Special Report

웹 취약점과 해킹 매커니즘 #6 SQL Injection 보안대책

■ 개요

SQL Injection 은 설계된 SQL 구문에서 사용자 입력값 검증이 미흡하여, 악의적인 명령을 실행하는 임의의 쿼리를 삽입해 공격이 가능한 취약점이다. 지금까지 공격 유형에 따른 세 가지 종류의 SQL Injection 을 살펴보았다. SQL Injection 취약점이 존재할 경우 인증 우회, 데이터베이스 접근과 같은 공격이 가능하기 때문에 데이터 유출, 변조, 삭제 등의 피해를 미칠 수 있다.

따라서 이번 Special Report 에서는 SQL Injection 의 보안대책을 다룬다. 적절한 보안대책을 통해 입력값을 검증하고 SQL Injection 공격을 예방해야 한다.

■ SQL Injection 보안 대책

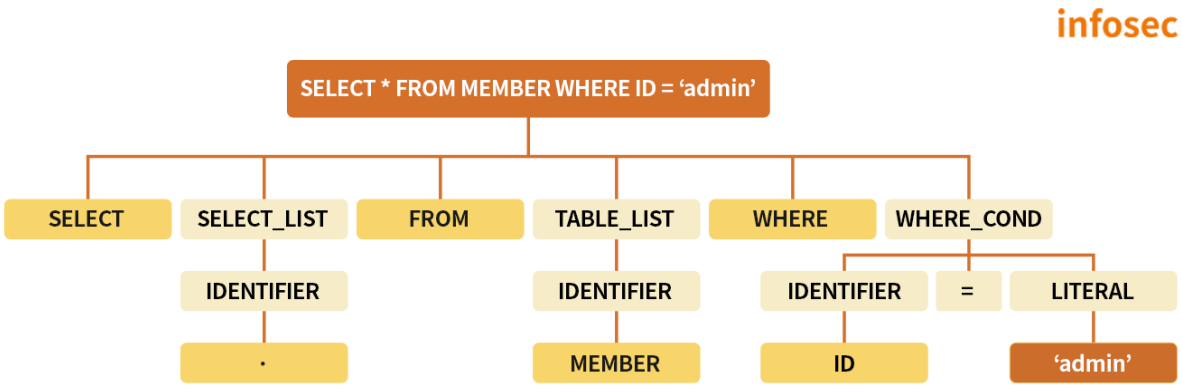
SQL Injection 보안 대책의 최선은 Prepared Statement를 사용하여 소스코드를 구성하는 것이다. Prepared Statement를 사용할 수 없는 환경이라면, 차선책으로 사용자 입력값 필터링 또는 정제를 통해 사용자 입력값이 SQL구문에 영향을 미치지 못하도록 해야 한다. 보안 대책에 대한 상세 설명은 다음과 같다.

1) Prepared Statement 사용

Prepared Statement는 SQL Injection을 방어하는 최선의 보안 대책이며, SQL구문이 미리 컴파일 되어 있어 입력값을 변수로 선언해 두고 필요에 따라 값을 대입하여 처리하는 방식이다.

