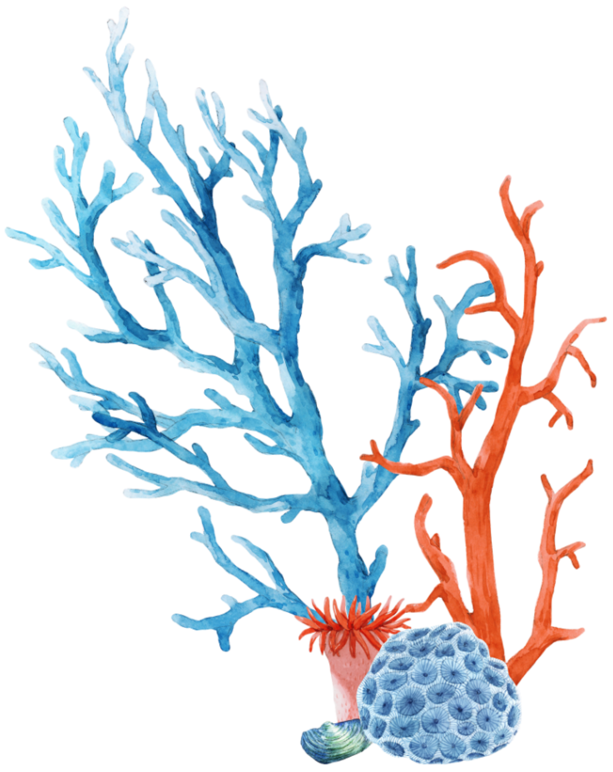
Node.js Secure Coding

Defending Against
Command Injection Vulnerabilities



LIRAN TAL

Node.js Secure Coding: Defending Against Command Injection Vulnerabilities

Liran Tal

Version v1.1, 01.05.2023:

Table of Contents

Node.js Secure Coding: Defending Against Command Injection Vulnerabilities
by Liran Tal

Revision history:

1. 2023-05-01
    a. New Appendix chapter includes self-assessment questions, reviews of closed-source and open-source real-world command injection vulnerability implications, and CVE list.
    b. Chapter 2: Argument Injection features citation of prior research.
2. 2023-04-07
    a. First edition.

This book is for sale at https://www.nodejs-security.com

Preface

Learn about secure coding practices with Node.js based on real-world CVE vulnerabilities in popular open-source npm packages.

This book takes an adventure-based approach to application security learning, where you will be playing detective who unravels the mysteries of common security vulnerabilities. Through these exercises you will learn about secure coding practices, and how to avoid security pitfalls that software developers and open-source maintainers get caught with.

Senior software engineers often recite how one of the most critical skills you should have as an engineer is the ability to read code. The more you read, the easier it becomes for you to understand code and the more context you gain. This book focuses exactly on that - reading vulnerable code, so we can learn from it. This activity creates patterns that our brain learns to identify and that later quickly turn into red flags that we detect and apply in our day-to-day programming and code review routines.

What you gain to learn

Designed for software developers and security professionals interested in command injection, this book provides a comprehensive understanding of the topic. It also demonstrates its impact and concerns on web application security.

Through insecure coding practices found in vulnerable open-source npm packages, this book examines the security aspects affecting JavaScript and Node.js applications. Developers of other languages such as Python will find references to insecure code and best practices relatively easy to transfer to other server-side languages and software ecosystems.

By completing this book you stand to gain:

- A high level of security expertise on the topic of command injection vulnerabilities.
- An understanding of application security jargon and conventions associated with security vulnerabilities management and severity classification.
- How real-world software libraries were found to be vulnerable and their methods of fixing security issues.
- Adopting a security-first mindset to recognize patterns of insecure code.
- Secure coding best practices to avoid command injection security vulnerabilities.

- Proficiency in performing secure code reviews as they apply to concerns and the scope of command injection security vulnerabilities.

Software developers

Software developers who build web applications, and specifically those who practice server-side JavaScript development on-top of the Node.js runtime will greatly benefit from the secure coding practices learned in this book.

As a software developer, you will engage in step-by-step code review of real-world popular libraries and their vulnerable code, through which you will investigate how security vulnerabilities manifest and understand the core reasons that lead to a security risk.

By reviewing code used in real-world software libraries, you will learn to recognize patterns of insecure code. In addition, you will learn secure coding best practices for working with system processes.

Security practitioners

Security professionals who wish to learn and investigate the source of insecure code and security implications concerned with vulnerable open-source and third-party libraries that make up an application’s software composition analysis (SCA).

How to read this book?

This book primarily focuses on the following knowledge-base sections:

- Introduction to application security
- A primer on command injection
- Chapters that review security vulnerabilities in-depth

If you have a high level of familiarity and understanding of application security concepts such as OWASP, NVD, and other security jargon then you can skip the ***Introduction to application security concepts***.

For readers who have an in-depth understanding of command injection vulnerabilities, such as those who have prior experience fixing them as a developer, or disclosing a command injection vulnerability through a bug bounty program, you can skip the command injection primer. Keep in mind, the command injection introduction chapter provides an elaborate foundation of different types and other insightful security considerations. It can still be effective educational content even for experienced practitioners.

At the core of this book is a deep-dive into real-world security vulnerabilities reviews. Each vulnerability that we review is assigned a security identifier, such as a CVE, and has impacted real-world npm packages, some of which you might even be using.

About the Author

Liran Tal is an accomplished software developer, respected security researcher, and prominent advocate for open-source software in the JavaScript community. He has earned recognition as a ***GitHub Star***, in part for his tireless efforts to educate developers and for his contributions to developing essential security tools and resources that help JavaScript and Node.js developers create more secure applications.

His leadership in open-source security extends to meaningful contributions to OWASP projects, recording supply chain security incidents at the CNCF, and various OpenSSF initiatives. His contributions to the Node.js community have been widely recognized, including being honored with the ***OpenJS Foundation’s Pathfinder for Security award*** for his significant contributions to advance the state of Node.js security. In his role as a security analyst in the Node.js Foundation’s Security Working Group, Liran reviewed hundreds of vulnerability reports for npm packages and created processes for responsible security disclosures and vulnerability triage.

Liran is also an accomplished security researcher and has disclosed security vulnerabilities in various open-source software projects, including being credited with CVEs impacting npm packages. His work on supply chain security research, including Lockfile Injection, was presented at Black Hat Europe 2021 cybersecurity conference.

As an experienced author and educator, Liran has written several widely respected books on software security. These include "Serverless Security" published by O’Reilly, as well as the self-published titles "Essential Node.js Security" and "Web Security: Learning HTTP Security Headers". He is passionate about sharing his knowledge and occasionally speaks on software security topics at academic institutions, such as presenting to students at the Electrical and Computer Engineering School at Purdue University.

Since joining Snyk, Liran has made a significant impact as a developer advocate, empowering developers with the knowledge and tools needed to build and deploy secure software at scale. His contributions to the developer community have been instrumental in advancing the state of application security and strengthening the adoption of secure coding practices.

Chapter 1

Introduction to Application Security

It’s necessary for software developers to understand the terminology used by security professionals, become aware of their standards and comprehend the role they play in application security. Doing so can assist in assimilating information on secure coding, which is a fundamental component of IT security.

Learnings

By the end of this chapter, you should be able to answer questions such as:

- What is a CVE and how is a CWE related to it?
- What is the OWASP Top 10?
- What is NVD?
- What is a source-to-sink?
- What is an attack vector?

1.1. Application Security Organizations

The following bodies of work are actively referenced and used as application security resources. They provide security tools, frameworks, documentation, libraries, working groups, events, industry-accepted standards, and maintain a security vulnerability database.

1.1.1. OWASP

The Open Web Application Security Project (OWASP) is a non-profit organization that aims to improve the quality of software on the internet through active work to make it more secure. The OWASP Foundation provides resources to help security professionals and developers create secure software. This extends to guides, tools, and documentation; developers should be aware of the Open Web Application Security Project (OWASP), as it is a widely recognized and respected source of information on software security.

In addition to the OWASP Top 10, developers should also be aware of other OWASP resources, such as the OWASP Application Security Verification Standard (ASVS) and the OWASP Secure Coding Practices Quick Reference Guide. These resources provide detailed guidance on how to write secure code and adhere to secure coding practices.

