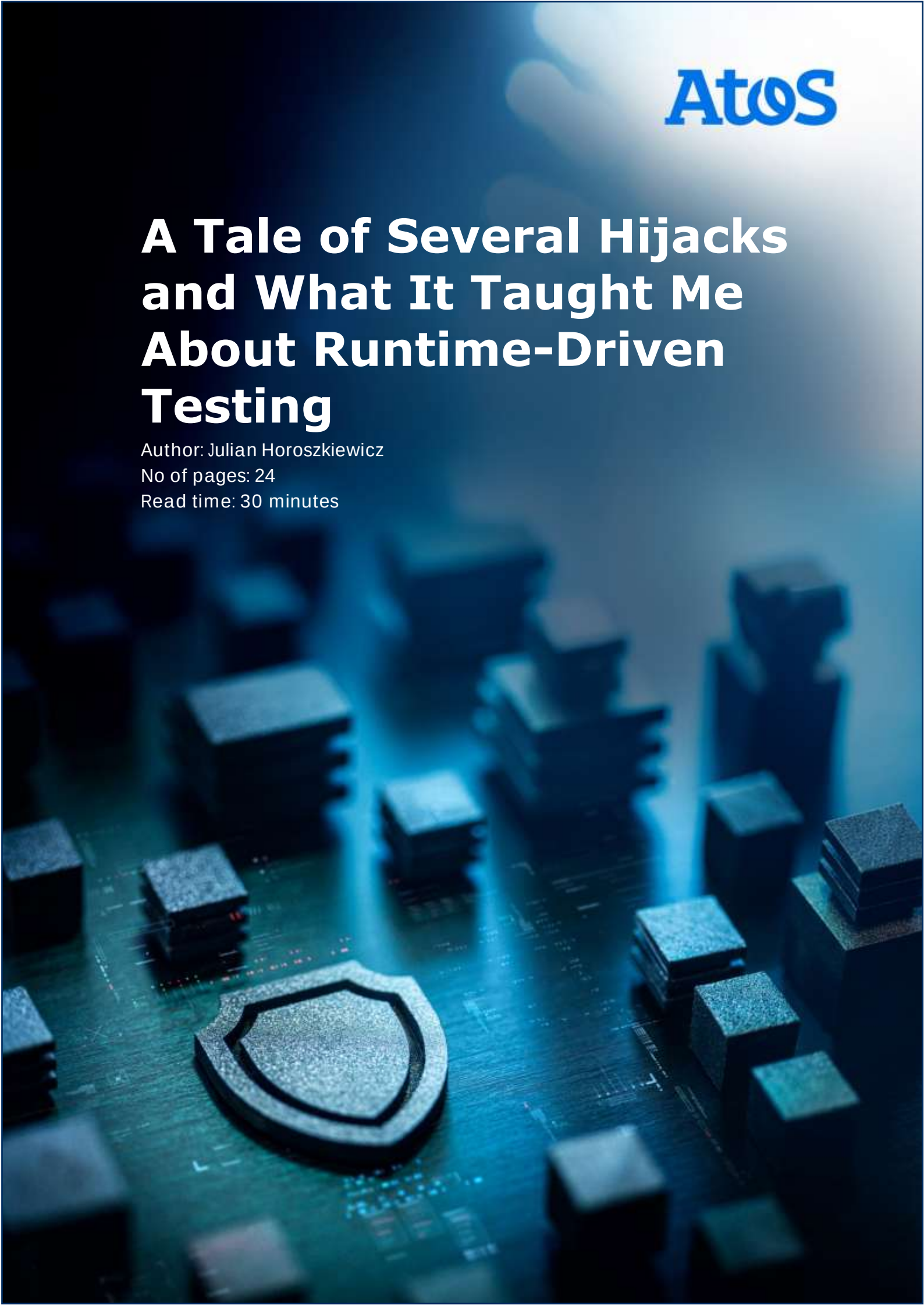


A Tale of Several Hijacks and What It Taught Me About Runtime-Driven Testing

Author: Julian Horoszkiewicz

No of pages: 24

Read time: 30 minutes

Table of Contents

1 Introduction

What do the Windows versions of Ghostscript, Apache, PHP, GnuPG, wget, and Microsoft Coreutils have in common?

This article walks through a series of recently discovered file-hijacking issues in widely used software. I break down the root causes: default OS permissions on common filesystem locations, code poorly ported from other platforms, assumptions that don't hold, insecure or missing installers, and various path construction issues such as failed environmental variable expansion or mismatches between APIs.

I also ask why these issues slip past static source-code analysis — looking at the practical differences between source, the compiled executable, and the running process—and how runtime-driven testing helps close that gap.

All in a story that started with a simple follow-up investigation.

2 Case studies

2.1 Ghostscript (CVE-2026-19547)

Ghostscript for Windows in versions up to 10.07.1 is vulnerable to cross-user code execution through PostScript resource file hijacking. During initialization, it searches for PostScript resource files in multiple directories - as demonstrated in the screenshot below:

Ghoscript_path_not_found

These paths are checked during every Ghostscript execution regardless of the invocation command line. Those PostScript resource files Ghostscript attempts to load are not passive data files — they are executable PostScript programs.

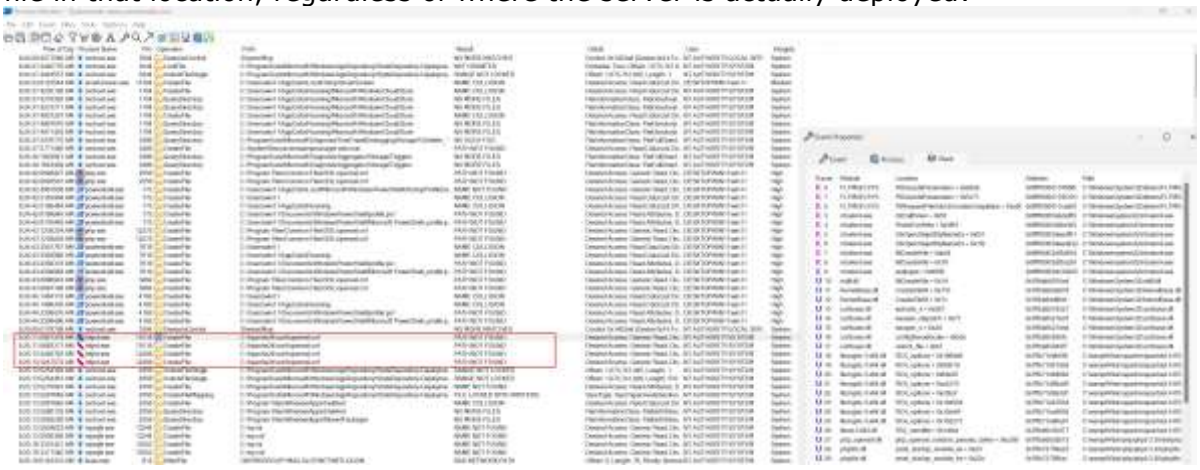
For example, a custom PostScript file created in C:\gs\fonts\IdiomSet\Pscript5Idiom executes every time Ghostscript is invoked. This was confirmed with a benign Postscript PoC, demonstrating arbitrary file writing capability. In practice that primitive can be utilizied for creation of an executable script, such as PowerShell profile (C:\Users\<user>\Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1).

In response to the bug ticket I filed (now publicly available), the developer referred to those references as "ancient, historical paths, dating back to the days of MS-DOS".

