SECURE CODE VILLAGE

Presentated by
Gaurav Bhosale
Hare Krishna Rai

Introduction

Secure Code Village is a hands-on security learning space focused on helping developers and security professionals build secure software from the ground up. Through workshops, live demos, and real-world challenges, we promote a shift-left mindset—embedding security early in the development lifecycle. Whether you're just starting out or deep into DevSecOps, Secure Code Village is where code meets security, practically.

About us

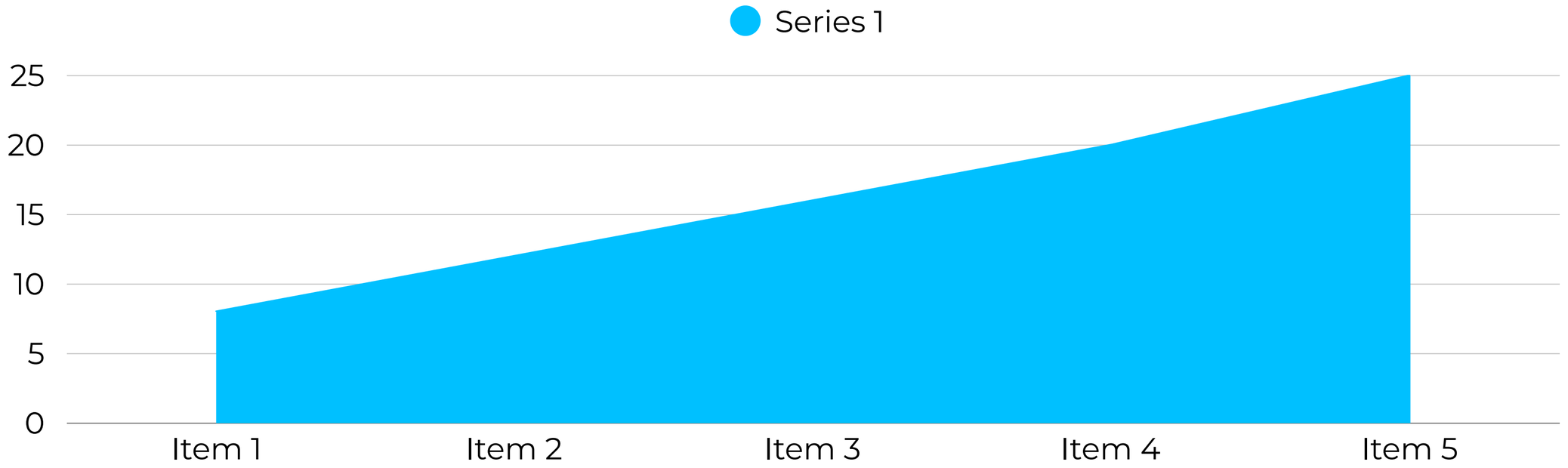
Gaurav Bhosale

Application Security Enginer | Ex-mastercard | Ex-Payatu
Speaker

Hare krishna Rai

Security Engineer @ Okta
Arsenal @Blackhat Asia, Europe

Catching Problem Early Means Less People Impacted

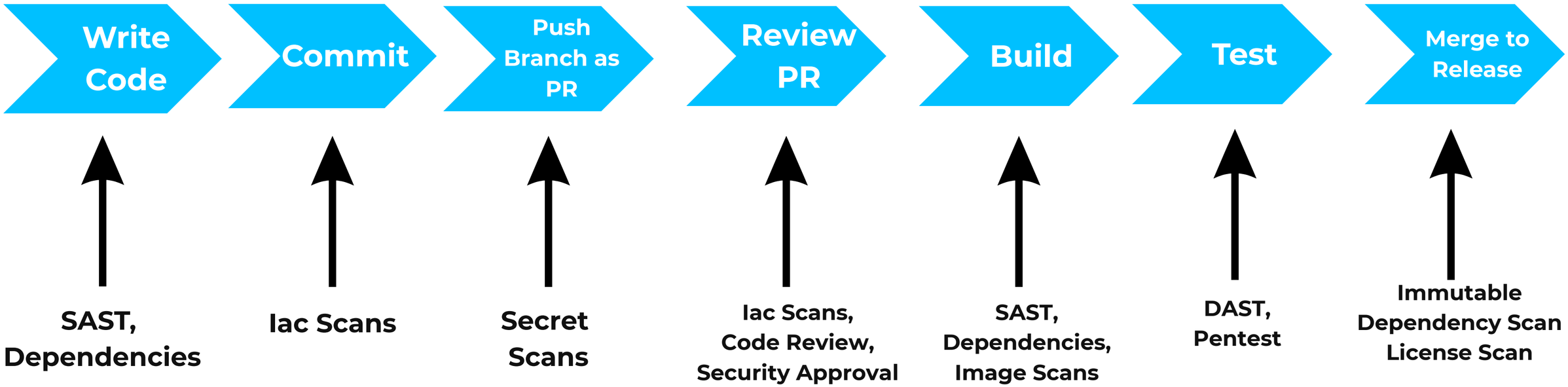


Typical Development Process

Security Integration Points

Integration Point	Integration
Developer IDE	Ability to run SAST & Dependency Scan
Commit Hook	Platform Iac Scanning & Linting
Push Hook	Secret Scanning (Prevents Secret reaching the git history)