It is essential for software developers to familiarize themselves with OWASP and make use of its resources to develop secure applications. This will ensure that any potential vulnerabilities are eliminated and organizations or people can be spared from expensive security breaches.

Some notable examples of OWASP-related security resources for Node.js developers:

- OWASP Top 10 - known commonly as a security weaknesses awareness document, the OWASP Top 10 is a list of the most common and most critical web application security risks. It however doesn't aim to provide an exhaustive list or claim that one weakness is more dangerous than another. The list is curated by OWASP Foundation members and other guests who are invited to share their expertise. It is reviewed every few years to make updates to the list. It provides an ideal starting point for developers to understand the types of vulnerabilities they should be aware of and work to prevent in their software.
- OWASP NodeGoat and OWASP Juice Shop - these are both open-source projects that present a security-focused learning platform for JavaScript and Node.js developers. You can clone them on GitHub and experience real-world misconfigurations and security issues. At the time of writing this book, they're known to cover all of OWASP's Top 10 vulnerabilities for developers to learn about and exploit in a controlled environment.
- OWASP Cheat Sheet Series - the OWASP Cheat Sheet Series provides comprehensive security advice for a wide range of languages, platforms, and development practices. In general, it provides guidance on secure coding practices for JavaScript and Node.js developers, such as NPM Security best practices, the Node.js Docker Cheat Sheet, and others.

FUN FACT

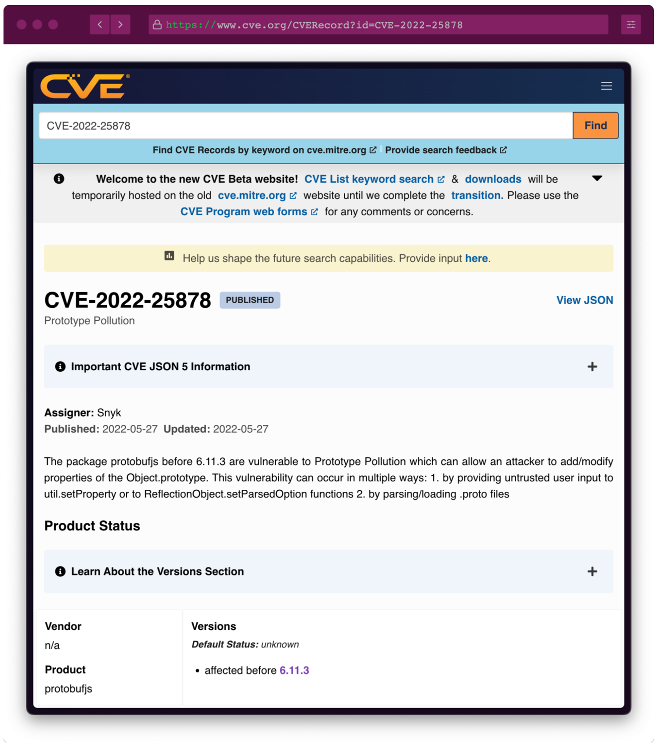
The author of this book has contributed to the OWASP Cheat Sheet Series. This includes the Node.js Docker Cheat Sheet and the NPM Security Cheat Sheet which have been widely referenced and recognized by the Node.js community.

1.1.2. MITRE, CVEs and NVD

MITRE is a non-profit organization that operates research and development centers sponsored by the US government. One of MITRE’s many areas of focus is cybersecurity and the development of application security frameworks, such as MITRE ATT&CK. In addition, MITRE provides other cybersecurity resources and tools to help organizations and individuals improve their cyber defenses.

One of MITRE’s most known contributions to the security industry is the establishment and maintenance of the Common Vulnerabilities and Exposures system, commonly referred to as CVE. This system tracks and maintains security vulnerabilities and assigns each of them an ID, referred to as a CVE ID, or for short, a CVE. It then categorizes them into specific classifications such as CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection'). This classification system is known as Common Weakness Enumeration (CWE).

MITRE maintains its list of open and public security vulnerabilities database through the National Vulnerability Database, known commonly by its acronym: NVD. As such, NVD is a website that provides access to the CVE database, which is a list of all publicly known security vulnerabilities and their associated CVE IDs.


https://www.cve.org/CVERecord?id=CVE-2022-25878

CVE-2022-25878 Find

Find CVE Records by keyword on cve.mitre.org Provide search feedback

Welcome to the new CVE Beta website! CVE List keyword search & downloads will be temporarily hosted on the old cve.mitre.org website until we complete the transition. Please use the CVE Program web forms for any comments or concerns.

Help us shape the future search capabilities. Provide input here.

CVE-2022-25878 PUBLISHED View JSON

Prototype Pollution

Important CVE JSON 5 Information

Assigner: Snyk
Published: 2022-05-27 **Updated:** 2022-05-27

The package protobufjs before 6.11.3 are vulnerable to Prototype Pollution which can allow an attacker to add/modify properties of the Object.prototype. This vulnerability can occur in multiple ways: 1. by providing untrusted user input to util.setProperty or to ReflectionObject.setParsedOption functions 2. by parsing/loading .proto files

Product Status

Learn About the Versions Section

Vendor	Versions
n/a	*Default Status:* *unknown*
Product	• affected before 6.11.3
protobufjs	

Figure 1. CVE-2022-25878 on the cve.org website

Useful resources:

- MITRE CVE - this is the home for the overall management of CVEs, providing access to CVE artifacts, a searchable list, the issuing of CVEs as well as modifications to existing CVEs. It also lists current working groups and other resources such as Common Numbering Authorities, known as CNAs. A CNA is an organization that has been approved and authorized by MITRE to handle disclosures of security vulnerabilities and issue CVE IDs

> As a maintainer of open-source npm packages, you may find yourself handling CVE reports pertaining to your project. The MITRE CVE website is the address to submit revocation requests or other modifications to be made to a CVE report that was issued to a project you are maintaining or contributing to.

- NVD - MITRE's CVE database is made publicly available for consumption through NVD, the National Vulnerability Database. In NVD, each vulnerability is well described with metadata, accompanying resources, a severity score, and the product or vendor information it is associated with in terms of impact.

