

Command injection: how it works, what are the risks, and how to prevent it | Snyk

Liran Tal November 24, 2020

6-7 minutes

Command injection attacks—also known as operating system command injection attacks—exploit a programming flaw to execute system commands without proper input validation, escaping, or sanitization, which may lead to arbitrary commands executed by a malicious attacker.

What are the risks of command injections?

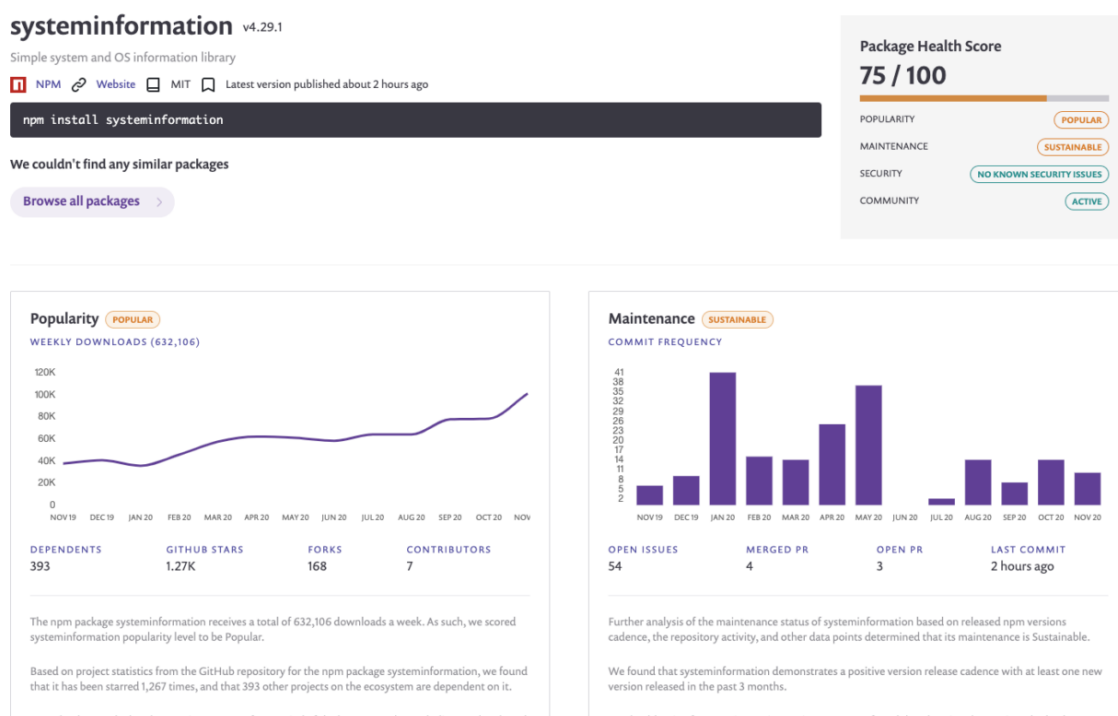
Depending on the setup of the application and the process configuration that executes it, a command injection vulnerability could lead to privilege escalation of the process or to spawn a

remote reverse shell that allows complete interaction by a malicious party.

How do command injection attacks work?

To understand programming flaws related to OS command injection attacks, let's explore a variety of command injection vulnerabilities that were discovered in Node.js based applications.

[systeminformation](#) is an Operating System (OS) information library that spans more than 500,000 downloads a week with regular maintenance (commits) and a community around it, as we can see using the [Snyk Advisor dependency health](#) page:



However, [all versions](#) of [systeminformation](#) below version 4.27.11 are vulnerable to OS command injection, due to several vulnerable methods to which the library exposes the users.

Let's have a look at the vulnerable `inetChecksite()` function code. Can you spot the security vulnerability in this code that shows a command injection example?

```
    let urlSanitized =
util.sanitizeShellString(url).toLowerCase()
    if (urlSanitized) {
        let t = Date.now();
        if (_linux || _freebsd ||
_openbsd || _netbsd || _darwin ||
_sunos) {
            let args = ' -I --connect-
timeout 5 -m 5 ' + urlSanitized + '
2>/dev/null | head -n 1 | cut -d " "
-f2';
            let cmd = 'curl';
            exec(cmd + args, function
(error, stdout) {
```

...

I can draw several conclusions from this code snippet:

- A `url` parameter is being sanitized. Perhaps I can bypass the sanitization functions with something the author didn't take into account?
- An insecure operating system process spawning API is used: `exec ()` which concatenates commands and their arguments into one shell command that is executed. This is also the reason the author had to resort to their own command line argument sanitization.