Pull Request	Microservice Iac Scanning, code Review, Code Owner for Secuirty Approval
Repository	Secret Scanning
Build Pipline	SAST (Block High & Medium), Dependecy Scan (Block Critical, High & Medium)
Release Process	Immutable Dependency & license Scans
Test Process	DAST Scan, Pentesting

Security Test Integration



Is this is good?

- its Comprehensive
- Multiple check
- Includes both early advisory checks and later governance checks

But.....

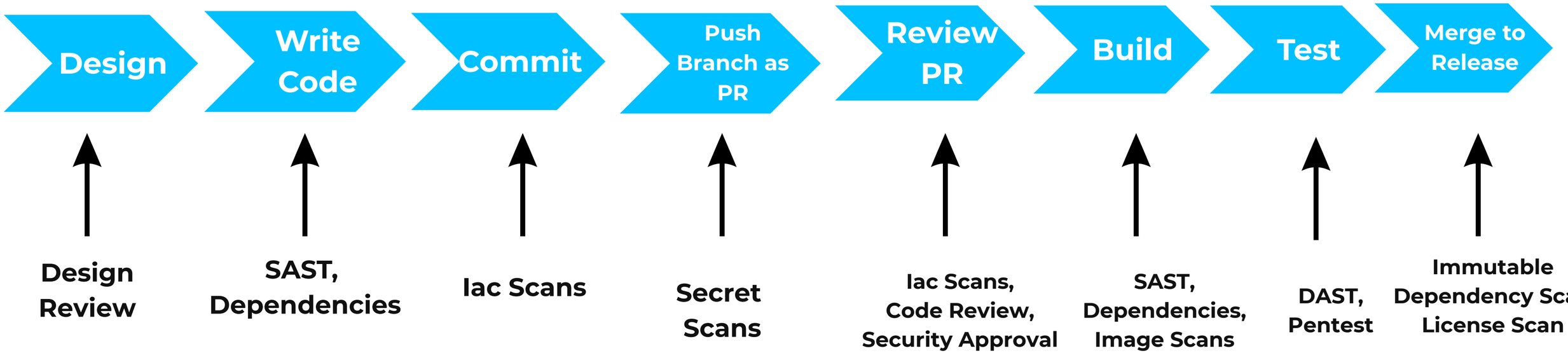
- Only addresses implementation issues not design or concept
- Missing controls only discovered at the end during pentest
- Fixing Problem rather than preventing them

Considering Security During Design

Design Reviews

- FInd problems Before implementation
- Can identify missing control early

Security Integration with Design Review



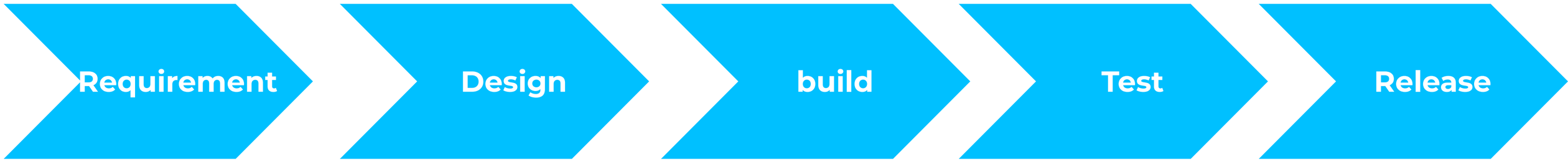
Definitely Better

- All the controls we had before
- Design review Catched problem earlier
- Now identify missing control ealry

BUT...

