# 웹 해킹 및 정보보안 실습

Chapter 3. WebGoat 모의 해킹 실습 (Injection ~ XSS)

# KEYWORD

이번 과정에서 다루게 될 핵심 취약점 키워드입니다.

# Keyword

## #Injection

SQL 쿼리문 조작 및 Blind 기반 데이터 추출

## #Path Traversal

디렉터리 조작 및 압축 파일 기반 시스템 침해

## #XSS

Cross Site Scripting을 통한 자바스크립트 실행

# Section 04. Injection

공격자가 설계한 악의적인 데이터가 개발자의
프로그래밍 코드에 침입해 발생하는 문제를 중점적으로 다룹니다.

# 01. SQL Injection (intro)

**SQL (Structured Query Language)**은 RDBMS(관계형 데이터베이스)의 데이터를 관리하기 위해 설계된 강력한 프로그래밍 언어입니다.

개발자가 짠 이 SQL 쿼리에, 공격자가 교묘하게 조작한 문자열이 섞여 들어가 데이터베이스가 완전히 털리는 것을 **SQL Injection**이라고 부릅니다.

# [기초] SQL 쿼리의 3가지 종류

데이터베이스를 조작하는 언어는 크게 세 부류로 나뉩니다.

- **DML (Data Manipulation):** 데이터를 직접 다루는 조작어.
- **DDL (Data Definition):** 테이블이나 구조 자체를 다루는 정의어.
- **DCL (Data Control):** 권한 및 확정을 다루는 제어어.

# DML (데이터 조작어) 상세

테이블 내부의 실제 데이터를 조작할 때 사용합니다.

- **SELECT:** 데이터를 조회합니다.
- **INSERT:** 데이터를 새롭게 추가합니다.
- **UPDATE:** 기존 데이터를 수정합니다.
- **DELETE:** 기존 데이터를 삭제합니다.

# DDL (데이터 정의어) 상세

데이터베이스나 테이블 등 구조적인 요소를 관리합니다.

- **CREATE:** 데이터베이스나 테이블 구조를 새로 생성합니다.
- **ALTER:** 기존 구조(컬럼 등)를 수정합니다.
- **DROP:** 구조 자체를 완전히 삭제합니다.
- **TRUNCATE:** 테이블 구조는 남기되, 내부의 모든 행을 삭제합니다.

# DCL (데이터 제어어) 상세

데이터베이스 접근 권한 및 트랜잭션을 관리합니다.

- **GRANT:** 사용자에게 특정 권한을 부여합니다.
- **REVOKE:** 부여한 권한을 회수합니다.
- **COMMIT:** 쿼리 작업 결과를 디스크에 확정(저장)합니다.
- **ROLLBACK:** 에러 발생 시 원래 상태로 복구합니다.

# 2번 실습: DML (SELECT) 기초

주어진 Employees 테이블에서 임직원 **Bob Franco**의 부서(department) 정보를 조회해야 합니다.

first_name이 'Bob' 이고 last_name이 'Franco'인 조건을 완성해 봅니다.

```
    select department from Employees where first_name='Bo
b' and last_name='Franco'
```

입력 후 Submit하면 'Marketing'이라는 부서 결과가 나옵니다.

# 3번 실습: DML (UPDATE) 기초

이번엔 임직원 Tobi Barnett의 부서를 'Sales'로 수정하는 문제입니다.

수정은 UPDATE 쿼리를 사용하며 SET으로 값을 넣고 WHERE로 대상을 지정합니다.

```
    update Employees set department='Sales' where first_na
me='Tobi' and last_name='Barnett'
```

실행 시 테이블 구조에서 부서가 완벽히 수정됨을 볼 수 있습니다.

# 4번 실습: DDL (ALTER) 기초

Employees 테이블의 스키마(구조) 자체를 변경하여, varchar(20) 타입의 'phone' 컬럼을 추가해야 합니다.

테이블 구조를 바꾸므로 ALTER 쿼리를 사용합니다.

```
alter table Employees add column phone varchar(20)
```

제출하면 테이블 맨 우측에 phone 이라는 새로운 빈 컬럼이 생성됩니다.

# 5번 실습: DCL (GRANT) 기초

unauthorized_user 계정에게 grant_rights 테이블에 대한 제어 권한을 새롭게 부여하는 문제입니다.

```
grant all privileges on grant_rights to unauthorized_u
ser
```

WebGoat가 사용하는 **HSQLDB** 공식 문서를 보면, 다른 DB와 다르게 별도의 DB명 명시 없이 테이블명만으로도 권한 부여가 정상적으로 성공합니다.

# 본격적인 SQL Injection의 원리

!

SQL Injection의 핵심은 **"개발자가 의도한 입력 문자열의 범위를 강제로 탈출"**하는 데 있습니다.

개발자가 `WHERE id='[입력값]'` 형태로 짜놓은 코드에,

공격자가 `admin'` 처럼 **싱글 쿼터(')**를 직접 치고 들어가면 쿼리 구조가 통째로 박살나며 해커의 명령어로 둔갑하게 됩니다.

# 전형적인 우회 패턴: ' or 1=1

다음 로그인 쿼리에 'aa' or '1'='1' 을 넣으면 어떻게 해석될까요?

```
Where id='aa' and pw='aa' or '1'='1'
```

- id='aa' and pw='aa': 이런 계정은 없으므로 **거짓(False)**
- or '1'='1': 1과 1은 무조건 같으므로 **참(True)**

# 왜 임의의 계정으로 로그인될까?

앞 조건이 틀려도 뒤의 OR True 조건 때문에 전체 결과가 **True**가 됩니다.

↓

DBMS는 "조건이 참이니까 테이블에 있는 데이터를 가져와야지" 라고 판단합니다.

↓

결과적으로 **테이블의 가장 첫 번째 계정(보통 관리자)** 정보를 반환하여 강제 로그인되는 마법이 펼쳐집니다.

# 9번 실습: String SQL Injection

코드 베이스가 다음과 같이 문자열을 직접 이어붙이도록(Concatenating) 짜여져 있습니다.

```
"SELECT * FROM user_data WHERE first_name = 'John' AND last_name = '" + LastName + "'";
```

목표는 특정 유저 이름 없이도 **전체 유저 테이블을 몽땅 조회**하는 것입니다.

입력칸에 ' or '1'='1 을 넣어 앞의 조건을 박살 내고 전체 True를 만들어 냅니다.

