Flatt SECURITY

Flatt SECURITY April 9, 2024

BatBadBut: You can't securely execute commands on Windows

RyotaK

BatBadBut: You can't securely execute commands on Windows

Posted on April 9, 2024 • 10 minutes • 2113 words

Table of contents

Introduction

Hello, I'm RyotaK (@ryotkak), a security engineer at Flatt Security Inc.

Recently, I reported multiple vulnerabilities to several programming languages that allowed an attacker to perform command injection on Windows when the specific conditions were satisfied.
Today, affected vendors published advisories of these vulnerabilities [1], so I'm documenting the details here to provide more information about the vulnerabilities and minimize the confusion regarding the high CVSS score.

TL;DR

The **BatBadBut** is a vulnerability that allows an attacker to perform command injection on Windows applications that indirectly depend on the `CreateProcess` function when the specific conditions are satisfied.

`CreateProcess()` implicitly spawns `cmd.exe` when executing batch files (`.bat`, `.cmd`, etc.), even if the application didn't specify them in the command line.
The problem is that the `cmd.exe` has complicated parsing rules for the command arguments, and programming language runtimes fail to escape the command arguments properly.
Because of this, it's possible to inject commands if someone can control the part of command arguments of the batch file.

The following simple Node.js code snippet, for example, may pop calc.exe on the server machine:

```
const { spawn } = require('child_process');
const child = spawn('./test.bat', ['<your-input-here>']);
```

This happens only if a batch file is explicitly specified in the command line passed to `CreateProcess()`, and it doesn't happen when a `.exe` file is specified.
However, since Windows includes `.bat` and `.cmd` files in the `PATHEXT` environment variable by default, some runtimes execute batch files against the developers' intention if there is a batch file with the same name as the command that the developer intended to execute. So, even the following snippet may lead to arbitrary command executions whereas it doesn't include .bat or .cmd explicitly:

```
cmd := exec.Command("test", "<your-input-here>")
cmd.Run()
```

Exploitation of these behaviors is possible when the following conditions are satisfied:

- The application executes a command on Windows
- The application doesn't specify the file extension of the command, or the file extension is `.bat` or `.cmd`
- The command being executed contains user-controlled input as part of the command arguments

- The runtime of the programming language fails to escape the command arguments for `cmd.exe` properly[2]

By exploiting these behaviors, arbitrary command execution might be possible.
I created a flowchart to determine if your applications are affected by this vulnerability, so please refer to Appendix A if you are unsure whether you are affected or not, and refer to Appendix B for the status of the affected programming languages.

CVSS Score

First of all, I have to mention that you shouldn't apply the CVSS score of library vulnerabilities to your application directly.
The user guide of CVSS v3.1 states that the CVSS score of a library should be calculated based on the worst-case scenario, and this is why the recent vulnerabilities for programming languages got high scores despite the requirement of specific conditions.

Instead of applying the CVSS score directly, you should recalculate the score based on the specific implementation:
https://www.first.org/cvss/v3.1/user-guide#3-7-Scoring-Vulnerabilities-in-Software-Libraries-and-Similar

Technical Details

While I'm not a fan of naming vulnerabilities that are not internet-breaking, I'm a fan of puns, so I decided to call this vulnerability `BatBadBut` because it's about **bat**ch files and **bad**, **but** not the worst.

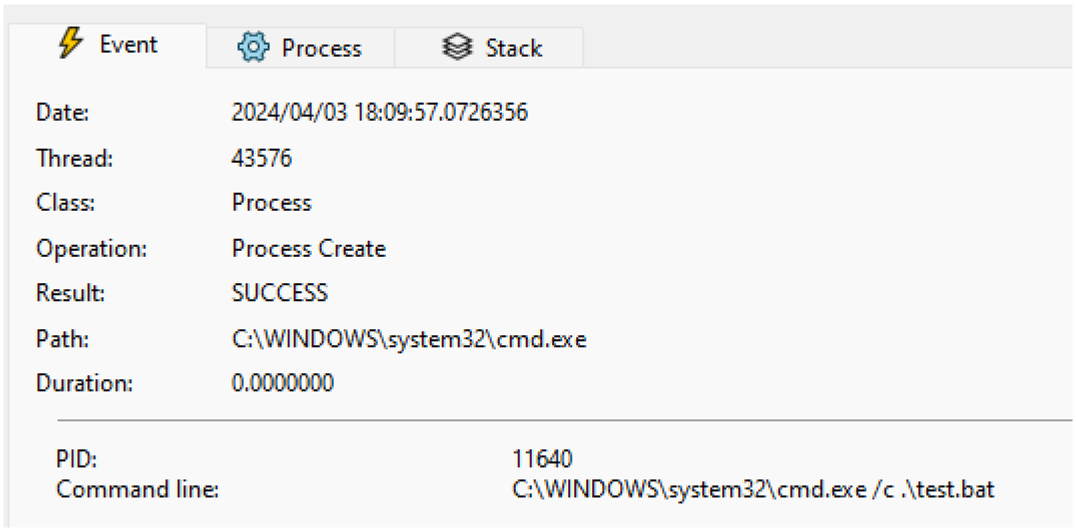
From this section, I'll explain the technical side of `BatBadBut` and why the command injection is possible.
Please note that some of the code snippets don't work on the latest version of runtimes, as some affected vendors already patched the issue.

