeForce Experience and Intel Driver & Support Assistant by Mark Barnes
- In Pulse Secure VPN client (unpatched) (collision with Matt Bush)

From arbitrary file write to privilege escalation

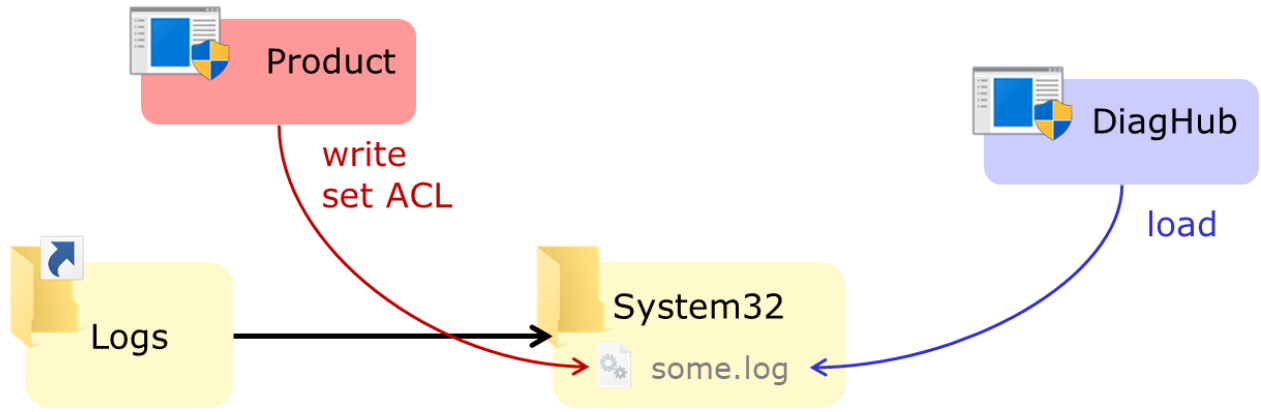
The two go-to techniques used to get code execution in the context of a privileged process with an arbitrary file write are:

- DLL hijacking: create a DLL in a location where a privileged process will load it (in the application's directory, in `System32`, `Windows`, or some other directory on `SYSTEM`'s `%PATH%`). It requires a way to (re)start that privileged process to load the payload, and a location the DLL will be loaded from before the hijacked one.
- Overwrite: replace an existing binary / script / config file / etc. that will give us code execution. Besides requiring (re)starting the process, it also needs the file write operation to allow overwriting existing files (plus the target file should not be locked), and is usually very specific to a given service/application.



There are at least two lesser-known techniques:

- Using `C:\Windows\System32\Wow64Log.dll` to load a 64-bits DLL in a privileged 32-bits process. This DLL does not exist by default (at least on consumer releases), and is loaded in all 32-bits processes. However, the DLL can't (always?) use imports from Kernel32 so it must use only NTDLL APIs, and of course this only works if you have an interesting (privileged) 32-bit process to inject into (and again a way to start it). AFAIK this trick was found by George Nicolaou (first reference I could find) and documented by Walied Assar.
- Using the “Diagnostics Hub Standard Collector Service”: this technique was found by – you guessed it – James Forshaw, explained in detail in a GPZ blog post and published in an example exploit for the bug used as a case study. The article is very well worth a read, but if you're in a hurry (seriously add it to your reading list though), the gist is: the DiagHub service (which runs as `SYSTEM`) can be made to load a file with any extension from `System32` as a DLL. So, if you can create a file `test.log` with your payload (the content of the file must still be a working DLL of course) in `System32`, just use this technique to have that DLL loaded in the privileged service. However, this technique is being mitigated in the upcoming version of Windows 10.