# 9번 실습 공격 성공

입력 결과, 서버에서 실행된 쿼리는 다음과 같이 조작됩니다.

```
SELECT * FROM user_data WHERE first_name = 'John' and last_name = ''
or '1' = '1'
```

AND 조건은 무시되고 OR 1=1 이 발동하며, Joe Snow, John Smith 등 테이블 내 모든 유저의 개인정보와 신용카드 번호가 화면에 쏟아져 나옵니다.

# 10번 실습: Numeric SQL Injection

이번엔 싱글 쿼터 없이 숫자형으로 이어진 쿼리를 타격하는 문제입니다.

```
      "SELECT * FROM user_data WHERE login_count = " + Login_Count + " AND
userid = " + User_ID;
```

Login_Count 칸에 123 #을 넣어보니 숫자 포맷 에러가 납니다. 즉 숫자 필터링이 있습니다.

반면 User_ID 에는 필터링이 없어, 여기에 123 or 1=1을 그대로 넣습니다. (숫자형은 쿼터가 필요 없습니다)

# 10번 실습 공격 성공

입력 결과, 실행된 쿼리입니다.

```
      SELECT * From user_data WHERE Login_Count = 123 and us
erid= 123 or 1=1
```

역시나 조건이 True가 되어 전체 데이터베이스의 기밀 정보를 모두 뽑아내게 됩니다.

# 02. SQL Injection (advanced)

조회를 넘어, 쿼리를 이어 붙이거나 응답이 없는 블라인드 환경에서
데이터를 강제로 뽑아내는 고급 기법을 실습합니다.

# 3번 실습: 다른 테이블 데이터 탈취

주어진 입력창은 원래 `user_data` 테이블을 조회합니다.

하지만 우리의 목표는 이 시스템 어딘가에 숨겨진 **user_system_data** 테이블에 있는 Dave의 패스워드를 훔쳐내는 것입니다.

이를 위해 세미콜론(;)을 사용하여 쿼리 여러 개를 연쇄적으로 실행하는 **SQL Query Chaining** 기법을 사용할 수 있습니다.

# 기법 1: Query Chaining 원리

세미콜론(;)을 사용해 앞 쿼리를 강제 종료하고 내 쿼리를 켭니다.

```
asdf'; select * from user_system_data--
```

- asdf': 조건문 강제 거짓 처리
- ;: 첫 번째 쿼리 문장 강제 종료

# Query Chaining 페이로드 분석

- `select * from ...`: 내가 원하는 테이블 조회 쿼리 삽입
- `--`: 뒤에 남은 기존 개발자의 쿼리를 전부 주석 처리하여 에러 방지

개발자가 의도한 쿼리 뒤에, 내가 보낸 완전한 SELECT 문장이 독립적으로 추가되어 연쇄적으로 실행되는 아주 파괴적인 기법입니다.

# Query Chaining 공격 성공

입력 결과, 서버는 첫 번째 쿼리(user_data)에서 빈 결과를 내지만, 두 번째 쿼리(user_system_data)가 연이어 성공하면서 화면에 **숨겨진 시스템 데이터 목록**을 뿌립니다.

이 결과로 Dave의 패스워드를 알아낼 수 있습니다. 하지만, 간혹 DB 설정이나 언어 제약으로 인해 세미콜론(;) 다중 실행이 막혀있는 서버가 존재합니다.

# 기법 2: UNION Based SQL Injection

백엔드 세팅에 따라 세미콜론(Chaining)이 막혀 있을 땐 **UNION** 연산자를 씁니다.

```
      asdf' union select userid, user_name, password, null, null, coo
kie, null from user_system_data--
```

UNION은 에러 없이 우회할 수 있지만, 지켜야 할 아주 까다로운 조건이 있습니다.

# UNION의 절대 제약 조건

UNION 연산자는 두 개의 쿼리 조회 결과를 표 하나로 합쳐주는 기능입니다. 이를 위해서는 다음 조건이 완벽히 맞아야 합니다.

> ⚠ **UNION 제약 사항**
>
> 1. 위 쿼리와 아래 쿼리의 **컬럼(열) 개수**가 완벽히 똑같아야 합니다.
> 2. 대응되는 위치의 **데이터 타입(문자열, 숫자 등)**이 일치해야 합니다.

# NULL을 이용한 컬럼 수 맞추기

원본 테이블(user_data)은 컬럼이 7개인데, 우리가 훔치려는 타겟 테이블(user_system_data)은 컬럼이 4개밖에 없습니다.

이럴 땐 부족한 컬럼 자리에 **데이터 타입에 구애받지 않는 'null'**을 대충 끼워 넣어서 강제로 짝수(7개)를 맞춰버립니다.

```
      union select userid, user_name, password, null, null,
cookie, null from ...
```

# UNION 공격 페이로드 적용

완성된 UNION 구문을 전송합니다.

```
     asdf' union select userid, user_name, password, null, null, cookie,
null from user_system_data--
```

위 페이로드를 제출하면 null 패딩 덕분에 에러 없이 병합되어, 마침내 숨겨져 있던 Dave의 패스워드인 passW0rD 탈취에 성공합니다.

# Blind SQL Injection

웹 서버가 보안을 위해 에러 로그나 결과를 가려버려서
화면에 아무것도 보이지 않을 때 쓰는 가장 무서운 공격 기법입니다.

# 보이지 않는데 어떻게 공격할까?

서버에게 참(True)/거짓(False) 단답형 질문을 집요하게 던집니다.

↓

**"비밀번호의 첫 글자가 A냐?"** → 에러 남 (False)

↓

**"비밀번호 첫 글자가 B냐?"** → 정상 로딩됨 (True)

↓

이 미세한 서버 반응 차이를 통해 텍스트를 한 땀 한 땀 역추적합니다.

# Blind SQLi의 3가지 판별법

- **Error based:** 고의로 수학적 에러(예: 1/0)를 내어 판별합니다.
- **Time based:** SLEEP(5) 등의 지연 명령어로 반응 속도를 측정합니다.
- **Content based:** 화면의 로직 결과(가입 성공/실패)가 달라지는 것을 보고 판별합니다.

# 5번 실습: 회원가입 우회 판별법 찾기

목표는 Tom 계정으로 로그인하는 것입니다. 하지만 비밀번호를 모릅니다.

REGISTER 탭에서 `tom' and 1<2--` (항상 참)을 넣고 가입을 시도하면 "User already exists"가 뜹니다.

`tom' and 1>2--` (항상 거짓)을 넣고 시도하면 "User created"가 뜹니다.

즉, **가입 실패 = 조건 참 / 가입 성공 = 조건 거짓** 이라는 확실한 Content based Blind 판별 기준을 얻었습니다!