Root Cause

The root cause of `BatBadBut` is the overlooked behavior of the `CreateProcess` function on Windows.
When executing batch files with the `CreateProcess` function, Windows implicitly spawns `cmd.exe` because Windows can't execute batch files without it.

For example, the following code snippet spawns `C:\Windows\System32\cmd.exe /c .\test.bat` to execute the batch file `test.bat`:

```
wchar_t arguments[] = L".\\test.bat";
STARTUPINFO si{};
PROCESS_INFORMATION pi{};
CreateProcessW(nullptr, arguments, nullptr, nullptr, false, 0, nullptr, nullptr,
```


Event Process Stack

```
Date:          2024/04/03 18:09:57.0726356
Thread:        43576
Class:         Process
Operation:     Process Create
Result:        SUCCESS
Path:          C:\WINDOWS\system32\cmd.exe
Duration:      0.0000000

PID:           11640
Command line:  C:\WINDOWS\system32\cmd.exe /c .\test.bat
```

While this isn't a problem itself, the issue arises when the programming language wraps the `CreateProcess` function and adds the escaping mechanism for the command arguments.

Wrapping `CreateProcess`

Most programming languages provide a function to execute a command, and they wrap the `CreateProcess` function to provide a more user-friendly interface.
For example, the `child_process` module in Node.js[3] wraps the `CreateProcess` function and provides a way to execute a command with arguments like the following:

```
const { spawn } = require('child_process');
const child = spawn('echo', ['hello', 'world']);
```

As you can see in the above code snippet, the `spawn` function takes the command and arguments as separate arguments.
It then internally escapes arguments to pass them to the `CreateProcess` function.

src/win/process.c line 444-518

```
/*
 * Quotes command line arguments
 * Returns a pointer to the end (next char to be written) of the buffer
 */
WCHAR* quote_cmd_arg(const WCHAR *source, WCHAR *target) {
  [...]

  /*
   * Expected input/output:
   *   input : hello"world
   *   output: "hello\"world"
   *   input : hello""world
   *   output: "hello\"\"world"
   *   input : hello\world
   *   output: hello\world
   *   input : hello\\world
   *   output: hello\\world
   *   input : hello\"world
   *   output: "hello\\\"world"
   *   input : hello\\"world
   *   output: "hello\\\\\"world"
   *   input : hello world\
   *   output: "hello world\\"
   */

  *(target++) = L'"';
  start = target;
  quote_hit = 1;

  for (i = len; i > 0; --i) {
    *(target++) = source[i - 1];

    if (quote_hit && source[i - 1] == L'\\') {
      *(target++) = L'\\';
    } else if(source[i - 1] == L'"') {
      quote_hit = 1;
      *(target++) = L'\\';
    } else {
      quote_hit = 0;
    }
  }
  target[0] = L'\0';
  _wcsrev(start);
  *(target++) = L'"';
  return target;
}
```

Most developers expect that the `spawn` function properly escapes the command arguments, and it's true in most cases.[4]

However, as I mentioned earlier, the `CreateProcess` function implicitly spawns `cmd.exe` when executing batch files.
And unfortunately, the `cmd.exe` has different escaping rules compared to the usual escaping mechanism.

Parsing rule of `cmd.exe`

Most shells for Unix-like systems have similar (or the same) escaping rules; backslashes (`\`) are used as an escape character.
So, if you want to escape a double quote (`"`) inside of a double-quoted string, you can use the backslash like the following:

```
echo "Hello \"World\""
```

Using backslash as the escape character seems to be a de facto standard, and other things like JSON or YAML also use it.

