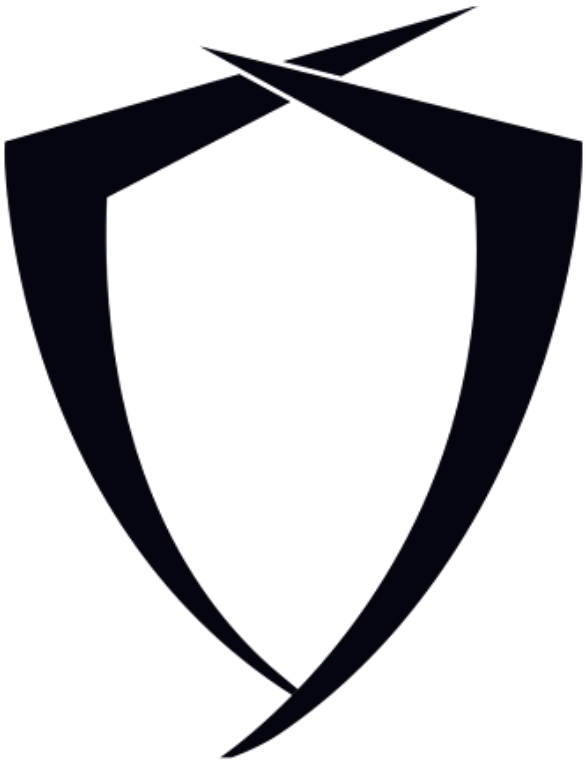


CODE WHITE

FINEST HACKING

FEBRUARY 4, 2016

Java and Command Line Injections in Windows

Everyone knows that incorporating user provided fragments into a command line is dangerous and may lead to command injection. That's why in Java many suggest using *ProcessBuilder* instead where the program's arguments are supposed to be passed discretely in separate strings.

However, in Windows, processes are created with a single command line string. And since there are different and seemingly confusing parsing rules for different runtime environments, proper quoting seems to be likewise complicated.

This makes Java for Windows still vulnerable to injection of additional arguments and even commands into the command line.

WINDOWS' *CREATEPROCESS* COMMAND LINES

In Windows, the main function for creating processes is the *CreateProcess* function. And in contrast to C API functions like *execve*, arguments are not passed separately as an array of strings but in a single command line. On the other side, the entry point function *WinMain* expects a single command line argument as well.

This circumstance requires the program to parse the command line itself for extracting the arguments. And although Windows provides a *CommandLineToArgvW* function and supports C and C++ API entry point functions where arguments are already parsed by the runtime and passed in a *argc*/*argv* style, the rules for quoting command line arguments with all their quirks can be quite confusing. And there is no definitive guide on how to quote properly, let alone something like a *ArgvToCommandLineW* function that does it for you. That's why many do it wrong, as "Everyone quotes command line arguments the wrong way" by Daniel Colascione observes.

You should definitely read the latter two linked pages first to understand the rest of this blog post.

For testing, we'll use the following Java class, which utilizes *ProcessBuilder* as suggested:

```
1  import java.io.IOException;
2  import java.io.InputStream;
3  import java.util.Arrays;
4
5  public class CommandLine {
6
7          public static void main(String[] args) throws IOException {
8                  args = new String[] {
9                          "C:\\Temp\\PrintArguments.exe",
10                         "arg 1",
11                         "arg 2",
12                         "arg 3"
13                 };
14                 start(args);
15         }
16
17         public static void start(String[] args) throws IOException {
18                 InputStream in = new ProcessBuilder(Arrays.asList(args)).start().
19
20                 byte[] buffer = new byte[1024];
21                 int len;
22                 while ((len = in.read(buffer)) != -1) {
23                     System.out.write(buffer, 0, len);
```

```
                }
        }

}
```

CommandLine.java hosted with ❤ by **GitHub** **view raw**

The resulting *CreateProcess* command line can be observed with the Windows Sysinternals' Process Monitor. And for how the command line gets parsed, you can use the following program, which prints the results of both the parsing of C command-line arguments (via the *argv* function parameter) and of the parsing of C++ command-line arguments (via *CommandLineToArgvW* function).

```
#include <Windows.h>

int main(int argc, char *argv[], char **envp)
{
        int nArgs;
        LPWSTR * szArgs;

        printf("argv arguments:\n");
        for (int i = 0; i < argc; ++i) printf("%u: '%s'\n", i, argv[i]);

        printf("CommandLineToArgvW arguments:\n");
        szArgs = CommandLineToArgvW(GetCommandLineW(), &nArgs);
        if (szArgs == NULL) return -1;
        for (int i = 0; i < nArgs; ++i) wprintf(L"%u: '%ws'\n", i, szArgs[i]);
        LocalFree(szArgs);

        return 0;
}
```

