1. Automating Security Tools in the Pipeline

Modern DevSecOps pipelines rely heavily on security automation to ensure that every build, test, and deployment follows security best practices without manual intervention.

Automation ensures:

- Consistency across all builds
- Faster detection of vulnerabilities
- Early feedback to developers
- Compliance with organizational policies
- Reduced manual effort and human error

Key Areas of Security Automation

1. Source Code Scanning

Triggered automatically when a developer pushes code or raises a pull request.
Includes:

- SAST (source code vulnerabilities)
- Secrets scanning
- Linting for secure coding standards

2. Dependency & Library Scanning

Triggered during the build phase.
Tools analyze:

- Third-party libraries
- Open-source components
- CVEs in direct and transitive dependencies

3. Container Security Scanning

Happens after image build.
Scans:

- Base image vulnerabilities
- System packages
- Security misconfigurations

4. Infrastructure as Code (IaC) Scanning

Checks Terraform, CloudFormation, ARM, Kubernetes YAML for:

- Misconfigured cloud resources

- Open security groups
- Unencrypted storage
- Privileged containers

5. Dynamic Scanning (DAST)

Runs against deployed applications in staging/QA environments.
Detects runtime issues such as:

- Authentication problems
- Injection risks
- Server misconfigurations

6. Policy Automation

Security policies defined as code using tools such as:

- OPA (Open Policy Agent)
- Conftest
- Gatekeeper
  Automation ensures every build follows governance rules.

---

2. SAST, SCA, and Secret Scanning in Continuous Integration (CI)

Security scanning becomes effective only when integrated into CI and runs automatically on every commit.

Static Application Security Testing (SAST)

Purpose

Analyzes source code to detect vulnerabilities such as:

- SQL injection
- XSS
- Command injection
- Insecure API usage
- Hardcoded strings

When It Runs

- Pre-commit hooks
- Pull requests
- CI build stage

Tools

- SonarQube
- Semgrep
- Checkmarx
- Fortify

Software Composition Analysis (SCA)

Purpose

Scans third-party dependencies for known vulnerabilities and license issues.

Checks Performed

- CVE lookup
- Version comparison
- Dependency tree mapping
- License compliance

When It Runs

- During build stage in CI
- As PR checks
- As scheduled scans

Tools

- Snyk
- Mend
- OWASP Dependency-Check
- JFrog Xray
- GitHub Dependabot

Secret Scanning

Purpose

Identifies credentials exposed in code repositories.

Issues Detected

- API keys
- Tokens
- Passwords

- Certificates
- Private keys

When It Runs

- Pre-commit
- On every code push
- During CI pipeline

Tools

- GitLeaks
- TruffleHog
- GitHub Secret Scanning
- GitLab Secret Detection

Why These 3 Scans Run in CI

- Prevent vulnerable code from entering the main branch
- Provide developers with immediate feedback
- Reduce security debt
- Automate compliance
- Stop unsafe builds early

---

3. Using Gates to Block Risky Builds

A “gate” in CI/CD is a set of automated rules that decides whether a build should proceed or fail.

Security gates ensure that only secure and compliant builds reach deployment stages.

Types of Security Gates

1. SAST Gate

Triggers fail when:

- Critical vulnerabilities are found
- Hardcoded secrets exist in source code
- High-risk patterns detected (SQL injection, unsafe calls)

2. SCA Gate

Stops the build if:

- Libraries contain high/critical CVEs

- License risk detected (GPL in a proprietary app)
- Patch or upgrade required

3. Secrets Gate

Blocks:

- API keys in code
- Private keys committed accidentally
- Database passwords in configs

4. Container Scan Gate

Fails build if:

- Base image contains severe OS vulnerabilities
- Containers run as root
- Unpatched system packages found

5. IaC Gate

Stops deployment if IaC templates contain:

- Public S3 buckets
- Wide-open security groups (0.0.0.0/0)
- Disabled encryption
- Privileged Kubernetes pods

6. DAST Gate

Used in staging/QA.
Blocks promotion if:

- Authentication flaws found
- Injection detected
- Sensitive headers missing

How Gates Work in a Pipeline

Example CI Flow:

1. Developer pushes code
2. CI triggers
3. SAST, SCA, and secret scans run
4. Build runs only if gates pass

5. Container image is created
6. Container is scanned
7. IaC policies validated
8. Deployment allowed only if all gates are green

If any gate fails:

- Build stops
- Developer receives alerts
- Vulnerabilities must be fixed before retry

Gate Example (Jenkinsfile)

```
stage('SAST Scan') {
   steps {
      script {
         def result = sh(returnStatus: true, script: 'sonar-scanner')
         if (result != 0) {
            error("Build failed: SAST scan returned vulnerabilities")
         }
      }
   }
}
```

Security Risks in Terraform / CloudFormation

Infrastructure as Code (IaC) tools such as Terraform and AWS CloudFormation help automate cloud provisioning.
However, misconfigured IaC templates can introduce serious, large-scale security risks.

