KISIA S-개발자 4기

3일차: API 보안과 데이터 보호

내 API를 지키는 법: 보안 (Security)

Day 3 of 5

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

3일차 교육 일정

09:00-09:20	복습 및 보안 사고 사례
09:20-09:50	인증(Authentication) vs 인가(Authorization)
09:50-10:20	세션 vs 토큰 (JWT 심층 분석)
10:20-10:30	☕ 쉬는 시간
10:30-11:00	OWASP API Security Top 10
11:00-11:30	BOLA 취약점 분석
11:30-12:00	[실습 5] Postman 해킹 체험
12:00-13:00	🍽️ 점심 시간
13:00-14:00	[실습 5] 계속 - 취약 서버 공격
14:00-15:00	[실습 6] 보안 코드 수정
15:00-15:10	☕ 쉬는 시간
15:10-16:00	[실습 6] 계속 - 소유권 검증 미들웨어
16:00-16:30	SQL Injection & Rate Limiting
16:30-17:00	3일차 요약 및 Q&A

3일차 학습 목표

인증과 인가의 차이를 명확히 구분할 수 있다

Authentication(신원 확인) vs Authorization(권한 확인)의 개념과 실제 구현

JWT 토큰의 구조를 이해하고 직접 해부할 수 있다

Header, Payload, Signature의 역할과 jwt.io를 통한 실습

OWASP API Security Top 10을 설명할 수 있다

가장 위험한 API 취약점 10가지와 각각의 방어 방법

BOLA 공격을 실습하고 방어 코드를 작성할 수 있다

Postman으로 취약한 API를 공격하고, 소유권 검증 미들웨어 구현

SQL Injection과 Rate Limiting의 기본을 이해한다

주입 공격의 원리와 API 호출 제한의 필요성

Section 1

보안 사고 사례

왜 API 보안이 중요한가?

실제 API 보안 사고 사례

- **2019년 Facebook: 5억 3천만 명 개인정보 유출**
  → API에서 전화번호 조회 기능의 접근 제어 미흡
- **2021년 Peloton: 사용자 프로필 API 무단 접근**
  → 인증 없이 /api/user/{id} 접근 가능 → BOLA 취약점
- **2022년 Optus (호주 통신사): 1천만 명 데이터 유출**
  → 인증이 필요 없는 API 엔드포인트 노출
- **2023년 T-Mobile: 3,700만 고객 정보 유출**
  → API 게이트웨이의 인증 우회 취약점
- **공통점: '기능은 잘 동작했지만, 보안은 고려하지 않았다'**

API 보안의 3가지 핵심 축 (CIA + AAA)

- **기밀성 (Confidentiality) - 허가된 사람만 데이터를 볼 수 있는가?**
  → HTTPS, 암호화, 접근 제어
- **무결성 (Integrity) - 데이터가 변조되지 않았는가?**
  → 디지털 서명, 해시, HMAC
- **가용성 (Availability) - 서비스가 정상 운영되고 있는가?**
  → Rate Limiting, DDoS 방어
- **인증 (Authentication) - 당신은 누구인가?**
- **인가 (Authorization) - 당신은 무엇을 할 수 있는가?**
- **감사 (Auditing) - 누가 무엇을 했는가?**

Section 2

인증(Authentication) vs 인가(Authorization)

"당신은 누구?" vs "당신은 뭘 할 수 있나?"

인증(Authentication): "당신은 누구입니까?"

- **비유: 건물 입구에서 신분증을 보여주는 것**
- **사용자의 신원(identity)을 확인하는 과정**
- **대표적인 인증 방법:**
  - → ID/Password: 가장 기본적인 인증
  - → OAuth 2.0: 소셜 로그인 (Google, Kakao)
  - → API Key: 서비스 간 통신에 사용
  - → Certificate: 인증서 기반 (mTLS)
- **인증 실패 시: 401 Unauthorized 응답**
- **인증 = '문 앞에서 신분증 확인'**

인증 미들웨어 구현 (Express.js)

JavaScript (Express.js)

```
// 기본 인증 미들웨어
const authenticate = async (req, res, next) => {
  const token = req.headers.authorization?.split(' ')[1];

  if (!token) {
    return res.status(401).json({
      error: 'Authentication required',
      message: '토큰이 없습니다. 로그인해주세요.'
    });
  }

  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    req.user = decoded;  // { id: 1, role: 'user' }
    next();
  } catch (err) {
    return res.status(401).json({
      error: 'Invalid token',
      message: '유효하지 않은 토큰입니다.'
    });
  }
};
```

인가(Authorization): "이 방에 들어올 수 있나요?"

- **비유: 건물 안에서 특정 층에 접근할 수 있는 카드키**
- **인증된 사용자가 특정 리소스에 접근할 권한이 있는지 확인**
- **대표적인 인가 모델:**
  - → RBAC (Role-Based): 역할 기반 (admin, user, viewer)
  - → ABAC (Attribute-Based): 속성 기반 (부서, 직급)
  - → ACL (Access Control List): 개별 리소스별 권한 목록
- **인가 실패 시: 403 Forbidden 응답**
- **인가 = '건물 안에서 특정 층 출입 카드'**
- **핵심: 인증 없이 인가 불가! (순서가 중요)**

인가 미들웨어 구현 (RBAC)

JavaScript (Express.js)

```
// 역할 기반 접근 제어 미들웨어
const authorize = (...allowedRoles) => {
  return (req, res, next) => {
    // 인증이 먼저 완료되어 req.user가 있어야 함
    if (!req.user) {
      return res.status(401).json({ error: '인증이 필요합니다' });
    }

    if (!allowedRoles.includes(req.user.role)) {
      return res.status(403).json({
        error: 'Forbidden',
        message: '이 리소스에 접근할 권한이 없습니다.',
        yourRole: req.user.role,
        requiredRoles: allowedRoles
      });
    }
    next();
  };
};
```

```
// 사용 예시
app.delete('/api/users/:id',
  authenticate,                 // 1단계: 인증
  authorize('admin'),           // 2단계: 인가 (admin만)
  deleteUserHandler             // 3단계: 비즈니스 로직
);
```