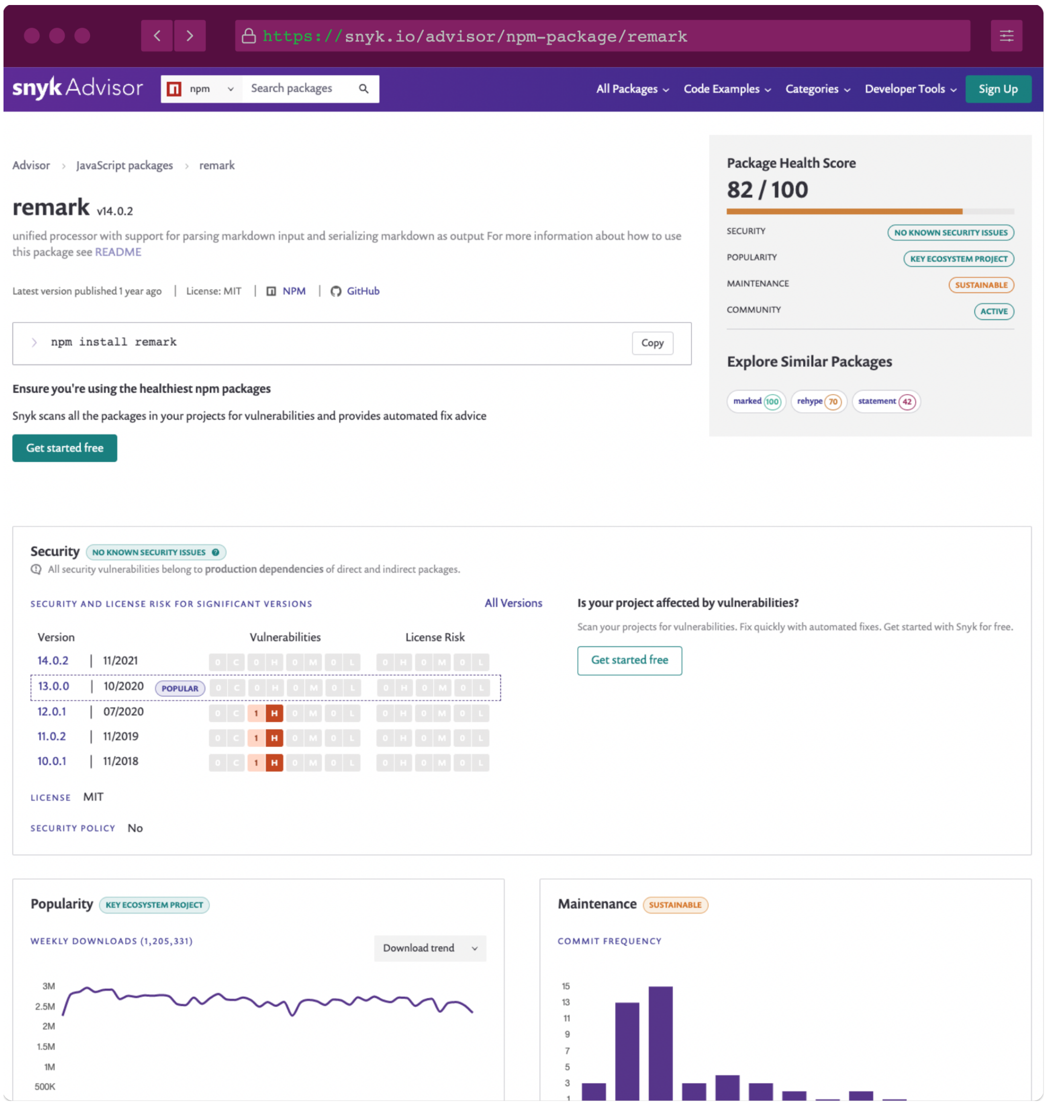
1.1.3. Snyk Advisor

Typically, developers integrate and use third-party open-source libraries to build applications. With the peak adoption of open-source software these days, reliance on community-powered open-source software extends to more than just security risks with libraries.

Developers may often perform due diligence and compare projects in order to vet their sustainability. As such, they may be concerned about issues such as:

- Is the library facing maintenance issues?
- Has the library's popularity been on the decline?
- How many active maintainers and contributors are working on the project?

The Snyk Advisor is a free web tool to help gauge a package's health status and curate a project's sustainability and security criteria into a holistic comparable health score. This helps developers and engineers make better decisions about open-source projects based on current factual data.



https://snyk.io/advisor/npm-package/remark

snyk Advisor | npm | Search packages | All Packages | Code Examples | Categories | Developer Tools | Sign Up

Advisor › JavaScript packages › remark

remark v14.0.2

unified processor with support for parsing markdown input and serializing markdown as output For more information about how to use this package see README

Latest version published 1 year ago | License: MIT | NPM | GitHub

```
> npm install remark
```

Copy

Ensure you're using the healthiest npm packages

Snyk scans all the packages in your projects for vulnerabilities and provides automated fix advice

Get started free

Package Health Score

82 / 100

SECURITY	NO KNOWN SECURITY ISSUES
POPULARITY	KEY ECOSYSTEM PROJECT
MAINTENANCE	SUSTAINABLE
COMMUNITY	ACTIVE

Explore Similar Packages

marked 100 | rehype 70 | statement 42

Security NO KNOWN SECURITY ISSUES

All security vulnerabilities belong to **production dependencies** of direct and indirect packages.

SECURITY AND LICENSE RISK FOR SIGNIFICANT VERSIONS — All Versions

Version		Vulnerabilities	License Risk
14.0.2	11/2021	0 C 0 H 0 M 0 L	0 H 0 M 0 L
13.0.0	10/2020 POPULAR	0 C 0 H 0 M 0 L	0 H 0 M 0 L
12.0.1	07/2020	0 C 1 H 0 M 0 L	0 H 0 M 0 L
11.0.2	11/2019	0 C 1 H 0 M 0 L	0 H 0 M 0 L
10.0.1	11/2018	0 C 1 H 0 M 0 L	0 H 0 M 0 L

LICENSE MIT

SECURITY POLICY No

Is your project affected by vulnerabilities?

Scan your projects for vulnerabilities. Fix quickly with automated fixes. Get started with Snyk for free.

Get started free

Popularity KEY ECOSYSTEM PROJECT

WEEKLY DOWNLOADS (1,205,331) — Download trend

Maintenance SUSTAINABLE

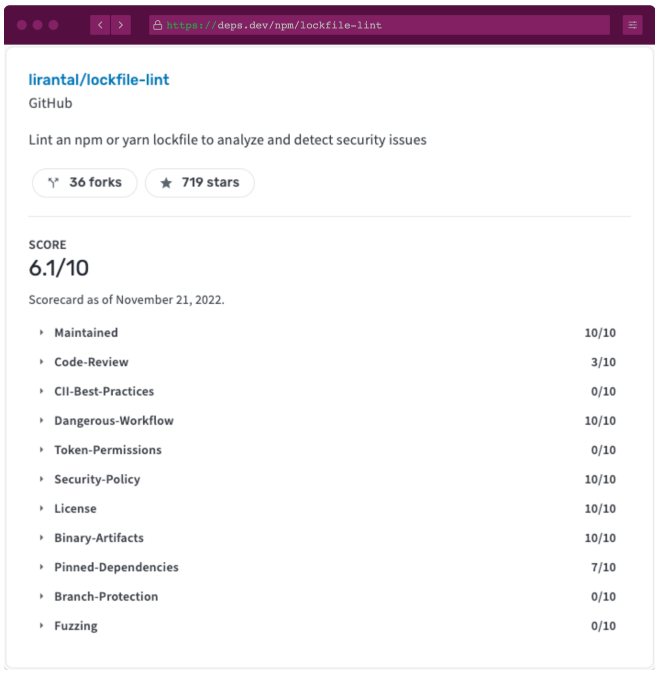
COMMIT FREQUENCY

Figure 2. Snyk Advisor package health score for the* `remark` *package

Another source of package health information is deps.dev which is a free web resource tool made available by Google and provides open access to the data through BigQuery Public Dataset. The following capabilities are powerful features to investigate dependencies beyond package health:

- A complete list of dependencies and dependent packages

- Visual diff to compare across published package versions
- Versions are annotated with information about a list of dependents
- License information includes the entire dependency list
- OpenSSF scorecard details



https://deps.dev/npm/lockfile-lint

lirantal/lockfile-lint
GitHub

Lint an npm or yarn lockfile to analyze and detect security issues

36 forks   719 stars

SCORE
6.1/10
Scorecard as of November 21, 2022.

Maintained	10/10
Code-Review	3/10
CII-Best-Practices	0/10
Dangerous-Workflow	10/10
Token-Permissions	0/10
Security-Policy	10/10
License	10/10
Binary-Artifacts	10/10
Pinned-Dependencies	7/10
Branch-Protection	0/10
Fuzzing	0/10

Figure 3. The `lockfile-lint` npm package OpenSSF scorecard details from deps.dev

1.2. Application Security Jargon

The following is a list of technical security terms and acronyms commonly used in security conversations, documentation and vulnerability communication. These are widely used throughout the book and defined here for reference.

Injection

A data payload provided to an application to deviate from its original execution intent. This is done by composing data in a specific way.

Security controls

a mechanism to protect data, applications, and systems from unwanted and unintended behavior such as unauthorized access, modification, or destruction. In relation to software and code, security controls are often implemented as a set of rules, algorithms, and practices. These controls are used to protect the application and its users from potential harm. For example, escaping user input is a security control used to prevent command injection attacks.

Responsible disclosure

Security responsible disclosure refers to the process of disclosing sensitive information about a security vulnerability found in a library, a product, or a flaw in a computer system. The goal of ***responsible security disclosure*** is to alert the appropriate parties to the vulnerability. This will enable them to triage, communicate with maintainers and collaborate to fix the issue. This will protect their systems and users from potential harm.

OWASP Top 10

A widely recognized and industry-accepted document that provides a concise, high-level summary of the top 10 most common weaknesses when it comes to web security.

Source to sink

Source to sink is a term used to describe the process of data flow within a program from where user input originates (the ***source***), to where it is used in some form (the ***sink***). In order to ensure the security of an application or the integrity of a system, it is common to implement security controls at the source. This includes input validation or input sanitization. Security controls can, and should, be employed at the sink. Some examples are output encoding and parametrized queries.

