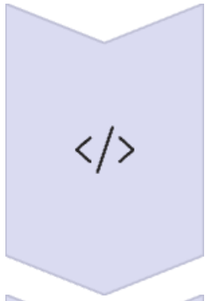
웹 애플리케이션 보안

1. XSS (Cross- site Scripting)

XSS(Cross-Site Scripting)는 웹 보안 취약점 중 하나로, 공격자가 사용자의 웹 브라우저에서 실행될 수 있는 스크립트를 웹 페이지에 삽입하는 공격이다.

스크립트는 주로 JavaScript로 작성되며, XSS 공격을 통해 공격자는 사용자의 세션 토큰, 쿠키, 개인 정보 등을 탈취할 수 있다. 또한, XSS 공격은 사용자를 가장하여 웹 애플리케이션에서 악의적인 행동을 할 수 있게 한다.

XSS(크로스 사이트 스크립팅)의 이해

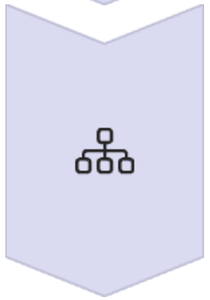


저장형 XSS

공격자가 악성 스크립트를 서버에 저장하여 다른 사용자가 페이지를 방문할 때 실행되게 하는 공격입니다. 주로 게시판, 댓글 등에서 발생합니다.

반사형 XSS

악성 스크립트가 포함된 URL을 사용자가 클릭하면 스크립트가 실행되는 공격입니다. 서버가 입력을 그대로 반환할 때 발생합니다.



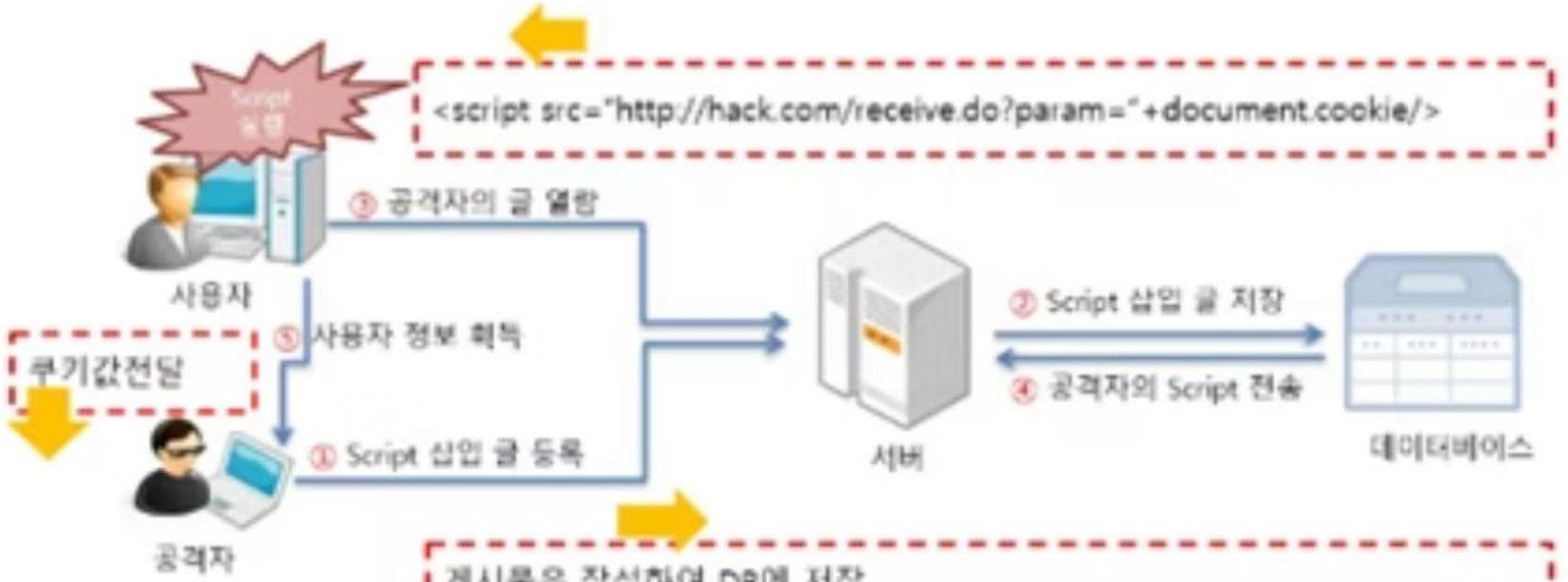
DOM 기반 XSS

클라이언트 측 스크립트가 DOM을 동적으로 수정할 때 발생하는 공격으로, 서버 응답은 변경되지 않지만 브라우저에서 악성 코드가 실행됩니다.