인증 vs 인가 한눈에 비교

인증 (Authentication)

- 누구인가? (Who are you?)
- 신원 확인 과정
- 로그인, 토큰 발급
- 실패 시: 401 Unauthorized
- 비유: 신분증 확인
- 방법: ID/PW, OAuth, JWT
- 순서: 항상 먼저 실행

인가 (Authorization)

- 무엇을 할 수 있나? (What can you do?)
- 권한 확인 과정
- 역할, 퍼미션 체크
- 실패 시: 403 Forbidden
- 비유: 출입 카드 확인
- 방법: RBAC, ABAC, ACL
- 순서: 인증 이후 실행

Section 3

세션 vs 토큰

왜 요즘은 토큰(JWT)을 더 많이 쓰나?

세션(Session) 기반 인증

- **전통적인 웹 애플리케이션 인증 방식**
- **동작 원리:**
  - → 1. 사용자가 ID/PW로 로그인
  - → 2. 서버가 세션 ID를 생성하고 서버 메모리/DB에 저장
  - → 3. 세션 ID를 쿠키(Cookie)로 클라이언트에 전달
  - → 4. 이후 요청마다 쿠키에 세션 ID를 포함
  - → 5. 서버가 세션 ID로 사용자 정보를 조회
- **장점: 서버에서 세션을 즉시 무효화 가능**
- **단점: 서버에 상태(State)를 저장해야 함 → 확장 어려움**
- **단점: 서버가 여러 대일 때 세션 공유 문제 발생**

토큰(Token) 기반 인증 - JWT

- **현대 API에서 가장 널리 사용되는 인증 방식**
- **동작 원리:**
  - → 1. 사용자가 ID/PW로 로그인
  - → 2. 서버가 JWT 토큰을 생성하여 클라이언트에 전달
  - → 3. 클라이언트가 토큰을 로컬에 저장 (localStorage 등)
  - → 4. 이후 요청마다 Authorization 헤더에 토큰 포함
  - → 5. 서버가 토큰의 서명을 검증 (DB 조회 불필요!)
- **장점: 서버 상태 저장 불필요 (Stateless) → 수평 확장 용이**
- **장점: 마이크로서비스, 모바일 앱에 적합**
- **단점: 발급 후 즉시 무효화 어려움 (만료 시간까지 유효)**

세션 vs 토큰 비교

세션 (Session)

- 서버에 상태 저장 (Stateful)
- 쿠키로 세션 ID 전달
- 서버 메모리/DB 필요
- 즉시 무효화 가능
- 수평 확장 어려움
- 브라우저 중심 (쿠키)
- CSRF 공격에 취약

토큰 (JWT)

- 클라이언트에 상태 저장 (Stateless)
- Authorization 헤더로 전달
- 서버 DB 조회 불필요
- 만료 전 무효화 어려움
- 수평 확장 용이
- 모바일/API에 적합
- XSS 공격에 주의 필요

JWT (JSON Web Token) 구조 분석

- **JWT는 3개의 파트로 구성: xxxxx.yyyyy.zzzzz**
- **Header (헤더) - 토큰 타입과 알고리즘 정보**

  → { "alg": "HS256", "typ": "JWT" }
- **Payload (페이로드) - 사용자 정보 (Claims)**

  → { "sub": "1234", "name": "홍길동", "role": "admin" }
- **Signature (서명) - 위변조 방지**

  → HMACSHA256(base64(header) + '.' + base64(payload), secret)
- **각 파트는 Base64Url로 인코딩되어 점(.)으로 연결**
- **주의: Payload는 암호화가 아닌 인코딩! 누구나 읽을 수 있음**
- **비밀번호 등 민감 정보를 Payload에 절대 넣지 말 것!**

JWT 토큰 실제 예시

JWT Structure

실제 JWT 토큰 (점으로 구분된 3파트)
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0IiwibmFtZSI6Iu2Zjeq4uOuPmSIsInJvbGUiOiJ1c2VyIiw
iaWF0IjoxNzA5MjAwMDAwLCJleHAiOjE3MDkyODY0MDB9.
SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

디코딩 결과
Header: { "alg": "HS256", "typ": "JWT" }

Payload: {
"sub": "1234", ← 사용자 ID
"name": "홍길동", ← 사용자 이름
"role": "user", ← 역할
"iat": 1709200000, ← 발급 시간 (Issued At)
"exp": 1709286400 ← 만료 시간 (Expiration)
}

Signature: HMACSHA256(header + "." + payload, "your-secret-key")

JWT 생성 및 검증 코드 (Express.js)

JavaScript (Express.js)

```
const jwt = require('jsonwebtoken');
const SECRET = process.env.JWT_SECRET;  // 환경변수에서 비밀키 로드

// 로그인 API - JWT 생성
app.post('/api/login', async (req, res) => {
  const { email, password } = req.body;
  const user = await User.findByEmail(email);

  if (!user || !bcrypt.compareSync(password, user.password)) {
    return res.status(401).json({ error: '이메일 또는 비밀번호 오류' });
  }

  // JWT 토큰 생성 (유효기간: 1시간)
  const token = jwt.sign(
    { id: user.id, email: user.email, role: user.role },
    SECRET,
    { expiresIn: '1h' }
  );

  res.json({ token, message: '로그인 성공!' });
});
```

HANDS-ON

JWT 실습: jwt.io에서 토큰 해부하기

1. 브라우저에서 https://jwt.io 접속
2. 왼쪽 Encoded 영역에 샘플 토큰 확인
3. 오른쪽 Decoded 영역에서 Header, Payload 확인
4. Payload의 name 값을 '내 이름'으로 변경
5. 토큰이 실시간으로 변하는 것을 확인
6. Secret 키를 변경하면 Signature가 바뀌는 것을 확인
7. Invalid Signature 경고 메시지 확인
8. 핵심: 서명 없이는 토큰을 위조할 수 없다!