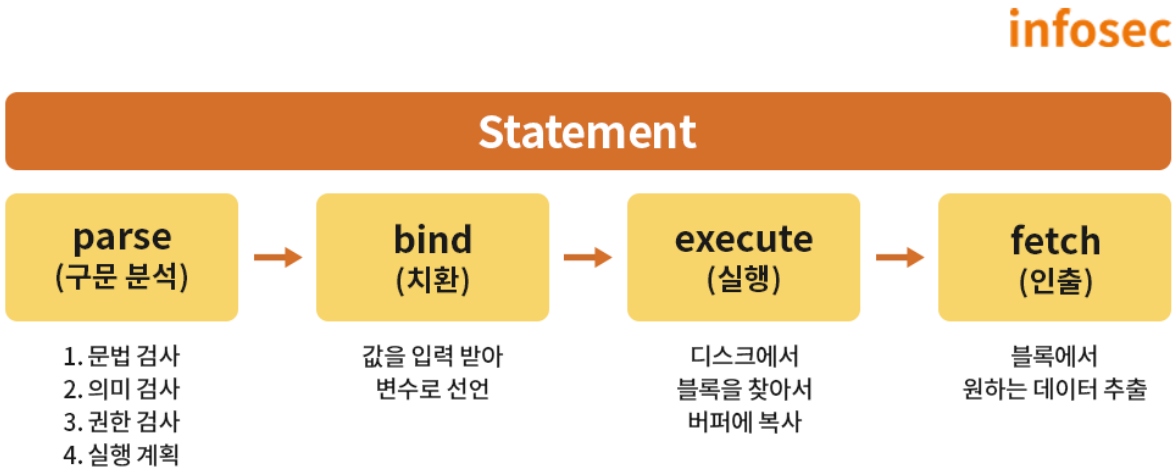
- Statement vs Prepared Statement

SELECT문은 DBMS 내부적으로 4단계의 과정(Parse, Bind, Execute, Fetch)을 거쳐 결과를 출력한다. 특히 구문 분석을 하는 parse 과정을 거치면 다음과 같은 파싱 트리가 생성된다.



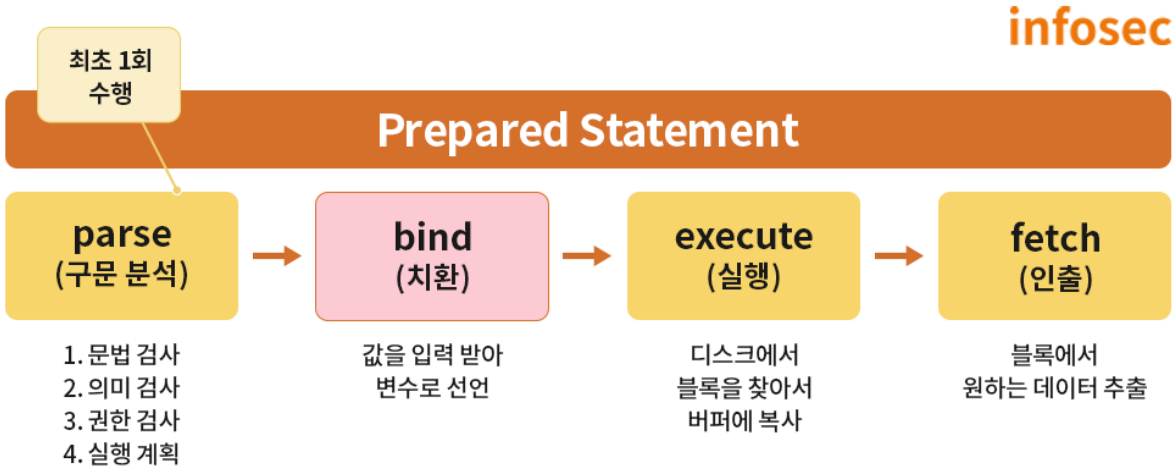
[파싱 트리]

일반적인 Statement의 경우, 구문 분석(parse)부터 인출(fetch)까지 모든 과정을 매번 수행한다. 따라서 입력값에 SQL구문에 영향을 미치는 특수문자나 예약어가 들어갈 경우 구문 분석 과정에서 SQL구문의 일부로 작용하여 SQL Injection 공격이 가능하다.



[Statement]

Prepared Statement의 경우, 구문 분석(parse) 과정을 최초 1회만 수행하여 생성된 결과를 메모리에 저장해 필요할 때마다 사용한다. 미리 구성된 파싱 트리를 반복적으로 사용하기 때문에 Statement에 비해 시간을 단축할 수 있다. 또한 SQL구문이 미리 컴파일 되어 사용자 입력값을 변수로 선언해 값을 대입하여 사용한다. 따라서 외부 입력값으로 SQL문법에 영향을 미치는 특수문자나 예약어가 입력되어도 문법적인 의미로 작용하지 못한다.



[Prepared Statement]

SQL Injection 공격에 취약한 Statement와 안전한 Prepared Statement로 구성된 소스코드는 다음과 같다.

• 일반적인 Statement 코드 (취약)

```
Connection conn = null;
Statement stmt = null;
ResultSet rs = null;
...
String sql = "SELECT * FROM MEMBER WHERE ID = '"+param_id+"' AND PW = '"+param_ passwd+"'";
stmt = conn.createStatement();
rs = stmt.executeQuery(sql);
```

사용자 입력값인 아이디와 패스워드 파라미터인 param_id, param_passwd에 SQL Injection 공격이 가능하다.