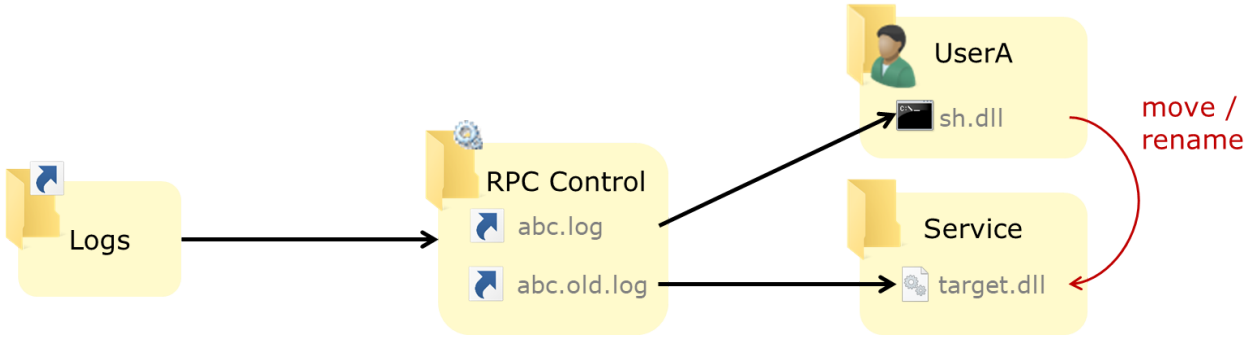
Controlling the content

These techniques require control over the content of the created file: if you can hijack the creation of a file to an arbitrary location, but can't control what's in that file, its usefulness is very limited. In our example bug we had a nice ACL set by the privileged program on the resulting file, but what if we don't have this luxury condition?

We can try to target other operations. In our logging example, assuming the logging feature rotates logs when they reach a certain size, a privilege process might move or rename the log file (e.g. from `abc.log` to `abc.old.log`). Then we can abuse this operation using symlinks:

- Replace the source file in the rename/move operation by a pseudo-symlink to our payload (`sh.dll`)
- Replace the destination file by a pseudo-symlink to the file we want to create or replace (here `target.dll`)

So, the layout when the rename operation takes place looks something like the following:



When the privileged process will try to move or rename `abc.log` to `abc.old.log`, it will in fact move/rename the user-owned file `sh.dll` to `target.dll`, placing our payload in the right place to be executed.

So privileged file move/rename/copy operations we can control are very interesting primitives:

- A controlled move or rename gives us arbitrary file write
- Same for a fully controlled (source & destination) copy

- A copy operation where we control the source but not the destination gives us an arbitrary file read (if the destination location is user-readable)
- A move/rename operation where we control the source but not the destination gives us an arbitrary file delete (kind of)

Side notes:

- The ability to overwrite the destination will depend on the options used by the process performing the operation
- If the target file already exists, we can use also a hard link instead of a pseudo-symlink
- A common way to abuse arbitrary file read is to get the `SAM`, `SECURITY` and `SYSTEM` hives to dump the `SAM` database and cached credentials

From arbitrary file delete to privilege escalation

We talked about arbitrary file read & write, what about delete? Aside from the obvious DoS potential, we can sometimes abuse arbitrary file delete bugs for EoP by deleting files that:

- are in a location we can write to, even if we can't overwrite existing ones, like `C:\ProgramData`
- will later be used for read or write operations by a privileged process (whether it is the same process we abuse for removal or a different one)

As an example, if we know how to trigger a move/rename from `C:\ProgramData\Product\foo` to `C:\ProgramData\Product\bar` but these files already exist and we don't have write access to them, we can use an arbitrary file delete bug to delete foo and bar, and re-create them ourselves (again assuming default rights for the `Product` subdirectory). We can use the previous techniques to abuse the write operation (pseudo-symlink if the `Product` directory is now empty, hard links otherwise) and complete the chain.

Exploiting AV

AV software is a prime target for this class of bugs, as it is highly-privileged software that, by design, must manipulate files, including user-owned files. Privileged process performing scanning, deletion, and sometimes restore can be tricked into performing interesting files operations for us, turning the defense component into a potential way in. (Of course, this just piles up to the lot of other attack surface AV has, and one should carefully weigh the risk-benefit ratio before deploying anyway.)

AV quarantine & restore

The quarantine and restore features are particularly interesting, especially when they can be triggered by unprivileged users (sometimes not in the UI but reachable via COM hijacking). The simplest way to trigger a quarantine (or removal) is of course to drop a

known detected file like EICAR onto the filesystem.