JWT 보안 주의사항

- **Payload에 민감 정보 넣지 않기 (비밀번호, 주민번호 등)**

  → Payload는 Base64 디코딩으로 누구나 읽을 수 있음!

- **반드시 HTTPS 사용 (토큰 탈취 방지)**

  → HTTP에서는 중간자 공격으로 토큰 탈취 가능

- **적절한 만료 시간 설정 (너무 길면 위험)**

  → Access Token: 15분~1시간 / Refresh Token: 7~30일

- **비밀키를 코드에 하드코딩하지 않기**

  → 환경변수(env) 또는 Vault 사용

- **alg: none 공격 방어 - 알고리즘 명시적 지정**

  → jwt.verify(token, secret, { algorithms: ['HS256'] })

Section 4

OWASP API Security Top 10

가장 위험한 API 취약점 10가지

OWASP란?

- **Open Web Application Security Project**
- **웹/API 보안 분야의 비영리 국제 재단**
- **보안 취약점 Top 10 목록을 주기적으로 발표**
  - → 웹 보안 Top 10: 2004년부터 발표
  - → API 보안 Top 10: 2019년 첫 발표, 2023년 업데이트
- **전 세계 개발자와 보안 전문가의 필독 자료**
- **API Security Top 10은 API 특화 취약점에 집중**
- **오늘 수업에서는 상위 5개를 집중적으로 다룹니다**

OWASP API Security Top 10 (2023)

- **API1: Broken Object Level Authorization (BOLA)**
  → 남의 데이터를 내 것처럼 접근
- **API2: Broken Authentication**
  → 인증 메커니즘의 결함
- **API3: Broken Object Property Level Authorization**
  → 객체 속성 수준의 권한 미흡
- **API4: Unrestricted Resource Consumption**
  → 리소스 소비 제한 없음 (Rate Limiting 부재)
- **API5: Broken Function Level Authorization**
  → 기능 수준의 권한 미흡 (일반 사용자가 관리자 API 호출)

OWASP API Security Top 10 (2023) - 계속

- **API6: Unrestricted Access to Sensitive Business Flows**
  → 비즈니스 로직 악용 (대량 구매, 자동화 봇)
- **API7: Server Side Request Forgery (SSRF)**
  → 서버가 공격자가 지정한 URL로 요청을 보내도록 유도
- **API8: Security Misconfiguration**
  → 보안 설정 오류 (CORS, 에러 메시지 노출)
- **API9: Improper Inventory Management**
  → API 버전 관리 미흡, 미사용 API 방치
- **API10: Unsafe Consumption of APIs**
  → 외부 API 응답을 신뢰하고 검증 없이 사용

API1: BOLA (Broken Object Level Authorization)

- 가장 흔하고 위험한 API 취약점 (Top 1!)
- 공격 원리: URL의 ID 값을 바꿔서 남의 데이터 접근

  → GET /api/orders/1001 → 내 주문 (정상)

  → GET /api/orders/1002 → 남의 주문 (공격!)
- 서버가 '이 주문이 요청자의 것인지' 검증하지 않을 때 발생
- 영향: 개인정보 유출, 금융 데이터 탈취, 계정 탈취
- 방어: 모든 API에서 소유권(ownership) 검증 필수

  → WHERE order_id = ? AND user_id = ?
- 가장 쉬운 공격이면서, 가장 자주 발생하는 취약점

API2: Broken Authentication (인증 결함)

- **인증 메커니즘 자체에 결함이 있는 경우**
- **대표적 사례:**
  - → 약한 비밀번호 정책 (1234, password 허용)
  - → Brute Force 공격 방어 없음 (로그인 시도 제한 없음)
  - → JWT 비밀키가 약하거나 노출됨
  - → 토큰 만료 시간이 너무 길거나 없음
  - → 비밀번호 재설정 토큰 예측 가능
- **방어 방법:**
  - → 강력한 비밀번호 정책 + MFA (다단계 인증)
  - → 로그인 실패 횟수 제한 + 계정 잠금
  - → JWT 비밀키 충분히 길게 (256bit 이상)

API3: Broken Object Property Level Authorization

- **객체의 속성(필드) 수준에서 권한 검증이 없는 경우**
- **두 가지 하위 취약점:**

  → Excessive Data Exposure: 필요 이상의 데이터 반환

  → Mass Assignment: 클라이언트가 보낸 데이터로 서버 필드 덮어쓰기
- **예시 - Excessive Data Exposure:**

  → GET /api/users/me → { name, email, password_hash, ssn }
- **예시 - Mass Assignment:**

  → PUT /api/users/me → { "role": "admin" } 로 권한 상승
- **방어: 응답에서 민감 필드 제거, 허용 필드 화이트리스트 사용**

API4: Unrestricted Resource Consumption

- **API 호출 횟수나 리소스 사용량에 제한이 없는 경우**
- **공격 시나리오:**
  - → 무한 API 호출로 서버 다운 (DoS)
  - → 대용량 파일 업로드로 디스크 가득 채우기
  - → SMS 인증 API를 반복 호출하여 비용 발생
  - → 페이지네이션 없이 전체 데이터 요청
- **방어 방법:**
  - → Rate Limiting: 분당/시간당 요청 수 제한
  - → 페이지네이션: 한 번에 반환하는 데이터 수 제한
  - → 파일 크기 제한, 요청 본문 크기 제한

