hijacking

Cmd Hijack - a command/argument confusion with path traversal in cmd.exe

Julian Horoszkiewicz
Jun 10, 2020 • 14 min read

This one is about an *interesting behavior* 🤭 I identified in cmd.exe in result of many weeks of intermittent (private time, every now and then) research in pursuit of some new OS Command Injection attack vectors.

So I was mostly trying to:

- find an encoding missmatch between some command check/sanitization code and the rest of the program, allowing to smuggle the ASCII version of the existing command separators in the second byte of a wide char (for a moment I believed I had it in the `StripQuotes` function - I was wrong ¯\_(ツ)_/¯),
- discover some hidden cmd.exe's counterpart of the unix shells' backtick operator,

- find a command separator alternative to |, & and \n - which long ago resulted in the discovery of an interesting and still alive, but very rarely occurring vulnerability - https://vuldb.com/?id.93602.

And I eventually ended up finding a command/argument confusion with path traversal ... or whatever the fuck this is 😜

For the lazy with no patience to read the whole thing, here comes the magic trick:

Tested on Windows 10 Pro x64 (Microsoft Windows [Version 10.0.18363.836]), cmd.exe version: 10.0.18362.449 (SHA256: FF79D3C4A0B7EB191783C323AB8363EBD1FD10BE58D8BCC96B07067743CA81D5). But should work with earlier versions as well... probably with all versions.

Some more context

Let's consider the following command line: `cmd.exe /c "ping 127.0.0.1"`, whereas `127.0.0.1` is the argument controlled by the user in an application that runs an external command (in this sample case it's *ping*). This exact syntax - with the command being preceded with the `/c` switch and enclosed in double quotes - is the default way cmd.exe is used by external programs to execute system commands (e.g. PHP `shell_exec()` function and its variants).

Now, the user can trick cmd.exe into running calc.exe instead of ping.exe by providing an argument like `127.0.0.1/../../../../../../../../../../windows/system32/calc.exe`, traversing the path to the executable of their choice, which cmd.exe will run instead of the ping.exe binary.

So the full command line becomes:

```
cmd.exe /c "ping 127.0.0.1/../../../../../../../../../../windows/system32/calc.exe"
```

The potential impact of this includes Denial of Service, Information Disclosure, Arbitrary Code Execution (depending on the target application and system).

Although I am fairly sure there are some other scenarios with OS command execution whereas a part of the command line comes from a different security context than the final command is executed with (Some services maybe? I haven't search myself yet) - anyway let's use a web application as an example.

Consider the following sample PHP code:

Due to the use of `escapeshellcmd()` it is not vulnerable to known command injection vectors (except for argument injection, but that's a slightly different story and does not allow RCE with the list of arguments *ping.exe* supports - no built-in execution arguments like *find*'s *-exec*).

And I know, I know, some of you will point out that in this case `escapeshellarg()` should be used instead - and yup, you would be right, especially since putting the argument in quotes in fact prevents this behavior, as in such case cmd.exe properly identifies the command to run (ping.exe). The trick does not work when the argument is enclosed in single/double quotes.

Anyway - the use of *escapeshellcmd()* instead of *escapeshellarg()* is very common. Noticed that while - after finding and registering CVE-2020-12669, CVE-2020-12742 and CVE-2020-12743 ended up spending one more week running automated source code analysis scans against more open source projects and manually following up the results - using my old evil SCA tool for PHP. Also that's what made me fed up with PHP again quite quickly, forcing me to get back to cmd.exe only to let me finally discover what this blog post is mostly about.

I am fairly sure there are applications vulnerable to this (doing OS command injection sanity checks, but failing to prevent path traversal and enclose the argument in quotes).

Also, the notion of similar behavior in other command interpreters is also worth entertaining.

An extended POC

Normal use:

Abuse:

Now, this is what normal use looks like in Sysmon log (process creation event):

So basically the child process (ping.exe) is created with command line equal to the value enclosed between the double quotes preceded by the `/c` switch from the parent process (cmd.exe) command line.

Now, the same for the above ipconfig.exe hijack:

And it turns out we are not limited to executables located in directories present in `%PATH%`. We can traverse to any location on the same disk.