# 백엔드 코드 분석 (REGISTER)

왜 저런 반응이 나올까요? 회원가입 로직 코드를 봅니다.

`src/main/java/org/owasp/webgoat/lessons/sqlinjection/advanced/SqlInjectionChallenge.java`

```
    String checkUserQuery = "select userid from sql_challenge_users wher
e userid = '" + username_reg + "'";
```

사용자가 입력한 username_reg 값을 문자열 더하기(+)로 쌩으로 붙이고 있습니다.

여기서 SQLi 취약점이 터진 것입니다.

# 백엔드 코드 분석 (LOGIN)

반면 로그인 로직 코드는 어떨까요?

`src/main/java/org/owasp/webgoat/lessons/sqlinjection/advanced/SqlInjectionChallengeLogin.java`

```
    var statement = connection.prepareStatement("select password from sql_challenge_users where userid = ? and password = ?");
```

로그인은 prepareStatement와 ? 를 써서 방어해 두었기 때문에, 로그인 폼에서는 아무리 쿼리를 날려도 공격이 먹히지 않았던 것입니다.

# 데이터 길이 파악 (Length)

이제 REGISTER 창의 취약점을 이용합니다. 알파벳을 찍어 맞추기 전에, 비밀번호가 몇 글자인지부터 찾아야 노가다를 줄입니다.

SQL 내장 함수인 `length()`를 사용합니다.

```
tom' and length(password)<30--
```

30자 미만인지 물어보니 가입 실패(True)가 뜹니다. 숫자를 조금씩 내려서 비교하다 보면 **길이가 정확히 23자리**라는 것을 알아낼 수 있습니다.

# 문자열 쪼개기 (Substring)

데이터가 23자리이니, 첫 글자부터 쪼개서 맞춰야 합니다.

`substring(데이터, 시작위치, 개수)` 함수를 씁니다.

- 예: `substring('abc', 1, 2)` → 'ab'가 추출됩니다.
- 우리는 `substring(password, 1, 1)`로 딱 하나만 뽑아냅니다.

# 아스키코드로 변환 (Ascii)

뽑아낸 문자를 그냥 부등호(<, >)로 비교하면 안 될까요?

SQL은 기본적으로 문자열 비교 시 **대소문자를 구별하지 못하는** 치명적인 단점이 있습니다. (A와 a를 같다고 인식함)

따라서 `ascii()` 함수를 겉에 씌워 문자를 '숫자(아스키코드)'로 변환해버리면, 완벽하게 대소문자를 구분하여 부등호로 범위를 좁혀갈 수 있습니다.

# 최종 페이로드 조합 및 수동 테스트

위 두 함수를 결합합니다.

```
tom' and ascii(substring(password,1,1))<100--
```

해석: "Tom의 비밀번호 첫 글자의 아스키코드가 100보다 작냐?"

입력 결과 가입 성공(False)이 뜹니다. 즉, 첫 글자의 아스키 코드는 100보다 크거나 같습니다.

# Burp Suite Intruder로 자동화 준비

23글자를 일일이 숫자를 바꿔가며 손으로 치는 것은 불가능에 가깝습니다. 프록시 툴 **Burp Suite의 Intruder** 기능을 써서 자동화합니다.

HTTP history에서 방금 쏜 회원가입 요청을 우클릭하여 **Send to Intruder**로 넘깁니다.

# Intruder 타겟 위치(Position) 지정

```
username_reg=tom'+and+ascii(substring(password,1,1))%3D§100§--...
```

자동으로 숫자가 대입될 부분인 100 영역을 드래그하고 **[Add §]** 버튼을 눌러 타겟을 잡습니다.

주의: 부등호(<) 대신 정확한 값을 찾기 위해 **등호(=)의 URL 인코딩 값인** %3D 로 변경해 줍니다.

# Intruder Payload 설정

[Payloads] 탭으로 이동해 설정합니다.

- Payload type: **Numbers**
- Number range Type: **Sequential**
- From: **32** / To: **126** (출력 가능한 아스키코드 범위)
- Step: **1** (1씩 증가하며 대입)

상단의 Start Attack을 누릅니다.

# 공격 결과 분석 및 탈취

결과 창에서 수백 개의 요청이 날아갑니다. [Length] 탭을 눌러 오름차순으로 정렬해 봅니다.

**유일하게 혼자 길이가 116인 결과값(Payload: 116)**이 튀어나옵니다. 116은 소문자 't'의 아스키코드입니다!

이 과정을 2번째 글자(`substring(password, 2, 1)`), 3번째 글자로 넘어가며 23번 반복하면 전체 비밀번호를 완벽히 탈취할 수 있습니다.

# 03. SQL Injection (mitigation)

지옥 같은 SQL Injection을 소스코드 단에서
어떻게 방어하고 완화할 수 있는지 구체적인 방안을 배웁니다.

# 방어 기법 1. Stored Procedure

개발 단에서 쿼리를 직접 짜지 않고, DB 서버 안에 쿼리를 함수화(Procedure)하여 올려둔 뒤, 필요할 때 데이터(파라미터)만 쏙쏙 던져주는 방식입니다.

전달된 데이터는 DB가 정해진 타입(정수형, 문자형)에 맞게 **강제 형변환(Casting)**을 해버리기 때문에, ' or 1=1 같은 텍스트를 넣어도 문법이 아닌 단순한 문자열로 취급되어 공격이 무력화됩니다.

# Stored Procedure의 맹점

프로시저를 쓴다고 100% 안전한 것은 아닙니다.

만약 프로시저 내부 코드에서 조차 다음과 같이 또다시 문자열을 더하기(+)로 이어붙이는 방식을 써버리면, 애써 프로시저를 도입한 의미가 사라지고 취약점이 다시 부활하게 됩니다.

# 동적 SQL의 위험성 (코드)

내부에서 문자열 결합을 사용하는 취약한 프로시저의 예시입니다.

```
      set @sql = 'SELECT * FROM users WHERE lastname = ' + @
LastName
```

공격자가 @LastName 변수에 ' or 1=1을 주입하면 프로시저 내부에서 그대로 공격 구문이 완성되어 버립니다.

# 방어 기법 2. Prepared Statement

가장 강력하고 보편적인 방어책입니다. **'미리 준비된 구문'**이라는 뜻입니다.

```
"SELECT * FROM user WHERE id = ?"
```

데이터 자리를 ? 로 비워둔 채 먼저 쿼리 뼈대를 DBMS에 보내 **컴파일(구조 분석)**을 끝내버립니다.