API5: Broken Function Level Authorization

- **일반 사용자가 관리자 기능을 호출할 수 있는 경우**
- **공격 시나리오:**
  - → 일반: GET /api/users (사용자 목록 조회)
  - → 공격: DELETE /api/users/123 (사용자 삭제 - 관리자 기능)
  - → 공격: PUT /api/settings (시스템 설정 변경)
- **흔한 실수: 프론트엔드에서만 버튼을 숨기고, 서버에서는 권한 체크 안 함**
- **방어: 모든 엔드포인트에서 역할(role) 기반 권한 검증**
  - → app.delete('/api/users/:id', authorize('admin'), handler)
- **원칙: '프론트엔드 제어 != 보안' - 서버에서 반드시 검증!**

Section 5

BOLA 취약점 심층 분석

남의 주문 번호를 넣어 정보를 훔치는 기법

BOLA 공격 시나리오 (상세)

- **시나리오: 온라인 쇼핑몰의 주문 조회 API**
- **1단계: 정상 사용자 Alice가 자신의 주문 조회**

  → GET /api/orders/1001 → Alice의 주문 정보 반환 (정상)
- **2단계: Alice가 URL의 ID를 변경**

  → GET /api/orders/1002 → Bob의 주문 정보 반환 (공격 성공!)
- **3단계: 자동화 스크립트로 대량 데이터 수집**

  → for id in range(1, 10000): GET /api/orders/{id}
- **결과: 전체 고객의 주문 정보 (이름, 주소, 전화번호) 유출**
- **원인: 서버가 '이 주문이 요청자의 것인지' 확인하지 않음**

BOLA 취약한 코드 (위험!)

JavaScript - VULNERABLE

```
// ❌ 취약한 코드 - 누구나 모든 주문을 조회 가능
app.get('/api/orders/:id', authenticate, async (req, res) => {
  const orderId = req.params.id;

  // 문제: 주문의 소유자를 확인하지 않음!
  const order = await db.query(
    'SELECT * FROM orders WHERE id = ?',
    [orderId]
  );

  if (!order) {
    return res.status(404).json({ error: '주문을 찾을 수 없습니다' });
  }

  // 누구의 주문이든 그냥 반환!
  res.json(order);
});

// 공격자: GET /api/orders/1002 (남의 주문)
// 결과: Bob의 주문 정보가 그대로 반환됨!
```

BOLA 방어 코드 (안전!)

JavaScript - SECURE

```
// ✅ 안전한 코드 - 소유권 검증 추가
app.get('/api/orders/:id', authenticate, async (req, res) => {
  const orderId = req.params.id;
  const userId = req.user.id;  // JWT에서 추출한 사용자 ID

  // 핵심: WHERE 절에 user_id 조건 추가!
  const order = await db.query(
    'SELECT * FROM orders WHERE id = ? AND user_id = ?',
    [orderId, userId]
  );

  if (!order) {
    // 주문이 없거나 남의 주문이면 404 반환
    // (403을 반환하면 주문의 존재 여부가 노출됨!)
    return res.status(404).json({ error: '주문을 찾을 수 없습니다' });
  }

  res.json(order);
});
```

```
// 공격자: GET /api/orders/1002 (남의 주문)
// 결과: 404 Not Found - 정보 유출 없음!
```

BOLA 공격 시연 (curl)

Bash (curl)

```
# 1. 로그인하여 JWT 토큰 획득 (Alice 계정)
curl -X POST http://localhost:3000/api/login \
  -H "Content-Type: application/json" \
  -d '{"email":"alice@test.com","password":"alice123"}'

# 응답: { "token": "eyJhbGci..." }

# 2. Alice의 주문 조회 (정상)
curl http://localhost:3000/api/orders/1001 \
  -H "Authorization: Bearer eyJhbGci..."
# 응답: { "id": 1001, "user": "Alice", "item": "노트북" }

# 3. Bob의 주문 조회 (BOLA 공격!)
curl http://localhost:3000/api/orders/1002 \
  -H "Authorization: Bearer eyJhbGci..."

# 취약한 서버: { "id": 1002, "user": "Bob", "address": "..." }
# 안전한 서버: { "error": "주문을 찾을 수 없습니다" }
```

Section 6

[실습 5] Postman을 이용한 API 해킹 체험

취약한 서버를 직접 공격해보자!

HANDS-ON

[실습 5-1] 실습 환경 설정

1. 취약한 API 서버 프로젝트 다운로드
2. git clone https://github.com/crattack/KISIA_API.git
3. 프로젝트 디렉토리로 이동
4. cd KISIA_API && npm install
5. 환경변수 설정
6. cp .env.example .env
7. 서버 실행
8. npm start (http://localhost:3000)
9. Postman에서 새 Collection 생성: 'Day3-Security-Lab'

[실습 5-2] 취약한 서버 코드 구조

JavaScript (server.js)

```
// server.js - 취약한 API 서버 (교육용)
const express = require('express');
const jwt = require('jsonwebtoken');
const app = express();

// 사용자 데이터 (실습용 하드코딩)
const users = [
  { id: 1, email: 'alice@test.com', pw: 'alice123', role: 'user' },
  { id: 2, email: 'bob@test.com',   pw: 'bob123',   role: 'user' },
  { id: 3, email: 'admin@test.com', pw: 'admin123',  role: 'admin' }
];

// 주문 데이터
const orders = [
  { id: 1001, userId: 1, item: '노트북',  price: 1500000, addr: '서울...' },
  { id: 1002, userId: 2, item: '키보드',  price: 150000,  addr: '부산...' },
  { id: 1003, userId: 1, item: '마우스',  price: 50000,   addr: '서울...' },
];
```