In terms of impact, it depends on the environment and setup. This issue does not pose risk to single-user systems in which Ghostscript is executed by the regular user. Contrary to multi-user environments and setups in which Ghostscript is invoked by a privileged process (e.g. a service).

2.2 Apache HTTP Server (CVE-2026-89281)

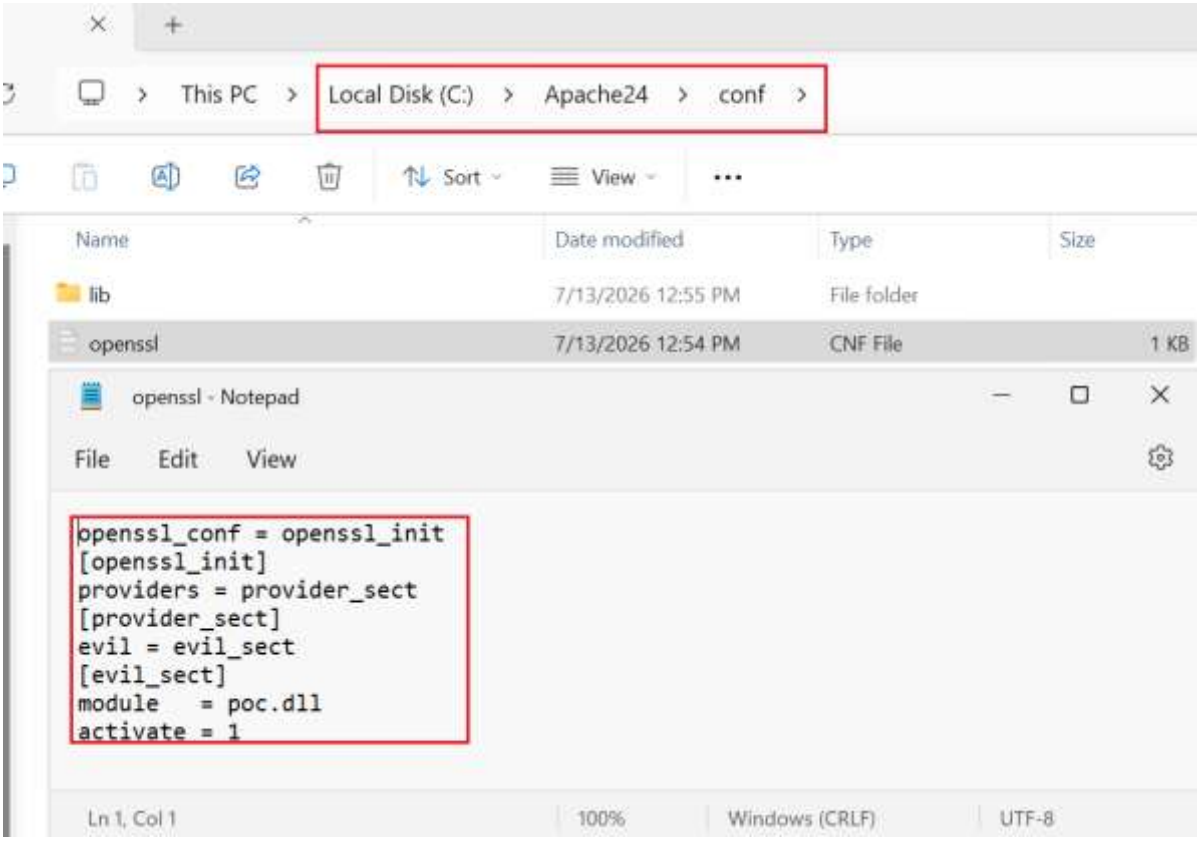
The generally available Windows builds provided and hosted by the Apache Lounge project, contain a hardcoded openssl config path in libcrypto-3-x64.dll - C:\Apache24\conf. Upon service start the process searches for the openssl.cnf config file in that location, regardless of where the server is actually deployed:



References to nonexistent openssl.conf file

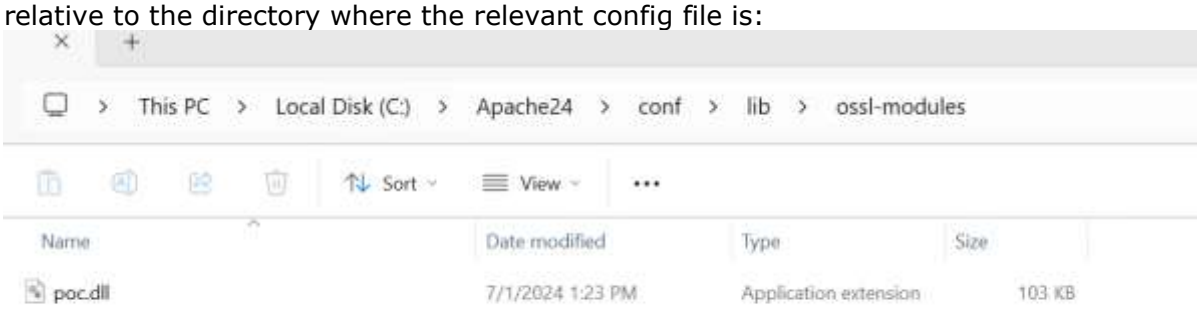
Proof of concept

A local user needs to create the relevant file and directory structure, as demonstrated in the screenshot below:



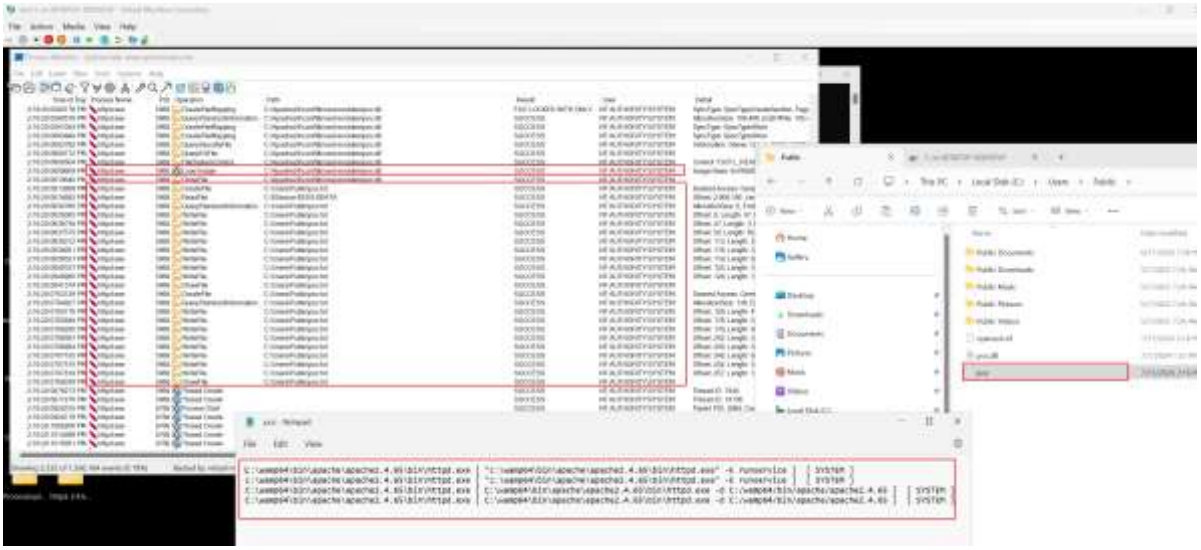
PoC conf file

The DLL file defined in the config is searched for in a subdirectory lib\ossl-modules, relative to the directory where the relevant config file is:



PoC location

When the server starts/restarts, the DLL of the fake crypto provider defined in the malicious openssl.cnf gets loaded and executed in the httpd.exe server process. The DLL I used for the demo only grabs the current process information and the user running it (to confirm where it got injected and with what privileges) and appends it into C:\Users\Public\poc.txt:



Code execution as SYSTEM confirmed

It is worth keeping in mind that these binaries are not only deployed directly as Apache Lounge distribution. They are also used in other software products (which by default do not use C:\Apache24 as the installation path), including WAMP, XAMPP and several proprietary solutions I am not going to disclose in this article.