Also, we are not limited to the *EXE* extension, neither to the list of "executable" extensions contained in the `%PATHEXT%` variable (which by default is `.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC` - basically these are the extensions cmd.exe will try to add to the name of the command if no extension is provided, e.g. when `ping` is used instead of explicit `ping.exe`). cmd.exe runs stuff regardless to the extension, something I noticed long ago (https://twitter.com/julianpentest/status/1203386223227572224).

And one more thing - more additional arguments between the original command and the smuggled executable path can be added.

Let's see all of this combined.

For the demonstrative purposes, the following C program was compiled and linked into a PE executable (it simply prints out its own command line):

Copied the EXE into *C:\xampp\tmp\cmd.png* (consider this as an example of ANY location a malicious user could write a file).

Action:

So we just effectively achieved an equivalent of actual (exec, not just read) PE Local File Inclusion in an otherwise-safe PHP ping script.

But I don't think that our options end here.

The potential for extending this into a full RCE without chaining with file upload/control

I am certain it is also possible to turn this into an RCE even without the possibility of fully/partially controlling any file in the target file system and deliver the payload *in the command line itself*, thus creating a sort of polymorphic malicious command line payload.

When running the target executable, cmd.exe passes to it the entire part of the command line following the `/c` switch.

For instance:

`cmd.exe /c "ping 127.0.0.1/../../../../../../../windows/system32/calc.exe"`

executes *c:\windows\system32\calc.exe* with command line equal `ping 127.0.0.1/../../../../../../../windows/system32/calc.exe`.

And, as presented in the extended POC, it is possible to hijack the executable even when providing multiple arguments, leading to command lines like:

`ping` **`THE PLACE FOR THE RCE PAYLOAD ARGS`** `127.0.0.1/../../path/to/lol.bin`

This is the command line `lol.bin` would be executed with. Finding a proxy execution LOLBin tolerant enough to invalid arguments (since we as attackers cannot fully control them) could turn this into a full RCE.
The LOLBin we need is one accepting/ignoring the first argument (which is the hardcoded command we cannot control, in our example "ping"), while also willing to accept/ignore the last one (which is the traversed path to itself). Something like https://lolbas-project.github.io/lolbas/Binaries/Ieexec/, but actually accepting multiple arguments while quietly ignoring the incorrect ones.

Also, I was thinking of powershell.

Running this:

`cmd.exe /c "ping ;calc.exe; 127.0.0.1/../../../../../../../../../windows/system32/WindowsPowerShell/v1.0/POWERSHELL.EXE"`

makes powershell start with command line of

`ping ;calc.exe 127.0.0.1/../../../../../../../../../../windows/system32/WindowsPowerShell/v1.0/POWERSHELL.EXE`

I expected it to treat the command line as a string of inline commands and run calc.exe after running ping.exe. Yes, I know, a semicolon is used here to separate ping from calc - but the semicolon character is NOT a command separator in cmd.exe, while it is in powershell (on the other hand almost all OS Command Injection filters block it anyway, as they are written universally with multiple platforms in mind - cause obviously the semicolon IS a command separator in unix shells).

A perfect supported syntax here would be some sort of simple base64-encoded code injection like powershell's `-EncodedCommand`, having found a way to make it work even when preceded with a string we cannot control. Anyway, this attempt led to powershell running in interactive mode instead of treating the command line as a sequence of inline commands to execute.

Anyway, at this point turning this into an RCE boils down to researching the behaviors of particular LOLbins, focusing on the way they process their command line, rather than researching cmd.exe itself (although yes, I also thought about self-chaining and abusing cmd.exe as the LOLbin for this, in hope for taking advantage of some nuances between the way it parses its command line when it does and when it does not start with the `/c` switch).

Stumbling upon and some analysis

I know this looks silly enough to suggest I found it while ramming that sample PHP code over HTTP with Burp while watching Procmon with proper filters... or something like that (which isn't such a bad idea by the way)... as opposed to writing a custom cmd.exe fuzzer (no, you don't need to tell me my code is far away from elegant, I did not expect anyone would read it neither that I would reuse it), then after obtaining rather boring and disappointing results, spending weeks on static analysis with Ghidra (thanks NSA, I am literally in love with this tool), followed up with more weeks of further work with Ghidra while simultaneously manually debugging with x64dbg while further expanding comments in the Ghidra project 😂

cmd.exe command line processing starts in the `CheckSwitches` function (which gets called from `Init`, which itself gets called from `main`). `CheckSwitches` is responsible for determining what switches (like `/c`, `/k`, `/v:on` etc.) cmd.exe was called with. The full list of options can be found in `cmd.exe /?` help (which by the way, to my surprise, reflects the actual functionality pretty well).