HANDS-ON

[실습 5-3] Postman으로 로그인하기

1. Postman에서 새 Request 생성
2. Method: POST, URL: http://localhost:3000/api/login
3. Body 탭 → raw → JSON 선택
4. { "email": "alice@test.com", "password": "alice123" }
5. Send 클릭 → JWT 토큰이 응답됨
6. 토큰을 복사해두기 (이후 요청에서 사용)
7. jwt.io에서 토큰을 붙여넣고 Payload 확인
8. 사용자 ID, 이메일, 역할이 보이는지 확인

HANDS-ON

[실습 5-4] 정상 주문 조회

1. Postman에서 새 Request 생성
2. Method: GET, URL: http://localhost:3000/api/orders/1001
3. Headers 탭에서 Authorization 추가:
4. Key: Authorization, Value: Bearer {복사한토큰}
5. Send 클릭 → Alice의 주문 정보 확인
6. 응답: { id: 1001, item: '노트북', price: 1500000 }
7. 이것은 정상적인 동작입니다

HANDS-ON

[실습 5-5] BOLA 공격 실행!

1. 같은 Alice 토큰으로 Bob의 주문 조회 시도
2. URL을 변경: GET /api/orders/1002
3. Send 클릭 → Bob의 주문 정보가 보이는가?
4. 만약 보인다면: BOLA 취약점 발견!
5. 1003, 1004, 1005... 순차적으로 시도
6. 모든 사용자의 주문 정보를 수집 가능
7. 이것이 실제 해킹에서 가장 많이 사용되는 기법!

[실습 5-6] 공격 자동화 스크립트

Python (attack.py)

```
# Python으로 BOLA 공격 자동화
import requests

TOKEN = "eyJhbGci..."  # Alice의 토큰
BASE_URL = "http://localhost:3000/api/orders"
headers = {"Authorization": f"Bearer {TOKEN}"}

# 1부터 100까지의 주문 ID를 순차 조회
stolen_data = []
for order_id in range(1, 101):
    resp = requests.get(f"{BASE_URL}/{order_id}", headers=headers)
    if resp.status_code == 200:
        data = resp.json()
        stolen_data.append(data)
        print(f"[STOLEN] Order {order_id}: {data['item']} - {data['addr']}")

print(f"\n총 {len(stolen_data)}건의 주문 정보 탈취 완료!")

# 결과: 전체 고객의 이름, 주소, 전화번호, 주문 내역 유출
```

HANDS-ON

[실습 5-7] Function Level Authorization 공격

1. Alice 토큰으로 관리자 전용 API 호출 시도
2. GET /api/admin/users - 전체 사용자 목록 조회
3. DELETE /api/users/2 - Bob 계정 삭제 시도
4. PUT /api/users/1 - { role: 'admin' } 권한 상승 시도
5. 각 요청의 결과를 기록
6. 취약한 서버는 이 모든 것을 허용할 수 있음!
7. 방어: 모든 관리자 엔드포인트에 authorize('admin') 적용

HANDS-ON

[실습 5-8] Excessive Data Exposure 확인

1. GET /api/users/me 호출 - 내 정보 조회
2. 응답에 불필요한 정보가 포함되어 있는지 확인:
3. password_hash가 노출되는가?
4. 내부 ID, 생성일시 등이 노출되는가?
5. GET /api/users/2 (Bob의 정보) 조회 시도
6. Bob의 이메일, 전화번호 등 민감 정보 확인
7. 기록: 어떤 정보가 불필요하게 노출되는지 메모
8. API는 필요한 최소한의 데이터만 반환해야 함!

[실습 5] 공격 결과 정리

- **발견한 취약점을 정리해봅시다:**
- **BOLA: 다른 사용자의 주문/정보 접근 가능**

  → 원인: 소유권 검증 없음
- **Broken Function Level Auth: 관리자 API 무단 접근**

  → 원인: 역할(role) 기반 권한 체크 없음
- **Excessive Data Exposure: 민감 정보 노출**

  → 원인: 응답 필드 필터링 없음
- **Broken Authentication: 약한 비밀번호 허용**

  → 원인: 비밀번호 정책 없음
- **다음 실습에서 이 모든 취약점을 수정합니다!**

Section 7

[실습 6] 보안 코드 수정

취약점을 직접 고쳐보자 - Defense in Depth

[실습 6-1] 소유권 검증 미들웨어 설계

- **목표: 모든 API에서 재사용 가능한 소유권 검증 미들웨어 만들기**
- **설계 원칙:**
  - → 단일 책임: 미들웨어는 소유권 검증만 담당
  - → 재사용성: 주문, 프로필, 게시글 등 모든 리소스에 적용
  - → 실패 안전: 검증 실패 시 404 반환 (정보 유출 방지)
- **구현 순서:**
  - → 1. checkOwnership 미들웨어 함수 작성
  - → 2. 데이터베이스 쿼리에 user_id 조건 추가
  - → 3. 모든 리소스 접근 API에 미들웨어 적용
  - → 4. 테스트: Alice 토큰으로 Bob 데이터 접근 불가 확인

[실습 6-2] 소유권 검증 미들웨어 구현

JavaScript (middleware)

```
// middleware/checkOwnership.js
const checkOwnership = (resourceType) => {
  return async (req, res, next) => {
    const resourceId = req.params.id;
    const userId = req.user.id;  // JWT에서 추출

    let query;
    switch (resourceType) {
      case 'order':
        query = 'SELECT id FROM orders WHERE id=? AND user_id=?';
        break;
      case 'profile':
        query = 'SELECT id FROM profiles WHERE id=? AND user_id=?';
        break;
      default:
        return res.status(500).json({ error: 'Unknown resource' });
    }

    const resource = await db.query(query, [resourceId, userId]);
```

```
// middleware/checkOwnership.js
if (!resource) {
      return res.status(404).json({ error: '리소스를 찾을 수 없습니다' });
    }

    req.resource = resource;  // 다음 핸들러에서 사용
    next();
  };
};
```