Key Security Risks

1. Publicly Accessible Resources

Accidentally exposing resources publicly.
Examples:

- Security Group allowing 0.0.0.0/0
- Public S3 buckets

- Public RDS databases
- Open AKS/EKS/K8s endpoints

Impact:
Attackers can directly reach internal services.

---

2. Unencrypted Storage

Missing encryption on storage resources:

- S3
- EBS
- RDS
- DynamoDB
- Azure Storage Accounts

Impact:
Sensitive data exposed if compromised.

---

3. Hardcoded Secrets

Storing credentials or keys directly in IaC files:

- AWS secret keys
- Database passwords
- Token strings

Impact:
Total environment compromise through a single leaked file.

---

4. Over-Privileged IAM Roles

IAM roles or policies that allow:

- s3:*
- iam:*
- ec2:*
- Wildcards "Action": "*"

Impact:
Privilege escalation and full account takeover.

---

5. Insecure Network Configurations

Examples:

- Open SSH/RDP access
- Missing NACL rules
- Flat VPC/Subnet design
- No segregation between private and public resources

Impact:
Network pivoting, lateral movement, and exposure.

---

6. Insecure Defaults

Examples:

- Unrestricted API Gateway
- Publicly exposed Lambda functions
- Default VPC usage
- No lifecycle rules for logs

Impact:
Unexpected public access and data retention risks.

---

7. Missing Logging/Monitoring

IaC missing resources such as:

- CloudTrail
- Config Recorder
- VPC Flow Logs
- GuardDuty

Impact:
No visibility into threats or changes.

---

8. Drift and Inconsistent Environments

If templates allow changes outside IaC control, untracked insecure states can occur.

---

9. Vulnerable Container Configurations

Terraform/K8s YAMLs may:

- Run containers as root
- Allow privilege escalation
- Use unscanned images

---

2. Scanning IaC for Misconfigurations

IaC security scanning tools automatically check Terraform or CloudFormation for misconfigurations before deployment.

Purpose of IaC Scanning

- Detect misconfigurations early (Shift-Left)
- Enforce security baselines
- Prevent insecure resources from deploying
- Block risky builds in CI/CD
- Ensure compliance (CIS, NIST, ISO)

Tools Used for IaC Scanning

Open-Source

- Checkov
- TFLint
- Terrascan
- KICS
- Trivy (IaC mode)

Cloud Native

- AWS Config
- Azure Policy
- GCP Security Scanner

Commercial

- Prisma Cloud
- Wiz
- Tenable
- Snyk IaC

What IaC Scanners Detect

Common Checks:

1. Public exposure (0.0.0.0/0)
2. Missing encryption (aws_kms_key not used)
3. Over-privileged IAM roles
4. Unrestricted network ACL rules
5. Missing logging or monitoring
6. Insecure KMS key policies
7. Dangerous Kubernetes settings
8. Missing MFA and password policies
9. IAM users instead of roles
10. Use of default VPCs

IaC Scanners analyze your files before creation of cloud resources, preventing misconfigurations from reaching production.

Integration in CI/CD Pipeline

Typical flow:

Commit → IaC Scan → Build → Deploy

If any high-risk misconfigurations are found:

- Build fails
- Deployment stops
- Developer receives actionable feedback

Pipeline Examples:

- GitHub Actions
- GitLab CI
- Jenkins
- Azure DevOps

---

3. Hands-On: Securing a Cloud Template

Below is a complete hands-on lab suitable for participants.

Hands-On Exercise: Scan & Fix Terraform Misconfigurations

Step 1: Install Checkov (IaC Scanner)

```
pip install checkov
```

Step 2: Create an intentionally insecure Terraform file

main.tf:

```
resource "aws_s3_bucket" "mybucket" {
  bucket = "demo-bucket-devsecops"
  acl    = "public-read"
}

resource "aws_security_group" "web_sg" {
  name   = "web-sg"
  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

This contains:

- Public S3 bucket
- Open SSH port
- Missing encryption

Step 3: Run IaC Scan

```
checkov -d .
```

Expected Findings:

- S3 bucket not encrypted
- S3 bucket publicly accessible
- Security group open to world
- Missing logging

Step 4: Fix the Security Issues

A. Secure the S3 bucket

