**1. What is DevSecOps and Why It Matters?**

**What is DevSecOps?**

DevSecOps = **Development + Security + Operations**
It is a modern approach where **security is built into every stage** of the software development lifecycle rather than added at the end.

Earlier:

- Developers wrote code
- Ops deployed it
- Security checked at the last stage

Problem → late detection, delays, security issues in production.

DevSecOps fixes this by:

- Integrating security tools inside CI/CD
- Automating security checks
- Making developers + DevOps + Security all responsible
- Detecting vulnerabilities early, cheaply, and faster

---

**Why DevSecOps Matters Today**

**Attacks are increasing**

Modern apps use APIs, microservices, containers, and open-source libraries.
These increase the attack surface.

**Security at the end is too late**

Fixing vulnerabilities late is:

- Slow
- Expensive
- Risky

DevSecOps prevents this.

**Faster releases need automated security**

Companies deploy hundreds of times a day.
Manual security doesn't scale.
DevSecOps automates:

- SAST
- SCA

- DAST
- Secrets scanning
- IaC security
- Container scanning

**Better collaboration**

Instead of “security vs developers,” DevSecOps creates a **security culture** across teams.

**Compliance and audit readiness**

DevSecOps pipelines automatically enforce:

- security policies,
- secure coding practices,
- and governance rules.

This makes organizations audit-ready at all times.

---

**2. The “Shift-Left” Approach**

**What does Shift-Left mean?**

Shift-Left means **moving security earlier in the SDLC pipeline**
(“left side” of the DevOps workflow).

Traditional SDLC:

Plan → Develop → Test → Deploy → Security (at the end)

Shift-Left SDLC:

Security embedded at: Design → Coding → Build → Test → Deploy

---

**Why Shift-Left is Important**

**Security issues detected earlier**

Fixing issues early reduces:

- cost
- rework
- production outages

**Improves code quality**

Developers get immediate feedback from:

- SAST

- SCA
- Secret scanning
- IaC scanning
- Container scanning

**Prevents vulnerable builds**

Shift-Left adds **quality gates** that block:

- critical vulnerabilities
- insecure dependencies
- misconfigured cloud resources

**Creates secure coding culture**

Developers become aware of:

- OWASP Top 10
- Secure code patterns
- Dependency hygiene
- Secrets management

**Supports rapid CI/CD**

Automation ensures:

- faster releases
- fewer manual reviews
- consistent security

---

**Shift-Left Tools in DevSecOps Pipeline**

| **Activity** | **Tools** |
|---|---|
| SAST | SonarQube, Semgrep, Checkmarx |
| SCA | Snyk, Mend, OWASP Dependency-Check |
| Secrets Scanning | GitLeaks, TruffleHog |
| IaC Security | Checkov, TFLint |
| Container Scanning | Trivy, Anchore |
| DAST | OWASP ZAP |

---

**3. Building a Shared Responsibility Model**

**What is a Shared Responsibility Model in DevSecOps?**

It means **security is not owned by a single team** anymore.
All teams—Dev, Ops, Security, Cloud—share responsibility.

---

**Why Shared Responsibility Is Needed**

**Modern applications are complex**

Security must be handled at:

- code level
- infrastructure level
- pipeline level
- runtime level

**Security can't scale as a separate department**

Traditional security teams are overloaded.
DevSecOps distributes security ownership.

**Faster releases require distributed security**

Every team plays its part.

---

**Who Is Responsible for What?**

**Development Team**

- Write secure code
- Fix SAST/SCA issues
- Avoid hardcoded secrets
- Implement secure API patterns
- Follow OWASP best practices

**DevOps Team**

- Build secure CI/CD pipelines
- Integrate scanners (SAST, SCA, DAST)
- Manage infrastructure automation
- Secure build agents, runners, registries

**Security / AppSec Team**

- Set policies

- Define vulnerability thresholds
- Threat modeling
- Perform penetration testing
- Review critical findings

**Cloud / Infra Team**

- Apply hardening
- Least privilege access (IAM)
- Network segmentation
- Logging & monitoring

**Management / Leadership**

- Promote security culture
- Approve tool investments
- Ensure compliance

---

**Shared Responsibility in Pipeline (Example)**

**Developer pushes code →**

- Secrets scanning
- SAST triggers
- Dependency scanning

**Build stage →**

- SCA
- IaC scans
- Container scanning

**Deploy stage →**

- DAST
- Policy-as-Code validation

**Runtime stage →**

- SIEM monitoring
- Anomaly detection
- Threat detection

Every team contributes to security in **every stage**.

## Introduction to Software Composition Analysis (SCA)

### What is SCA?

**Software Composition Analysis (SCA)** is the process of detecting and managing security risks in **open-source components, third-party libraries, and dependencies** used in an application.