I spent a good deal of time analyzing it carefully, looking for hidden switches, logic issues allowing to smuggle multiple switches via the command line by jumping out of the double quotes, quote-stripping issues and whatever else would just manifest to me as I dug in.

The beginning of the CheckSwitches function after some naming editions and notes I took

If the `/c` switch is detected, processing moves to the actual command line enclosed in double quotes - which is the most common mode cmd.exe is used and the only one the rest of this write-up is about:

The same mode can be attained with the `/r` switch:

After some further logic, doing, among other things, parsing the quoted string and making some sanity fixes (like removing any spaces if any found from its beginning), a function with a very encouraging and self-explanatory name is called:

Disassembly view:

Decompiler view:

At this point it was clear it was high time for debugging to come into play.

By default x64dbg will set up a breakpoint at the entry point - `mainCRTStartup`.

This is a good opportunity to set an arbitrary command line:

Then start cmd.exe once again (`Debug-> Restart`).

We also set up a breakpoint on the top of the `SearchForExecutable` function, so we catch all its instances.

We run into the first instance of `SearchForExecutable`:

We can see that the double-quoted proper command line (after cmd.exe skips the preceding `cmd.exe /c`) along with its double quotes is held in `RBX` and `R15`. Also, the value on the top of the stack (right bottom corner) contains an address pointing at `CheckSwitches` - it's the saved `RET`. So we know this instance is called from `CheckSwitches`.

If we hit `F9` again, we will run into the second instance of `SearchForExecutable`, but this time the command line string is held in `RAX`, `RDI` and `R11`, while the call originates from another function named `ECWork`:

This second instance resolves and returns the full path to ping.exe.

Below we can see the body of the `ECWork` function, with a call to `SearchForExecutable` (marked black). This is where the `RIP` was at when the screenshot was taken - right before the second call of `SearchForExecutable`:

Now, on below screenshot the `SearchForExecutable` call already returned (note the full path to ping.exe pointed at with the address held in `R14`). Fifteen instructions later the `ExecPgm` function is called, using the newly resolved executable path to create the new process:

So - seeing `SearchForExecutable` being called against the whole `ping 127.0.0.1` string (uh yeah, those evil spaces) suggests potential confusion between the full command line and an actual file name... So this gave me the initial idea to check whether the executable could be hijacked by literally creating one under a name equal to the command line that would make it run:

Uh really? Interesting. I decided to have a look with Procmon in order to see what file names cmd.exe attempts to open with `CreateFile`:

So yes, the result confirmed opening a copy of calc.exe from the file literally named `ping .PNG` in the current working directory:

Now, interestingly, I would not see any results with this Procmon filter (*Operation* = *CreateFile*) if I did not create the file first...

One would expect to see cmd.exe mindlessly calling `CreateFile` against nonexistent files with names being various mutations of the command line, with `NAME NOT FOUND` result - the usual way one would search for potential DLL side loading issues... But NOT in this case - cmd.exe actually checks whether such file exists before calling `CreateFile`, by calling `QueryDirectory` instead:

For this purpose, in Procmon, it is more accurate to specify a filter based on the payload's unique magic string (like `PNG` in this case, as this would be the string we as attackers could potentially control) occurring in the *Path* property instead of filtering based on the *Operation*.

"So, anyway, this isn't very useful" - I thought and got back to x64dbg.

"We can only hijack the command if we can literally write a file under a very dodgy name into the target application's current directory... " - I kept thinking - *"... Current directory... u sure ONLY current directory?"* - and at this point my path traversal reflex lit up, a seemingly crazy and desperate idea to attempt traversal payloads against parts of the command line parsed by `SearchForExecutable`.

Which made me manually change the command line to `ping 127.0.0.1/../calc.exe` and restart debugging... while already thinking of modifying the cmd.exe fuzzer in order to throw a set payloads generated for this purpose with psychoPATH against cmd.exe... But that never happened because of what I saw after I hit `F9` one more time.