In response to the report, the latest versions published on the website include a "Security.txt" file, which links to this forum post https://www.apachelounge.com/viewtopic.php?p=44459#44459.

2.3 PHP default include_path

The official Windows builds of PHP are vulnerable to local cross-user code execution via include/require script hijacking by Authenticated Users. **Both CLI and HTTP installations are affected.**

Whenever a PHP script calls include, include_once, require, or require_once, and the C:\php\pear directory exists, PHP includes that directory into the list of searched paths when looking for the included file if the argument is only a filename with no path. Since on Windows regular users can create new directories on C:, they can hijack such include calls and attain code execution with the current privileges of the PHP process.

Root cause

The C:\php\pear string can be found in the default value of the include_path directive on Windows: .;C:\php\pear

This can be easily confirmed by running a simple test on the latest binary PHP distribution for Windows (php-8.5.10-Win32-vs17-x64 at the time of writing this), by invoking phpinfo() and filtering its output for include_path in command line.

test.php:

```
<?php phpinfo(); ?>
```

```
php.exe test.php | findstr include_path
include_path => .;C:\php\pear => .;C:\php\pear
```

This string is not derived from the installation path, it is baked into the php8ts.dll binary.

Now, let's look at the value itself.

".;C:\php\pear" are just two directories separated by a semicolon. First, the current script directory alias (.) and then C:\php\pear.

This order suggests that the current script directory takes precedence over C:\php\pear during full path resolution.
And that is true, but with one exception, which decides when this becomes exploitable.

Exploitability - CLI example

Let's run a simple proof of concept using a CLI-only (no webserver) PHP script run by the victim user and injected into by the attacker.
Our CLI-only application consists of one script including another - a pattern found in almost every PHP application:

some_app.php:

```
<?php
    echo "Before config inclusion. \n";
    include "config.inc.php";
    echo "After config inclusion. \n";
?>
```

config.inc.php:

```
<?php
    echo "Legitimate config included!\n";
?>
```

We have both files in the current directory, let's run some_app.php:

```
C:\Users\victim\php-8.5.10-Win32-vs17-x64>whoami
desktopwin11\victim

C:\Users\victim\php-8.5.10-Win32-vs17-x64>php some_app.php
Before config inclusion.
Legitimate config included!
After config inclusion.
```

Nothing unexpected so far, this is how this script is supposed to work.

Now, let's set up the hijack while operating as the "attacker" user:

```
C:\Windows\System32>whoami
desktopwin11\attacker

C:\Windows\System32>mkdir C:\php\pear

C:\Windows\System32>echo MALICIOUS CODE > C:\php\pear\config.inc.php
```

Now let's see what happens when the victim runs the script again:

```
C:\Users\victim\php-8.5.10-Win32-vs17-x64>whoami
desktopwin11\victim

C:\Users\victim\php-8.5.10-Win32-vs17-x64>php some_app.php
Before config inclusion.
Legitimate config included!
After config inclusion.
```

No change - the malicious config.inc.php from C:\php\pear did not get included.

Now, let's see what happens if the file and directory structure of the app is slightly different, with the config.inc.php located in a different directory, let's say includes\, so now some_app.php looks like this:

```
<?php
	echo "Before config inclusion. \n";
	include "includes/config.inc.php";
	echo "After config inclusion. \n";
?>
```

In the new config.inc.php located in the includes/ subdirectory we have an additional include with no absolute path (a second order include):

```
<?php
	echo "Legitimate config included!\n";
	include "config.extension.php";
?>
```

For simplicity, the demo config.extension.php (the second order include) contains just a small piece of text:

```
C:\Users\victim\php-8.5.10-Win32-vs17-x64>type includes\config.extension.php
THIS IS LEGITIMATE CONFIG EXTENSION
```

So, the normal script behavior is:

```
C:\Users\victim\php-8.5.10-Win32-vs17-x64>php some_app.php
Before config inclusion.
Legitimate config included!
THIS IS LEGITIMATE CONFIG EXTENSION
After config inclusion.
```

Now, planting the hijack:

C:\Windows\System32>whoami
desktopwin11\attacker

C:\Windows\System32>echo MALICIOUS CODE > C:\php\pear\config.extension.php

Now, after C:\php\pear\config.extension.php is created:

C:\Users\victim\php-8.5.10-Win32-vs17-x64>whoami
desktopwin11\victim

C:\Users\victim\php-8.5.10-Win32-vs17-x64>php some_app.php
Before config inclusion.
Legitimate config included!
MALICIOUS CODE
After config inclusion.

The file from C:\php\pear\config.extension.php took precedence over C:\Users\victim\php-8.5.10-Win32-vs17-x64\includes\config.extension.php, leading to cross-user code execution!

So, in this case, when there is a second order include from a script located in a different directory than the main script and no absolute path is provided, the search order becomes:

1. C:\Users\victim\php-8.5.10-Win32-vs17-x64 (the current script directory). config.extension.php is not there, so the search continutes.
2. C:\php\pear (the alternative directory defined in *include_path*). If C:\php\pear\config.extension.php is there, the search ends here.
3. C:\Users\victim\php-8.5.10-Win32-vs17-x64\includes, which is where the first order include config.inc.php is located. This location works as expected, but only if C:\php\pear does not satisfy the search already.

This pattern of includes is very common in PHP applications, in both CLI and web setups. Even the welcome page of the WAMP server is built this way.

WAMP welcome page example

Let's see how this works using the WAMP welcome page, which in default installations is at C:\wamp64\www\index.php.

The script includes the following require call:

```
require $server_dir.'scripts/config.inc.php';
```

$server_dir is C:\wamp64.

The PHP script in C:\wamp64\scripts\config.inc.php makes another require call:

```
if(!defined('WAMPTRACE_PROCESS')) require 'config.trace.php';
```

Both files - config.inc.php and exist in the same directory. The former is the one making the second include call, the latter is the target of the first-order include call:

```
C:\wamp64\scripts>dir | findstr config
11/06/2025  03:34 PMs          39,263 config.inc.php
11/04/2025  11:39 AM             995 config.trace.php
```

Sending a GET request to http://localhost/ displays the welcome page as expected:

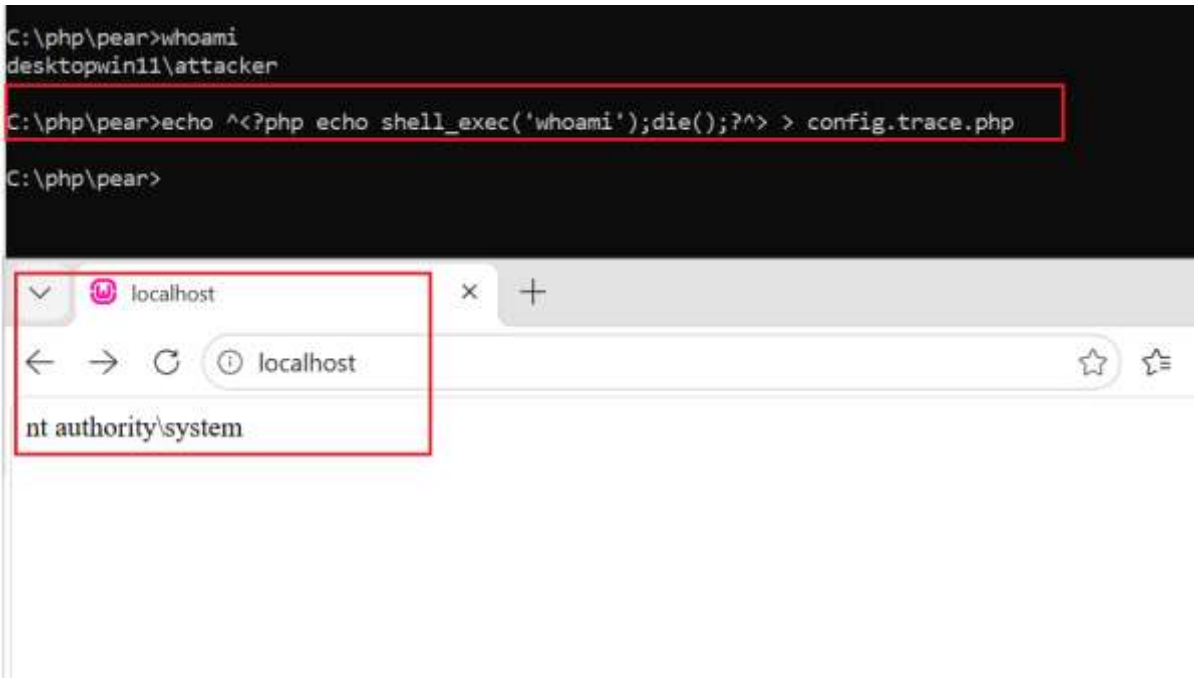
GET localhost before hijack

Now let's see what happens when C:\php\pear\config.trace.php is created:

```
C:\php\pear>whoami
desktopwin11\attacker
```

```
C:\php\pear>echo ^<?php echo shell_exec('whoami');die();?^> > config.trace.php
```

Now requesting GET http://localhost/ shows nt authority\system instead of the welcome page:



GET localhost before hijack

Notes on target code invocation

Usually, the code invoking the include will be a part of a web application running in the security context of the HTTP server process (Apache/WAMP as SYSTEM, IIS as ApplicationPoolIdentity / a service account, etc.), and in such case the attacker attains code execution with those privileges. In those scenarios the attacker will usually be able to trigger code execution by sending a web request by themselves. Otherwise, code execution will be delayed until another user or process invokes the code making the include (CLI applications are affected too).

Final notes on exploitability

The only Windows installations where this is NOT exploitable are: - installations where C:\php\pear exists (**not default**) with **non-default** permissions, preventing Authenticated Users from creating or changing files, - systems explicitly hardened to prevent Authenticated Users from creating directories in the C:\ root directory (**not default**), - the include_path php.ini variable is explicitly set to a **non-default** value, pointing at a location with secure permissions, - none of the deployed PHP applications have the exploitable structure of includes (depends on the environment).

Also, the attacker needs to know the file name/names of the existing second-order non-absolute path includes. For known software products this information is easy to obtain. An alternative approach is to use a dictionary of common include names and create hardlinks for them in C:\php\pear, all pointing at the malicious file. On Windows hardlinks can be created by non-administrative users if the target file is owned by the same user who requests hardlink creation. Alternatively, instead of hardlinks (which save disk space) a bunch of copies with different names can be created (and avoid potentially suspicious event of hardlink creation).

Notes on the affected scope

1. The vulnerable include pattern is a **foundational idiom** of the pre-Composer PHP era and remains prevalent in deployed WAMP/XAMPP/IIS-PHP installations.
2. Even modern Composer-based deployments frequently ship at least one third-party package in vendor/ that emits a bare-filename require, so the vulnerability class is **not limited to legacy targets**.
3. Any PHP tool that itself was distributed via PEAR (a substantial slice of the Windows PHP CLI toolchain) is essentially guaranteed to be exploitable when invoked, because those scripts *depend on* include_path resolution to load their own modules.

2.4 GnuPG

On Windows, GnuPG uses C:\ProgramData\GNU\etc\gnupg\ as its system-wide configuration directory (sysconfdir). This directory does not exist by default after installation from the regular EXE installers such as https://gnupg.org/ftp/gcrypt/binary/gnupg-w32-2.5.21_20260702.exe. Due to default Windows ProgramData DACL inheritance, BUILTIN\Users can create arbitrary subdirectories and files under C:\ProgramData.

An attacker can plant a gpg-agent.conf file containing pinentry-program C:\attacker\path\malicious.exe.

When any user on the system runs GPG and a passphrase prompt is triggered, gpg-agent launches the attacker-specified executable. Code executes in the context of the victim user.

In response to my report the maintainer decided to introduce a warning message as a form of mitigation: https://github.com/gpg/gnupg/commit/56eb3148c7b88eb1c0804141b58cb54b9007f48d .

2.5 wget (CVE-2026-94574)

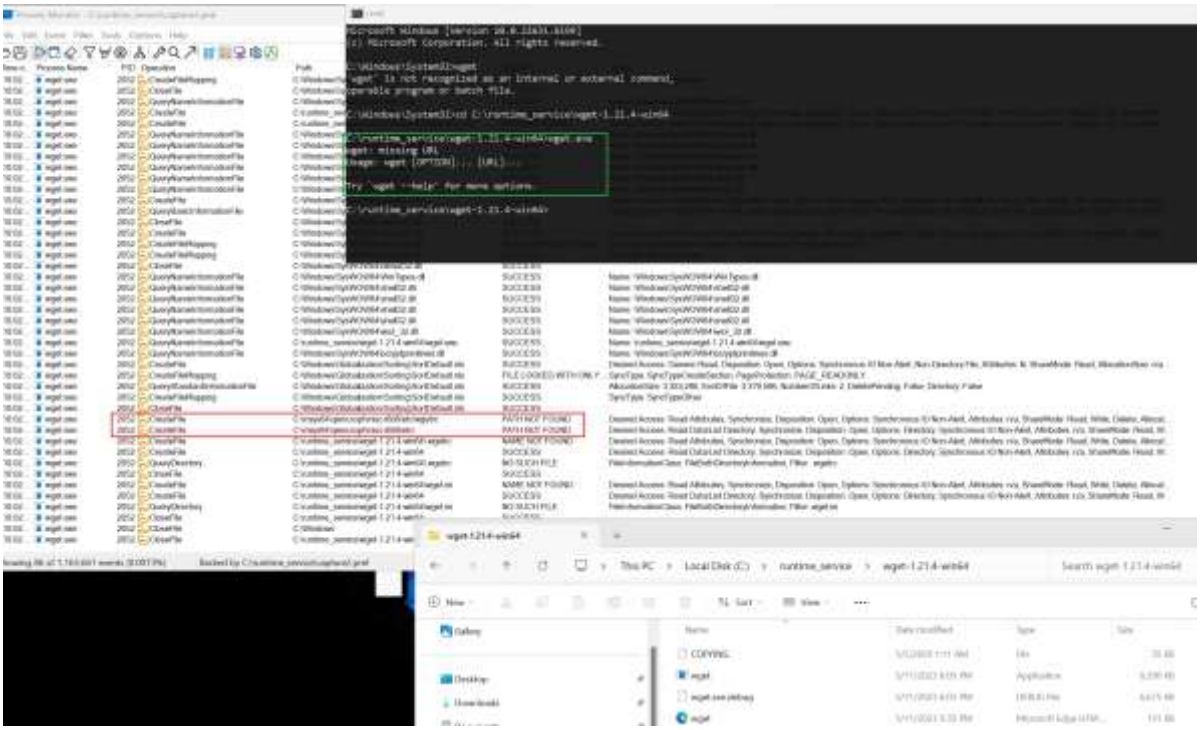
wget.exe (up to GNU Wget 1.21.4, the same wget version that gets installed via Chocolatey) has a hardcoded SYSTEM_WGETRC path compiled from the MSYS2/MinGW build environment. Depending on the build (x86 vs x64), it is one of:

C:\msys64\qemu\opt\misc-i686\etc\wgetrc (x86)
C:\msys64\qemu\opt\misc-x64\etc\wgetrc (x64)

This path is confirmed by wget.exe --version output:

Wgetrc: C:/msys64/qemu/opt/misc-i686/etc/wgetrc (system)
Wgetrc: C:/msys64/qemu/opt/misc-x64/etc/wgetrc

Procmon evidence:



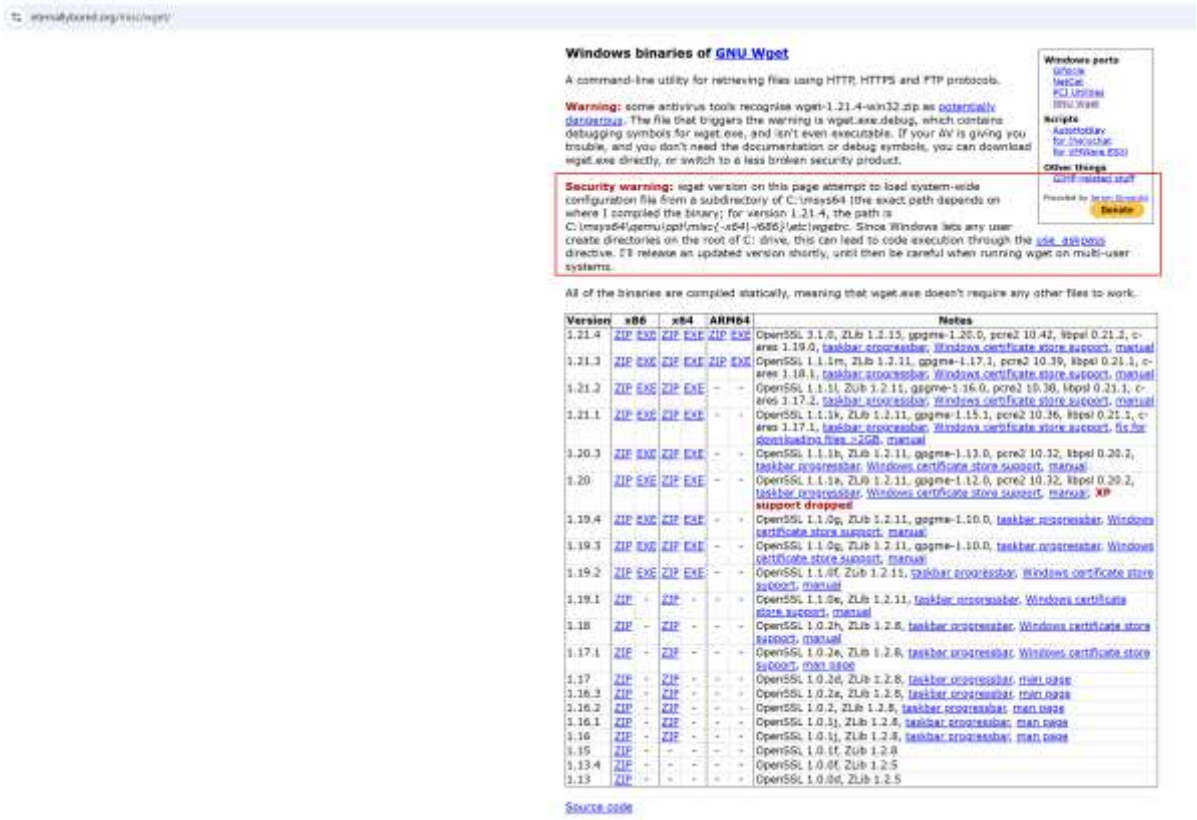
wget - PATH NOT FOUND

Code execution PoC

A wgetrc file containing the directive:
`use_askpass = C:\path\to\malicious.exe`
causes wget to execute the specified program whenever it needs to prompt for credentials (e.g., HTTP 401, --user= flag). The malicious program runs in the context of the user executing wget.

At the time of writing this the issue is not yet addressed, and the maintainer put a warning on the download page:

wget version on this page attempt to load system-wide configuration file from a subdirectory of C: (the exact path depends on where I compiled the binary; for version 1.21.4, the path is C:. Since Windows lets any user create directories on the root of C: drive, this can lead to code execution through the use_askpass directive. I'll release an updated version shortly, until then be careful when running wget on multi-user systems.



Windows binaries of GNU Wget

A command-line utility for retrieving files using HTTP, HTTPS and FTP protocols.

Warning: some antivirus tools recognise wget-1.21.4-win32.zip as potentially dangerous. The file that triggers the warning is wget.exe.debug, which contains debugging symbols for wget.exe, and isn't even executable. If your AV is giving you trouble, and you don't need the documentation or debug symbols, you can download wget.exe directly, or switch to a less broken security product.

Security warning: wget version on this page attempt to load system-wide configuration file from a subdirectory of C:\msys64 (the exact path depends on where I compiled the binary; for version 1.21.4, the path is C:\msys64\gemu\opt\mlocl-x64\-i686\)\etc\wgetrc. Since Windows lets any user create directories on the root of C: drive, this can lead to code execution through the use, system directive. I'll release an updated version shortly, until then be careful when running wget on multi-user systems.

All of the binaries are compiled statically, meaning that wget.exe doesn't require any other files to work.