XSS 공격은 사용자의 세션 토큰, 쿠키, 개인 정보 등을 탈취할 수 있으며, 사용자를 가장하여 웹 애플리케이션에서 악의적인 행동을 할 수 있게 합니다. 이러한 공격은 주로 JavaScript를 이용하여 이루어집니다.

입력데이터 검증 부재 - Stored XSS

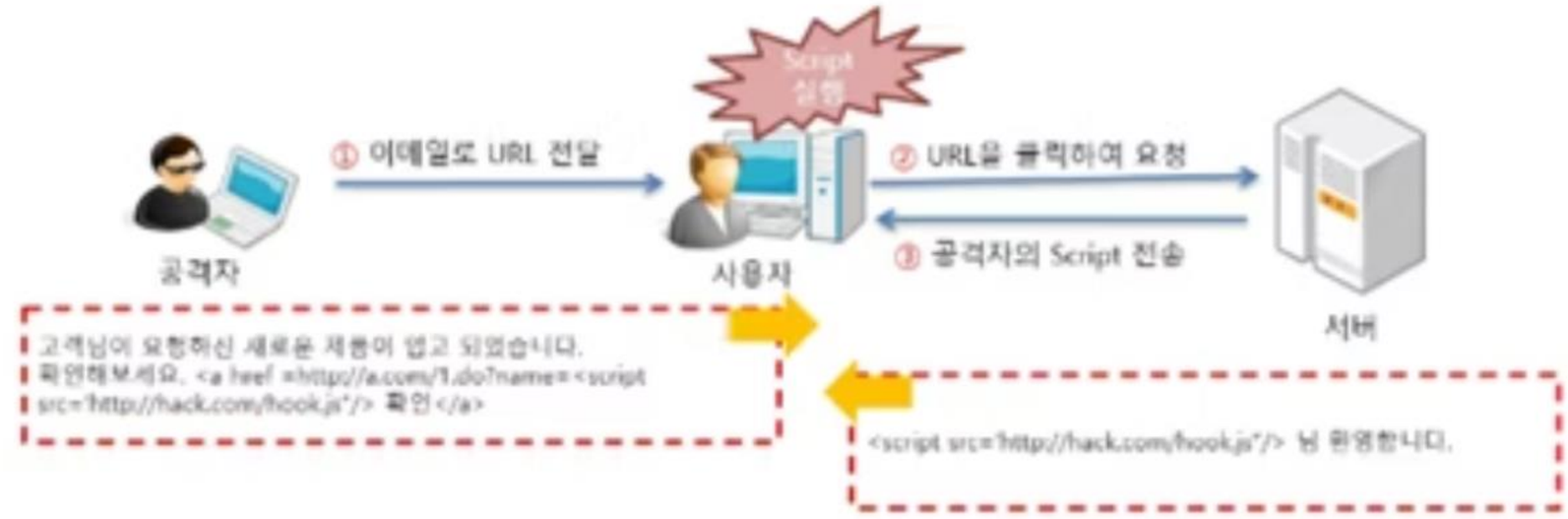
공격자가 서버에 저장한 스크립트를 사용자가 열람하는 경우 스크립트가 다운로드 되어 사용자의 브라우저에서 실행



입력데이터 검증 부재 – Reflective XSS

XSS(크로스사이트스크립트)는 웹애플리케이션에서 사용자 입력값에 대한 필터링이 제대로 이루어지지 않을 경우, 공격자가 입력 가능한 폼에 악의적인 스크립트를 삽입하여 해당 스크립트가 희생자 브라우저에서 동작하도록 하는 보안 약점이다.

사용자가 요청한 값을 서버에서 검증하지 않고 응답으로 사용하는 경우 발생



입력데이터 검증 부재 – DOM XSS

자바스크립트로 DOM 객체 정보를 사용하여 Document에 write를 수행하는 경우 스크립트가 사용자의 브라우저에서 실행

③ 자바스크립트로 URL의 일부인 msg 값을 document.write()로 화면에 출력하는 경우 브라우저에서 스크립트가 실행

XSS 방어 전략: 출력 이스케이프와 코드 예제

Java에서의 이스케이프

Java에서는 OWASP Java Encoder나 Spring의 HtmlUtils.htmlEscape() 메소드를 사용하여 사용자 입력을 안전하게 처리할 수 있습니다.