# Prepared Statement의 방어 원리

이후에 자바 코드에서 `ps.setString(1, pUsername)` 처럼 값을 쏙 넣어줍니다.

해커가 ? 자리에 `' or 1=1` 이라는 해킹 구문을 쑤셔 넣어도, 이미 뼈대 분석이 끝난 DB는 이를 문법이 아니라 **'or 1=1 이라는 텍스트 이름을 가진 유저'**를 찾는 단순 데이터로 취급하여 완벽히 방어해 냅니다.

# 방어 기법 3. ORM (JPA, Mybatis 등)

개발자가 직접 SQL 쿼리를 길게 타이핑하지 않고, 자바 객체 코드를 짜면 프레임워크가 알아서 안전한 쿼리로 바꿔 DB와 소통해주는 도구입니다. (내부적으로 Prepared Statement 사용)

> **※ MyBatis 사용 시 치명적 예외 (바인딩 실수)**
>
> **#{parameter}:** 안전하게 Prepared Statement 형태로 동작합니다. (권장)
>
> **${parameter}: 파라미터를 문자열 그대로 이어붙여버립니다. 이 방식을 쓰면 ORM을 쓰더라도 SQLi가 터집니다! 절대 주의!**

# 위험한 착각: 입력값 검증(필터링)

가장 많이 하는 초보적인 실수가 **"입력값에 스페이스바나 특정 키워드가 들어오면 튕겨내야지"** 하고 단순 필터링만 거는 것입니다.

이는 근본적인 조치가 아닌 임시방편에 불과합니다. 해커는 우회(Bypass) 기법을 통해 방화벽을 뚫어버립니다.

# 9번 실습: 공백 필터링 우회

이전의 UNION 공격 구문을 넣었더니 "Using spaces is not allowed!" 라고 공백이 막힙니다.

공백을 의미하는 건 스페이스바만 있는 게 아닙니다. **SQL 표준 다중 라인 주석 기호인 /**/를 띄어쓰기 대신 촘촘하게 박아 넣습니다.**

```
asdf'/**/union/**/select/**/userid,user_name,password...
```

보란 듯이 공백 필터링을 뚫고 100% 동일하게 전체 데이터 탈취에 성공합니다.

# 10번 실습: 예약어 (Keyword) 우회 준비

공백을 막아도 뚫리자, 이번엔 개발자가 "아예 SELECT, FROM 같은 SQL 예약어가 보이면 중간에서 지워버려야지" 하고 필터링을 추가했습니다.

실험 삼아 `111select222from333` 을 넣어보니, 쿼리에는 `111222333` 만 남았습니다.
서버가 중간에 낀 키워드를 정확히 지워버리고 있습니다.

# 글자 안에 글자 숨기기 기법

해커는 **글자 안에 글자를 겹쳐서 숨기는 기법**을 씁니다.

seSELECTlect , frFROMom 형태로 입력합니다.

방화벽 로직이 가운데 진짜 단어만 쏙 뽑아 지우고 나면, 양옆의 앞뒤 껍데기 알파벳이 다시 철썩 합쳐지며 완벽한 SELECT 와 FROM 이 서버 내부에서 부활되어 완성되어 버립니다.

# Prepared Statement의 예외: ORDER BY

만능 방패인 Prepared Statement나 ORM도 딱 하나 뚫리는 곳이 있습니다. 바로 **정렬 기준을 정하는** `ORDER BY` **구문**입니다.

ORDER BY 뒤에는 단순 데이터가 아니라 '컬럼명'이라는 뼈대 구조 자체가 와야 합니다. 여기에 파라미터 ? 를 쓰면 문법 에러가 나므로, 어쩔 수 없이 문자열을 더하기(+)로 이어붙이는 코딩을 하게 되며 여기서 치명적인 취약점이 부활합니다.

# 12번 실습: ORDER BY 구문 타격 (1)

목표: 블라인드 방식을 통해 webgoat-prd 서버의 IP 앞 3자리 알아내기.

Hostname 정렬 버튼을 눌러보니 `column=hostname` 으로 넘어갑니다. 이 파라미터가 백엔드에서 `ORDER BY hostname` 으로 이어붙여지고 있었습니다.

# 12번 실습: ORDER BY 구문 타격 (2)

여기에 column=ip 를 넣으니 IP순 정렬이 됩니다.

확신을 갖고 싱글 쿼터(')를 던져보니 서버가 500 에러를 뿜으며 자바 스택 트레이스 코드를 구구절절 노출합니다.

에러 로그에 ORDER BY ' 가 찍혀 있는 것을 보고, 이 구문이 SQLi에 완벽히 뚫려있음을 확신하게 됩니다.

# ORDER BY 기반 Blind SQLi (CASE WHEN)

ORDER BY 구문에 넣을 수 있는 조건 제어문인 CASE WHEN 문법을 주입합니다.

```
(case when (1<2) then hostname else ip end)
```

해석: "만약 1이 2보다 작으면(True) Hostname으로 정렬하고, 아니면 IP로 정렬하라"

서버가 Hostname으로 정렬해서 응답하면 내 조건이 참이고, IP로 정렬해서 응답하면 거짓입니다. 완벽한 참/거짓 판별 기준이 완성되었습니다.

# 서브쿼리 주입 및 자동화 탈취

이제 1<2 자리에 진짜 알아내고 싶은 복잡한 서브쿼리를 박아 넣습니다.

```
     substring((select ip from servers where hostname='webgoat-prd'),1,1)
='9'
```

Burp Suite Intruder를 켜서 9 부분을 타겟으로 잡고, 숫자 범위를 32~126(아스키코드)로 돌립니다.

정렬 결과가 뒤집히는 찰나를 캐치하여 IP 첫 글자가 104 임을 완벽하게 추출해 냅니다.

# 취약점 원인 분석: 소스코드 리뷰

왜 처음에 회원가입 단에서 SQL Injection이 터졌을까요? 백엔드 코드를 열어봅니다.

`src/main/java/org/owasp/webgoat/lessons/sqlinjection/advanced/SqlInjectionChallenge.java`

```
    String checkUserQuery = "select userid from sql_challenge_users wher
e userid = '" + username_reg + "'";
```

위와 같이 username_reg 파라미터를 쿼리 문자열에 그냥 + 기호로 쌩으로 이어붙인 구시대적 코딩 방식이 문제의 근원이었습니다.

# 올바른 방어 로직 비교 (Login 컨트롤러)

반면, 같은 프로젝트인데도 로그인을 담당하는 컨트롤러는 공격이 안 뚫렸습니다.