[실습 6-3] 미들웨어를 API에 적용

JavaScript (routes)

```
// routes/orders.js - 수정 후

// 주문 조회 (소유권 검증 적용)
app.get('/api/orders/:id',
  authenticate,                // 1. 인증
  checkOwnership('order'),     // 2. 소유권 검증
  async (req, res) => {
    res.json(req.resource);    // 3. 이미 검증된 데이터 반환
  }
);
```

```
// routes/orders.js - 수정 후

// 주문 수정 (소유권 검증 + 입력 검증)
app.put('/api/orders/:id',
  authenticate,
  checkOwnership('order'),
  validateInput(['item', 'quantity']),  // 허용 필드만 수정 가능
  async (req, res) => {
    const { item, quantity } = req.body;
    await db.query('UPDATE orders SET item=?, qty=? WHERE id=?',
      [item, quantity, req.params.id]);
    res.json({ message: '주문이 수정되었습니다' });
  }
);
```

[실습 6-4] 관리자 API 보안 강화

JavaScript (authorize)

```
// middleware/authorize.js
const authorize = (...allowedRoles) => {
  return (req, res, next) => {
    if (!req.user) {
      return res.status(401).json({ error: '인증이 필요합니다' });
    }
    if (!allowedRoles.includes(req.user.role)) {
      // 로깅: 권한 없는 접근 시도 기록!
      console.warn(
        `[SECURITY] User ${req.user.id} attempted ${req.method} `
        + `${req.originalUrl} (role: ${req.user.role})`
      );
      return res.status(403).json({ error: '권한이 없습니다' });
    }
    next();
  };
};
```

```
// 관리자 전용 API에 적용
app.get('/api/admin/users', authenticate, authorize('admin'), ...);
app.delete('/api/users/:id', authenticate, authorize('admin'), ...);
app.put('/api/settings', authenticate, authorize('admin'), ...);
```

[실습 6-5] Excessive Data Exposure 방어

JavaScript

```
// ❌ Before: 모든 필드를 그대로 반환
app.get('/api/users/:id', authenticate, async (req, res) => {
  const user = await db.query('SELECT * FROM users WHERE id=?', [id]);
  res.json(user);
  // { id, email, password_hash, phone, ssn, created_at, ... }
});
```

```
// ✅ After: 필요한 필드만 선택하여 반환
app.get('/api/users/:id', authenticate, async (req, res) => {
  const user = await db.query(
    'SELECT id, name, email FROM users WHERE id=?',
    [req.params.id]
  );
  // DTO (Data Transfer Object) 패턴으로 한번 더 필터링
  const safeUser = {
    id: user.id,
    name: user.name,
    email: user.email,
    // password_hash, phone, ssn 등은 절대 포함하지 않음!
  };
  res.json(safeUser);
});
```

[실습 6-6] Mass Assignment 방어

JavaScript

```
// ❌ 취약: 클라이언트가 보낸 모든 필드로 업데이트
app.put('/api/users/me', authenticate, async (req, res) => {
  await db.query('UPDATE users SET ? WHERE id=?',
    [req.body, req.user.id]);  // role: 'admin' 도 반영됨!
});
```

```
// ✅ 안전: 허용된 필드만 화이트리스트로 지정
app.put('/api/users/me', authenticate, async (req, res) => {
  // 수정 가능한 필드를 명시적으로 지정
  const allowedFields = ['name', 'email', 'phone'];
  const updates = {};

  for (const field of allowedFields) {
    if (req.body[field] !== undefined) {
      updates[field] = req.body[field];
    }
  }

  // role, is_admin 등은 절대 수정 불가!
  await db.query('UPDATE users SET ? WHERE id=?',
    [updates, req.user.id]);
  res.json({ message: '프로필이 수정되었습니다' });
});
```

[실습 6-7] 입력 검증 미들웨어 (Joi 라이브러리)

JavaScript (Joi validation)

```
const Joi = require('joi');

// 주문 생성 스키마
const orderSchema = Joi.object({
  item: Joi.string().min(1).max(100).required(),
  quantity: Joi.number().integer().min(1).max(99).required(),
  address: Joi.string().min(5).max(200).required(),
  // userId, role 등은 스키마에 없으므로 자동 거부됨!
});

// 입력 검증 미들웨어
const validate = (schema) => {
  return (req, res, next) => {
    const { error, value } = schema.validate(req.body, {
      stripUnknown: true  // 스키마에 없는 필드 자동 제거!
    });
    if (error) {
      return res.status(400).json({
        error: '입력값 오류',
        details: error.details.map(d => d.message)
      });
    }
    req.body = value;  // 검증된 값만 사용
    next();
  };
};
```

HANDS-ON

[실습 6-8] 수정된 코드 보안 테스트

1. 서버 재시작: npm start
2. Alice 토큰으로 다시 로그인
3. 테스트 1: GET /api/orders/1002 → 404 확인 (BOLA 방어)
4. 테스트 2: DELETE /api/users/2 → 403 확인 (권한 방어)
5. 테스트 3: PUT /api/users/me {role:'admin'} → role 변경 안됨
6. 테스트 4: GET /api/users/me → password_hash 미노출 확인
7. 테스트 5: POST /api/orders (잘못된 입력) → 400 확인
8. 모든 테스트를 통과하면 실습 완료!