However, when you execute the following command on the command prompt, `calc.exe` will be executed:

```
echo "\"&calc.exe"
```

This is because the command prompt doesn't use the backslash as an escape character, and uses the caret (`^`) instead.

Back to the `child_process` example, it escapes the double quotes (`"`) in command arguments using a backslash (`\`).
Due to the escaping rules of `cmd.exe` mentioned above, this escaping is not sufficient when executing the batch file, so the following snippet spawns `calc.exe` even though the argument is separated properly, and the `shell` option[5] is not enabled:

```
const { spawn } = require('child_process');
const child = spawn('./test.bat', ['"&calc.exe']);
```

Because of this behavior, a malicious command line argument might be able to perform command injection, and this is the main problem of `BatBadBut`.

Mitigation

Escaping double quotes?

The problem here is that the double-quoted string is broken by the double quote inside of the string.
So, it seems that escaping double quotes (`"`) with a caret (`^`) is sufficient to prevent the command injection.[6]

But in fact, that is not enough to prevent the command injection.
Surprisingly, the command prompt parses and expands variables (e.g., `%PATH%`) before any other parsing.

This means that the following command will execute `calc.exe` although the `&calc.exe` is inside of the double-quoted string:

```
SET VAR=^"
echo "%VAR%&calc.exe"
```

While the default environment variables of Windows don't contain the double quote (`"`) in their value, there is a special variable called `CMDCMDLINE`, that contains the command line used to start the current command prompt session.

Assuming that the following command is executed on the PowerShell, `"C:\WINDOWS\system32\cmd.exe" /c "echo %CMDCMDLINE%"` will be printed:

```
cmd.exe /c "echo %CMDCMDLINE%"
```

And, by using the variable substring extraction in the command prompt, it's possible to extract the double quote (`"`) from this variable.
So, the following command spawns `calc.exe` when executed on PowerShell:

```
cmd.exe /c 'echo "%CMDCMDLINE:~-1%&calc.exe"'
```

Due to this behavior, escaping double quotes with a caret is insufficient to prevent the command injection when executing the batch file, and requires further escaping. I'll explain about it in the next section.

As a Developer

Since not all programming languages patched the issue[2], you should be careful when executing commands on Windows.

As a developer who executes commands on Windows, but doesn't want to execute batch files, you should always specify the file extension of the command.
For example, the following code snippet may execute `test.bat` instead of `test.exe` if the user places `test.bat` in the directory included in the `PATH` environment variable:

```
cmd := exec.Command("test", "arg1", "arg2")
```

To prevent this, you should always specify the file extension of the command like the following:

```
cmd := exec.Command("test.exe", "arg1", "arg2")
```

If you want to execute batch files, and your runtime doesn't escape the command arguments properly for the batch file, you must escape user-controlled input before using it as command arguments.

Since spaces can't be escaped properly outside of the double-quoted string[7], you have to use double quotes to wrap the command arguments.
However, inside the double-quoted string, `%` can't be escaped properly[8].

To solve this situation, the following tricky escaping is required:[9]

1. Disable the automatic escaping that uses the backslash (`\`) provided by the runtime.
2. Apply the following steps to each argument:
    1. Replace percent sign (`%`) with `%%cd:~,%`.
    2. Replace the backslash (`\`) in front of the double quote (`"`) with two backslashes (`\\`).
    3. Replace the double quote (`"`) with two double quotes (`""`).
    4. Remove newline characters (`\n`).
    5. Enclose the argument with double quotes (`"`).

By replacing `%` with `%%cd:~,%`, `%cd:~,%` will be expanded to an empty string, and the command prompt fails to expand the actual variable, so the `%` will be treated as a normal character.

Please note that if delayed expansion is enabled via the registry value `DelayedExpansion`, it must be disabled by explicitly calling `cmd.exe` with the `/V:OFF` option.
Also, note that the escaping for `%` requires the command extension to be enabled. If it's disabled via the registry value `EnableExtensions`, it must be enabled with the `/E:ON` option.

As a User

To prevent the unexpected execution of batch files, you should consider moving the batch files to a directory that is not included in the `PATH` environment variable.

In this case, the batch files won't be executed unless the full path is specified, so the unexpected execution of batch files can be prevented.

As a Maintainer of the runtime

If you maintain a runtime of a programming language, I'd recommend you implement an additional escaping mechanism for batch files.
Even if you don't want to fix it on the runtime layer, you should at least document the issue and provide a proper warning to the users, as this problem is not well-known.