CVE

A Common Vulnerabilities and Exposures is an identifier assigned to a publicly disclosed security vulnerability. It provides a common reference for identifying and tracking vulnerabilities. It is recognized as a standard for identifying vulnerabilities across the industry. As a developer, you can think of CVE IDs as backlog ticket IDs for security vulnerabilities.

CVSS

Common Vulnerability Scoring System is a standardized method for assessing the severity of security vulnerabilities. It is commonly used in conjunction with CVEs to provide a quantitative measure of the potential impact of a vulnerability. CVSS assigns a score to a vulnerability based on a number of factors. These factors include the impact on the confidentiality, integrity, and availability of the affected program or underlying system. The vulnerability is also evaluated based on its ease of exploitation and likelihood of being exploited. The resulting score is a value between 0 and 10, with higher scores indicating more severe vulnerability.

CWE

Common Weakness Enumeration is a standardized classification of common software weaknesses that can lead to security vulnerabilities. It was developed by MITRE as a way to establish a common software vulnerability categorization. Additionally, it provides the basis for tools and services that can assist organizations in identifying and addressing these vulnerabilities. A CWE is also structured hierarchically and contains metadata about vulnerability classes, mitigation and prevention.

Vulnerability

A security vulnerability is a weakness in a computer program or a computer system that can be exploited by a malicious party to gain unauthorized access to sensitive data or to disrupt the normal functioning of the system. Security vulnerabilities can take many forms, including design weaknesses or computational logic flaws in applications and systems. These vulnerabilities when exploited lead to adverse consequences.

Exploit

An exploit is a technique, method or program code that is developed and used to take advantage of a vulnerability in a computer system or a software application. Exploits are then executed in order to gain advantage of a vulnerable system.

Attack vector

An attack vector is a path or means by which an attacker can gain access to a computer system or software application to exploit a vulnerability. Attack vectors can take many forms, such as manipulating input data in a way that causes the system to behave unintentionally. In addition, they can use social engineering techniques to trick users into divulging sensitive information or access credentials.

Payload

A payload is commonly used in exploits and is part of an attack that is delivered to a target system or application that may perform malicious actions.

Attack surface

refers to all available interfaces and components accessible to an attacker and can potentially be exploited to perform malicious actions on a system.

OS Command injection

A software security vulnerability that allows attackers to execute arbitrary system commands in the context of an operating system (OS). These vulnerabilities apply to web applications but extend to other technologies such as connected end-point devices, routers, and printers.

Argument injection

A type of command injection vulnerability that manifests in the form of an attacker’s ability to modify or abuse the command-line arguments of a given command, even if they cannot modify the command itself.

Blind command injection

A type of attack that employs fuzzing and randomly generated payloads intended to exploit command injection vulnerabilities with a special payload that triggers "phone home" behavior to allow positive confirmation of a vulnerable application.

1.3. Test Your Knowledge

In this section, you can check your understanding of the concepts and best practices presented in this chapter through multiple-choice questions. Answer them to the best of your ability, and check your answers at the end of the section.

Select the correct answer (some questions may have multiple correct answers):

1. **What does OWASP stand for?**
   a. Open Web Application Security Project
   b. Open Worldwide Application Security Program
   c. Online Web Application Scanning Platform
   d. Organization of Web Application Security Professionals
2. **What is the purpose of OWASP?**
   a. To promote open source software
   b. To create a community of software developers
   c. To improve the security of software
   d. To improve website design
3. **What is the OWASP Top 10?**
   a. A list of the most critical web application security risks
   b. A list of the top ten programming languages
   c. A list of the top ten web development frameworks
   d. A list of the top ten web hosting providers
4. **What is injection?**
   a. A type of input validation technique
   b. A type of encryption algorithm
   c. A type of attack where untrusted data is sent to an interpreter as part of a command or query
   d. A type of cross-site scripting attack
5. **What is source-to-sink in software development?**
   a. The process of compiling source code into machine code
   b. The process of running a program and observing its output
   c. The flow of data within a program from user input to where it is used
   d. The process of debugging and fixing errors in code
6. **What is the difference between a CVSS and a CVE?**

   a. CVSS is a scoring system used to assess the severity of a vulnerability, while CVE is a database of known vulnerabilities
   b. CVE is a scoring system used to assess the severity of a vulnerability, while CVSS is a database of known vulnerabilities
   c. Both CVSS and CVE are databases of known vulnerabilities, but CVE focuses on the impact of the vulnerability, while CVSS focuses on its severity
   d. Both CVSS and CVE are scoring systems used to assess the severity of a vulnerability, but CVSS is more widely used in industry

7. **What is the difference between MITRE and NVD?**
   a. MITRE is a government organization that focuses on cyber security research, while NVD is a database of known vulnerabilities
   b. NVD is a government organization that focuses on cyber security research, while MITRE is a database of known vulnerabilities
   c. Both MITRE and NVD are databases of known vulnerabilities, but MITRE focuses on the impact of the vulnerability, while NVD focuses on its severity
   d. Both MITRE and NVD are organizations that focus on cyber security research, but MITRE is more widely known in the industry.

1.3.1. Answers

The correct answers are as follows:

1. a)
2. c)
3. a)
4. c)
5. c)
6. a)
7. b)

Chapter 2

Command Injection

In this introductory chapter we learn about command injection as a security vulnerability. We also learn why software is commonly vulnerable to this type of vulnerability, and its impact on applications and software libraries. We also expand upon different types of command injection vulnerabilities and how the security community classifies this vulnerability.

Learnings

By the end of this chapter, you should be able to answer questions such as:

- What makes command injection vulnerabilities so common?
- What are the types of command-injection vulnerabilities?
- How do you classify command injection vulnerabilities?
- Command injection vulnerabilities and their impact on applications and software libraries.
- Patterns of insecure code that lead to command injection vulnerabilities and identifying them in other programming languages.

2.1. What is Command Injection?

Command injection is a specific form of injection attack, such as SQL injection and Cross-site Scripting injection (XSS). The attack exploits vulnerabilities in program code that executes system commands. It insecurely concatenates user input, or completely misses to properly sanitize or encode user input that is passed to the command being executed.

When code that is meant to spawn system processes cannot distinguish between the programmer's original intention and dangerous user input. This results in an unsafe and unsantizied command being executed.

The class of injection attacks has been featured at the top of OWASP's Top 10 web security risks for over two decades. These types of attacks have been a pivotal, recurring, and dangerous set of vulnerabilities that developers have struggled with mitigating for a long time.

2.2. Command Injection Types

It may be surprising, but command injection takes on many shapes and forms, beyond the common seemingly obvious code pattern of string concatenation into a system command.

In this section, we will explore the different types of command injection vulnerabilities and how they can be exploited. Among others, you will learn:

1. The different types of command injection vulnerabilities.
2. How to identify various insecure code patterns.

At the end of each vulnerability chapter you will also learn about root causes and how to apply secure coding practices and other conclusions. This will effectively help you avoid these types of vulnerabilities when writing code.

NOTE

The Command Injection vulnerability is classified as "CWE-77: Improper Neutralization of Special Elements used in a Command ('Command Injection')", which is the parent Common Weakness Enumeration (CWE) classification of other command injection types.

2.2.1. OS Command Injection

OS command injection is classified formally as CWE-78 which describes it as ***Improper Neutralization of***

Special Elements used in an OS Command. It refers specifically to an attacker's ability to inject commands that are executed by an operating system (OS).

The Node.js runtime enables developers to execute operating system commands using the Child Process API accessible through `node:child_process` which provides synchronous and asynchronous methods to spawn subprocesses.

A Node.js command injection vulnerability is shown below as an example of OS command injection:

./app.js

```
const { execSync } = require('child_process');
execSync('git clone ' + user_specified_git_repository);
```

In this example, the variable `user_specified_git_repository` is user-controlled input of a remote Git repository to clone, which is concatenated with the `git` command.

As another reference to vulnerable code, the following is an OS command injection example in PHP:

./app.php

```
$username = $_POST["username"];
system('ls -l /home/' . $username);
```

In both code snippets above, a user is in control of input such as the URL for a Git repository to be cloned in the Node.js example. In the PHP example, a user can list files in a user's home directory on a server. This user input that we refer to as a ***source*** is then concatenated to operating system commands (`git` and `ls`, respectively) using sensitive APIs (`execSync` and `system`, respectively) which we refer to as a ***sink***.