- Still fixing problems rather than preventing them

Introducing Threat Modelling



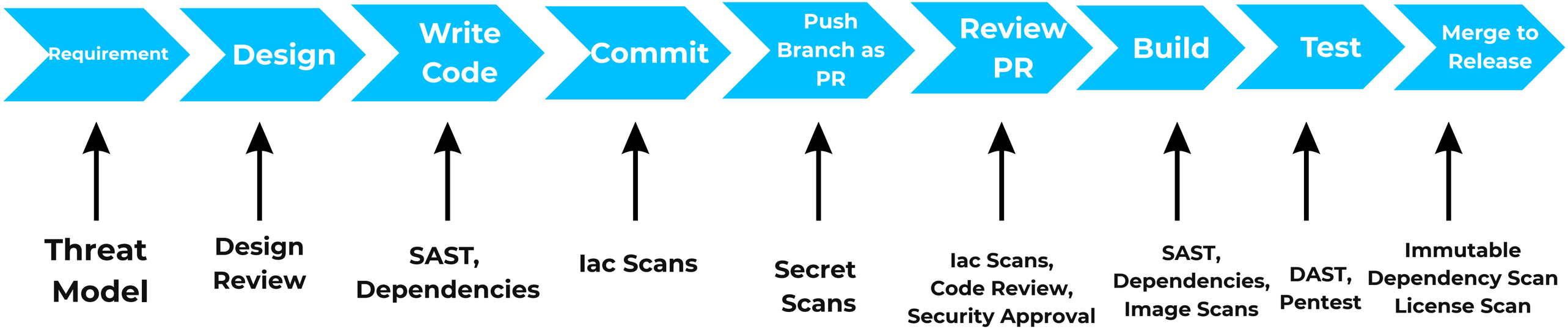
- Consider threats as early as possible
- Can be done before the design to help identify concers

Light Weigh Threat Modeling

- Anyone can do it
- Security Champion can facilitate
- Living Document Not stable Spredsheets
- Bring Value much more early
- No Secret Sauce, it is easy

- It’s not rocket science - just good ol’ overthinking, like my ex at a shopping mall.

End to End Security Integration



Mujhe break lena hai

A DEEP DIVE INTO STATIC APPLICATION SECURITY TESTING

VULNCON

2025

SECURECODEVILLAGE.COM

Static Application Security Testing

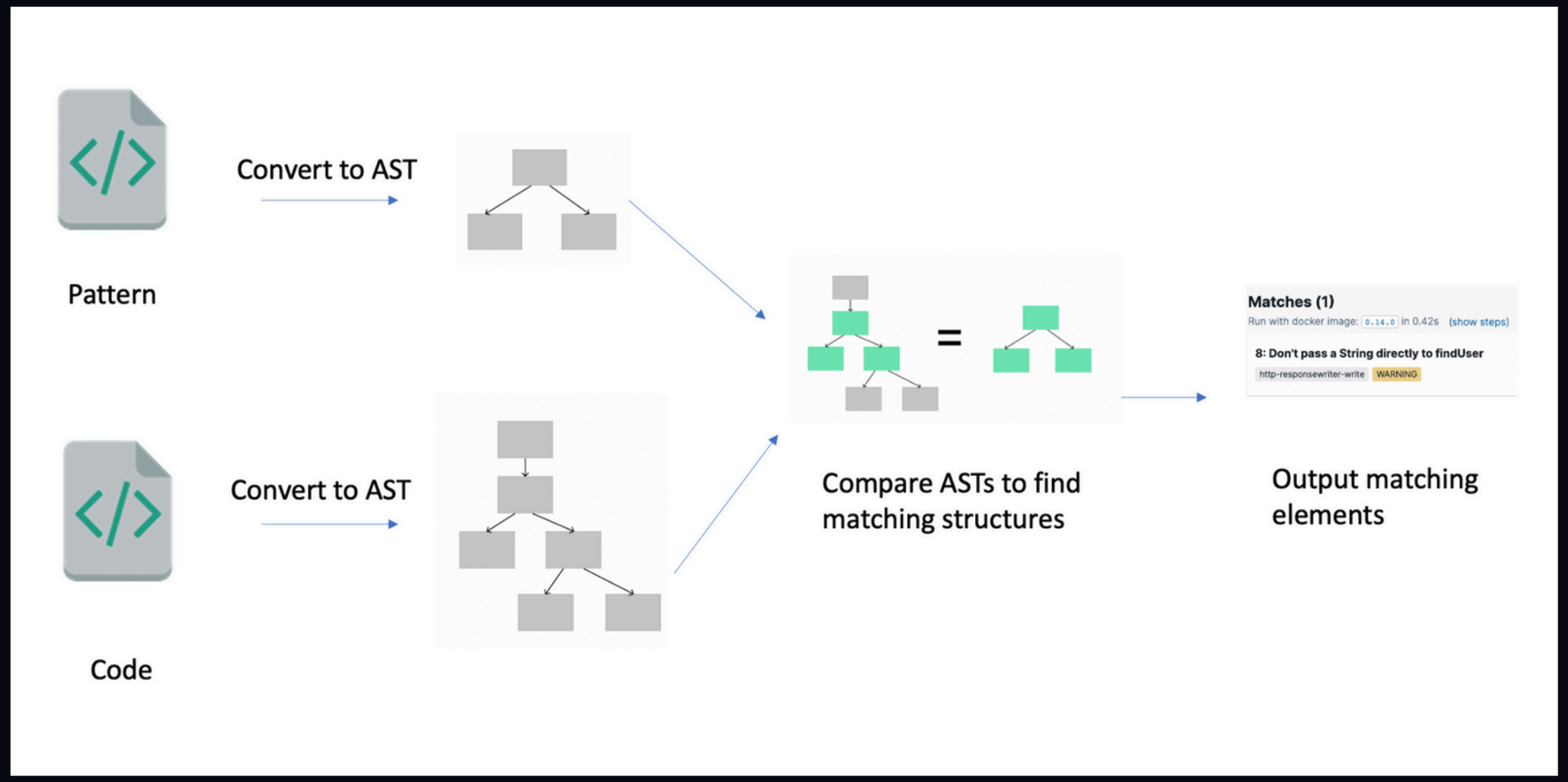
- Analyzes Source Code without execution
- Originally meant for Developers
- Most of OWASP vulns are found at code Level
- Cost effective
- Early identification of Vulnerabilities Various stages of Implementation