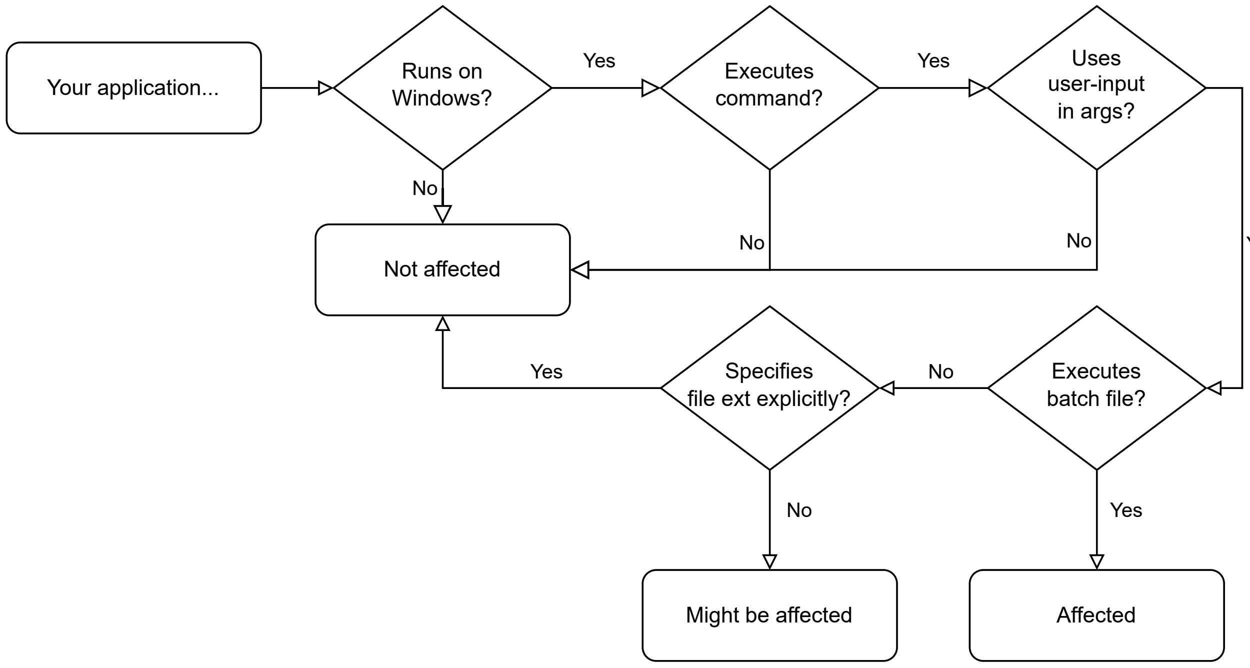
Conclusion

In this article, I explained the technical details of `BatBadBut`, a vulnerability that allows an attacker to perform command injection on Windows when the specific conditions are met.
As I mentioned several times in this article, this issue doesn't affect most applications, but in case you are affected, you should properly escape the command arguments manually.

I hope that this article helps you to understand the severity of this issue and mitigate the issue properly.

Appendix

Appendix A: Flowchart to determine if your applications are affected



Appendix B: Status of the affected programming languages

Project	Status
Erlang	Documentation update
Go	Documentation update
Haskell	Patch available
Java	Won’t fix
Node.js	Patch will be available
PHP	Patch will be available
Python	Documentation update
Ruby	Documentation update
Rust	Patch available

1. The advisory on CERT/CC will be available soon ↩

2. Please refer to the Appendix B for the status of the affected programming languages. ↩ ↩
3. As I use Node.js mostly, I'm using it as an example here. However, the issue is not specific to Node.js, and it affects other programming languages as well. ↩
4. In fact, many programming languages guarantee that the command arguments are escaped properly, and/or don't use shell. ↩
5. When the `shell` option is disabled, the `child_process` module doesn't spawn `cmd.exe` and directly spawns the command instead. However, Windows implicitly spawns `cmd.exe` when executing batch files, so the `shell` option is silently ignored when executing batch files. ↩
6. Of course, you need to escape the caret itself. ↩
7. When executing `.\\test.bat arg1^ arg2`, `arg1` and `arg2` will be recognized as separate arguments. ↩
8. `.\\test.bat "100^%"` will be recognized as `100^%` instead of `100%`. ↩
9. While the testing shows that this prevents the command injection, I'm still unsure if this escaping is the best way to prevent it, so if you are aware of a better way, please let me know. ↩