By controlling these inputs, attackers exploit an ***OS Command Injection*** vulnerability, with a payload such as `; touch /tmp/pwned`. This payload uses the special character of a semicolon (`;`) which instructs a shell interpreter to terminate a command, and begin another command to be executed. In that payload, `touch` is a Unix command that creates empty new files at a given path. However, more destructive inputs such as deleting all files, reading environment variables and sending them to a remote attacker, would've been just as easy to abuse on a vulnerable system.

2.2.2. Argument Injection

CONTENTS NOT INCLUDED IN BOOK SAMPLE VERSION

```
/**
 * CONTENTS NOT INCLUDED IN BOOK SAMPLE VERSION
 */
```

2.2.3. Blind Command Injection

CONTENTS NOT INCLUDED IN BOOK SAMPLE VERSION

2.3. Command Injection in CWE

2.4. Command Injection in Other Languages

CONTENTS NOT INCLUDED IN BOOK SAMPLE VERSION

2.4.1. Command Injection in C

CONTENTS NOT INCLUDED IN BOOK SAMPLE VERSION

2.5. Test Your Knowledge

In this section, you can check your understanding of the concepts and best practices presented in this chapter through multiple-choice questions. Answer them to the best of your ability, and check your answers at the end of the section.

Select the correct answer (some questions may have multiple correct answers):

1. **What is Command Injection?**
    a. A vulnerability that allows an attacker to steal commands from running servers

b. A method of injecting CSS code into a website

c. A security vulnerability that allows an attacker to execute arbitrary commands on a server

d. A type of operating system system call that can be abused by attackers

2. **Which module in Node.js is vulnerable to Command Injection?**

    a. The `crypto` module

    b. The `http` module

    c. The `child_process` module

    d. The `fs` module

3. **How can Command Injection be prevented in Node.js?**

    a. By using a firewall to block incoming requests

    b. By disabling the `child_process` module

    c. By validating and sanitizing user input, and using a secure process execution API such as `execFile`

    d. By encrypting the server’s file system

4. **What is the impact of a successful Command Injection attack?**

    a. An attacker can modify the server`s DNS settings

    b. An attacker can launch a DDoS attack against the server

    c. An attacker can steal user credentials

    d. An attacker can execute arbitrary commands on a server, potentially gaining access to sensitive data or causing system damage

5. **What is Argument Injection?**

    a. A security vulnerability that allows an attacker to modify the return value of a function call in order to execute malicious code

    b. A security vulnerability that allows an attacker to modify the functionality of a function call in order to execute malicious code

    c. A security vulnerability that allows an attacker to modify the arguments of a function call in order to execute malicious code

    d. A security vulnerability that allows an attacker to modify the parameters of a function call in order to execute malicious code

6. **What is the CWE ID for Command Injection vulnerabilities?**

    a. CWE-79

    b. CWE-78

    c. CWE-119

    d. CWE-89

2.5.1. Answers

The correct answers are as follows:

1. c)
2. c)
3. c)
4. d)
5. c)
6. b)