HOW is SAST used to be done?

How can you leverage grep,git grep for the SAST in your manual code review?

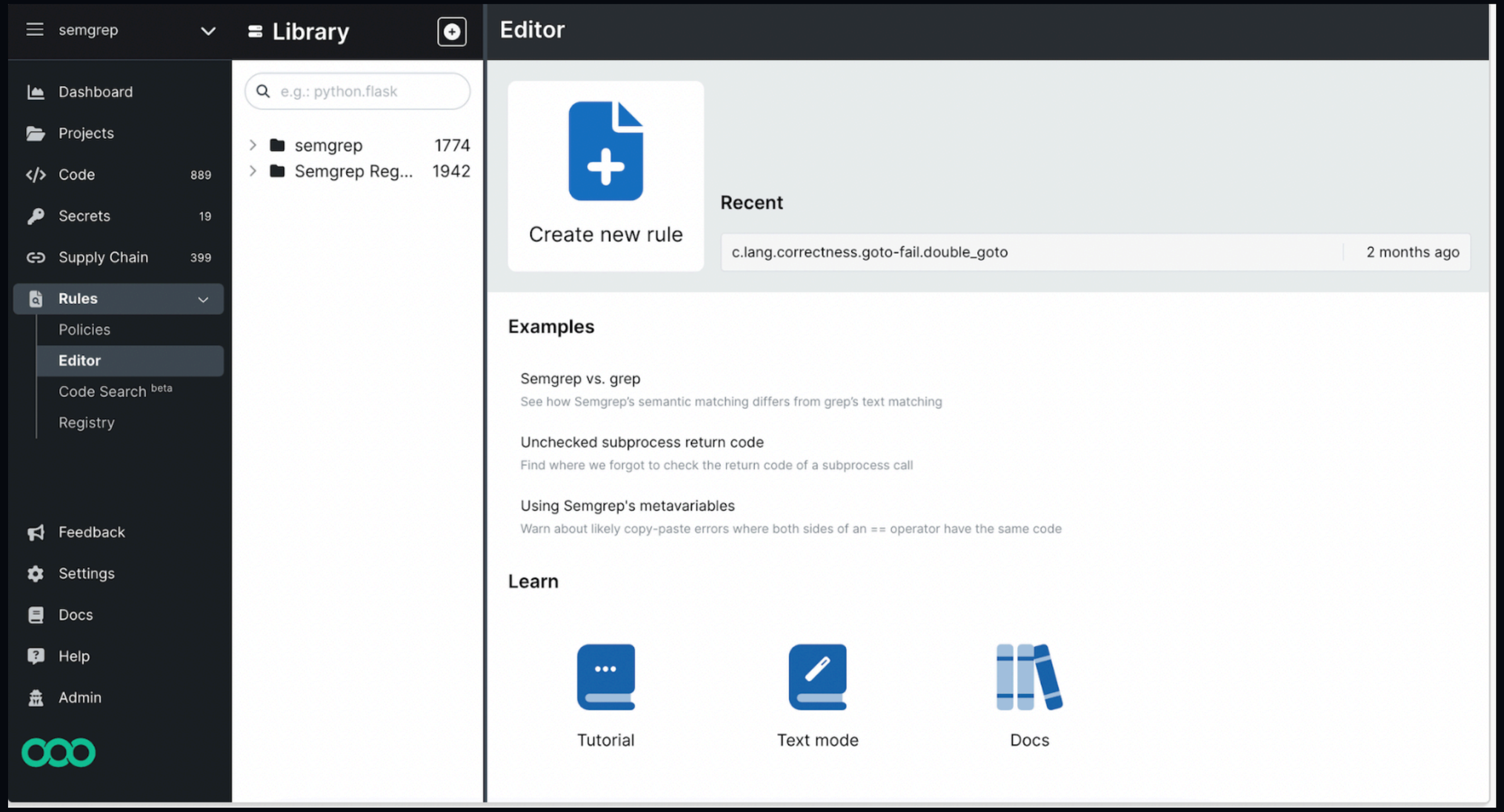
Abstract syntax tree

- What is AST?
- How AST is done?
- how it benefits SAST tools to identify the vulnerabilities?