PrintArguments.c hosted with ❤ by **GitHub** **view raw**

This already produces different and frankly surprising results in some cases:

```
Arguments in Java:
  0: 'C:\Temp\PrintArguments.exe'
  1: '"""""'
CreateProcess command line:
  "C:\Temp\PrintArguments.exe" """""
argv arguments:
  0: 'C:\Temp\PrintArguments.exe'
  1: '"""'
CommandLineToArgvW arguments:
  0: 'C:\Temp\PrintArguments.exe'
```

```
  1: '"'

Arguments in Java:
  0: 'C:\Temp\PrintArguments.exe'
  1: '" "" "'
CreateProcess command line:
  "C:\Temp\PrintArguments.exe" " "" "
argv arguments:
  0: 'C:\Temp\PrintArguments.exe'
  1: ' " '
CommandLineToArgvW arguments:
  0: 'C:\Temp\PrintArguments.exe'
  1: ' "'
  2: ''

Arguments in Java:
  0: 'C:\Temp\PrintArguments.exe'
  1: '"" '
CreateProcess command line:
  C:\Temp\PrintArguments.exe """ "
argv arguments:
  0: 'C:\Temp\PrintArguments.exe'
  1: '" '
CommandLineToArgvW arguments:
  0: 'C:\Temp\PrintArguments.exe'
  1: '"'
  2: ''

Arguments in Java:
  0: 'C:\Temp\PrintArguments.exe'
  1: '"" '
CreateProcess command line:
  C:\Temp\PrintArguments.exe """" "
argv arguments:
  0: 'C:\Temp\PrintArguments.exe'
  1: '"'
  2: ''
CommandLineToArgvW arguments:
  0: 'C:\Temp\PrintArguments.exe'
  1: '" '
```

examples.txt hosted with ❤ by **GitHub** **view raw**

The last two are remarkable as one additional quotation mark swaps the results of *argv* and *CommandLineToArgvW*.

JAVA'S COMMAND LINE GENERATION IN WINDOWS

With the knowledge of how *CreateProcess* expects the command line arguments to be quoted, let's see how Java builds the command line and quotes the arguments for Windows.

If a process is started using *ProcessBuilder*, the arguments are passed to the static method *start* of *ProcessImpl*, which is a platform-dependent class. In the Windows implementation of *ProcessImpl*, the *start* method calls the private constructor of *ProcessImpl*, which creates the command line for the *CreateProcess* call.

```
1  private ProcessImpl(String cmd[],
2                      final String envblock,
3                      final String path,
4                      final long[] stdHandles,
5                      final boolean redirectErrorStream)
6      throws IOException
7  {
8      String cmdstr;
9      SecurityManager security = System.getSecurityManager();
10     boolean allowAmbiguousCommands = false;
11     if (security == null) {
12         allowAmbiguousCommands = true;
13         String value = System.getProperty("jdk.lang.Process.allowAmbiguousCommands
14         if (value != null)
15             allowAmbiguousCommands = !"false".equalsIgnoreCase(value);
16     }
17     if (allowAmbiguousCommands) {
18         /* 331-344 skipped; legacy mode */
19     } else {
20         /* 346-383 skipped; strict mode */
21     }
22
23     handle = create(cmdstr, envblock, path,
24                     stdHandles, redirectErrorStream);
25     /* 389-418 skipped */
26 }
```

ProcessImpl.java hosted with ❤ by **GitHub** **view raw**

In the private constructor of *ProcessImpl*, there are two operational modes: the legacy mode and the strict mode. These are the result of issues caused by changes to *Runtime.exec*. The legacy mode is only performed if there is no *SecurityManager* present and the property *jdk.lang.Process.allowAmbiguousCommands* is not set to *false*.

The Legacy Mode

In the legacy mode, the first argument (i. e., the program to execute) is quoted if required and then the command line is created using *createCommandLine*.

```
// Legacy mode.

// Normalize path if possible.
String executablePath = new File(cmd[0]).getPath();

// No worry about internal, unpaired ["], and redirection/piping.
if (needsEscaping(VERIFICATION_LEGACY, executablePath) )
    executablePath = quoteString(executablePath);

cmdstr = createCommandLine(
    //legacy mode doesn't worry about extended verification
    VERIFICATION_LEGACY,
    executablePath,
    cmd);
```

ProcessImpl_legacy.java hosted with ❤ by **GitHub** **view raw**

The *needsEscaping* method checks whether the value is already quoted using *isQuoted* or wraps it in double quotes if it contains certain characters.

```
private static boolean needsEscaping(int verificationType, String arg) {
    // Switch off MS heuristic for internal ["].
    // Please, use the explicit [cmd.exe] call
    // if you need the internal ["].
    //    Example: "cmd.exe", "/C", "Extended_MS_Syntax"

    // For [.exe] or [.com] file the unpaired/internal ["]
    // in the argument is not a problem.
    boolean argIsQuoted = isQuoted(
        (verificationType == VERIFICATION_CMD_BAT),
        arg, "Argument has embedded quote, use the explicit CMD.EXE call.");

    if (!argIsQuoted) {
        char testEscape[] = ESCAPE_VERIFICATION[verificationType];
        for (int i = 0; i < testEscape.length; ++i) {
            if (arg.indexOf(testEscape[i]) >= 0) {
                return true;
            }
        }
    }
    return false;
```

```
}
```