Chapter 3

CVE-2022-XYZ: Command Injection in `--REDACTED--`

CONTENTS NOT INCLUDED IN BOOK SAMPLE VERSION

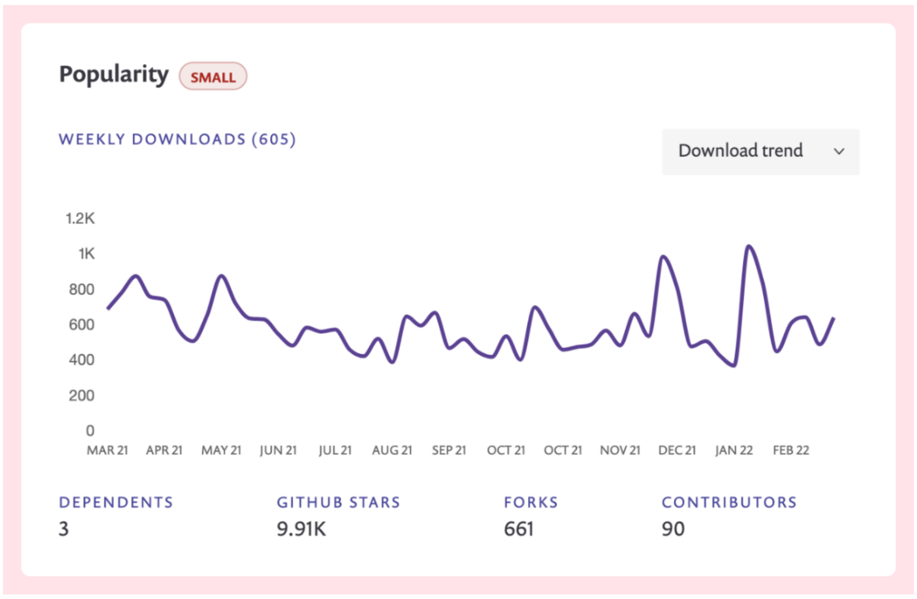


Figure 4. The `--REDACTED--` npm package popularity ranking

3.1. About the Security Vulnerability

3.2. Setting Up a Vulnerable Test Environment

3.3. Exploiting the Security Vulnerability

3.4. Reviewing the Security Fix

3.5. Lessons Learned

Chapter 4

CVE-2022-XYZ: Command Injection in `--REDACTED--`

CONTENTS NOT INCLUDED IN BOOK SAMPLE VERSION

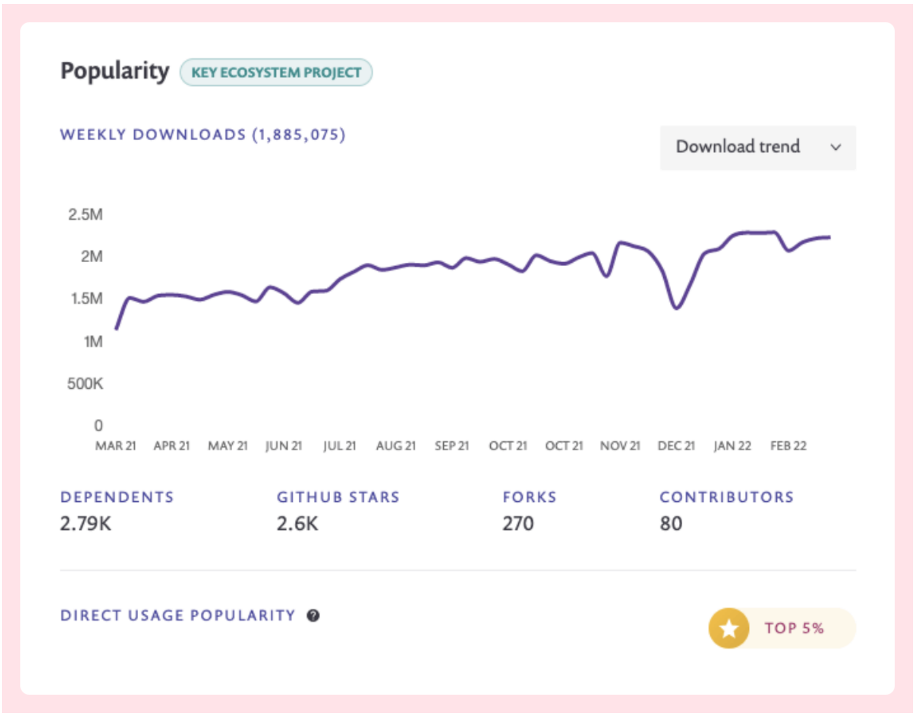


Figure 5. The `--REDACTED--` npm package popularity ranking

4.1. About the Security Vulnerability

4.2. Setting Up a Vulnerable Test Environment

4.3. Exploiting the Security Vulnerability

4.4. Reviewing the Security Fix

4.5. Lessons Learned

Chapter 5

CVE-2022-XYZ: Command Injection in `--REDACTED--`

CONTENTS NOT INCLUDED IN BOOK SAMPLE VERSION

5.1. About the Security Vulnerability

5.2. Setting Up a Vulnerable Test Environment

5.3. Exploiting the Security Vulnerability

5.4. Reviewing the Security Fix

5.5. Lessons Learned

Chapter 6

CVE-2022-XYZ: Command Injection in `--REDACTED--`

CONTENTS NOT INCLUDED IN BOOK SAMPLE VERSION

6.1. About the Security Vulnerability

6.2. Setting Up a Vulnerable Test Environment

6.3. Exploiting the Security Vulnerability

6.4. Reviewing the Security Fix

6.5. Lessons Learned

Chapter 7

CVE-2022-XYZ: Command Injection in `--REDACTED--`

CONTENTS NOT INCLUDED IN BOOK SAMPLE VERSION

7.1. About the Security Vulnerability

7.2. Setting Up a Vulnerable Test Environment

7.3. Exploiting the Security Vulnerability

7.4. Reviewing the Security Fix

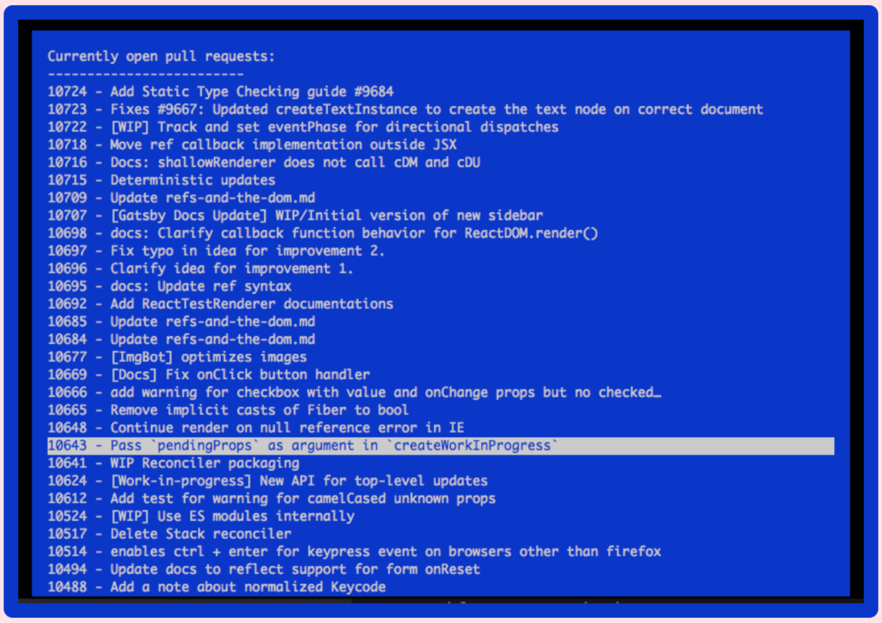
7.5. Lessons Learned

Chapter 8

CVE-2018-25083: Command Injection in `pullit`

The pullit package on npm, also known as ***Pull It***, is a terminal user interface (TUI) that displays and pulls remote Git branches of open GitHub pull requests in Github repositories.

This command-line tool allows developers to quickly browse a repository's open pull requests and select one by its title to switch your local Git working branch across.



```
Currently open pull requests:
-------------------------
10724 - Add Static Type Checking guide #9684
10723 - Fixes #9667: Updated createTextInstance to create the text node on correct document
10722 - [WIP] Track and set eventPhase for directional dispatches
10718 - Move ref callback implementation outside JSX
10716 - Docs: shallowRenderer does not call cDM and cDU
10715 - Deterministic updates
10709 - Update refs-and-the-dom.md
10707 - [Gatsby Docs Update] WIP/Initial version of new sidebar
10698 - docs: Clarify callback function behavior for ReactDOM.render()
10697 - Fix typo in idea for improvement 2.
10696 - Clarify idea for improvement 1.
10695 - docs: Update ref syntax
10692 - Add ReactTestRenderer documentations
10685 - Update refs-and-the-dom.md
10684 - Update refs-and-the-dom.md
10677 - [ImgBot] optimizes images
10669 - [Docs] Fix onClick button handler
10666 - add warning for checkbox with value and onChange props but no checked…
10665 - Remove implicit casts of Fiber to bool
10648 - Continue render on null reference error in IE
10643 - Pass `pendingProps` as argument in `createWorkInProgress`
10641 - WIP Reconciler packaging
10624 - [Work-in-progress] New API for top-level updates
10612 - Add test for warning for camelCased unknown props
10524 - [WIP] Use ES modules internally
10517 - Delete Stack reconciler
10514 - enables ctrl + enter for keypress event on browsers other than firefox
10494 - Update docs to reflect support for form onReset
10488 - Add a note about normalized Keycode
```

Figure 6. A screenshot of* `pullit` *CLI in action

Command line (CLIs) tools are usually not as ubiquitous as application libraries due to their nature of being less of a hoisted dependency that trickles into an application's dependency tree. Another notable information about CLIs is that they are rarely open to Internet-public user interaction like web applications are, and so they are often ignored as a significant threat. As such, it is common for developers to wave off such vulnerabilities and deeming any vulnerability associated with CLIs as ***false positive***.

NOTE

The argument provided by developers or security practitioners who rule out vulnerabilities in CLIs is most often that ***if an attacker can provide command line arguments as well as bee able***

to run the CLI the way they want, it's already game over.

However, that's sometimes an understatement as we'll learn with the case of a command injection found in the open-source `pullit` npm package.

8.1. About the Security Vulnerability

In 2018, as a command-line tool enthusiast[1] building terminal user interface applications and CLIs myself, I heard about `pullit` and started using it for my own projects.

Developers being curious creatures, I poked into its source code to learn how it manages its visual interface. However, I didn't expect to find a potential command injection flow in the process. The following is the relevant source code at the time, trimmed down for brevity:

pullit/src/index.js

```javascript
const GitHubApi = require('github');
const Menu = require('terminal-menu');
const {
  execSync
} = require('child_process');
const parse = require('parse-github-repo-url');

class Pullit {
  fetch(id) {
    return this.github.pullRequests
      .get({
        owner: this.owner,
        repo: this.repo,
        number: id
      })
      .then(res => {
        const branch = res.data.head.ref;
        execSync(
          `git fetch origin pull/${id}/head:${branch} && git checkout
${branch}`
        );
      })
      .catch(err => {
        console.log('Error: Could not find the specified pull request.');
      });
  }

```

```
27   display() {
28     this.fetchRequests().then(results => {
29       const menu = Menu({
30        width: process.stdout.columns - 4,
31         x: 0,
32         y: 2
33       });
34       menu.reset();
35       menu.write('Currently open pull requests:\n');
36       menu.write('-------------------------\n');
37 
38       results.data.forEach(element => {
39         menu.add(`${element.number} - ${element.title} - ${element.head.user.login}`);
40       });
41 
42       menu.add(`Exit`);
43 
44       menu.on('select', label => {
45         menu.close();
46         this.fetch(label.split(' ')[0]);
47       });
48   }
49 }
50 
51 module.exports = Pullit;
```

Perhaps the use of `execSync = require('child_process')` provides some hint at the risk involved. Specifically, line 19 in the above source code executes the following Git system commands:

```
execSync(
  `git fetch origin pull/${id}/head:${branch} && git checkout ${branch}`
);
```

However, how would an external threat actor interact with it? Here is where things get interesting and educate us on how user input flows into applications, or CLIs in this case, in unexpected ways.

At this point, I wondered what if I could create Git branches with special shell-related characters in them? Even if I could do that, I'd need to push them to a GitHub repository for them to be used in an attack on developers. Won't GitHub filter out these weirdly named Git branches?

Let's explore an idea for a branch name which, when used as the source for a command execution API like Node.js's `execSync()`, will trigger arbitrary user-controlled system commands:

```
";{echo,hello,world}>/tmp/c"
```

This input is a fully acceptable and legitimate branch name. In fact, you can run the following command to demonstrate how to create a Git branch with this name in your local development environment:

```
git checkout -b ";{echo,hello,world}>/tmp/c"
```

When the above command is run as-is, the branch name part, identified by the command-line value passed to the `-b` command flag, is properly quoted and treated as a branch name to be created.

However, what happens when the same input is passed into a process execution API such as the following code?

```
const branch = ";{echo,hello,world}>/tmp/c"
execSync(
  `git checkout ${branch}`
);
```

It is passed as a string of text to be evaluated by the shell interpreter:

```
const branch = ";{echo,hello,world}>/tmp/c"
execSync(
  `git checkout ;{echo,hello,world}>/tmp/c`
);
```

The result is that a seemingly innocent branch name is now interpreted as a command to be executed by the shell interpreter.