How are the different tools different from each other in terms of their working or analysis mechanisms - like checkmarx, fortify, semgrep, opengrep?

Basics of Semgrep - Semgrep Rules?



SEMGREP RULES CONSTRUCTS

Construct	Description	Example
...	Any number of arguments/statements	func(..., x=1, ...)
$_	Any single argument/expression	print($_)
$X	Capture a value for reuse	$FUNC($ARG)
pattern-not	Exclude a pattern	Exclude os.system("safe")
pattern-inside	Match only within some code	Inside a def block
pattern-regex	Match via regex instead of AST	pattern-regex: exec\s*\(

SEMGREP RULES EXAMPLES FROM THE PORTAL - DEMO

A DEEP DIVE INTO SOFTWARE COMPOSITION ANALYSIS

VULNCON

2025

SECURECODEVILLAGE.COM

“With great power comes great responsibility... and great vulnerabilities”

TABLE OF CONTENTS

INTRODUCTION

Open Source Software (OSS) powers modern applications, from web frameworks to AI models. But with its widespread adoption comes a hidden cost: vulnerabilities.

- 80% of modern codebases contain OSS components.
- OSS vulnerabilities can lead to catastrophic breaches.
- Software Composition Analysis (SCA) is critical for identifying and mitigating risks.

Today, we'll explore OSS vulnerabilities, SCA challenges, and how to secure your software supply chain.

WHY OSS IS EVERYWHERE ?

- Cost-effective: Free to use and modify.
- Community-driven: Rapid development and innovation.
- Flexibility: Integrates into any stack.

WHAT IS SOFTWARE COMPOSITION ANALYSIS (SCA)?

SCA tools identify and manage OSS components in a codebase, tracking:

- Dependencies and their versions.
- Known vulnerabilities (e.g., CVE database).
- License compliance.

Why it matters: Manual tracking is impossible in complex projects.