JavaScript에서의 이스케이프

DOMPurify 같은 라이브러리를 사용하거나, textContent 속성을 사용하여 HTML이 해석되지 않도록 합니다. innerHTML 대신 안전한 대안을 사용하세요.

프레임워크 활용

React, Angular, Vue 같은 현대적인 프레임워크는 기본적으로 XSS 방어 기능을 제공합니다. 이러한 기능을 적극 활용하고 올바르게 구성하세요.

XSS 방어의 핵심은 출력 시 사용자 입력을 항상 이스케이프 처리하는 것입니다. 입력 검증만으로는 모든 XSS 공격을 방어할 수 없으므로, 출력 시 이스케이프와 함께 여러 방어층을 구축하는 것이 중요합니다. 특히 템플릿 엔진이나 프레임워크의 보안 기능을 활용하는 것이 효과적입니다.

사용자 입력을 HTML에 출력할 때는 **문자 이스케이프 처리**.

문자	변환
`<`	`<`
`>`	`>`
`"`	`&quot;`
`'`	`&#x27;`
`/`	`&#x2F;`

```
const handleConfirm = () => {
  let username = escape(document.getElementById("username").value);
  console.log("username:", username);
};
```

콘텐츠 보안 정책(CSP)을 통한 XSS 방어

```
http

Content-Security-Policy: default-src 'self'

자기자신(self)와 지정된 신뢰할 수 있는 도메인(trusted.cdn.com)에서만 스크립트를 로드하고 실행하도록 제한
```

인라인 스크립트 제한

'unsafe-inline'을 금지하여 HTML 내에 직접 작성된 JavaScript 코드와 HTML 속성에 직접 작성된 코드의 실행을 방지합니다.

외부 리소스 제어

승인된 출처에서만 스크립트, 이미지, 폰트 등을 로드하도록 설정하여 악성 외부 리소스의 실행을 차단합니다.

XHR 및 Fetch API 요청 제한

connect-src 지시문을 통해 JavaScript의 fetch(), XMLHttpRequest 등으로 요청할 수 있는 대상 서버를 제한합니다.

HTTPS 강제 적용

HTTPS 사이트에서 HTTP 리소스를 불러오는 것을 차단하여 보안을 강화합니다. 이는 CSP와 브라우저 기본 정책 모두에 의해 차단됩니다.