`src/main/java/org/owasp/webgoat/lessons/sqlinjection/advanced/SqlInjectionChallengeLogin.java`

```
    var statement = connection.prepareStatement("select password from sql_challenge_users where userid = ? and password = ?"); statement.setString(1, username_login); statement.setString(2, password_login);
```

이렇게 prepareStatement와 ? 를 활용해 안전하게 바인딩을 수행했기 때문에 해커의 장난질이 먹히지 않은 것입니다.

# 자바 백엔드에서의 방어 조치법 (1)

뚫렸던 회원가입 코드를 열고, 로그인 컨트롤러를 벤치마킹하여 리팩토링할 준비를 합니다.

`src/main/java/org/owasp/webgoat/lessons/sqlinjection/advanced/SqlInjectionChallenge.java`

```
    String checkUserQuery = "select userid from sql_challe
nge_users where userid = '" + username_reg + "'"; Statement
statement = connection.createStatement();
```

# 자바 백엔드에서의 방어 조치법 (2)

아래처럼 똑같이 ? 구조로 안전하게 패치를 적용합니다.

src/main/java/org/owasp/webgoat/lessons/sqlinjection/advanced/SqlInjectionChallenge.java

```
    String query = "select userid from sql_challenge_users where userid = ?"; var statement = connection.prepareStatement(query); statement.setString(1, username_reg); var resultSet = statement.executeQuery();
```

# 패치 검증 및 재빌드

터미널에서 컨테이너 내부로 들어가 Maven으로 서버를 다시 빌드합니다.

```
root@...:# ./mvnw spotless:apply && ./mvnw clean insta
ll -DskipTests
```

빌드 완료 후 `./mvnw spring-boot:run`으로 서버를 켭니다.

이제 `tom' and 1<2--` 구문이 그대로 일반 텍스트 문자로 DB에 박혀 들어가며, 공격이 완벽히 방어됨을 확인합니다.

# 04. Path Traversal

애플리케이션의 디렉터리 경로 조작을 통해
서버의 파일 시스템 권한을 무력화시키는 공격을 알아봅니다.

# Path Traversal (Directory Traversal) 이란?

상위 폴더로 이동하는 명령어인 `../` 문자열을 악용하여, 개발자가 허락한 공개 웹 폴더 범위를 강제로 벗어나 시스템 루트나 내부 중요 폴더를 헤집고 다니는 취약점입니다.

- **파일 다운로드 시:** 리눅스 서버의 `/etc/passwd` (계정 정보 파일) 같은 시스템 심장부 파일을 훔쳐냅니다.
- **파일 업로드 시:** 서버 시스템 구동에 필요한 필수 파일을 업로드 기능으로 덮어써서 서비스를 파괴하거나 악성 웹쉘(WebShell)을 심어 원격 제어권을 탈취합니다.

# 2번 실습: 업로드 경로 탈주 공격

목표: 일반적인 프로필 사진 업로드 폴더가 아닌, 더 상위 폴더인 `/root/.webgoat-2023.3/PathTraversal` 경로에 내 이미지를 강제 저장시키기.

UI에서 프로필 사진을 지정하고 정보를 채워 넣습니다. 이때 Full Name 항목의 데이터가 최종 저장 파일 이름에 관여한다는 힌트가 있습니다.

Full Name 칸에 `../test` 라고 넣고 Update 버튼을 누릅니다.

# 업로드 위치 조작 성공

제출 결과, 화면 하단 메시지에 내 파일이 저장된 절대 경로가 찍혀 나옵니다.

```
Profile has been updated, your image is available at: /root/.webgoat-2023.3/PathTraversal/webhacking/test
```

원래는 webhacking 하위의 더 깊은 폴더에 들어갔어야 할 파일이, 내가 심어둔 ../ 구문으로 인해 한 단계 상위인 webhacking 디렉터리 바로 아래 test라는 이름으로 무단 생성되며 해킹에 성공합니다.

# 3번 실습: ../ 필터링 우회

개발자가 화들짝 놀라 입력값에서 `../` 이 보이면 지워버리는 필터링 로직을 급하게 추가했습니다.

해커는 지워지는 성질을 역이용하여 `....//test` 라고 겹쳐서 입력합니다.

정가운데 위치한 `../` 만 필터링에 걸려 삭제되고 나면, 양옆에 떨어져 있던 파편들(`..` 와 `/`)이 서로 철썩 달라붙어 완벽한 `../` 로 부활합니다. 필터링이 허무하게 뚫렸습니다.

# 4번 실습: Burp Suite를 활용한 파일명 변조

이번엔 개발자가 폼 입력란(Full Name)을 못 믿겠다며, 내가 컴퓨터에서 올린 로컬 파일 이름 원본 자체를 서버 저장명으로 쓰도록 코드를 바꿨습니다.

윈도우나 맥에서는 애초에 파일 이름에 특수문자인 / 나 \를 넣고 저장할 수 없어 막힌 듯 보입니다.

하지만 파일을 올리고 Update를 누르는 순간 **Burp Suite로 패킷을 낚아챕니다 (Intercept).**

# 패킷 조작으로 한계 돌파

가로챈 멀티파트(multipart/form-data) 패킷 본문을 봅니다.

```
Content-Disposition: form-data; name="uploadedFileRemoveUserInput"; filename="tem
p.png"
```

클라이언트(브라우저) 영역을 이미 떠난 패킷 텍스트 상태이므로, OS 제약 따위는 무시하고 `filename="../temp.png"` 로 쿨하게 변조해 줍니다.

Forward로 보내면 손쉽게 방어 로직이 뚫리며 상위 폴더에 파일이 박힙니다.

# 5번 실습: 다운로드 경로 역추적

업로드가 아닌 다운로드/조회 기능에서의 취약점입니다.

화면의 고양이 사진을 보여주는 버튼을 눌러보니 /random-picture?id=10.jpg 형태로 사진을 불러옵니다.

여기서 id 값에 존재하지 않는 파일인 asdf를 넣어 서버 에러를 고의로 냅니다.

친절한 서버는 에러 메시지로 현재 파일이 위치한 /root/.webgoat-2023.3/PathTraversal/cats/ 의 전체 내부 디렉터리 구조를 줄줄 읊어버립니다.

# URL 인코딩을 통한 방어 우회

목표는 cats 폴더보다 더 위에 숨겨져 있는 `path-traversal-secret.jpg` 를 뽑아내는 것입니다.