The `urlSanitized` variable is concatenated as part of the arguments provided to the CLI.

That last point about `urlSanitized` is where this security vulnerability lies. The command itself, referenced with the `cmd` variable is the networking util `curl`. What if we could simply pass a command argument to `curl` as the URL?

One may attempt command injection using the following proof of concept, with the URL provided to the vulnerable function, as shown in the first argument passed to the `inetChecksite ()`

function. The following is a command injection example:

```
const si =  
require('systeminformation');  
si.inetChecksite("https://snyk.io;  
touch /tmp/abc", (result) =>  
console.log(result));
```

That however doesn't work in this case, because this npm package sanitized dangerous user input such as the semicolon (;).

However, command line arguments usually don't require special characters, and we could provide a URL as user input such as `https://snyk.io -o hihi` which includes a space and concatenates a curl positional argument `-o` which outputs the result of the website into a file that we control.

Here is another command injection example of how that exploitation might take place and is indeed working as expected with a vulnerable version of `systeminformation@4.27.10`:

```
const si =  
require('systeminformation');
```

```
si.inetChecksite("https://snyk.io -o  
hihi", (result) =>  
console.log(result));
```

This commit addressed sanitization in a way that aims to further limit any such command injection vulnerabilities:

```

8  lib/internet.js
@@ -34,7 +34,13 @@ function inetChecksite(url, callback) {
34     return new Promise((resolve) => {
35         process.nextTick(() => {
36
37 -         const urlSanitized =
38             util.sanitizeShellString(url).toLowerCase();
39
40         let result = {
41             url: urlSanitized,
42             ok: false,
43
44 +         let urlSanitized =
45 +             util.sanitizeShellString(url).toLowerCase();
46 +             urlSanitized = urlSanitized.replace(/ /g, '');
47 +             urlSanitized = urlSanitized.replace(/\$/g, '');
48 +             urlSanitized = urlSanitized.replace(/\./g, '');
49 +             urlSanitized = urlSanitized.replace(/\(/g, '');
50 +             urlSanitized = urlSanitized.replace(/\)/g, '');
51 +             urlSanitized = urlSanitized.replace(/{/g, '');
52 +             urlSanitized = urlSanitized.replace(/}/g, '');
53
54         let result = {
55             url: urlSanitized,
56             ok: false,
57
58         }
59     })
60 }

```

Command injection cheatsheet

We can learn much from other command injection examples, where we identified security vulnerabilities in open source packages and how they mitigated the issue, such as the following vulnerabilities [here](#) and [here](#).

To prevent command injection attacks, consider the following practices:

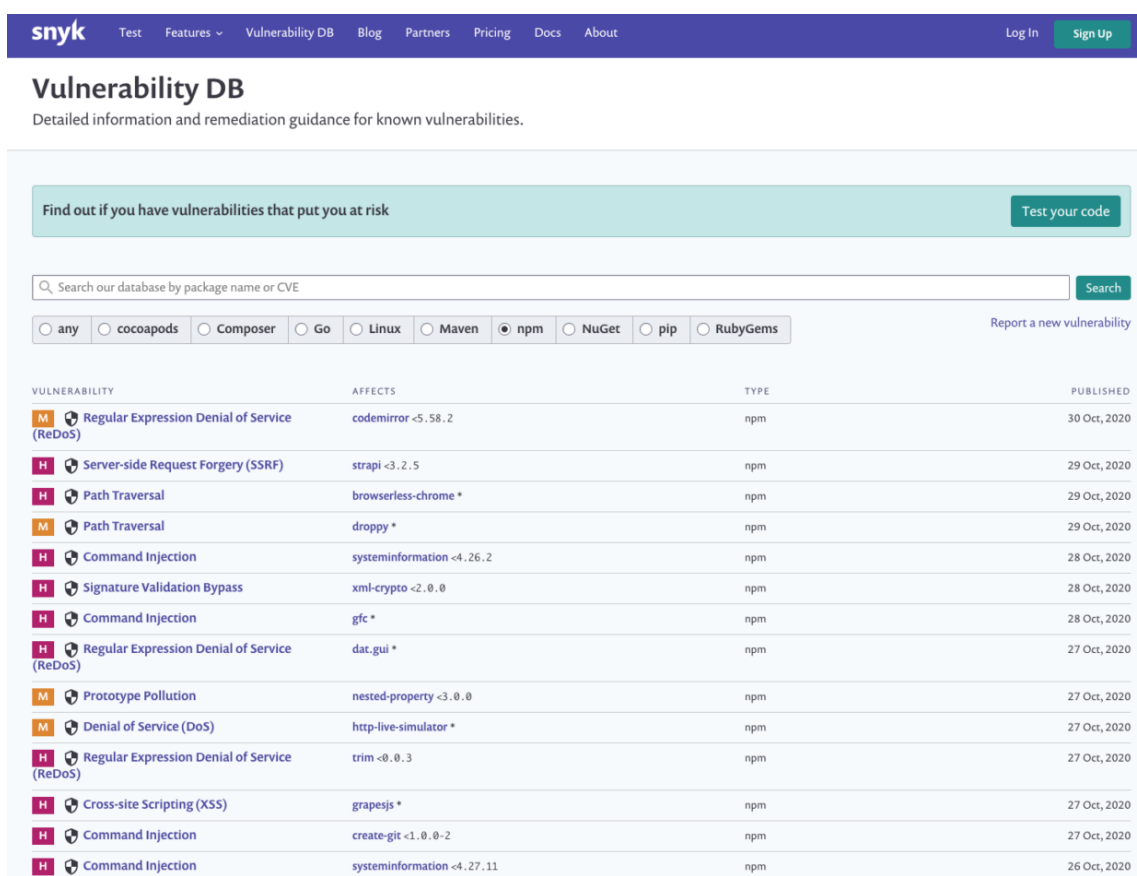
1. Do not allow any user input to commands your application is executing.