Why and how does this branch name work for command injection?

- The leading `;` character instructs the shell interpreter to terminate the previous command (`git checkout`), and start the next command to be executed.
- The `{…}` is a special syntax for the shell interpreter referred to as command grouping which defines a list of commands to be executed. In this case `{echo,hello,world}` expands into the command `echo hello world`.
- The last part of this user input which acts as a malicious payload is `>/tmp/c` that instructs the shell program to direct all standard output (`>` is the notation for **stdout**) of the command (`echo hello world`) into the file path identified as `/tmp/c`.

The question at this point becomes - is it really possible to name a Git branch

`;{echo,hello,world}>/tmp/c` ? The remarkable answer is yes. It's absolutely possible, and in fact you will be able to this branch to GitHub and it will show up as is:

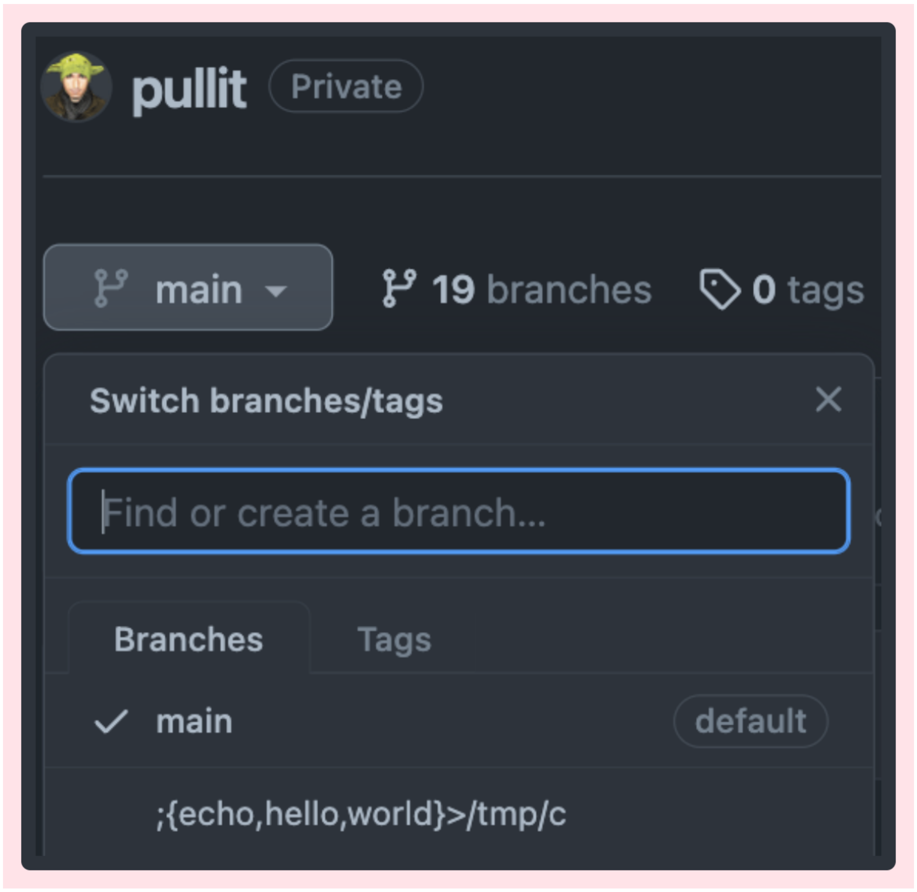


Figure 7. GitHub UI shows a perfectly valid and legitimate Git branch name with dangerous characters impacting operating system shell interpreters and may result in command injection.

8.2. Exploiting the Security Vulnerability

As we learned by reviewing `pullit's source code, the` **`pullit`** `npm package makes insecure use of system process API (such as employing the user of `exec()` or `execSync()` with insecure user input). This makes the tool vulnerable to malicious user input based on a remote branch name on the GitHub platform (and potentially other Git source control management systems).

This is made especially severe due to the ***GitHub workflow*** for open-source contributions which embraces forking projects by third-party contributors who control their branch name. This results in the risk of tricking innocent users who use the `pullit` tool to pull their branch and execute arbitrary commands.

1. Create a branch that could potentially terminate an exec() command and concatenate a new command: `git checkout -b ";{echo,hello,world}>/tmp/c"`.
2. Push it to GitHub and create a pull request with this branch name.

3. Run `pullit`, select the pull request with the title matching the dangerous source branch to checkout locally.
4. Confirm the following file has been created `/tmp/c` with the contents of "hello world".

8.3. Reviewing the Security Fix

To fix this issue, the maintainer applied the following commit, most notably exchanging `execSync` with `execFileSync` which separates the command from its arguments, and provides protection against user input concatenation. Following is a partial snippet of the committed fix:

```
execFileSync("git", ["fetch", "origin", `pull/${id}/head:${branch}`]);
execFileSync("git", ["checkout", branch]);
```

This was indeed an effective fix against string concatenation in which user input flowed into the overall command passed to the shell interpreter.

8.4. Lessons Learned

In this chapter, we reviewed the security vulnerability in the `pullit` package discovered in 2018. The vulnerability allowed an attacker to execute arbitrary commands on the host system by specifying a specially crafted Git branch name.

One of the key takeaways from this case is that command-line tools are not immune to security vulnerabilities, despite being less commonly used than application libraries. In fact, CLIs can be just as vulnerable as web applications or other types of software, and should be treated as such. Developers and security practitioners should not immediately ignore vulnerabilities associated with CLIs as false positives.

We also learned about the dangers of assuming input always comes from expected and trusted sources. We saw how seemingly innocuous data sources can be exploited to execute harmful commands. Specifically, we learned that user input may originate from unexpected sources, such as a Git branch name. In addition, characters like the semicolon (`;`) can take on a different meaning when flowing into sensitive system APIs such as command execution. This highlights the importance of implementing proper input validation and sanitization techniques to prevent potentially harmful commands from being executed on a system.

Eventually, the `pullit` vulnerability was caused by the package's misplaced trust in Git naming conventions. This naive assumption was abused by an attacker to execute arbitrary commands.

Overall, this chapter highlights the importance of proactive security approaches in software development. It also highlights the importance of following security best practices such as using `execFile` vs exec. This would have helped mitigate this security vulnerability.

[1] Command-line tools by Liran Tal: https://github.com/lirantal/dockly, https://github.com/lirantal/awesome-nodejs-security and https://github.com/lirantal/npq

Chapter 9

Defending Against Command Injection

Following are curated secure coding best practices for preventing command injection attacks in Node.js applications. We look at the different ways command injection vulnerabilities can be introduced. We reflect on the subtleties of incorrectly using child process APIs, and address each attack vector with practical secure coding advice.

Learnings

By the By the end of this summary chapter, you will be skilled at secure coding practices, perform secure code reviews and answer questions such as:

- Which Node.js process executing APIs are recommended as safe methods to execute commands?
- What are the security implications of using the `shell` option of the `child_process` APIs?
- When and how should you escape user input to prevent command injection vulnerabilities?
- How do you effectively protect against argument injection attacks?
- What are the security implications associated with invoking the `npm` package manager's run-scripts?

9.1. Node.js `child_process`: Choosing the Right API for Secure Command Execution

When writing secure code in Node.js that involves executing child processes, it's critical to know the potential vulnerabilities that exist. It's also important to follow secure coding conventions to avoid them.

The Node.js core module for process execution is child_process. However, some of its APIs, such as `exec`, can lead to command injection security vulnerabilities, even when developers attempt to sanitize user input.

The following properties of the `exec` function make it an extremely dangerous programming interface and highly vulnerable to command injection attacks.