`?id=../../path-traversal-secret` 를 넣었더니 "Illegal characters are not allowed" (특수문자 금지) 에러가 뜹니다.

Burp Decoder에서 `../` 를 **URL 인코딩(%2e%2e%2f)** 하여 보냅니다.

# 은닉 파일 추출 성공

필터링은 URL 인코딩된 문자를 인식하지 못하고 무사통과시킵니다. 이후 백엔드 서버 로직에서 디코딩되며 정상 명령어로 작동합니다.

결과적으로 금단의 구역에 있던 `path-traversal-secret.jpg` 파일이 화면에 딸려 내려오며 메시지를 출력합니다.

"You found it! submit the SHA-512 hash of your username as answer"

# 취약점 원인 분석: getQueryString()의 함정

왜 URL 인코딩 우회가 성공했을까요? 소스 코드를 봅니다.

`src/main/java/org/owasp/webgoat/lessons/pathtraversal/ProfileUploadRetrieval.java`

```
    var queryParams = request.getQueryString(); if (queryParams != null && (queryParam
s.contains("..") || queryParams.contains("/"))) { return badRequest...
```

getQueryString() 메서드는 파라미터를 꺼내는 것이 아니라 URL 주소 표시줄 전체 텍스트를 그대로 가져옵니다. **즉, URL 인코딩된 데이터를 디코딩해 주지 않습니다.**

# 안전한 데이터 검증 패치 준비

getQueryString() 대신 어떻게 짜야 할까요?

스프링과 서블릿에서 제공하는 파라미터 파싱 전용 메서드를 써야 합니다.

이 메서드는 스프링이 친절하게 URL 인코딩을 풀고 평문으로 변환한 찐 알맹이 값을 돌려주므로 필터 로직에 무조건 걸리게 됩니다.

# getParameter()를 통한 완벽 방어

아래와 같이 코드를 수정합니다.

`src/main/java/org/owasp/webgoat/lessons/pathtraversal/ProfileUploadRetrieval.java`

```
var id = request.getParameter("id");
```

또한 에러 났을 때 전체 디렉터리를 까발려주던 불필요한 로직을 지우고 단순히 "no data" 문자열만 반환하도록 코드를 리팩토링하여 정보 누출을 차단합니다.

# 7번 실습: Zip Slip (압축 파일 취약점)

목표: 압축 파일(.zip)을 올려서 압축이 풀리는 타이밍에 백엔드의 `webhacking.jpg` 원본 프로필 사진을 무단 덮어쓰기.

이 서비스는 사용자가 업로드한 압축 파일을 해제하고, **압축 파일 내부에 들어있던 각각의 파일 이름**을 그대로 살려서 서버 디렉터리에 복사해 놓는 로직을 가집니다.

만약 압축파일 안에 `../../해킹파일.jpg` 라는 경로명을 가진 파일이 들어있다면 대참사가 발생합니다.

# 리눅스 환경의 무적의 압축 파일 만들기

윈도우 알집으로는 저런 괴상한 이름의 파일을 압축할 수 없습니다.

Docker에 임시 리눅스(Ubuntu) 컨테이너를 띄우고, `mkdir` 로 목표 타겟과 똑같은 가짜 폴더 구조를 만듭니다.

그리고 아래 명령어로 **상대 경로 구문이 듬뿍 발려진 무적의 압축 파일**을 강제로 조립해 냅니다.

```
zip zipslip.zip ../../../../////../../root/.webgoat-2023.3/.../webhacking.jpg
```

# Zip Slip 공격 성공

Docker 명령어 docker  cp를 써서 이 오염된 압축파일을 내 윈도우 바탕화면으로 빼옵니다.

WebGoat에 접속해 이 압축파일을 업로드하고 Update를 누르면, 압축이 팡 풀리면서 안에 도사리고 있던 엄청난 길이의 ../../ 구문이 폭발합니다.

개발자가 설정한 안전 구역을 아득히 벗어나 시스템 최상위 루트로 뚫고 올라간 뒤, 정확히 타겟팅한 원본 프로필 사진을 덮어써 버리고 성공 메시지를 띄웁니다.

# 취약점 원인 분석: 백엔드 파일 복사 로직

어디가 문제였을까요? 코드를 살펴봅니다.

`src/main/java/org/owasp/Webgoat/lessons/pathtraversal/ProfileZipSlip.java`

```
    File f = new File(tmpZipDirectory.toFile(), e.getName()); InputStream is = zip.get
InputStream(e); Files.copy(is, f.toPath(), StandardCopyOption.REPLACE_EXISTING);
```

`e.getName()` 으로 압축파일 내부의 오염된 파일명을 그대로 빼오고, **이 파일이 지정된 폴더 안에서 얌전히 생성되는지 검증하는 로직이 1도 없습니다.** 그대로 `Files.copy` 로 덮어써 버립니다.

# getCanonicalPath()를 활용한 철벽 방어

getCanonicalPath() 메서드는 지저분한 ../ ./ 같은 상대 경로 요소를 전부 깔끔히 계산해버린 후 **최종 목적지의 진짜 절대 경로 주소**를 반환해 주는 훌륭한 함수입니다.

코드를 수정하여 파일 생성 직전에 아래 방어막을 한 겹 씌웁니다.

```
     if (f.getCanonicalPath().startsWith(tmpZipDirectory.toString())) { Files.copy(...)
} else { throw new IOException(); }
```

해석: "새로 생길 파일의 최종 절대 경로가, 내가 허락한 tmpZipDirectory 구역 내부에서 시작하는 게 확실히 맞을 때만 복사해라!"

# 05. Cross Site Scripting (XSS)

웹 애플리케이션의 클라이언트(브라우저) 측에서 악의적인 스크립트가 동작하도록 유도하여 사용자의 세션을 탈취하는 공격을 알아봅니다.

# Cross Site Scripting 이란?

개발자가 의도하지 않은 자바스크립트가 웹 서비스를 통해 **다른 사용자의 웹 브라우저 안에서 동작하도록** 만드는 취약점입니다.

**주요 특징:** 서버 시스템 자체가 털리는 SQLi와 달리, **공격 대상이 서버가 아니라 나와 동일한 다른 사용자(Client)**라는 점이 핵심입니다.

다른 사용자의 브라우저 쿠키에 담긴 세션 ID를 훔쳐서 해커 서버로 전송시키거나, 화면을 아예 조작된 피싱 사이트로 덮어씌우는 등 막대한 피해를 입힙니다.