```
resource "aws_s3_bucket" "mybucket" {
  bucket = "demo-bucket-devsecops"
}

resource "aws_s3_bucket_acl" "bucket_acl" {
  bucket = aws_s3_bucket.mybucket.id
  acl    = "private"
}

resource "aws_s3_bucket_server_side_encryption_configuration" "sse" {
  bucket = aws_s3_bucket.mybucket.id
  rule {
    apply_server_side_encryption_by_default {
      sse_algorithm = "AES256"
    }
  }
}
```

B. Secure the SSH Access

```
cidr_blocks = ["10.0.0.0/16"] # Private network only
```

Step 5: Re-run the scan

```
checkov -d .
```

Expected:

- No high-risk findings
- Warnings may remain but build can now proceed

Optional: CloudFormation Hands-On

Given a CloudFormation template:

BucketName: PublicBucket

AccessControl: PublicRead

Participants secure it using:

- Private ACL
- Logging enabled
- Encryption enabled

Then scan with:

checkov -f template.yaml

<u>**Introduction to Dynamic Application Security Testing (DAST)**</u>

What is DAST

Dynamic Application Security Testing (DAST) is a black-box security testing technique used to identify vulnerabilities **in a running application**.
DAST does not access the source code. Instead, it interacts with the application from the outside, just like an attacker would.

DAST simulates real-world attacks by sending inputs, requests, and payloads to the application and observing how the system behaves.

Characteristics of DAST

- Tests the application at runtime
- Detects vulnerabilities arising from configuration, server behavior, and application logic
- Language-agnostic (works for Java, .NET, Node, Python, etc.)
- Identifies flaws that are not visible in static code analysis
- Mimics attacker behavior

What DAST Can Detect

- Authentication and session management flaws
- Input validation issues
- Cross-Site Scripting
- SQL Injection

- Server configuration issues
- API endpoint exposure
- Missing security headers
- Weak error handling
- Overly permissive CORS

Limitations of DAST

- Cannot detect code-level vulnerabilities
- Requires a deployed/staged environment
- Testing may be slower than static analysis
- Needs proper configuration to reduce false positives

Common DAST Tools

- OWASP ZAP
- Burp Suite
- Nessus (web app scanning)
- AppScan
- Netsparker
- Acunetix

OWASP ZAP is widely used in DevSecOps pipelines due to its open-source availability and automation support.

---

2. Scanning a Running Application (DAST Process)

To run a DAST scan, the application must be deployed and accessible on a URL or API endpoint.

Typical workflow:

1. Deploy the application in a testing/staging environment
2. Configure the DAST scanner
3. Provide the application's URL (example: http://app-dev.testing.local/)
4. Run crawl/spider to map all available endpoints
5. Execute attack/active scan
6. Collect, analyze, and triage results

7. Fix vulnerabilities in code or configurations
8. Re-run the scan for validation

Components of a DAST Scan

- **Spider/Crawler**: Automatically discovers pages and endpoints
- **Passive Scan**: Observes traffic without sending attacks
- **Active Scan**: Sends attack payloads to exploit weaknesses
- **Report Generation**: Produces vulnerability reports with severity levels

---

When to Run DAST

- During QA/staging environment testing
- As part of nightly security testing
- As a release gate before production deployment

---

3. Hands-on: DAST Scan Demo (OWASP ZAP)

Below is a hands-on demonstration that participants can follow.
This uses OWASP ZAP in Docker for simplicity.

Step 1: Start OWASP ZAP Docker Container

```
docker run -u root -p 8080:8080 -p 8090:8090 \

-d owasp/zap2docker-stable
```

Step 2: Identify the Application URL

Example running application:
http://localhost:3000/
or a demo vulnerable app such as Juice Shop:
http://localhost:3000

Step 3: Perform a Baseline Scan (Passive Scan)

```
docker run --network="host" owasp/zap2docker-stable zap-baseline.py \

-t http://localhost:3000 \

-r baseline-report.html
```

Output includes:

- Information findings
- Missing headers
- Cookie flags
- Outdated server configurations

Step 4: Perform an Active Scan

```
docker run --network="host" owasp/zap2docker-stable zap-full-scan.py \
-t http://localhost:3000 \
-r active-scan-report.html
```

Active scan attempts to exploit the application by sending attack payloads.

Step 5: View the Report

Open generated HTML files:

```
baseline-report.html
active-scan-report.html
```

Reports include:

- Vulnerability name
- Severity (High, Medium, Low)
- Affected URL
- Attack payload
- Suggested fix

Step 6: Fix and Re-test

Developers and DevSecOps teams review the findings:

- Authentication issues
- Missing headers
- Injection risks
- Error messages
- Server misconfigurations

Fix issues, deploy updated version, and re-run DAST.