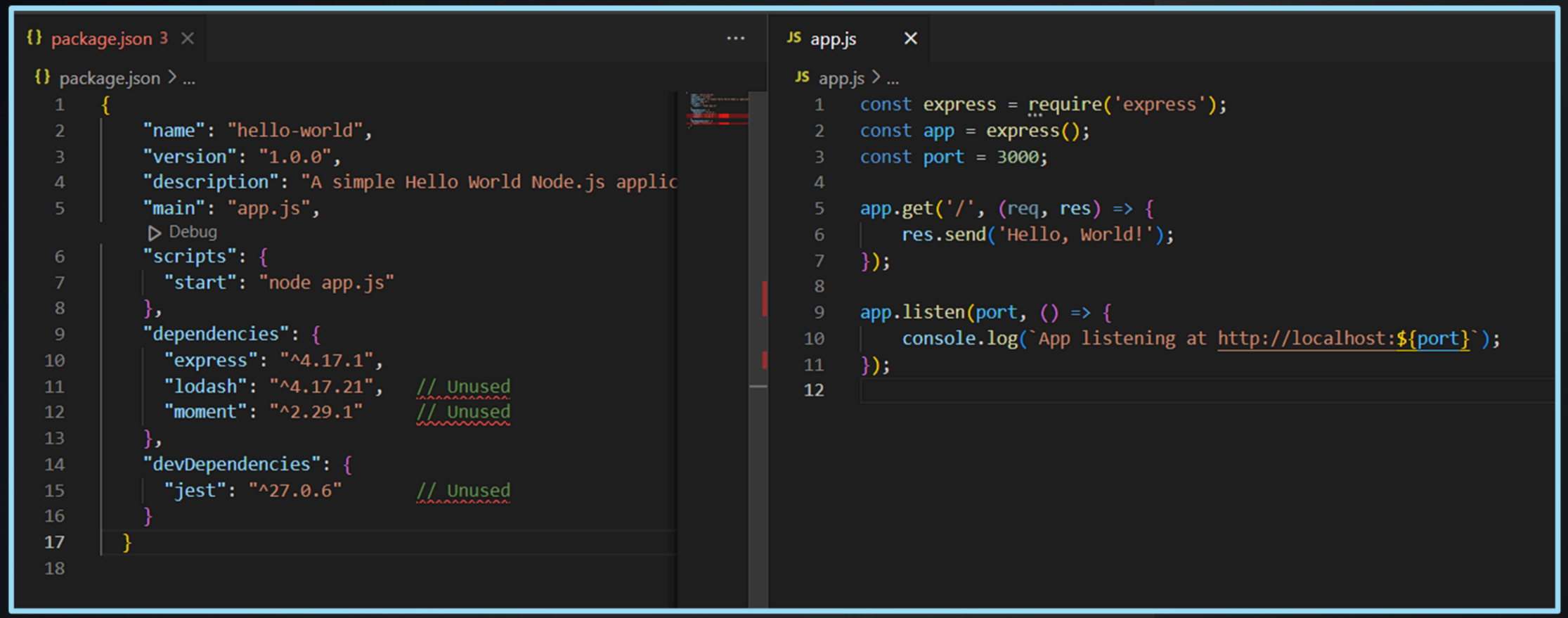
TYPES OF SCA SCANNING

SCA scanning can be categorized into several types based on what and how they analyze:

- Manifest Scanning: Analyzes package manifest files (e.g., package.json, pom.xml) to identify declared dependencies.
- Binary Scanning: Examines compiled binaries to detect embedded OSS components, useful when source code isn't available.
- Snippet Scanning: Looks for code snippets that match known OSS code, even if not declared as dependencies.
- Reachability Scanning: Determines if vulnerable code in dependencies is actually used by the application, prioritizing exploitable vulnerabilities.

Each type offers unique insights, and advanced SCA tools often combine multiple approaches for comprehensive analysis.

EXAMPLE - MANIFEST SCANNING



package.json 3

package.json > ...

```
{
    "name": "hello-world",
    "version": "1.0.0",
    "description": "A simple Hello World Node.js applic
    "main": "app.js",
    ▷ Debug
    "scripts": {
      "start": "node app.js"
    },
    "dependencies": {
      "express": "^4.17.1",
      "lodash": "^4.17.21",    // Unused
      "moment": "^2.29.1"      // Unused
    },
    "devDependencies": {
      "jest": "^27.0.6"        // Unused
    }
}
```

app.js

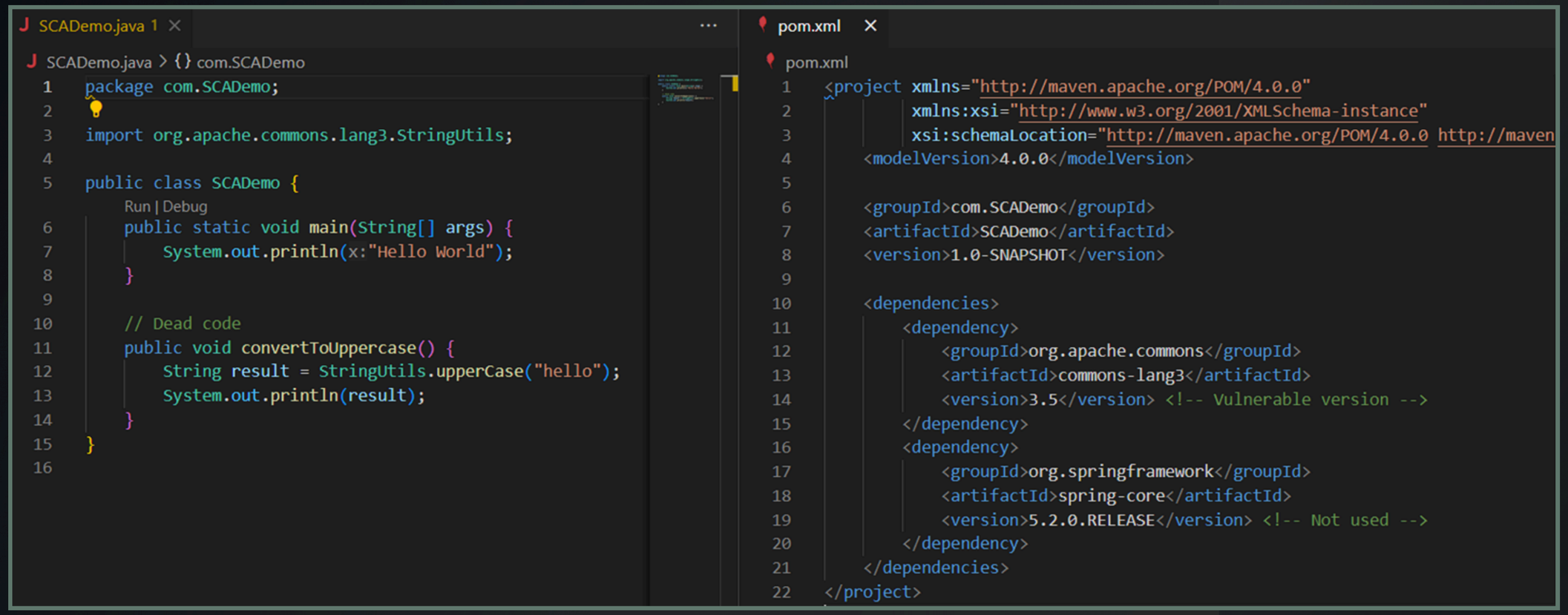
app.js > ...

```
const express = require('express');
const app = express();
const port = 3000;

app.get('/', (req, res) => {
    res.send('Hello, World!');
});

app.listen(port, () => {
    console.log(`App listening at http://localhost:${port}`);
});
```