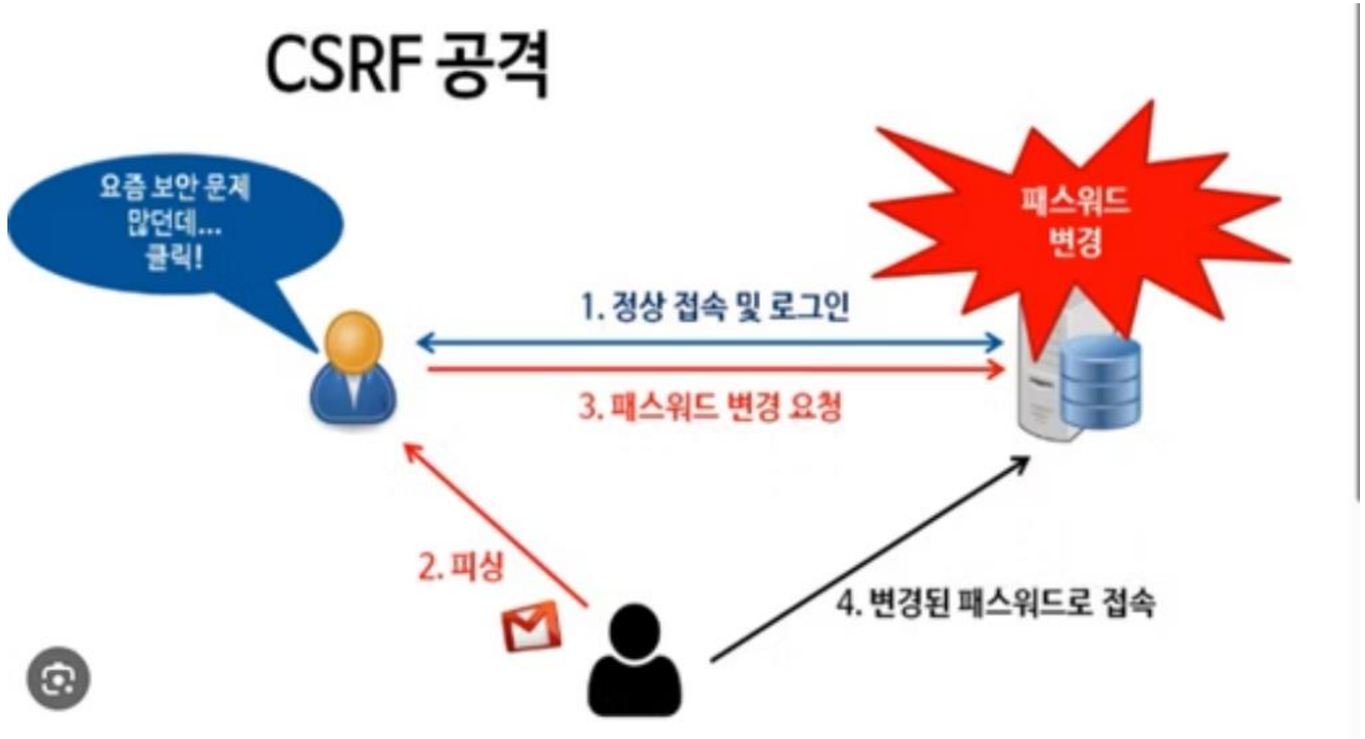
콘텐츠 보안 정책(CSP)은 XSS 공격을 효과적으로 방어할 수 있는 강력한 도구입니다. HTTP 응답 헤더에 Content-Security-Policy를 추가하여 웹 페이지에서 로드하고 실행할 수 있는 리소스를 세밀하게 제어할 수 있습니다. CSP는 이스케이프 처리와 함께 사용할 때 더욱 효과적인 방어층을 형성합니다.

방어 전략 요약

방어 방법	설명
HTML 이스케이프	출력 시 `<`, `>` 등 특수문자 변환
JavaScript 이스케이프	JS 안에서 출력 시 따로 escape
CSP 헤더 적용	외부 스크립트 차단 등
보안 라이브러리 사용	OWASP Java Encoder, StringEscapeUtils 등
프레임워크의 기본 보안 기능 활용	Thymeleaf, JSP, React 등은 자동 이스케이프

CSRF(크로스 사이트 요청 위조)

CSRF는 Cross Site Request Forgery의 약자로 웹 애플리케이션 취약점 중 하나로, 인터넷 사용자가 자신의 의지와는 무관하게 공격자가 의도한 행위(수정, 삭제, 등록 등)를 특정 웹 사이트에 요청하게 만드는 공격이다