Version	x86	x64	ARM64	Notes
1.21.4	ZIP EXE	ZIP EXE	ZIP EXE	OpenSSL 3.1.0, ZLib 1.2.13, gpgme-1.20.0, pcre2 10.42, libpsl 0.21.2, c-ares 1.19.0, taskbar progressbar, Windows certificate store support, manual
1.21.3	ZIP EXE	ZIP EXE	ZIP EXE	OpenSSL 1.1.1m, ZLib 1.2.11, gpgme-1.17.1, pcre2 10.39, libpsl 0.21.1, c-ares 1.18.1, taskbar progressbar, Windows certificate store support, manual
1.21.2	ZIP EXE	ZIP EXE	- -	OpenSSL 1.1.1l, ZLib 1.2.11, gpgme-1.16.0, pcre2 10.38, libpsl 0.21.1, c-ares 1.17.2, taskbar progressbar, Windows certificate store support, manual
1.21.1	ZIP EXE	ZIP EXE	- -	OpenSSL 1.1.1k, ZLib 1.2.11, gpgme-1.15.1, pcre2 10.36, libpsl 0.21.1, c-ares 1.17.1, taskbar progressbar, Windows certificate store support, fix for downloading files >2GB, manual
1.20.3	ZIP EXE	ZIP EXE	- -	OpenSSL 1.1.1h, ZLib 1.2.11, gpgme-1.13.0, pcre2 10.32, libpsl 0.20.2, taskbar progressbar, Windows certificate store support, manual
1.20	ZIP EXE	ZIP EXE	- -	OpenSSL 1.1.1a, ZLib 1.2.11, gpgme-1.12.0, pcre2 10.32, libpsl 0.20.2, taskbar progressbar, Windows certificate store support, manual; XP support dropped
1.19.4	ZIP EXE	ZIP EXE	- -	OpenSSL 1.1.0g, ZLib 1.2.11, gpgme-1.10.0, taskbar progressbar, Windows certificate store support, manual
1.19.3	ZIP EXE	ZIP EXE	- -	OpenSSL 1.1.0g, ZLib 1.2.11, gpgme-1.10.0, taskbar progressbar, Windows certificate store support, manual
1.19.2	ZIP EXE	ZIP EXE	- -	OpenSSL 1.1.0f, ZLib 1.2.11, taskbar progressbar, Windows certificate store support, manual
1.19.1	ZIP -	ZIP -	- -	OpenSSL 1.1.0e, ZLib 1.2.11, taskbar progressbar, Windows certificate store support, manual
1.18	ZIP -	ZIP -	- -	OpenSSL 1.0.2h, ZLib 1.2.8, taskbar progressbar, Windows certificate store support, manual
1.17.1	ZIP -	ZIP -	- -	OpenSSL 1.0.2e, ZLib 1.2.8, taskbar progressbar, Windows certificate store support, man page
1.17	ZIP -	ZIP -	- -	OpenSSL 1.0.2d, ZLib 1.2.8, taskbar progressbar, man page
1.16.3	ZIP -	ZIP -	- -	OpenSSL 1.0.2a, ZLib 1.2.8, taskbar progressbar, man page
1.16.2	ZIP -	ZIP -	- -	OpenSSL 1.0.2, ZLib 1.2.8, taskbar progressbar, man page
1.16.1	ZIP -	ZIP -	- -	OpenSSL 1.0.1j, ZLib 1.2.8, taskbar progressbar, man page
1.16	ZIP -	ZIP -	- -	OpenSSL 1.0.1j, ZLib 1.2.8, taskbar progressbar, man page
1.15	ZIP -	- -	- -	OpenSSL 1.0.1f, ZLib 1.2.8
1.13.4	ZIP -	- -	- -	OpenSSL 1.0.0f, ZLib 1.2.5
1.13	ZIP -	- -	- -	OpenSSL 1.0.0d, ZLib 1.2.5

Source code

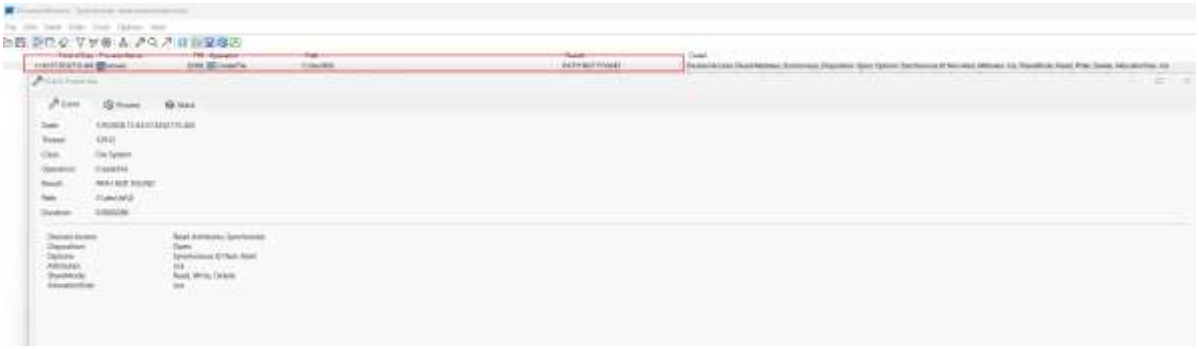
wget - warning on the website

2.6 Microsoft Core Utils tail.exe

In Microsoft Core Utils versions including 2026.9.3, the tail.exe utility can have its **standard output fully controlled by any member of the local Authenticated Users group**. Because tail.exe is a transparent conduit — it selects lines/bytes from its input and emits them verbatim — full control of its output means an unprivileged local user can force the command to emit arbitrary bytes, withhold legitimate data, or produce unbounded output, in security contexts other than their own (e.g., another user, a service, or a SYSTEM Scheduled Task that invokes tail.exe).

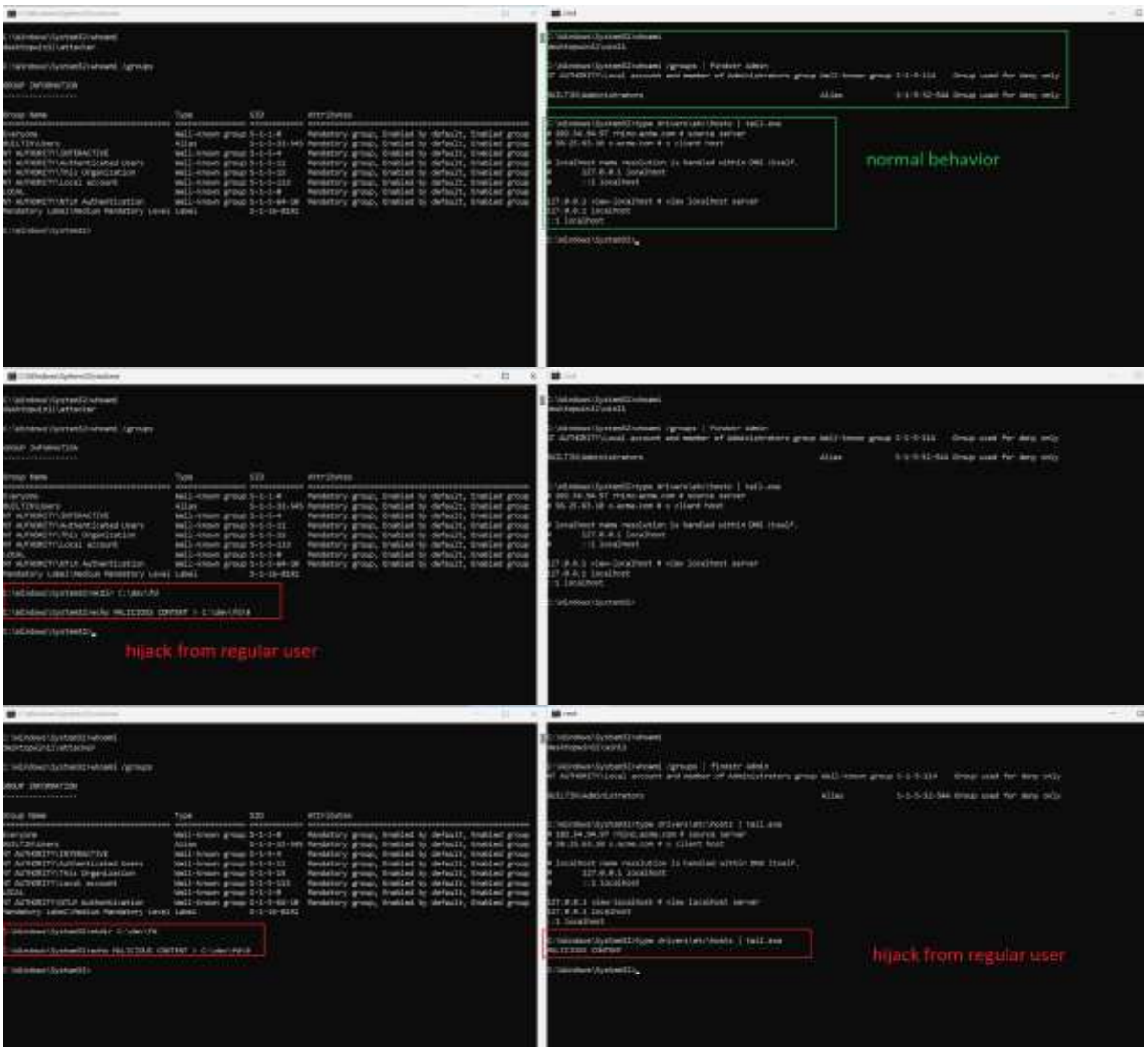
The root cause is tail's attempt to use a nonexistent C:\dev\fd\0 file (Windows translation of the native /dev/fd/0 standard input pseudo device) and, in case of its existence, its preference over the real standard input.