Today, 70–90% of application code is **open-source packages**, not custom code.
Example:

- Spring Boot → dozens of libraries
- NodeJS → hundreds of dependencies
- Python → pip installs multiple libraries

SCA helps you identify:

- Known vulnerabilities (CVE database)
- Outdated/open-source packages
- Malicious libraries
- License compliance issues (GPL, MIT, Apache)
- Dependency risks from transitive packages

### Why SCA matters in DevSecOps?

#### Detect vulnerabilities early

Most breaches come from vulnerable libraries (e.g., Log4Shell).

#### Prevent risky builds

The CI pipeline blocks deployments when libraries are insecure.

#### Reduce security debt

SCA prevents vulnerability accumulation in codebases.

#### Compliance & license governance

Organizations need to avoid legal issues from risky licenses.

#### Enables Shift-Left security

SCA integrates into:

- Pre-commit
- CI pipelines
- PR review

This ensures vulnerabilities never reach production.

---

**Popular SCA Tools**

- **Snyk**
- **OWASP Dependency-Check**
- **Mend (WhiteSource)**
- **JFrog Xray**
- **GitHub Dependabot**
- **GitLab Dependency Scanning**
- **Trivy (SCA + container scan)**

**2. Finding Vulnerabilities in Dependencies**

Dependencies often have:

- Publicly known vulnerabilities (CVE list)
- Unsafe outdated versions
- Transitive dependencies that contain risks
- Unpatched third-party packages

Example:

flask==0.12.2
Contains CVE-2018-1000656 → Remote Code Execution

**Where vulnerabilities come from?**

**A. Direct dependencies**

Defined in:

- package.json
- pom.xml
- requirements.txt

**B. Transitive dependencies**

These are dependencies required by your dependencies.

Example:
express → pulls 20 more libs automatically.

**C. Vulnerable base images**

Docker images may use:

- Outdated OS packages

- Vulnerable versions (e.g., openSSL)

**D. Unmaintained libraries**

Packages abandoned for years → high risk.

**How DevSecOps pipelines detect dependency vulnerabilities**

**Pipeline flow:**

Commit → SCA Scan → Build → Container Scan → Deploy

**Scans include:**

- CVE lookup
- Version comparison
- Package authenticity
- License check
- Malware scanning (in advanced tools)

---

**Examples of real dependency vulnerabilities**

**1. Log4j (CVE-2021-44228)**

Affected: Java apps
Impact: Remote code execution
Fix: Upgrade to log4j-core ≥ 2.17.1

**2. Lodash (JS Library)**

Prototype pollution
Fix: Upgrade to 4.17.21

**3. Apache Struts**

Critical RCE
Fix: Patch immediately

**3. Hands-On with an SCA Tool (Trainer-Friendly Demo)**

Here is a full **hands-on lab** using **OWASP Dependency-Check** and **Snyk**.
You can use *either* depending on participant comfort.

---

**Option 1: Hands-On with Snyk (Simple & Cloud-Based)**

**Step 1: Install Snyk CLI**

curl -s https://static.snyk.io/cli/latest/snyk-linux -o snyk

chmod +x snyk

sudo mv snyk /usr/local/bin/

**Step 2: Authenticate**

snyk auth

**Step 3: Run SCA Scan**

For Node:

snyk test

For Java:

snyk test --file=pom.xml

For Python:

snyk test --file=requirements.txt

**Output Includes:**

- Vulnerable libraries
- CVE IDs
- Severity
- Fix version
- Remediation suggestions

**Step 4: Monitor continuously**

snyk monitor

---

**Option 2: Hands-On with OWASP Dependency-Check**

**Install dependency-check**

sudo apt update

sudo apt install dependency-check -y

**Scan Java project**

dependency-check.sh --project MyApp --scan .

**Output report generated as:**

dependency-check-report.html

Open in browser:

file:///home/ubuntu/dependency-check-report.html

**Report shows:**

- Libraries with vulnerabilities
- CVE list
- Severity score
- Suggested fixes
- Dependency tree

---

**Option 3: Containerized SCA using Trivy (Bonus)**

**Install:**

sudo apt install trivy -y

**Scan dependencies:**

trivy fs .

**Scan Docker image:**

trivy image python:3.9

---

**How SCA integrates in CI/CD pipelines**

Example: **GitHub Action**

```
name: SCA Scan
on: [push]

jobs:
  sca:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Run Snyk
        uses: snyk/actions/node
        with:
          command: test
```

Example: **Jenkinsfile**

```
stage('SCA') {
  steps {
```

```
    sh 'snyk test'
  }
}
```

---

**Key Takeaways for Participants**

**✓ SCA checks third-party libraries & dependencies**

**✓ Automatically detects known CVEs**

**✓ Prevents deployment of vulnerable packages**

**✓ Essential part of "shift-left" security**

**✓ Must be integrated into CI/CD**

**✓ Tools: Snyk, OWASP Dependency-Check, Mend, Trivy**

**Common Code Flaws (OWASP Top 10)**

The OWASP Top 10 is a globally recognized list of the most critical security risks in modern applications.
These flaws commonly appear in code and can be identified using SAST, code reviews, and security scanning.