• Prepared Statement 코드 (양호)

```
String param_id=request.getParameter("id");
String param_passwd=request.getParameter("passwd");


Connection conn = null;
PreparedStatement pstmt = null;
ResultSet rs = null;
...
String sql = "SELECT * FROM MEMBER WHERE ID = ? AND PW = ?";
pstmt.setString(1, param_id);
pstmt.setString(2, param_passwd);
rs = pstmt.executeQuery(sql);
```

SQL구문이 미리 컴파일 되어 있고 사용자 입력값을 받는 부분을 '?'로 바인딩 처리를 하여 setString 메소드를 통해 외부 입력값을 받기 때문에 SQL Injection 공격이 불가능하다.

Prepared Statement를 통해 SQL Injection에 대응하는 것이 근본적인 해결책이지만, 문법적/비즈니스 로직 상 사용 불가한 로직이 존재하기도 한다. 예를 들어, Prepared Statement는 데이터를 파라미터로 전달하는 역할을 하기 때문에 데이터 정렬의 기능을 하는 ORDER BY절에서는 쓰이지 않는다.

또한, 실제 운영 중인 서버의 경우엔 소스코드 수정이 어려울 수 있다. 따라서 이러한 점을 고려하여 Prepared Statement 사용이 불가피한 경우에는 문자열 필터링 또는 정제를 통해 사용자 입력값을 검증하는 방법을 사용해야 한다.

2) 입력값 필터링 및 정제

서비스 운영 등의 이유로 Prepared Statement 사용이 불가피할 경우 차선책을 고려해야 한다.

WhiteList Filter는 허용할 문자열을 제외한 모든 문자를 필터링하는 방법이다. 이때 개별 문자에 대해 지정하는 것보단 정규식 등을 이용해 패턴화해두는 것이 유용하다. 입력값 필터링 시에는 DBMS마다 대소문자 구분이 상이하다는 점을 고려하여 대소문자를 모두 필터링하는 것을 권장한다. 또한 입력값에 길이 제한을 두어 공격 구문 삽입이 어렵도록 해야 한다.

입력값 정제는 사용자 입력값이 SQL구문에서 문법적인 의미를 갖지 못하도록 입력값을 다른 값으로 치환하는 방법이다. 치환된 값을 통해 데이터를 추출하기 때문에 사용자가 입력한 공격 구문 실행을 방지할 수 있다.

대표적인 예시는 HashMap을 사용하는 것이다. 사용자 입력값이 해시테이블의 값과의 매핑을 통해 정제되어 전달되기 때문에 SQL구문에 영향을 미치지 못한다.

로직상 필터링 및 정제할 문자열을 미리 지정할 수 없는 경우에는 BlackList Filter 방식을 통해 필터링 해야 한다. BlackList Filter 방식은 공격에 사용될 가능성이 있는 예약어 및 특수문자를 모두 필터링 하는 방법이다. 필터링 시 DBMS별로 쓰이는 예약어 및 특수문자가 다르기 때문에 주의가 필요하다.

아래의 표는 Oracle 데이터베이스 사용 시 필터링 해야 할 문자열이다.

infosec

구분	필터링 문자열									
공통 (특수문자)	‘	(	)	--	/*	*/	%	+	-	/
공통 (예약어)	AND	OR	SELECT	FROM	WHERE	UPDATE	CASE	WHEN	THEN	
	SET	INSERT	INTO	DELETE	DROP	JOIN	ELSE	END	IF	
데이터 검색	ALL_TABLES			TABLE_NAME			ALL_TAB_COLUMNS			
	USER_TABLES			COLUMN_NAME						
UNION SQL Injection	UNION			ORDER BY			NULL			
Error Based SQL Injection	UTL_INADDER.GET_HOST_NAME			UTL_INADDER.GET_HOST_ADDRESS			GROUP BY			
	ORDSYS.ORD_DICOM GETMAPPINGXPATH			CTXSYS.DRITHSX.SN			UTL_INADDER.GET_HOST_NAME			
Blind SQL Injection	SUBSTR		ASCII		>		<		=	

[Black List Filter 문자열 예시(Oracle)]

DBMS 별로 공격에 사용되는 문자열이 다르기 때문에 상세 내용은 다음의 링크를 참고하면 된다.