출처: https://rjswn0315.tistory.com/173

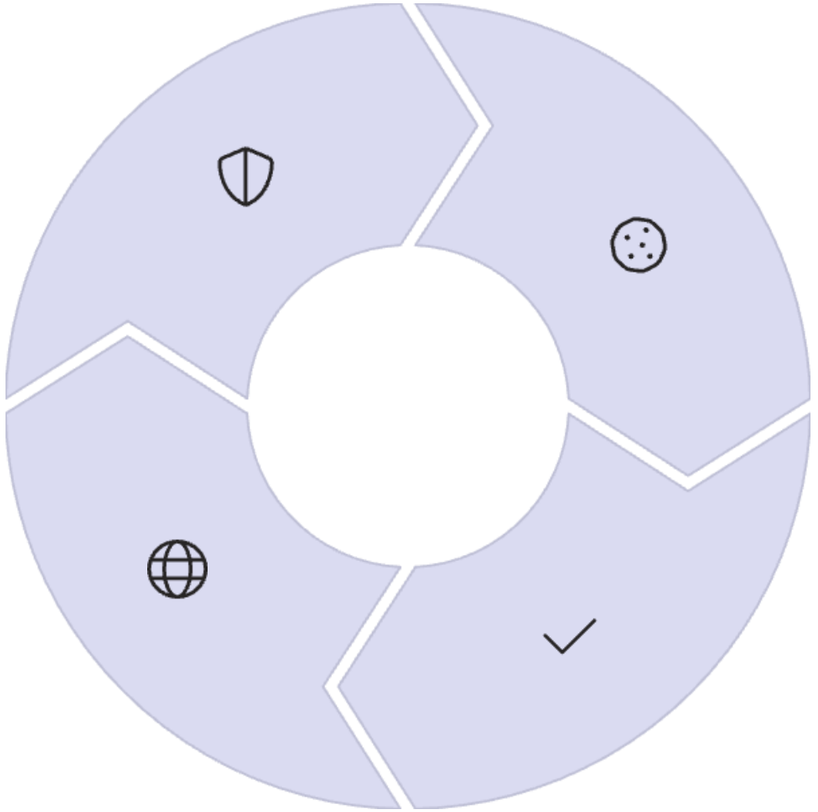
CSRF(크로스 사이트 요청 위조) 공격과 방어

CSRF 토큰 사용

사용자 세션과 연관된 고유한 토큰을 생성하여 모든 중요 요청에 포함시킵니다.

CORS 정책 설정

적절한 CORS(Cross-Origin Resource Sharing) 정책을 구성하여 허용된 도메인에서만 리소스 접근을 허용합니다.



SameSite 쿠키 속성

쿠키에 SameSite=Strict 또는 Lax 속성을 설정하여 크로스 사이트 요청에서 쿠키 전송을 제한합니다.

Referer 검증

HTTP 요청의 Referer 헤더를 검사하여 요청이 유효한 출처에서 왔는지 확인합니다.

CSRF 공격은 사용자가 인증된 상태에서 공격자가 의도한 요청을 사용자 모르게 전송하도록 하는 공격입니다. 이 공격은 사용자의 신원을 도용하여 권한 있는 작업을 수행할 수 있습니다. 효과적인 방어를 위해서는 위의 방법들을 조합하여 사용하는 것이 좋습니다. 특히 CSRF 토큰과 SameSite 쿠키 속성은 현대 웹 애플리케이션에서 필수적인 방어 수단입니다.

XXE(XML 외부 엔티티) 공격 이해와 방어

XXE 공격 이해

XML 파서가 외부 엔티티를 처리할 때 발생하는 취약점입니다.

취약점 식별

XML 입력을 처리하는 애플리케이션이 DTD를 활성화한 경우 취약할 수 있습니다.

방어 전략 구현

외부 엔티티 처리를 비활성화하고 XML 파서 설정을 안전하게 구성합니다.

XXE 공격은 XML 외부 엔티티 참조를 처리하는 취약한 XML 파서를 이용하여 서버의 파일을 읽거나, 내부 네트워크를 스캔하거나, 서비스 거부 공격을 수행할 수 있습니다. Spring 프레임워크에서는 XML 파서 설정에서 외부 엔티티 처리를 명시적으로 비활성화해야 합니다. JAXB, DOM, SAX, StAX 등의 XML 파서를 사용할 때는 항상 외부 엔티티 참조를 비활성화하는 것이 중요합니다.

```java
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;
import org.w3c.dom.Document;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import java.io.StringReader;
import jakarta.xml.bind.annotation.XmlRootElement;
import org.xml.sax.InputSource;

@RestController
public class XmlController {

    @PostMapping("/upload-xml")
    public String uploadXml(@RequestBody String xml) throws Exception {
        DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
        DocumentBuilder db = dbf.newDocumentBuilder(); //  보안 설정 없음
        InputSource is = new InputSource(new StringReader(xml));
        Document doc = db.parse(is);

        String content = doc.getElementsByTagName("data").item(0).getTextContent();
        return "Received: " + content;
    }
}
```

취약한 spring boot 코드

```xml
<?xml version="1.0"?>
<!DOCTYPE foo [
  <!ENTITY xxe SYSTEM "file:///etc/passwd">
]>
<root>
  <data>&xxe;</data>
</root>
```

공격자가 보내는 악성 xml 요청 예시

서버는 Entity의 xxx를 해석하기 위해 부분을 읽어서 <data> 태그 내부 출력하게 된다. 서버의 내부 파일 passwd파일이 공개 된다.