Interestingly, some AVs will perform privileged operations on a detected file before removing it, such as:

- create/delete temporary files in the same directory
- copy or move the infected file in a user-writable location
- copy or move the infected file to a user-readable quarantine location (beware not to nuke you `SAM` file if you exploit this)

The temporary and quarantine files are sometimes encoded and/or padded, and if you want to look at the algorithm (to read the resulting files) a good place to check before firing up IDA/Ghidra is DeXRAY by Hexacorn.

And the restore process, if it can be triggered by an unprivileged user, is another instance or a privileged file write bug (examples here). To control the content, either look for potential TOCTOU during delete or restore, or make your payload “malicious” enough to be quarantined in the first place.

Diverting a file deletion/quarantine

A fun trick we can use if the AV does not lock (or otherwise prevent access to) a detected file between detection and removal/quarantine (TOCTOU) is to replace its parent directory by a junction – after detection but before removal. If we want to remove some file we don't have access to (say, `C:\Windows\System32\license.rtf`), we can proceed as such:

1. Drop EICAR (or any detectable file) in a directory we created, with the same name as the target file e.g. `C:\Temp\Test\licence.rtf`
2. Wait for it to be detected by AV
3. Remove or rename the parent directory `C:\Temp\Test`
4. Replace it with a junction to `C:\Windows\System32`
5. AV deletes `C:\Temp\Test\licence.rtf` which resolves to `C:\Windows\System32\licence.rtf`

The proper way to do that is to use oplocks, however in practice it is not always that easy, as the file can be accessed a (variable) number of times before it is removed, and the lack of granular control can render this tricky. A quick & dirty way is to simply create the junction alongside the directory and make a loop that keeps exchanging the two. Depending on the way AV retrieves the path to delete and deletes the file, we may have a reasonable chance of deletion hitting the junction, and we can retry as necessary. (Spoiler: it did work on several AV products, not at all on others.)

Obviously, all of this is not very stealthy, but if proper event forwarding is not in place, the first file you target might as well be the event log :)

Bugs found

Several such bugs have been found in common software products early last year and reported to their respective vendors, some of which have issued fixes. The following table summarizes the bug we found and their status:

Product	ID	Vulnerability	Arbitrary file	Reported	Fix
Symantec Endpoint Protection 12 & 14	CVE-2017-13680	TOCTOU in the quarantine GUI	Deletion Read	09/2017	Available 11/2017
Symantec Endpoint Protection 12 & 14	CVE-2018-5236	TOCTOU during file deletion	Deletion	11/2017	Available 06/2018
Symantec Endpoint Protection 12 & 14	CVE-2018-5237	Check bypass in file restore	Write	11/2017	Available 06/2018
AV product A	TBD	Over-privileged file deletion	Deletion	03/2018	In progress
AV product B	TBD	Over-privileged file restore	Write	05/2018	In progress
McAfee Endpoint Security 10	CVE-2019-3582	Overpermissive access rights Over-privileged file creation	Write Deletion	05/2018	Available 10/2018 & 02/2019
AV product C	TBD	TOCTOU during file deletion	Deletion	05/2018	In progress
AV product D	TBD	TOCTOU during file deletion	Deletion	05/2018	In progress
F-Secure SAFE/CS/CP	(none)	Over-privileged file copy	Write Read Delete	07/2018	Available 08/2018
Pulse Secure VPN client	CVE-2018-11002 (collision)	Overpermissive access rights Over-privileged file creation	Write	06/2018	Available 05/2019
Product F	TBD	Over-privileged file creation	Write	07/2018	In progress
Product G	TBD	Over-privileged file creation	Write	07/2018	In progress

Product	ID	Vulnerability	Arbitrary file	Reported	Fix
Product H	TBD	Over-privileged file creation	Write	08/2018	In progress
Intel PROSet / Wireless	INTEL-SA-00182 / CVE-2018-12177 (collision)	Overpermissive access rights	DACL set	08/2018	Available 01/2019

Missing product names & additional details will be published as fixes become available (or bugs are made public).

Details are available for the following bugs:

Conclusion

I hope this article will get some new people started on finding file-related logical privilege escalation bugs, and as many reasons for vendors and developers to harden/fix their products against this type of bugs.

Thanks to James Forshaw of course, for giving us these cool techniques, to those of the vendors that were responsive enough (special mention for the F-Secure security team), to GreHack and TROOPERS who let me on stage to talk about it, and to my teammates who helped me prepare the talks and this article.