2. Only use secure APIs for executing commands, such as [execFile\(\)](#). Unlike other APIs, it accepts a command as the first parameter and an array of command line arguments as the second function parameter. This behavior ensures that the command itself has to be a valid program, unrelated to dangerous input.
3. When using `execFile()`, make sure the user has no control over the program name. Furthermore, ensure you are mapping user input to command arguments so that user input isn't passed as-is into program execution.
4. As a further act of precaution, validate that user input is conforming to the expected form. For example, if a function parameter expects a URL, validate that the input received is a valid URL. This would have prevented the above-mentioned [command injection vulnerability in npm package systeminformation](#).





























OWASP Top 10 command injection attacks

As a testament of how common OS command

injection attacks are, I looked up all publicly known security vulnerabilities that the Snyk Security Team published for October 2020, filtered for the [JavaScript ecosystem on npm](#). As you can see, quite a few Command injection security vulnerabilities have been identified for npm packages in the last week of October alone.



The screenshot shows the Snyk Vulnerability DB interface. At the top, there's a navigation bar with links like Test, Features, Vulnerability DB, Blog, Partners, Pricing, Docs, and About. Below this, the 'Vulnerability DB' section is highlighted, with a subtitle 'Detailed information and remediation guidance for known vulnerabilities.' A search bar is present with the text 'Find out if you have vulnerabilities that put you at risk' and a 'Test your code' button. Below the search bar, there's a filter section with radio buttons for various package managers: any, cocoapods, Composer, Go, Linux, Maven, npm (selected), NuGet, pip, and RubyGems. A 'Report a new vulnerability' link is also visible. The main content is a table of vulnerabilities.

VULNERABILITY	AFFECTS	TYPE	PUBLISHED
  Regular Expression Denial of Service (ReDoS)	codemirror <5.58.2	npm	30 Oct, 2020
  Server-side Request Forgery (SSRF)	strapi <3.2.5	npm	29 Oct, 2020
  Path Traversal	browserless-chrome *	npm	29 Oct, 2020
  Path Traversal	dropgy *	npm	29 Oct, 2020
  Command Injection	systeminformation <4.26.2	npm	28 Oct, 2020
  Signature Validation Bypass	xml-crypto <2.0.0	npm	28 Oct, 2020
  Command Injection	gfc *	npm	28 Oct, 2020
  Regular Expression Denial of Service (ReDoS)	dat.gui *	npm	27 Oct, 2020
  Prototype Pollution	nested-property <3.0.0	npm	27 Oct, 2020
  Denial of Service (DoS)	http-live-simulator *	npm	27 Oct, 2020
  Regular Expression Denial of Service (ReDoS)	trim <0.0.3	npm	27 Oct, 2020
  Cross-site Scripting (XSS)	grapesjs *	npm	27 Oct, 2020
  Command Injection	create-git <1.0.0-2	npm	27 Oct, 2020
  Command Injection	systeminformation <4.27.11	npm	26 Oct, 2020

To conclude, command injection vulnerabilities are more common than you'd think and there's a good reason injection attacks have been infamously featured on [OWASP Top 10 web security risks](#)—they are common and take many forms, such as [SQL injection attacks](#).

I highly recommend connecting your [git repositories with Snyk to test and fix](#) for command injection vulnerabilities. Maybe Snyk will find others too? Snyk is free and it's a quick setup to get your projects monitored!