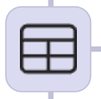
```java
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();

// XXE 방어 설정
dbf.setFeature("http://apache.org/xml/features/disallow-doctype-decl", true);
dbf.setFeature("http://xml.org/sax/features/external-general-entities", false);
dbf.setFeature("http://xml.org/sax/features/external-parameter-entities", false);
dbf.setFeature("http://apache.org/xml/features/nonvalidating/load-external-dtd", false);
dbf.setXIncludeAware(false);
dbf.setExpandEntityReferences(false);

DocumentBuilder db = dbf.newDocumentBuilder();
```

방어하기

SQL 인젝션 공격과 방어 기법

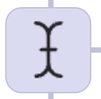
인증 우회

로그인 폼에 ' OR '1'='1' 같은 입력을 삽입하여 비밀번호 검증 없이 로그인을 우회합니다.



데이터 유출

UNION 공격을 통해 데이터베이스의 테이블 이름, 열 정보 등 민감한 정보를 획득합니다.



데이터 조작

INSERT, UPDATE, DELETE 쿼리에 악성 코드를 주입하여 데이터를 변경하거나 삭제합니다.

관리자 권한 획득

데이터베이스 관리자 권한으로 쿼리를 실행하여 시스템 전체를 제어합니다.

SQL 인젝션은 여전히 가장 위험한 웹 애플리케이션 취약점 중 하나입니다. 공격자는 입력 필드를 통해 악의적인 SQL 코드를 삽입하여 데이터베이스를 조작할 수 있습니다. 공격자는 오류 메시지, 시간 지연, 조건부 응답 등을 통해 공격 결과를 확인합니다. 이러한 공격은 애플리케이션의 데이터 무결성과 기밀성에 심각한 위협이 됩니다.

예시:

사용자가 입력한 사용자명과 비밀번호로 로그인하는 SQL 쿼리:

```sql
SELECT * FROM users WHERE username = 'user' AND password = 'pass';
```

사용자가 `user' OR '1'='1` 과 같이 입력하면 쿼리는 다음과 같이 변경된다.

```sql
SELECT * FROM users WHERE username = 'user' OR '1'='1' AND password = 'pass';
```

이로 인해 조건이 항상 참이 되어 인증이 우회된다.

```sql
' UNION SELECT table_name, null FROM information_schema.tables WHERE table_schema = '데이터베이스명'
```

```sql
SELECT * FROM users
WHERE username = ''
UNION SELECT table_name, null
FROM information_schema.tables
WHERE table_schema = 'my_database' --' AND password = '입력값';
```

공격자가 주입 결과를 알아내는 방법들 (주요 시나리오)

· 에러 기반 공격 (Error-based Injection)

주입한 결과로 일부러 SQL 에러를 발생시켜서 에러 메시지 안에 테이블 이름이나 컬럼명이 노출되게 한다. (예: table 'my_database.users' doesn't exist 같은 에러)

· Blind SQL Injection (블라인드 인젝션)

직접 결과를 못 보더라도, 조건을 바꿔서 서버 반응 차이(HTTP 500 vs 200)를 관찰한다.

예: 테이블이 존재하면 응답 속도가 느려진다든지, 특정 문구가 포함된다든지.

· Time-based Blind Injection

SQL 주입문에 IF 조건 THEN SLEEP(n) 같은 코드를 추가해서, 맞는 조건일 때 서버 응답을 고의로 지연시키고,그 차이로 존재 여부를 추측합니다.

SQL 인젝션 방어 전략 및 요약

1

Prepared Statements

쿼리와 데이터를 분리하여 SQL 인젝션 공격을 원천 차단하는 가장 효과적인 방법입니다.

2

ORM 사용

객체 관계 매핑 도구를 사용하여 SQL 쿼리 생성 시 SQL 인젝션 위험을 크게 줄일 수 있습니다.

3

저장 프로시저

데이터베이스 내에서 매개변수를 받아 처리하는 저장 프로시저를 통해 안전하게 쿼리를 수행합니다.

4

최소 권한 원칙

데이터베이스 사용자에게 필요한 최소한의 권한만 부여하여 공격의 영향을 제한합니다.

웹 애플리케이션 보안은 지속적인 과정입니다. XSS, CSRF, XXE, SQL 인젝션과 같은 주요 보안 취약점에 대한 이해와 방어 전략을 적용하여 애플리케이션을 안전하게 보호해야 합니다. 이 프레젠테이션에서 다룬 방어 기법들을 실제 개발 과정에 적용하고, 보안 테스트와 코드 리뷰를 통해 지속적으로 보안 취약점을 식별하고 수정하는 것이 중요합니다. 항상 최신 보안 패치를 적용하고, 보안 모범 사례를 따르는 것을 잊지 마세요.