ProcessImpl_needsEscaping.java hosted with ❤ by **GitHub** **view raw**

```
private static boolean isQuoted(boolean noQuotesInside, String arg,
        String errorMessage) {
    int lastPos = arg.length() - 1;
    if (lastPos >=1 && arg.charAt(0) == '"' && arg.charAt(lastPos) == '"') {
        // The argument has already been quoted.
        if (noQuotesInside) {
            if (arg.indexOf('"', 1) != lastPos) {
                // There is ["] inside.
                throw new IllegalArgumentException(errorMessage);
            }
        }
        return true;
    }
    if (noQuotesInside) {
        if (arg.indexOf('"') >= 0) {
            // There is ["] inside.
            throw new IllegalArgumentException(errorMessage);
        }
    }
    return false;
}
```

ProcessImpl_isQuoted.java hosted with ❤ by **GitHub** **view raw**

The vertification type *VERIFICATION_LEGACY* passed to *needsEscaping* makes *noQuotesInside* in *isQuoted* being *false*, which would allow quotation marks within the path. It also makes *needsEscaping* test for space and tabulator characters only.

But let's take a look at the *createCommandLine* method, which creates the command line:

```
private static String createCommandLine(int verificationType,
                                        final String executablePath,
                                        final String cmd[])
{
    StringBuilder cmdbuf = new StringBuilder(80);

    cmdbuf.append(executablePath);

    for (int i = 1; i < cmd.length; ++i) {
        cmdbuf.append(' ');
        String s = cmd[i];
```

```
12          if (needsEscaping(verificationType, s)) {
13              cmdbuf.append('"').append(s);
14
15              // The code protects the [java.exe] and console command line
16              // parser, that interprets the [\"] combination as an escape
17              // sequence for the ["] char.
18              //     http://msdn.microsoft.com/en-us/library/17w5ykft.aspx
19              //
20              // If the argument is an FS path, doubling of the tail [\]
21              // char is not a problem for non-console applications.
22              //
23              // The [\"] sequence is not an escape sequence for the [cmd.exe]
24              // command line parser. The case of the [""] tail escape
25              // sequence could not be realized due to the argument validation
26              // procedure.
27              if ((verificationType != VERIFICATION_CMD_BAT) && s.endsWith("\\")) {
28                  cmdbuf.append('\\');
29              }
30              cmdbuf.append('"');
31          } else {
32              cmdbuf.append(s);
33          }
34      }
35      return cmdbuf.toString();
36  }
```

ProcessImpl_createCommandLine.java hosted with ❤ by **GitHub** **view raw**

Again, with the verification type *VERIFICATION_LEGACY* the *needsEscaping* method only returns *true* if it is already wrapped in quotes (regardless of any quotes within the string) or if it is not wrapped in quotes and contains a space or tabulator character (again, regardless of any quotes within the string). If it needs quoting, it is simply wrapped in quotes and a possible trailing backslash is doubled.

Ok, so far, so good. Now let's recall Daniel Colascione's conclusion:

> **Do not:**
>
> 1. Simply add quotes around command line argument arguments without any further processing.
> 2. […]

Yes, exactly. This can be exploited to inject additional arguments:

- A value that is *considered to be quoted*:

    - passed argument value: `"arg 1" "arg 2" "arg 3"`
    - quoted argument value: no quoting needed as it's "already quoted"
    - parsed argument values: [`'arg 1'`, `'arg 2'`, `'arg 3'`]
- A value that is *considered to be not quoted but requires quotes*:
    - passed argument value: `"arg" 1" "arg 2" "arg 3`
    - quoted argument value: `""arg" 1" "arg 2" "arg 3"`
    - parsed argument values: [`'arg 1'`, `'arg 2'`, `'arg 3'`]