EXAMPLE - REACHABILITY SCANNING



SCADemo.java 1

SCADemo.java > {} com.SCADemo

```java
package com.SCADemo;

import org.apache.commons.lang3.StringUtils;

public class SCADemo {
    Run | Debug
    public static void main(String[] args) {
        System.out.println(x:"Hello World");
    }

    // Dead code
    public void convertToUppercase() {
        String result = StringUtils.upperCase("hello");
        System.out.println(result);
    }
}
```

pom.xml

pom.xml

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.SCADemo</groupId>
    <artifactId>SCADemo</artifactId>
    <version>1.0-SNAPSHOT</version>

    <dependencies>
        <dependency>
            <groupId>org.apache.commons</groupId>
            <artifactId>commons-lang3</artifactId>
            <version>3.5</version> <!-- Vulnerable version -->
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-core</artifactId>
            <version>5.2.0.RELEASE</version> <!-- Not used -->
        </dependency>
    </dependencies>
</project>
```

SCA TOOLS

ENDOR LABS

snyk

Socket
socket.dev

JFrog Xray

Checkmarx

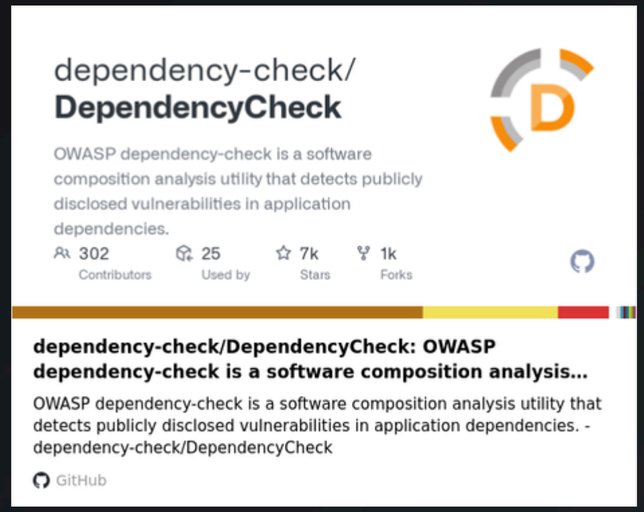
mend.io

vet

oligo

Semgrep

Kodem



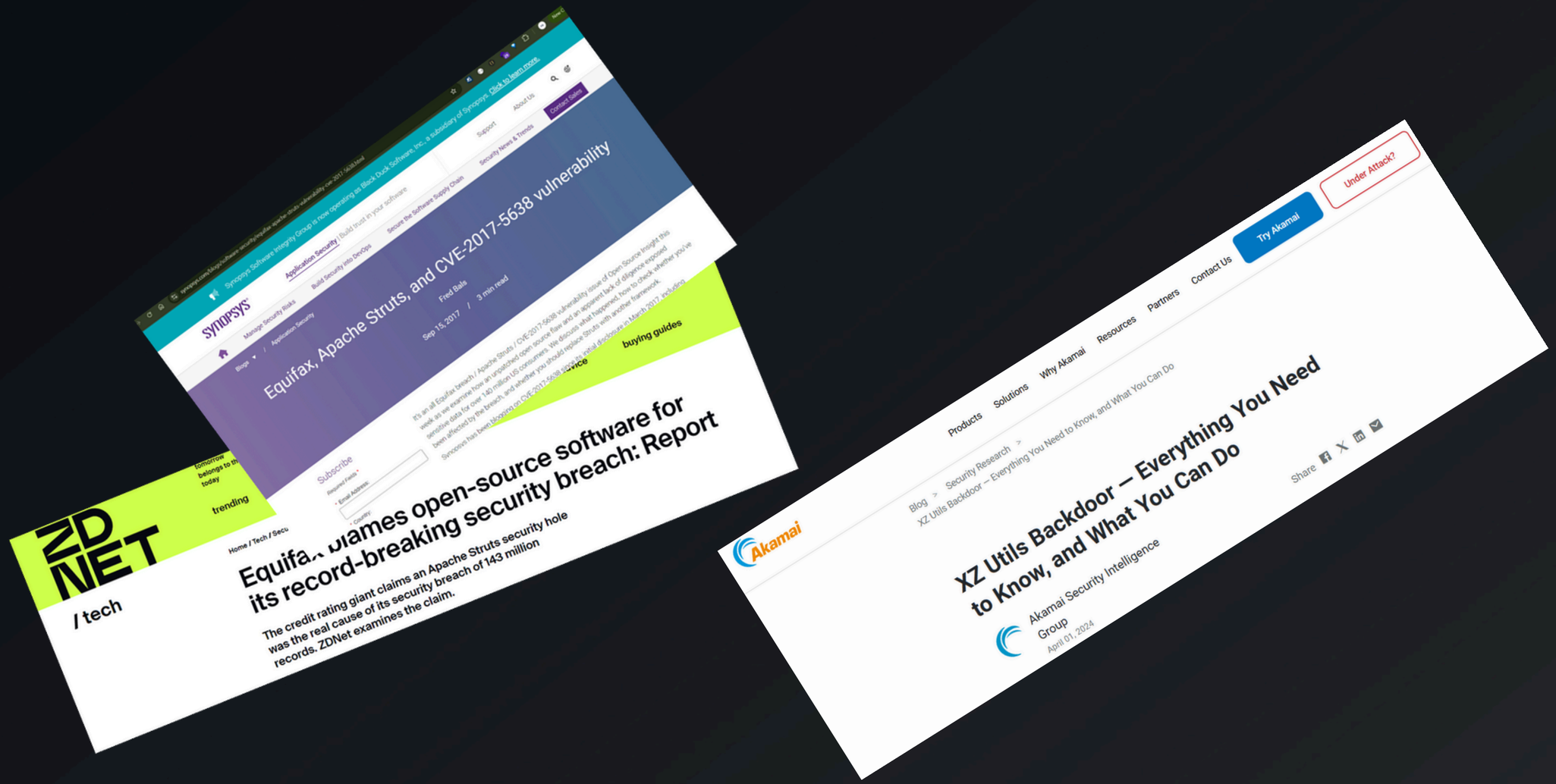
google/**osv-scanner**

WHAT ARE THE COMMON ISSUES IN SCA?

- New Vulnerabilities : CVEs identified in the open source package.
- Outdated dependencies: Unpatched vulnerabilities persist.
- Malicious packages: Typosquatting and supply chain attacks.
- Unmaintained projects: No updates, no fixes.
- Transitive dependencies: Hidden risks in nested libraries.

In 2024, 70% of OSS vulnerabilities were in transitive dependencies.

REAL-WORLD OSS DISASTERS



DAMN-VULNERABLE-SCA (SCAGOAT)

The Damn Vulnerable SCA project is a deliberately vulnerable application designed to help users understand and test SCA tools.