[실습 6] 보안 체크리스트

- **BOLA 방어: 모든 리소스 접근에 소유권 검증 적용**
  → checkOwnership 미들웨어가 모든 GET/PUT/DELETE에 적용됨
- **인가 강화: 관리자 API에 역할 기반 접근 제어**
  → authorize('admin') 미들웨어가 관리자 엔드포인트에 적용됨
- **응답 필터링: 민감 정보가 API 응답에 포함되지 않음**
  → SELECT * 대신 필요한 필드만 명시적 조회
- **Mass Assignment 방어: 수정 가능한 필드 화이트리스트**
  → allowedFields 배열로 허용 필드만 수정 가능
- **입력 검증: Joi 스키마로 모든 입력값 검증**
  → stripUnknown: true로 예상치 못한 필드 자동 제거

Section 8

SQL Injection & Rate Limiting

추가적인 API 보안 기법

SQL Injection이란?

- 사용자 입력에 SQL 코드를 삽입하여 DB를 조작하는 공격
- 가장 오래되었지만 여전히 위험한 공격 기법
- 공격 예시:

  → 로그인 폼에 ' OR 1=1 -- 입력

  → 검색창에 '; DROP TABLE users; -- 입력
- 결과: 인증 우회, 데이터 유출, 데이터 삭제 가능
- 원인: 사용자 입력을 SQL 쿼리에 직접 연결
- 방어: Parameterized Query (준비된 구문) 사용
- 원칙: '절대 사용자 입력을 신뢰하지 마라!'

SQL Injection: 취약한 코드 vs 안전한 코드

JavaScript

```
// ❌ 취약한 코드 - 문자열 연결로 SQL 생성
app.post('/api/login', async (req, res) => {
  const { email, password } = req.body;
  // 사용자 입력이 SQL에 직접 삽입됨!
  const query = `SELECT * FROM users
    WHERE email = '${email}' AND password = '${password}'`;
  const user = await db.query(query);
  // email에 ' OR 1=1 -- 을 넣으면 모든 사용자 반환!
});
```

```
// ✅ 안전한 코드 - Parameterized Query 사용
app.post('/api/login', async (req, res) => {
  const { email, password } = req.body;
  // ? 자리에 파라미터가 안전하게 바인딩됨
  const query = 'SELECT * FROM users WHERE email = ? AND password
= ?';
  const user = await db.query(query, [email, password]);
  // SQL Injection 불가능!
});
```

SQL Injection 공격 시연

Bash (attack)

```
# 정상 로그인 요청
curl -X POST http://localhost:3000/api/login \
  -H "Content-Type: application/json" \
  -d '{"email":"alice@test.com","password":"alice123"}'

# SQL Injection 공격 - 인증 우회
curl -X POST http://localhost:3000/api/login \
  -H "Content-Type: application/json" \
  -d '{"email":"' OR 1=1 --","password":"anything"}'

# 실행되는 SQL:
# SELECT * FROM users WHERE email = '' OR 1=1 --' AND password = '...'
# OR 1=1 이 항상 true이므로 모든 사용자가 반환됨!
# -- 이후는 주석 처리되어 password 검증이 무시됨!

# 데이터 삭제 공격
# email에 '; DROP TABLE users; -- 를 입력하면?
# 테이블 전체가 삭제될 수 있음!
```

Rate Limiting (속도 제한)

- **일정 시간 내 API 호출 횟수를 제한하는 보안 기법**
- **목적:**
  - → DDoS 공격 방어 (서버 과부하 방지)
  - → Brute Force 공격 방어 (로그인 시도 제한)
  - → BOLA 자동화 공격 방어 (대량 데이터 수집 방지)
  - → API 비용 관리 (SMS, 이메일 API 남용 방지)
- **일반적인 기준:**
  - → 일반 API: 분당 60~100회
  - → 로그인 API: 분당 5~10회
  - → SMS 인증: 분당 1회, 시간당 5회
- **초과 시: 429 Too Many Requests 응답**

Rate Limiting 구현 (express-rate-limit)

JavaScript (express-rate-limit)

```
const rateLimit = require('express-rate-limit');

// 전역 Rate Limiter (모든 API에 적용)
const globalLimiter = rateLimit({
  windowMs: 60 * 1000,  // 1분
  max: 100,             // 최대 100회
  message: {
    error: 'Too many requests',
    message: '요청이 너무 많습니다. 1분 후 다시 시도해주세요.'
  }
});
app.use('/api/', globalLimiter);
```

```
// 로그인 전용 Rate Limiter (더 엄격!)
const loginLimiter = rateLimit({
  windowMs: 15 * 60 * 1000,  // 15분
  max: 5,                    // 최대 5회
  message: {
    error: '로그인 시도 횟수 초과',
    message: '15분 후 다시 시도해주세요.'
  }
});
app.post('/api/login', loginLimiter, loginHandler);
```

보안 헤더 설정 (Helmet.js)

JavaScript (helmet + cors)

```
const helmet = require('helmet');

// Helmet: 주요 보안 HTTP 헤더 자동 설정
app.use(helmet());

// 개별 설정
app.use(helmet.contentSecurityPolicy());    // XSS 방어
app.use(helmet.hsts());                     // HTTPS 강제
app.use(helmet.noSniff());                  // MIME 타입 고정
app.use(helmet.frameguard());               // 클릭재킹 방어
```

```
// CORS 설정 (허용된 도메인만!)
const cors = require('cors');
app.use(cors({
  origin: ['https://myapp.com', 'https://admin.myapp.com'],
  methods: ['GET', 'POST', 'PUT', 'DELETE'],
  credentials: true
}));

// 에러 메시지에 스택 트레이스 노출 방지
app.use((err, req, res, next) => {
  console.error(err.stack);  // 서버 로그에만 기록
  res.status(500).json({ error: '서버 오류가 발생했습니다'
});
});
```