Here is what happens by default, when the file does not exist. The tool works properly and the nonexistent file is ignored:



Procmon evidence - when C:\0 does not exist

The screenshots below demonstrate a simple PoC. The attacker user creates the file and the victim user consumes it instead of the real standard input delivered to tail.exe:



PoC

This can be achieved by any member of the Authenticated Users group, as by default Windows allows all members of that group to create new directories on C:\.

The confirmed, deterministic impact is a local Denial of Service. It is also not hard to imagine scenarios in which this could lead to execution flow steering or affecting security decisions (if tail.exe is used to process log files).

So, this one is not as impactful as code execution, but as an example it is a quite interesting one.

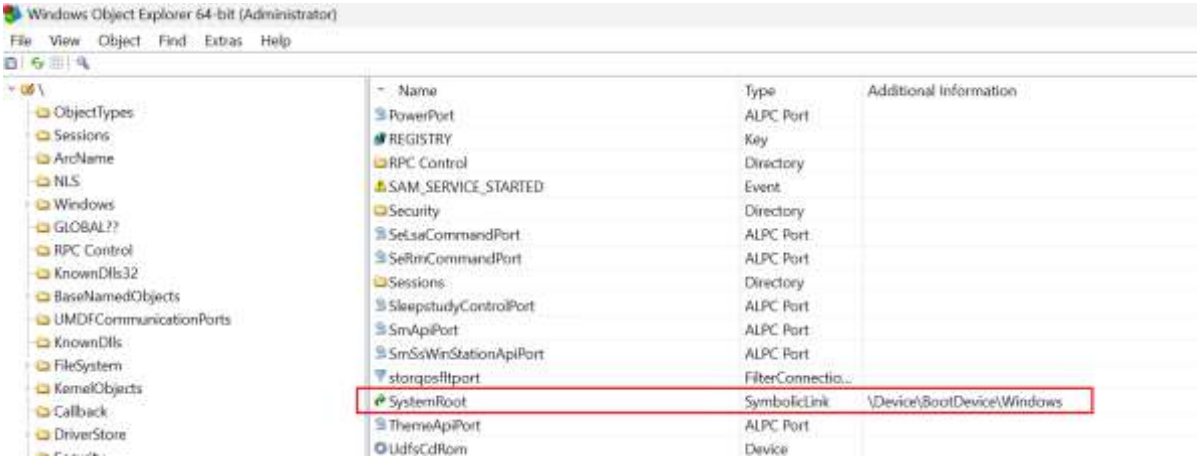
According to Microsoft, the issue does not meet their definition of a security vulnerability and is below threshold for servicing.

In this case we have a clear porting issue (Unix dependency and path not stripped from the code for a Windows product).

2.7 Undisclosed software product #1

Another very interesting example I observed (in a product whose name I am not going to disclose in this article) is an NT Object Manager path being fed to a Win32 path parser. A process running as NT AUTHORITY\SYSTEM makes references to nonexistent paths starting with C:\SystemRoot.

Obviously, the C:\SystemRoot folder does not exist by default, while SystemRoot is a well-known Object Manager symlink:



SystemRoot in WinObjEx

The affected software product and the impact are not relevant in this example. I am only providing it because it adds another interesting root cause. An OMNS path is passed to a regular WinAPI CreateFile call without the proper prefix, and gets misinterpreted into C:\SystemRoot. The event itself gets triggered from a scheduled task.

2.8 Undisclosed software product #2

Another example that needs mentioning is a software product running as a service (SYSTEM), which attempts to find an executable it intends to launch by improperly handling spaces in the original installation path - the exact same root cause as in unquoted service path vulnerabilities. The only difference between this example and actual unquoted service paths is that this particular improperly constructed path was referenced by an already running service process, and that the execution attempt resided in a code branch that could be triggered by a regular user via dedicated named pipe interaction.

I included this one only to exemplify another real-world root cause.

3 Cause and exploitability analysis

Let's break down the root causes and distinctive aspects of these issues.

3.1 Exploitability root cause

As a quick reminder - on Windows there are several common locations where non-administrative users can create new files and directories.
The most important ones are the C:\ root directory, C:\ProgramData and C:\Windows\TEMP.
Let's have a quick look at how they differ:

3.1.1 The C:\ root directory.

This is the most impactful and, based on my impression, the one mostly forgotten about.
The default DACL is:

BUILTIN\Administrators:(OI)(CI)(F)
NT AUTHORITY\SYSTEM:(OI)(CI)(F)
BUILTIN\Users:(OI)(CI)(RX)
NT AUTHORITY\Authenticated Users:(OI)(CI)(IO)(M)
NT AUTHORITY\Authenticated Users:(AD)
The NT AUTHORITY\Authenticated Users:(OI)(CI)(IO)(M) ACE means: (OI)Object Inherit — child files inherit this ACE (CI)Container Inherit — child folders inherit this ACE (M)Modify (read, write, execute, delete, change attributes; not change owner / change permissions)

This means that directory created on the C:\ root and its contents are almost fully controlled by any member of the Authenticated Users group. This means that **any executables deployed this way are world-writable by default**.
The NT AUTHORITY\Authenticated Users:(AD) ACE grants the Append Data (add subdirectory) permission. In practice it means that Authenticated Users can create new directories on the C:\ root. That is why references to nonexistent files can be satisfied without administrative privileges.

3.1.2 C:\ProgramData.

This directory is quite frequently used by software products. The default DACL is:

NT AUTHORITY\SYSTEM:(OI)(CI)(F)
BUILTIN\Administrators:(OI)(CI)(F)
CREATOR OWNER:(OI)(CI)(IO)(F)

BUILTIN\Users:(OI)(CI)(RX)
BUILTIN\Users:(CI)(WD,AD,WEA,WA)
In this case BUILTIN\Users by default cannot modify objects already created by someone else. But they can create new files and directories, which opens the way for exploitation of all file hijacking issues.

While ProgramData was never supposed to host executables and generally fewer and fewer software products use it for that purpose, it is very often used to store other types of files. That creates opportunities for data-based attacks - especially config hijacks, which are still quite common.

In addition to the GnuPG example, I have another one pending disclosure (code execution, affecting a very common software product that very often can be seen in multi-user environments). I also came across several instances with little to no impact.

One more example I want to mention is a pentest I did in a multi-user production environment, where creation of new directories on C:\ was blocked, but C:\ProgramData was at its default. The issue was a local privilege escalation from Builtin/Users to SYSTEM, caused by the installer's **failure to disable permission inheritance** on the C:\ProgramData\<PRODUCT> subdirectory. And while the developer was aware of the default permissions on ProgramData subdirectories in Windows in general, he was at the same time very surprised after receiving my report, because up until that point he had been convinced that the installer did remove the inheritance.