Below we can see x64dbg with cmd.exe ran with `cmd.exe /c "ping 127.0.0.1/../calc.exe"` command line (see `RDI`). We are hanging right after the second `SearchForExecutable` call, the one originating from the bottom of the `ECWork` function. Just few instructions before calling `ExecPgm`, which is about to execute the PE pointed by `R14`. The full path to `C:\Windows\System32\calc.exe` present `R14` is the result of the just-returned `SearchForExecutable("ping 127.0.0.1/../calc.exe")` call preceding the current `RIP`:

The traversal appears to be relative to a subdirectory of the current working directory (calc.exe is at `c:\windows\system32\calc.exe`):

"Or maybe this is just a result of a failed path traversal sanity check, only removing the first occurrence of `../` *?"* - I kept wondering.

So I dug further into the `SearchForExecutable` function, also trying to find the answer why variants of the argument created by splitting it by spaces are considered and why the most-to-the-right one is chosen first when found.

I narrowed down the culprit code to the instructions within the `SearchForExecutable` function, between the call of `mystrcspn` at *14000ff64* and then the call of the `FullPath` function at *14001005b* and `exists_ex` at *140010414:*

In the meantime I received the following feedback from Microsoft:

We do have a blog post that helps describe the behavior you have documented: [https://docs.microsoft.com/en-us/dotnet/standard/io/file-path-formats.](https://docs.microsoft.com/en-us/dotnet/standard/io/file-path-formats)

Cmd.exe first tries to interpret the whole string as a path: "ping 127.0.0.1/../../../../../../../../../../windows/system32/calc.exe" string is being treated as a relative path, so "ping 127.0.0.1" is interpreted as a segment in that path, and is removed due to the preceding "../" this should help explain why you shouldn't be able to use the user controlled input string to pass arguments to the executable.

There are a lot a cases that would require that behaviour, e.g. cmd.exe /c "....\Program Files (x86)\Internet Explorer\iexplore.exe" we wouldn't want that to try to run some program "....\Program" with the argument "Files (x86)\Internet Explorer\iexplore.exe".

It's only if the full string can't be resolved to a valid path, that it splits on spaces and takes everything before the first space as the intended executable name (hence why "ping 127.0.0.1" does work).

So yeah... those evil spaces and quoting.

From this point, I only escalated the issue by confirming the possibility of traversing to arbitrary directories as well as the ability to force execution of PE files with arbitrary extensions.

Interestingly, this slightly resembles the common unquoted service path issue, except that in this case the most-to-the-right variant gets prioritized.

The disclosure

Upon discovery I documented and reported this peculiarity to MSRC. After little less than six days the report was picked up and reviewed. About a week later Microsoft completed their assessment, concluding that this does not meet the bar for security servicing:

On one hand, I was little disappointed that Microsoft would not address it and I was not getting the CVE in cmd.exe I have wanted for some time.

On the other hand, at least nothing's holding me back from sharing it already and hopefully it will be around for some time so we can play with it 😀 It's not a vulnerability, it's a technique 😀

I would like thank Microsoft for making all of this possible - and for being nice enough to even offer me a review of this post! Which was completely unexpected, but obviously highly appreciated.

Some reflections

Researching stuff can sometimes appear to be a lonely and thankless journey, especially after days and weeks of seemingly fruitless drudging and sculpturing - but I realized this is just a short-sighted perception, whereas success is exclusively measured by the number of uncovered vulnerabilities/features/interesting behaviors (no point to argue about the terminology here 😀). In offensive security we rarely pay attention to the stuff we tried and failed, even though those failed attempts are equally important - as if we did not try, we would never know what's there (and risk false negatives). Curiosity and the need to know. And software is full of surprises.

Plus, simply dealing with a particular subject (like analyzing a given program/protocol/format) and gradually getting more and more familiar with it feeds our minds with new mental models, which makes us automatically come up with more and more ideas for potential bugs, scenarios and weird behaviors as we keep hacking. A journey through code accompanied by new inspirations, awarded with new knowledge and the peace of mind resulting from answering questions... sometimes ending with great satisfaction from a unique discovery.

Slipping through the cracks - the imperfections and nuances of CVE

Introduction This is mostly dedicated to people working with vulnerabilities affecting commonly used software: vulnerability management teams,...

Oct 24, 2024 · 13 min read

Breaking out from stripped tokens using process injection

Intro What I am going to showcase here isn't anything new, but it lays out the groundwork for what I am going to publish next, soon-ish. This...

Oct 13, 2024 · 22 min read