API 보안 아키텍처 전체 그림

- **1단계: 네트워크 레벨 - HTTPS, 방화벽, WAF**
  → 모든 통신은 암호화, 악성 요청 필터링
- **2단계: API Gateway - Rate Limiting, 인증**
  → 요청 수 제한, JWT 검증, API 키 확인
- **3단계: 애플리케이션 레벨 - 인가, 입력 검증**
  → RBAC, 소유권 검증, Joi/Zod 입력 검증
- **4단계: 데이터 레벨 - 암호화, 접근 제어**
  → 비밀번호 해싱(bcrypt), DB 접근 권한 최소화
- **5단계: 모니터링 - 로깅, 알림, 감사**
  → 이상 행위 탐지, 보안 이벤트 알림

Defense in Depth (심층 방어)

외부 방어선

- HTTPS (TLS 1.3)
- WAF (Web Application Firewall)
- DDoS 방어 (Cloudflare)
- API Gateway (Kong, AWS)
- Rate Limiting
- IP Whitelist / Blacklist

내부 방어선

- JWT 인증 (Authentication)
- RBAC 인가 (Authorization)
- 소유권 검증 (Ownership)
- 입력 검증 (Input Validation)
- Parameterized Query
- 응답 필터링 (DTO)

보안 이벤트 로깅

JavaScript (winston)

```
const winston = require('winston');

// 보안 전용 로거
const securityLogger = winston.createLogger({
  level: 'warn',
  format: winston.format.json(),
  transports: [
    new winston.transports.File({ filename: 'security.log' })
  ]
});

// 보안 이벤트 기록 미들웨어
const logSecurityEvent = (event, req, details = {}) => {
  securityLogger.warn({
    event,
    timestamp: new Date().toISOString(),
    ip: req.ip,
    userId: req.user?.id || 'anonymous',
    method: req.method,
    path: req.originalUrl,
    userAgent: req.headers['user-agent'],
    ...details
  });
};

// 사용 예시
// logSecurityEvent('BOLA_ATTEMPT', req, { targetId: req.params.id });
// logSecurityEvent('AUTH_FAILURE', req, { email: req.body.email });
// logSecurityEvent('RATE_LIMIT_EXCEEDED', req);
```

API 보안 Best Practices

- 1. 모든 API에 인증(Authentication) 적용 - 예외 없음!
- 2. 모든 리소스 접근에 인가(Authorization) 적용
- 3. HTTPS 필수 - HTTP는 절대 사용 금지
- 4. 입력값 항상 검증 - 서버 측에서 반드시 수행
- 5. 최소 권한 원칙 - 필요한 것만 허용
- 6. 민감 데이터 암호화 - 저장 시, 전송 시 모두
- 7. Rate Limiting 적용 - 남용 방지
- 8. 보안 로깅 및 모니터링 - 이상 행위 탐지
- 9. 의존성 보안 업데이트 - npm audit 정기 실행
- 10. 보안 테스트 자동화 - CI/CD에 보안 스캔 포함

Section 9

3일차 요약

"보안은 기능 구현보다 우선되어야 한다"

핵심 개념 요약 (1/3) - 인증과 인가

✓ 인증(Authentication): 사용자 신원 확인 (401 Unauthorized)

✓ 인가(Authorization): 리소스 접근 권한 확인 (403 Forbidden)

✓ 세션은 서버 상태 저장 (Stateful), 토큰은 클라이언트 저장 (Stateless)

✓ JWT = Header + Payload + Signature (점으로 구분)

✓ JWT Payload는 암호화가 아닌 인코딩! 민감 정보 금지

✓ 인증 → 인가 순서가 반드시 지켜져야 함

핵심 개념 요약 (2/3) - OWASP & BOLA

✓ OWASP API Security Top 10: API 특화 취약점 목록

✓ BOLA (API1): 가장 흔하고 위험한 취약점 - ID 변조 공격

✓ 방어: SQL WHERE 절에 user_id 조건 추가 필수

✓ 소유권 검증 미들웨어로 모든 리소스 접근 보호

✓ 응답 필터링으로 불필요한 민감 정보 제거

✓ Mass Assignment 방어: 수정 가능 필드 화이트리스트

핵심 개념 요약 (3/3) - 추가 보안 기법

✓ SQL Injection: Parameterized Query로 방어

✓ Rate Limiting: 분당 요청 수 제한 (429 Too Many Requests)

✓ 보안 헤더: Helmet.js로 XSS, 클릭재킹 방어

✓ CORS: 허용된 도메인만 명시적으로 설정

✓ 보안 로깅: 이상 행위를 기록하고 모니터링

✓ Defense in Depth: 여러 겹의 보안 계층 구축

오늘 구현한 보안 코드 정리

- **authenticate - JWT 토큰 검증 미들웨어**

  → req.headers.authorization에서 토큰 추출 → jwt.verify

- **authorize(...roles) - 역할 기반 접근 제어**

  → req.user.role이 허용된 역할 목록에 있는지 확인

- **checkOwnership(type) - 소유권 검증**

  → 리소스의 user_id가 요청자의 ID와 일치하는지 확인

- **validate(schema) - 입력 검증**

  → Joi 스키마로 req.body 검증, 미허용 필드 제거

- **rateLimit - 요청 속도 제한**

  → express-rate-limit으로 분당/시간당 요청 수 제한

내일 예고: 4일차 - API 설계 패턴과 문서화

- **RESTful API 설계 원칙 심화**

  → 리소스 네이밍, URL 설계, HTTP 메서드 활용

- **API 버전 관리 전략**

  → URI 버전, 헤더 버전, 언제 버전을 올려야 하나?

- **에러 처리 표준화**

  → 일관된 에러 응답 형식, 에러 코드 체계

- **API 문서화 (Swagger/OpenAPI)**

  → 자동 문서 생성, Swagger UI 실습

- **페이지네이션, 필터링, 정렬**

  → 대용량 데이터 처리를 위한 API 패턴

Q & A

"보안은 기능이 아니라 기본이다. 모든 API의 첫 번째 요구사항이어야 한다."

내일 예고: 4일차: API 설계 패턴과 문서화 (Swagger)