3.1.3 C:\Windows\TEMP

While none of the examples I demonstrate in this article involves C:\Windows\TEMP, those are easy to find (e.g. CVE-2019-17435) and should be mentioned for the sake of completeness.

The default DACL is:
BUILTIN\Users:(CI)(S,WD,AD,X)
BUILTIN\Administrators:(F)
BUILTIN\Administrators:(OI)(CI)(IO)(F)
NT AUTHORITY\SYSTEM:(F)
NT AUTHORITY\SYSTEM:(OI)(CI)(IO)(F)
CREATOR OWNER:(OI)(CI)(IO)(F)
All these different DACLs show that the default filesystem permissions on Windows are more nuanced than just the ability to create new directories on C:\ by NT AUTHORITY\Authenticated Users.

3.2 Existence root cause #1 - Unix paths ported to Windows

This is a pattern I have observed in multiple software products that originated from the Unix world. The failure to remove/avoid references to Unix paths when the code is running on Windows. The difference between the permission model defaults between the two systems turn trusted, root-only controlled locations such as /etc into their

Windows counterparts such as C:\etc, which do not exist but can be satisfied by any member of the Authenticated Users group.

Sometimes those incorrect references cannot be spotted with regular static analysis, **because they exist in external libraries and make it to the executable only during the build process**.

For example, the Windows versions of Terraform up until v1.15.6 have a built-in reference to the Unix uname command, making the executable try to find and launch "uname.exe" from the same directory where terraform.exe is located. While this one is not a security issue as long as the installation directory permissions prevent regular users from creating that file, I am including it as an example, because uname invocation was embedded in a dependency:

terraform.exe trying to invoke uname

Another example is a bundled build of OpenSSL used in a piece of security-related software running as a service (name redacted). OpenSSL attempts to read and parse files from a bunch of nonexistent paths that come to existence by translation of their original Unix versions - such as /etc/ssl/certs or /usr/lib/ssl/certs - into their Windows counterparts - C:\etc\ssl\certs or C:\usr\lib\ssl\certs:

Bundled OpenSSL build with Unix paths

I have even seen these failed Unix path references in software products we all know but would never suspect them to have had anything to do with the Unix world.

3.3 Existence root cause #2 - default paths

The hardcoded C:\Apache24\conf path in Apache Lounge and C:\php\pear in PHP are examples of assumptions about installation paths.

Sometimes the hardcoded path does not even have anything to do with the deployment directory. In one case (pending disclosure) the process searches for a config file in

C:\home\devuser\product\config, which is a combination of a Unix path ported to Windows and an environmental setting that should have never made it to release.

3.4 Existence root cause #3 - insecure installers

I have noticed that software installers are in a sense a grey area when it comes to the holistic picture of responsibility distribution.

If a product comes with an installer, it would be reasonable to assume that the installer will take care of securing the installation directory. But oftentimes the software manufacturer assumes that whoever deploys the software will do that. So, in reality automated installers quite often do not change the default directory permissions if when the software is deployed into a directory directly on C:\. Insecure by default applies to all software distributed in extractable ZIP packages. A very good example is the Terraform deployment tutorial, served as the top search result on Google.

Deployment into dedicated folders created directly in the root C:\ directory is very common (most likely because the path is shorter), and both EDR telemetry and known software defaults prove it. In those cases the software manufacturer assumes that the person installing the software will take care of securing the installation directory manually. Sometimes that part is only covered in the documentation, not the installer itself.

It is worth emphasizing that these assumptions usually do not have a tangible security impact, because most such deployments happen on single-user workstations (so there are no other Authenticated Users in the system). However, the default Windows permissions apply to servers as well. Multi-user environments such as jumphosts are at most risk (higher likelihood of exploitation, higher number of credentials to steal and move laterally). By extension the assumption made by manufacturers is that servers **should** be hardened by their administrators - but positive assumptions in security is what eventually leads to issues. And as we have already seen in the Exploitability root cause section, the default permissions on Windows are nuanced (e.g. C:\ vs C:\ProgramData) and it is unrealistic to assume that every user, developer and even administrator knows them by heart.

3.5 Existence root cause #4 - path construction failures

There are many reasons paths can get incorrectly constructed or **improperly used**.

API-level mismatch like in the case of Undisclosed software product #1 and issues with space-handling (Undisclosed software product #2) are just two examples of silent path construction failures that I have come across during runtime-driven research.

Other possible scenarios in this category I can think of are, for instance, CWD-related behaviors (e.g. drive-relative vs. drive-rooted paths; C:\Windows and C:Windows are not the same path), environment expansion used as path construction, and \\?\ / \\.\ / \??\ prefix confusions.

4 Source code != binary, binary != process

While investigating one of the examples above, I realized that I love reverse engineering so much that I even reverse engineer open source.

But jokes aside, this statement makes more sense after we realize that the actual subject of interest is not the source code, but the process - created from a compiled and linked binary, running in a live environment. And both the build process and the environment can introduce nuances that change the outcome.

Based on the cases studies provided above we can see a pattern. A combination of factors, among which only one exists in the source code and is not an issue alone from that perspective. Sometimes the other factor contributing to the security impact is environmental, other times it comes from dependency or mutually-exclusive assumptions.

I found most of these issues with an automated runtime setup. Static analysis took place only to establish the possible impact, after events of interest were already generated and captured. Initially I used Process Monitor but eventually switched to Dtrace. While Process Monitor is a great tool for quick manual investigations, I find DTrace much better-suited for automation. Hopefully I will soon be able to share my setup in its entirety.

Personally, I have always perceived static analysis as the best, deepest and most comprehensive way to learn about software and its potential weaknesses. I have always appreciated having the best of the two worlds while looking for vulnerabilities, but I always approached it by using static analysis as the starting point. Especially when my goal was to provide full coverage - as opposed to being opportunistic. A live installation served mostly as a helper to quickly identify the most obvious interaction vectors and to verify findings that static analysis brought. Several months ago, I was not even considering any research revolving around file-based hijacks. I found the Ghotscript issue “accidentally” (while examining a different software product that happens to use Ghostscript as a dependency), and only that gave me incentive to start looking for more in a deliberate way.

This experience made me realize that there are gaps which runtime-driven testing fills efficiently. Security issues that arise from subtle discrepancies between multiple interconnected systems have been known for a long time. Especially the ones revolving around protocol and format implementations. But even when just one computer system is involved, there are multiple subsystems it consists of. And runtime is where these systems meet.

Also, I am convinced that some of the examples demonstrated in this article were at some point highlighted by static analysis, but not prioritized either due to the missing environmental link, or due to assumptions that the production environment will be hardened by the administrator. Having multiple contributing factors, none of which alone seems like a problem, and each one is the responsibility of a different party, is what makes such issues more likely to occur.

5 Conclusion

While static analysis is the most efficient way to get deep coverage, runtime-driven testing can close the gap between expected behavior and what actually happens once code meets infrastructure. After all, "I don't always test my code, but when I do, I do it in production" is more than just a meme.

Until next time!

About Atos Group

Atos Group is a global leader in digital transformation with c. 56,000 employees and annual revenue of c. €7.2 billion (at the go-forward perimeter), operating in 54 countries under two brands - Atos for services and Eviden for products and systems. European number one in cybersecurity and a leader in cloud, Atos Group is committed to a secure and decarbonized future and provides tailored AI-powered, end-to-end solutions for all industries. Atos Group is listed on Euronext Paris.

Find out more about us

atos.net
atos.net/career