# XSS의 3가지 주요 유형

## Reflected XSS

요청값에 섞여 들어간 스크립트가 응답 메시지에 그대로 반사(Reflected) 되어 실행되는 1회성 공격.

## Stored XSS

스크립트가 게시판 등 DB에 아예 저장(Stored) 되어, 누구나 글을 읽을 때마다 지속적으로 감염시키는 파급력이 큰 공격.

## DOM based XSS

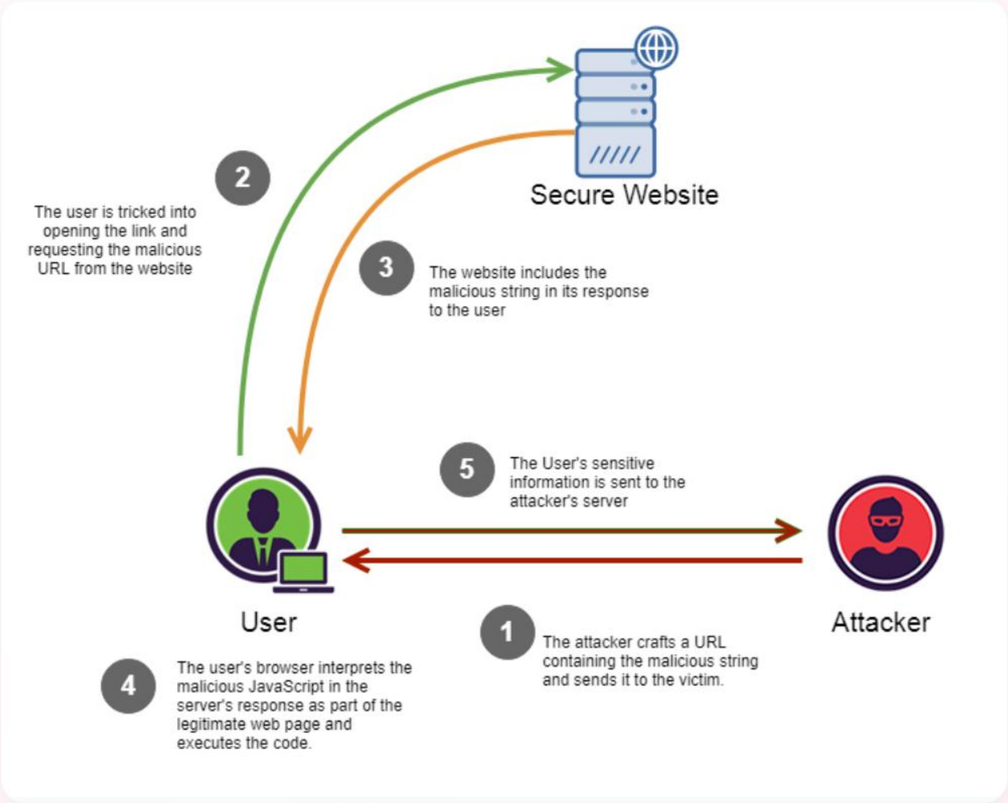
서버 통신 없이, 브라우저 내부 프론트엔드의 DOM 처리 로직 결함으로 인해 실행되는 공격.

# Reflected XSS 작동 흐름

주로 URL의 검색 파라미터나 폼 데이터에 스크립트를 삽입합니다.

공격자는 `https://test.com/search?q=` 악성코드 같은 URL을 이메일이나 메신저로 피해자에게 보내 클릭하도록 **유혹(사회공학적 기법)**합니다.

피해자가 클릭 시, 서버는 파라미터를 화면에 띄워주는 과정에서 악성코드를 포함해 반환하고 피해자의 브라우저는 이를 실행해버립니다.

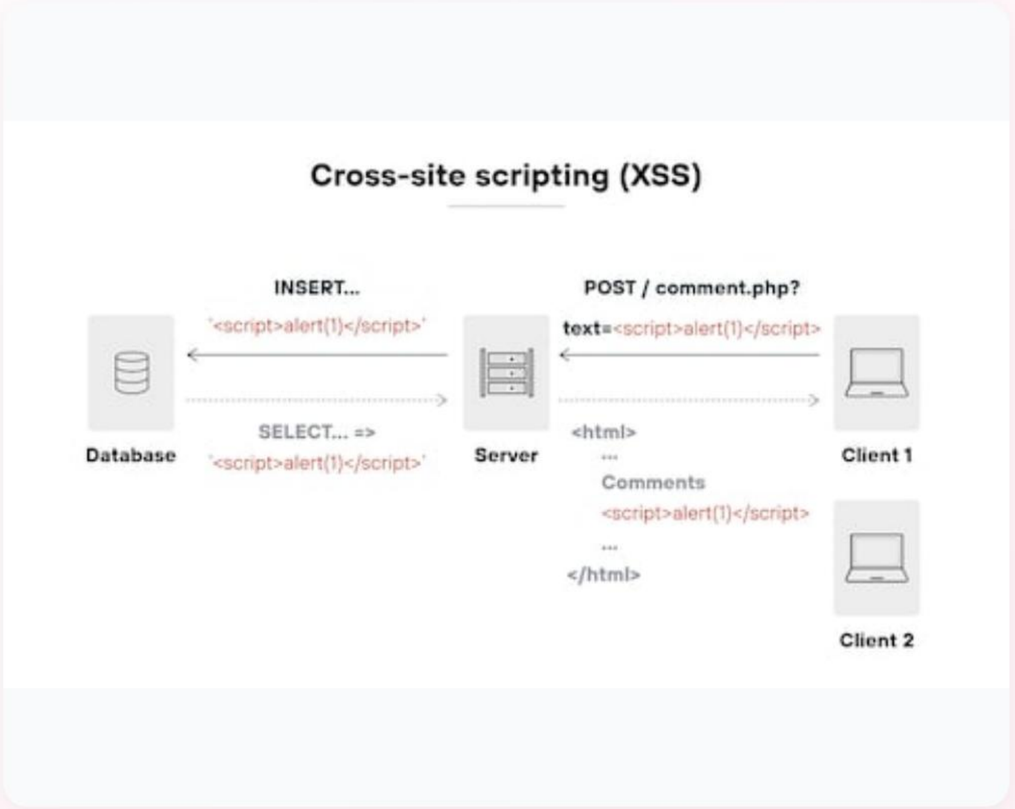


# Stored XSS 작동 흐름

공격자가 취약한 게시판이나 댓글창에 악성 스크립트를 작성하여 글을 등록합니다. (서버 DB에 영구 저장됨)

이후 불특정 다수의 일반 사용자들이 해당 게시글을 열람하기 위해 접근합니다.