- 참고: https://pentestmonkey.net/category/cheat-sheet/sql-injection

3) 에러 메시지 출력 제한

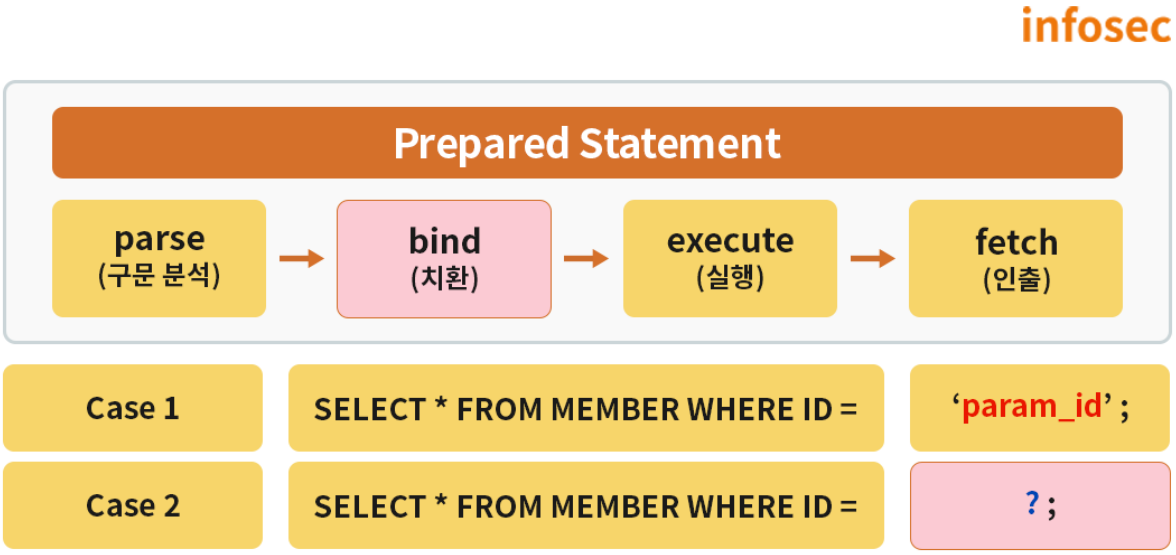
공격자가 악용할 가능성이 있는 에러 메시지가 노출될 경우 Error Based SQL Injection과 같은 공격이 가능하다. 따라서 Default 에러 메시지가 아닌 사전에 정의한 에러 페이지를 반환하도록 대체해야 한다. 또한, 디버깅용 에러 메시지 창은 실제 소스코드에서 제거하여 시스템 내부 정보가 노출되지 않도록 해야 한다.

■ Prepared Statement 사용 시 주의할 점

Prepared Statement를 사용한다고 해서 무조건 SQL Injection에 안전하지는 않다. 반드시 바인딩 처리를 통해 외부 입력값이 문법적 의미를 갖지 않도록 구성하는 것이 중요하다.

SELECT문이 실행되는 4단계 과정 중 bind(치환) 단계에서 입력값을 처리하는 방법에 따라 SQL Injection에 취약할 수 있다.

Case 1은 입력값을 파라미터 그대로 전달받아 문법적인 요소로 작용하기 때문에 SQL Injection 공격이 가능하다. Case 2는 입력값을 바인딩 처리하여 문법적인 요소로 작용하지 않기 때문에 공격이 불가능하다.



[Prepared Statement 사용 시 입력값 처리 예시]

Case 1) 취약한 Prepared Statement

```
String param_id=request.getParameter("id");
String param_passwd=request.getParameter("passwd");

Connection conn = null;
PreparedStatement pstmt = null;
ResultSet rs = null;
…
String sql = "SELECT * FROM MEMBER WHERE ID = '"+param_id+"' AND PW = '"+param_ passwd +"'";
rs = pstmt.executeQuery();
```

Prepared Statement로 선언했지만, 외부 입력값을 바인딩 처리하지 않아 param_id, param_passwd 파라미터에 SQL Injection 공격이 가능하다.