---

**A01: Broken Access Control**

Occurs when users can access data or functionalities they should not.
Examples:

- IDOR (Insecure Direct Object Reference)
- Accessing another user's data using modified IDs
- Missing authorization checks

---

**A02: Cryptographic Failures**

Issues caused by weak or missing encryption.
Examples:

- Storing passwords in plaintext
- Using weak algorithms (MD5, SHA1)
- Missing TLS/SSL
- Hardcoded encryption keys

---

**A03: Injection**

User input is executed as a command or query.
Examples:

- SQL Injection
- Command Injection
- LDAP Injection
  Cause: Input not sanitized or parameterized.

---

**A04: Insecure Design**

Weaknesses introduced due to flawed architecture or design.
Examples:

- Missing validation layers
- Insecure workflows
- Business logic bypasses

---

**A05: Security Misconfiguration**

Incorrect security settings in application, server, or cloud.
Examples:

- Default passwords
- Open S3 buckets
- Debug mode enabled
- Unpatched systems

---

**A06: Vulnerable and Outdated Components**

Using outdated libraries, frameworks, or dependencies with known CVEs.
Examples:

- Log4j vulnerability
- Old JavaScript packages
- Unmaintained open-source components

---

**A07: Identification and Authentication Failures**

Problems related to login and session handling.
Examples:

- Weak passwords
- Session fixation
- Missing MFA
- Long-lived sessions

---

**A08: Software and Data Integrity Failures**

Issues where data or software integrity is compromised.
Examples:

- Untrusted deserialization
- Unsigned code
- Unsigned container images

---

**A09: Security Logging and Monitoring Failures**

Lack of proper logs, alerting, or monitoring.
Examples:

- Missing audit logs
- Logs without context
- No alerting for suspicious activity

---

**A10: Server-Side Request Forgery (SSRF)**

Application fetches a URL provided by a user without validation.
Examples:

- User provides URL to internal services
- Attackers access metadata endpoints (cloud-specific)

---

**2. Introduction to Static Application Security Testing (SAST)**

**What is SAST**

SAST (Static Application Security Testing) is a method of analyzing source code, bytecode, or binaries to identify security vulnerabilities without executing the application.

It is performed early in the SDLC and supports the shift-left approach.

---

**How SAST Works**

- Scans application source code
- Uses rule sets and patterns to detect insecure coding practices
- Maps findings to OWASP Top 10 and CWE standards
- Highlights line of code, file path, and remediation advice

---

**What SAST Detects**

- SQL injection patterns
- Cross-site scripting vulnerabilities
- Hardcoded secrets
- Unsafe system calls
- Cryptographic misuse
- Insecure API usage
- Command injection
- Error handling issues
- Unsafe deserialization

---

**Advantages of SAST**

- Early detection during development
- Varies by language support
- Integrates into CI/CD
- Provides code-level remediation
- No need to deploy or run the application

---

**Limitations of SAST**

- False positives may occur
- Cannot detect runtime-only issues
- Struggles with complex logic vulnerabilities

---

**3. Hands-On with a SAST Tool**

Below is a hands-on practical session guide, ideal for participants.

You may choose any of the following SAST tools:

- SonarQube
- Semgrep
- Checkmarx (commercial)
- Fortify (commercial)

Here, we take SonarQube and Semgrep as examples.

---

**Hands-On: SAST with SonarQube**

**Step 1: Start SonarQube (Docker example)**

docker run -d --name sonar -p 9000:9000 sonarqube:lts-community

Open:

http://<server-ip>:9000

Default login: admin / admin

---

**Step 2: Create a project**

- Create a local project
- Get the generated scanner commands

---

**Step 3: Install SonarScanner**

For Linux:

sudo apt install unzip -y

wget https://binaries.sonarsource.com/Distribution/sonar-scanner-cli/sonar-scanner-5.0.1.zip

unzip sonar-scanner-5.0.1.zip

export PATH=$PATH:~/sonar-scanner-5.0.1/bin

---

**Step 4: Run a SAST scan**

Inside project folder:

```
sonar-scanner \
  -Dsonar.projectKey=app \
  -Dsonar.sources=. \
  -Dsonar.host.url=http://<server-ip>:9000 \
```

-Dsonar.login=<token>

---

**Step 5: Review results**

SonarQube displays:

- Vulnerabilities
- Code smells
- Security hotspots
- Severity levels
- Exact file and line number
- Recommended fix

---

**Hands-On: SAST with Semgrep (Lightweight Option)**

**Step 1: Install Semgrep**

pip install semgrep

---

**Step 2: Run scan**

semgrep --config=auto .

---

**Step 3: View findings**

Semgrep automatically detects insecure patterns across:

- Java
- Python
- JavaScript
- Go
- C#
- PHP

---

**Step 4: Run OWASP Top 10 ruleset**

semgrep --config="p/owasp-top-ten" .

---