서버는 DB에서 스크립트가 섞인 글 내용을 퍼서 전달하고, **글을 읽은 모든 사용자는 일제히 스크립트 감염 피해**를 입게 됩니다.



# DOM based XSS 작동 흐름

웹 페이지의 자바스크립트가 클라이언트 측 데이터를 동적으로 처리할 때 발생합니다.

서버를 거치지 않습니다. 공격자는 URL의 Fragment(#) 해시 뒤에 `javascript:alert(1)` 구문을 붙여서 보냅니다.

프론트엔드의 JS 코드가 이 Fragment 문자열을 가져와 `</div>` 등의 DOM 속성에 무지성으로 때려 박아버리면 브라우저 내에서 즉각 스크립트가 폭발합니다.

# 7번 실습: Reflected XSS 찌르기

쇼핑몰 장바구니 결제 페이지입니다. 신용카드 번호 란에 내 카드 번호 대신 고전적인 테스트 구문 `alert(1)`를 넣고 Purchase를 누릅니다.

화면에 **경고창 팝업 "1"** 이 번쩍 뜹니다.

개발자 도구를 켜서 HTML을 까보니 `We have charged credit card:alert(1)` 형태로, 내가 넣은 문자열이 HTML 태그 사이의 뼈대로 완벽하게 삽입되어 브라우저에서 실행된 것을 목격합니다.

# 10번 실습: DOM-Based XSS 추적 준비

프론트엔드 코드 어딘가에 숨겨진 낡은 '테스트 라우트' 코드를 찾아 그쪽으로 DOM XSS를 날리는 탐정 미션입니다.

힌트를 보면 이 사이트의 기본 라우팅 포맷은 `start.mvc#lesson/...` 형태입니다.

F12를 눌러 개발자 도구를 켜고 **Ctrl+Shift+F (전체 소스코드 검색)**로 `lesson/` 키워드를 쳐서 관련된 JS 라우터 파일의 흔적을 쫓아갑니다.

# GoatRouter.js 백도어 라우트 발견

검색된 GoatRouter.js 파일을 열어 소스를 스크롤 해보니, 운영 환경에는 절대 있어서는 안 될 낡은 잔재 코드가 발견됩니다.

`127.0.0.1:8080/WebGoat/js/goatApp/view/GoatRouter.js`

```
    routes: { 'welcome': 'welcomeRoute', 'lesson/:name': 'lessonRoute', 'test/:param': 'testRoute', <-- 은밀히 열려있는 백도어 라우트 }
```

이 코드는 주소창 # 뒤에 test/블라블라 가 들어오면, 블라블라 데이터를 가져다 testRoute 함수로 전달한다는 뜻입니다.

# testRoute의 종착지 (DOM 삽입부)

testRoute 함수가 어디로 이어지는지 꼬리에 꼬리를 물고 검색을 이어나가면 최종적으로 LessonContentView.js 파일의 showTestParam 함수에 당도합니다.

`127.0.0.1:8080/WebGoat/js/goatApp/view/LessonContentView.js`

```
    showTestParam: function (param) { this.$el.find('.lesson-content').html("test:" + param); }
```

경악스러운 코드가 나옵니다. 제이쿼리의 .html() 함수를 써서, 넘어온 데이터를 아무런 살균(Sanitization) 없이 화면의 DOM 트리에 바로 꽂아 넣고 있습니다!

# 11번 실습: DOM-Based XSS 정밀 타격

원리를 알았으니 즉각 타격합니다.

브라우저 주소창에 `http://.../start.mvc#test/webgoat.customjs.phoneHome()` 를 칩니다.

화면엔 아무 변화가 없지만, 개발자 도구의 **Console** 탭을 띄워보면 내부 스크립트가 폭발적으로 동작하여 "phoneHome Response is -1585819090" 이라는 시스템 은닉 난수를 토해냅니다. 이 난수를 입력하면 미션 클리어.

# 가장 확실한 XSS 방어책: HTML Entity 인코딩

해커가 게시판에  라는 글자를 남겼다고 칩시다.

이 글자를 다른 사람 화면에 그려줄 때, < 를 < 로, > 를 > 로 강제로 **HTML Entity 변환(치환)**하여 뿌려주면 끝납니다.

브라우저는 이를  라는 그림(문자열)으로만 정직하게 화면에 그릴 뿐, 절대로 명령어 코드(태그)로 인식하고 실행하지 않습니다.

# 자바 백엔드에서의 방어 조치법 (1)

7번 실습의 취약한 소스코드를 열어 분석합니다.

`src/main/java/org/owasp/Webgoat/lessons/xss/CrossSiteScriptingLesson5a.java`

```
        cart.append("We have charged credit card:" + field1 +
"");
```

입력받은 field1 텍스트를 아무런 검열 없이 cart HTML 덩어리에 그대로 퍼붓고 있는 것이 원인입니다.

# 자바 백엔드에서의 방어 조치법 (2)

이 코드를 스프링 프레임워크에서 제공하는 HtmlUtils 도구를 이용해 **안전하게 살균 처리**합니다.

src/main/java/org/owasp/Webgoat/lessons/xss/CrossSiteScriptingLesson5a.java

```
    cart.append("... credit card:" + HtmlUtils.htmlEscape
(field1) + "");
```

이제 서버에서 넘어올 때부터 특수문자가 < 등으로 자동 치환되어 안전해집니다.

# 방어 성공 확인

메이븐 재빌드 후 다시 신용카드 결제 란에 `alert(1)`를 넣고 결제 버튼을 누릅니다.

이제는 짜증나는 팝업창이 뜨지 않고, 화면에 문구 그대로 얌전히 표시됩니다.

개발자 도구 [Edit as HTML]로 뜯어보면 서버에서 넘어올 때부터 이미 `<script>` 로 완벽히 무력화되어 넘어왔음을 확인할 수 있습니다.

# 수고하셨습니다.

Injection의 정수부터 Path Traversal, XSS까지
웹 보안의 핵심 공격 벡터와 완벽한 코드 방어법을 모두 체득하셨습니다.

**긴 시간 대단히 수고하셨습니다!**

# Image Sources

- 

https://miro.medium.com/1*o_asKsD_JqunhqggHoxodw.png

Source: medium.com

---

- 

https://www.paloaltonetworks.com/content/dam/pan/en_US/images/cyberpedia/xss-cross-site-scripting/cross-site-scripting-xss.jpg?imwidth=480

Source: www.paloaltonetworks.com