Case 2) 안전한 Prepared Statement

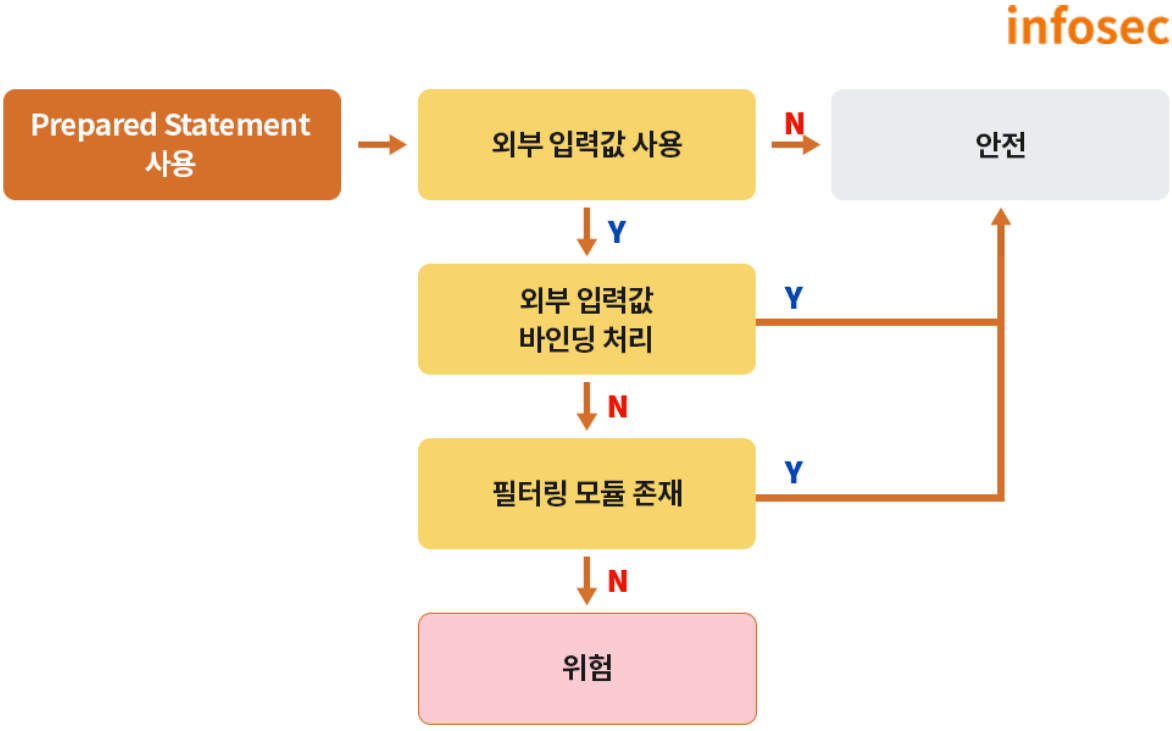
```
String param_id=request.getParameter("id");
String param_passwd=request.getParameter("passwd");

Connection conn = null;
PreparedStatement pstmt = null;
ResultSet rs = null;
…
String sql = "SELECT * FROM MEMBER WHERE ID = ? AND PW = ?";
pstmt.setString(1, param_id);
pstmt.setString(2, param_passwd);
rs = pstmt.executeQuery(sql);
```

Prepared Statement를 선언했고, 외부 입력값을 바인딩 처리하여 setString과 같은 메소드를 통해 입력받고 있다. 따라서 미리 컴파일된 구문에 외부 입력값이 변수로 들어가 SQL구문에 영향을 미치지 않기 때문에 SQL Injection 공격이 불가능하다.

* 바인딩 된 데이터는 SQL문법이 아닌 컴파일 언어로 처리되기 때문에 문법적인 의미를 가질 수 없다.

이처럼 Prepared Statement를 사용할 때 바인딩 처리를 하지 않으면 SQL Injection에 취약한 것을 알 수 있다. 따라서 반드시 바인딩 처리를 통해 외부 입력값이 구문 분석(parse) 단계에서 처리되지 않도록(문법적인 의미를 갖지 않도록) 구성하는 것이 중요하다.

Prepared Statement 사용 시, SQL Injection에 안전한 로직을 구성하는 방법은 다음과 같다.



[Prepared Statement 사용 시 로직 구성]

■ 맺음말

지금까지 SQL Injection의 유형별 공격 방법과 보안대책에 대해 알아보았다. SQL Injection은 사용자 입력값 검증이 미흡할 경우 발생하는 취약점이며, 데이터베이스에 직접 접근이 가능하여 중요 정보 유출, 데이터 변조 및 삭제 등의 피해를 일으킨다.

SQL Injection의 가장 근본적인 보안 대책은 입력값에 대한 바인딩 처리와 함께 **Prepared Statement**를 사용하여 외부 입력값이 문법적 의미를 갖지 않도록 소스코드를 구성하는 것이다. 문법적/비즈니스 로직 상 소스코드 수정이 불가피할 경우 **특수문자 및 예약어 필터링**을 적용하는 등 보안대책에 따라 대응해야 한다.

이어지는 『웹 취약점과 해킹 매커니즘#7』에서는 사용자 입력값에 대한 검증이 미흡하여 악성 스크립트를 삽입할 수 있는 공격인 XSS(Cross Site Script)에 대한 내용을 다룬다.