- Purpose: Provides a safe environment to learn about OSS vulnerabilities and how SCA tools detect them.
- Features: Includes known vulnerabilities like Log4Shell, malicious packages, and more.
- Usage: Clone the repo, build the Docker image, and run SCA scans to see vulnerabilities in action.

Why it matters: Hands-on experience helps security professionals and developers better understand SCA outputs and remediation steps.

harekrishnarai/**Damn-vulnerable-sca**

Damn Vulnerable SCA Application

6 Contributors | 1 Issue | 36 Stars | 32 Forks

harekrishnarai/Damn-vulnerable-sca: Damn Vulnerable SCA Application

Damn Vulnerable SCA Application. Contribute to harekrishnarai/Damn-vulnerable-sca development by creating an account on GitHub.

GitHub

MITIGATION STRATEGIES

- Automate SCA: Integrate into CI/CD pipelines.
- Patch promptly: Use tools like Dependabot for auto-updates.
- Minimize dependencies: Audit and remove unused libraries.
- Monitor actively: Subscribe to vulnerability feeds (e.g., NVD).
- Educate teams: Train developers on secure powołanie and dependency management.

FUTURE OF OSS SECURITY

- AI-driven SCA: Predictive vulnerability detection.
- Zero-trust supply chains: Verifying every component’s origin.
- Regulatory push: Laws like EU’s CRA mandating SBOMs.
- Community efforts: Projects like OpenSSF improving OSS security.

OSS is a double-edged sword: powerful but risky. SCA is your shield, but it’s not foolproof.

- Understand your dependencies.
- Leverage SCA tools effectively.
- Stay proactive with updates and monitoring.

"Secure software starts with secure components."

Q&A

THANK YOU

hi@harekrishnarai.me

harekrishnarai.me

linkedin.com/in/harekrishnarai

github.com/harekrishnarai