The Strict Mode

In the strict mode, things are a little different.

```
String executablePath;
try {
    executablePath = getExecutablePath(cmd[0]);
} catch (IllegalArgumentException e) {
    // Workaround for the calls like
    // Runtime.getRuntime().exec("\"C:\\Program Files\\foo\" bar")

    // No chance to avoid CMD/BAT injection, except to do the work
    // right from the beginning. Otherwise we have too many corner
    // cases from
    //    Runtime.getRuntime().exec(String[] cmd [, ...])
    // calls with internal ["] and escape sequences.

    // Restore original command line.
    StringBuilder join = new StringBuilder();
    // terminal space in command line is ok
    for (String s : cmd)
        join.append(s).append(' ');

    // Parse the command line again.
    cmd = getTokensFromCommand(join.toString());
    executablePath = getExecutablePath(cmd[0]);

    // Check new executable name once more
    if (security != null)
        security.checkExec(executablePath);
}

// Quotation protects from interpretation of the [path] argument as
// start of longer path with spaces. Quotation has no influence to
// [.exe] extension heuristic.
```

```
32  cmdstr = createCommandLine(
33          // We need the extended verification procedure for CMD files.
34          isShellFile(executablePath)
35              ? VERIFICATION_CMD_BAT
36              : VERIFICATION_WIN32,
37          quoteString(executablePath),
38          cmd);
```

ProcessImpl_strict.java hosted with ❤ by **GitHub** **view raw**

If the path contains a quote, *getExecutablePath* throws an exception and the *catch* block is executed where *getTokensFromCommand* tries to extract the path.

However, the rather interesting part is that *createCommandLine* is called with a different verification type based on whether *isShellFile* denotes it as a shell file.

```
1  private boolean isShellFile(String executablePath) {
2      String upPath = executablePath.toUpperCase();
3      return (upPath.endsWith(".CMD") || upPath.endsWith(".BAT"));
4  }
```

ProcessImpl_isShellFile.java hosted with ❤ by **GitHub** **view raw**

But I'll come back to that later.

With the verification type *VERIFICATION_WIN32*, *noQuotesInside* is still *false* and both injection examples mentioned above work as well.

However, if *needsEscaping* is called with the verification type *VERIFICATION_CMD_BAT*, *noQuotesInside* becomes *true*. And without being able to inject a quote we can't escape the quoted argument.

CREATEPROCESS' SILENT *CMD.EXE* PROMOTION

Remember the *isShellFile* checked the file name extension for `.cmd` and `.bat`? This is due to the fact that *CreateProcess* executes these files in a *cmd.exe* shell environment:

> […] the decision tree that *CreateProcess* goes through to run an image is as follows:
>
> - […]
> - If the file to run has a `.bat` or `.cmd` extension, the image to be run becomes *Cmd.exe*, the Windows command prompt, and *CreateProcess* restarts at Stage 1. (The name of the batch file is passed as the first parameter to *Cmd.exe*.)
> - […]

— Windows Internals, 6th edition (Part 1)

That means a '`file.bat …`' becomes '`C:\Windows\system32\cmd.exe /c "file.bat …"`' and an additional set of quoting rules would need to be applied to avoid command injection in the command line interpreted by *cmd.exe*.

However, since Java does no additional quoting for this implicit *cmd.exe* call promotion on the passed arguments, injection is even easier: `&calc&` does not require any quoting and will be interpreted as a separate command by *cmd.exe*.

This works in the legacy mode just like in the strict mode if we make *isShellFile* return *false*, e. g., by adding whitespace to the end of the path, which tricks the *endsWith* check but are ignored by *CreateProcess*.

CONCLUSION

Command line parsing in Windows is not consistent and therefore the implementation of proper quoting of command line argument even less. This may allow the injection of additional arguments.

Additionally, since *CreateProcess* implicitly starts `.bat` and `.cmd` in a *cmd.exe* shell environment, even command injection may be possible.

As a sample, Java for Windows fails to properly quote command line arguments. Even with *ProcessBuilder* where arguments are passed as a list of strings:

- Argument injection is possible by providing an argument containing further quoted arguments, e. g., '`"arg 1" "arg 2" "arg 3"`'.
- On *cmd.exe* process command lines, a simple '`&calc&`' alone suffices.

Only within the most strictly mode, the *VERIFICATION_CMD_BAT* verification type, injection is not possible:

- Legacy mode:
  - *VERIFICATION_LEGACY*: There is no *SecurityManager* present and *jdk.lang.Process.allowAmbiguousCommands* is not explicitly set to *false* (no default set)
    - allows argument injection
    - allows command injection in *cmd.exe* calls (explicit or implicit)
- Strict mode:
  - *VERIFICATION_CMD_BAT*: Most strictly mode, file ends with `.bat` or `.cmd`

    - does not allow argument injection
    - does not allow command injection in *cmd.exe* calls
  - *VERIFICATION_WIN32*: File does not end with `.bat` or `.cmd`
    - allows argument injection
    - allows command injection in *cmd.exe* calls (explicit or implicit)

However, Java's check for switching to the *VERIFICATION_CMD_BAT* mode can be circumvented by adding whitespace after the `.bat` or `.cmd`.

Posted by Markus Wulftange at February 04, 2016
Tags Vulnerability Details

Newer Post Home Older Post

2020 © Code White GmbH. Powered by Blogger.