9.1.1. Commands Passed as Strings

When the command to execute is passed as a string, developers often use string concatenation to build the command. This may lead to command injection vulnerabilities. Even when developers attempt to apply security controls such as sanitizing user input, they often miss edge cases that can lead to vulnerabilities.

```
const { exec } = require('child_process');

exec(`echo ${userInput}`, (err, stdout, stderr) => {
  if (err) {
    console.error(`exec error: ${err}`);
    return;
  }
  console.log(`stdout: ${stdout}`);
});
```

THE REST OF THIS CHAPTER'S CONTENTS NOT INCLUDED IN BOOK SAMPLE VERSION

9.1.2. --REDACTED--

9.1.3. --REDACTED--

9.1.4. --REDACTED--

9.2. Best Practice #2: --REDACTED--

9.3. Best Practice #3: --REDACTED--

9.4. Best Practice #4: --REDACTED--

9.5. Best Practice #5: --REDACTED--

Chapter 10. Appendix

10.1. Test Your Knowledge

In this section, you can check your understanding of the concepts and best practices presented in the book through multiple-choice questions, fill-the-blank stories and yes-no questions. Answer them to the best of your ability, and check your answers at the end of the section.

TIP

If you are using this book to train others, it is highly recommended that you use these sets of questions to test your audience's understanding of the concepts presented in the book and how well they have internalized the material.

To do that, you should use these questions to assess their skill-set before and after training. This will help you identify areas of improvement and better understand the effectiveness of the expertise gained by reading and practicing the exercises in this book.

Select the correct answer (some questions may have multiple correct answers), fill in the blanks, and answer to the best of your knowledge the following questions:

1. **What are the types of command injection vulnerabilities?**
   a. Argument Injection
   b. Command Injection
   c. Blind Command Injection
   d. Stored Command Injection
2. **What are the best practices to prevent command injection vulnerabilities in Node.js?**
   a. Use a secure API that provides a safe way to execute shell commands, such as `child_process.execFile`
   b. Validate user input to expected schema, length and types
   c. Use the POSIX double-dash to separate arguments from options
   d. Map user input to an allow-list of pre-defined command-line arguments

    e. Use a firewall to block incoming requests

3. **What is the danger of using `child_process.exec` to execute shell commands in Node.js?**
    a. It is slower than other methods of executing shell commands
    b. It does not work on all operating systems
    c. It is not compatible with other Node.js APIs
    d. It allows an attacker to inject arbitrary shell commands and execute them on the server

17 MORE QUESTIONS ARE'NT AVAILABLE IN THE BOOK SAMPLE VERSION

10.2. Command Injection in the Wild

10.2.1. Command Injection in GitHub Actions

Can you spot the vulnerability in the following GitHub Actions workflow?

```
name: app-ci
on:
  issue_comment:
    types: [created]
jobs:
  comment-action:
    runs-on: ubuntu-latest
    steps:
      - name: Echo issue comment
        run: |
          echo ${{ github.event.comment.body }}
```

CONTENTS NOT INCLUDED IN BOOK SAMPLE VERSION

10.2.2. Command Injection in Networking & Security Appliances

CONTENTS NOT INCLUDED IN BOOK SAMPLE VERSION

Exercises

CONTENTS NOT INCLUDED IN BOOK SAMPLE VERSION

10.2.3. Vulnerable `sketchsvg`

SketchSVG is an official eBay library for converting Sketch files to SVGs.

The CVE-2023-26107 security advisory published on March 6th, 2023, describes a command injection vulnerability in the `runCmdLine` function of the `sketchsvg` library. The vulnerability is caused by the `shell.exec` function, which is vulnerable to command injection. The `shell.exec` function executes a command line tool to convert Sketch files to SVGs.

The vulnerable code manifests in lines 109-120 of the `sketchsvg/lib/index.js` file as follows:

sketchsvg/lib/index.js

```
runCmdLine(allLayers, fileName) {
  return new Promise((resolve, reject) => {
    allLayers.layers.forEach((layerObj, idx) => {
      const id = layerObj.id;
      const name = encodeURIComponent(layerObj.name);
      /* eslint-disable max-len */
      shell.exec(`${sketchTool} export layers ${fileName} --item=${id}
--filename=${Math.random()}--${name}.svg --output=${__dirname}/tmpsvgs
--formats=svg`);
      count++;
      resolve();
    });
  });
}
```

Exercises

The following are recommended exercises to engage your team in application security:

1. Identify the vulnerability in the code snippet above.
2. Identify the ***source*** and ***sink*** of the vulnerability.
3. Are there other vulnerable functions in this version of the library's source code?

4. Describe several ways to fix this vulnerability.
5. What is the ideal fix for this vulnerability?
6. How did the library maintainer fix this vulnerability?

10.2.4. Vulnerable `versionn`

The versionn npm package is a library for managing version numbers for Node.js projects packaged as npm modules.

The CVE-2023-25805 security advisory published on February 19th, 2023, describes a command injection vulnerability in the `gitfn.js` file of the `versionn` library. The vulnerability is caused by an insecure use of the `child_process.exec` function to commit and tag the version number. This security vulnerability was fixed in version 1.1.0.

The vulnerable version of the `versionn` npm package is as follows:

versionn/lib/gitfn.js

```
var child = require('child_process')

function GitFn (version, options) {
  this._version = version
  this._options = {
    cwd: options.dir,
    env: process.env,
    setsid: false,
    stdio: [0, 1, 2]
  }
}
module.exports = GitFn

GitFn.prototype = {
  tag: function (cb) {
    var cmd = ['git', 'tag', 'v' + this._version].join(' ')
    this._exec(cmd, cb)
  },
  untag: function (cb) {
    var cmd = ['git', 'tag', '-d', 'v' + this._version].join(' ')
    this._exec(cmd, cb)
  },
  commit: function (cb) {
    var cmd = ['git', 'commit', '-am', '"' + this._version + '"'].join(' ')
    this._exec(cmd, cb)
  },
  _exec: function (cmd, cb) {
    child.exec(cmd, this._options, cb)
  }
}
```

Exercises

For your team to become more aware of application security, I recommend the following exercises:

1. Can developers on your team identify and explain the primary reason for the vulnerability?
2. Can developers in your team suggest which alternative Node.js APIs should be used instead, to mitigate the command injection vulnerability?
3. Was this security vulnerability fixed on time?
4. Review the security fixes and discuss:
    a. Is the technique used to fix the vulnerability the best approach? How else could the vulnerability have been fixed?
    b. What else can you learn from the security fix? Hint: Software testing is a key part of software development

10.2.5. Vulnerable `--REDACTED--`

CONTENTS NOT INCLUDED IN BOOK SAMPLE VERSION

Exercises

CONTENTS NOT INCLUDED IN BOOK SAMPLE VERSION

10.2.6. Vulnerable `--REDACTED--`

CONTENTS NOT INCLUDED IN BOOK SAMPLE VERSION

Exercises

CONTENTS NOT INCLUDED IN BOOK SAMPLE VERSION

10.2.7. Vulnerable `--REDACTED--`

CONTENTS NOT INCLUDED IN BOOK SAMPLE VERSION

Exercises

CONTENTS NOT INCLUDED IN BOOK SAMPLE VERSION

10.2.8. Vulnerable `--REDACTED--`

CONTENTS NOT INCLUDED IN BOOK SAMPLE VERSION

Exercises

CONTENTS NOT INCLUDED IN BOOK SAMPLE VERSION

10.3. CVEs in This Book

CONTENTS NOT INCLUDED IN BOOK SAMPLE VERSION

applied secure coding techniques through exploiting command injections and insecure code

In this adventure-based approach to application security learning, you will become a detective and uncover the mysteries of command injection vulnerabilities.

This in-depth book provides a comprehensive understanding of command injection vulnerabilities and their impact on web application security, while also teaching you how to avoid common pitfalls through analyzing insecure code in real-world npm packages.

With step-by-step code reviews and secure coding best practices, you'll develop a security-first mindset and gain expertise that will benefit you in your day-to-day programming and code review routines.

Liran Tal is an accomplished software developer, respected security researcher, and prominent advocate for open source software in the JavaScript community. He has been recognized as a GitHub Star for his tireless efforts to educate and inspire developers. In recognition of his work in advancing Node.js security and building developer security tools, he was awarded the OpenJS Foundation's Pathfinder for Security Award. An experienced author and educator, Liran has authored several published books on software security, including "Serverless Security" published by O'Reilly and "Essential Node.js Security". Liran's leadership in open source security extends to significant contributions to OWASP projects, CNCF, and OpenSSF initiatives. His work on supply chain security research, including Lockfile Injection, was presented at the Black Hat Europe 2021 cybersecurity conference.