- Web 모의해킹 -

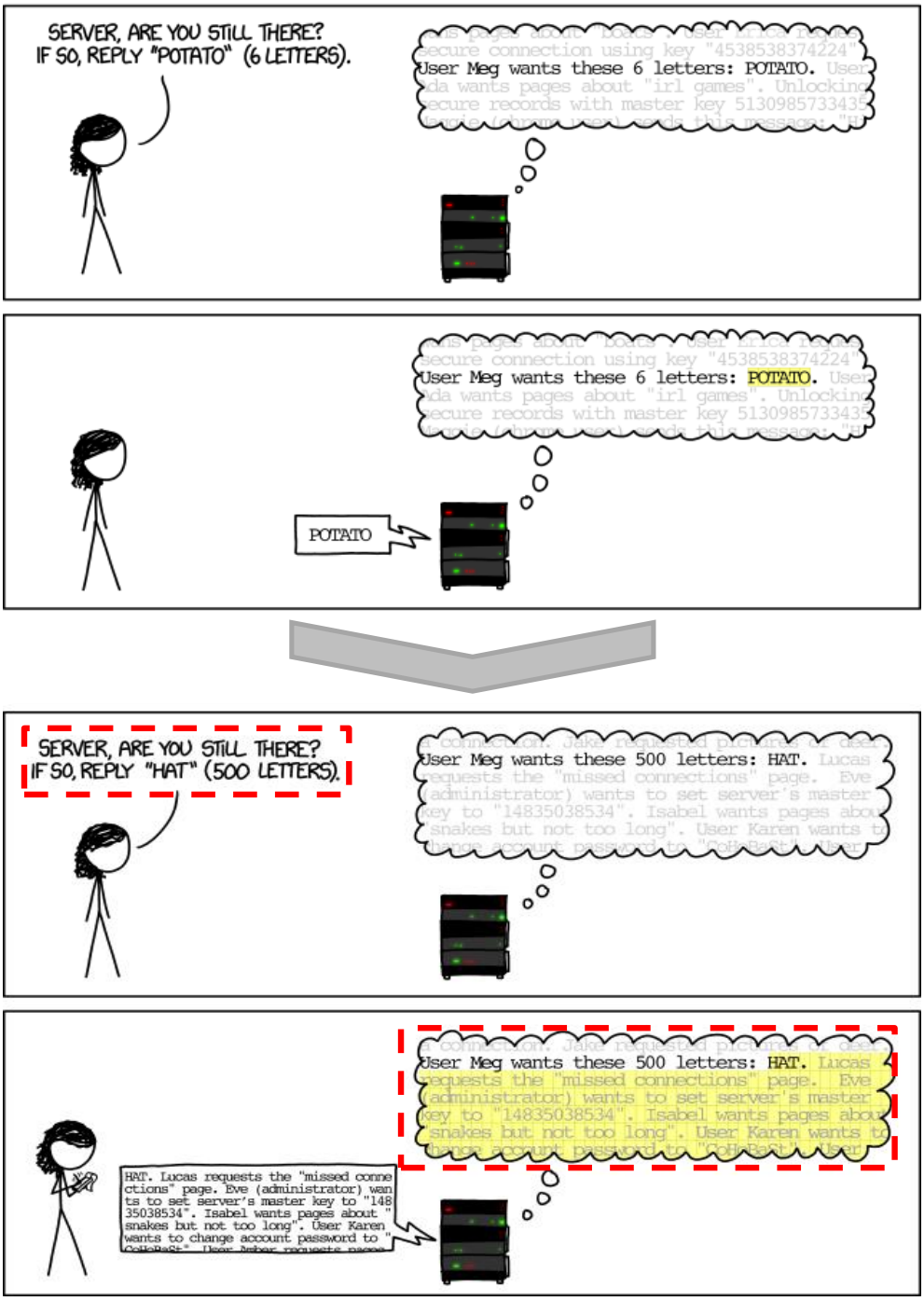
취약한 시큐어 코딩 사례

SK 인포섹, 이호석

leehs2@sk.com

1. 개요

SSL 통신에서 사용하는 OpenSSL 이라는 오픈소스 SW 가 있습니다. 이 OpeenSSL 에서 2014 년 4 월 하트블리드라는 취약점이 발견되었는데, 이 취약점을 통해 클라이언트와 서버 통신간 패킷크기를 변조하여 서버 메모리에 저장된 정보를 탈취하는 것이 가능하였습니다.



전세계 표준이라 대부분의 웹사이트에서 공격이 일어났는데 그 피해가 심각하였습니다. (캐나다 국세청 홈페이지, 야후, 베틀넷, LOL 홈페이지등에서 개인정보 탈취)

이 취약점은 Robin 이라는 엔지니어가 OpenSSL 에 Heartbeat 를 업데이트 하면서 발생하였는데, 패킷 가변 길이 체크를 간과하고 코드를 제출한 것이 원인이라고 밝혔습니다. 이런 실수는 코드 리뷰 단계에서 걸러지는게 일반적이지만 이번 버그는 리뷰어도 발견하지 못하고 통과가 되었습니다.

현재 개발하고 있는 웹 시스템도 동일합니다. 내가 작성한 소스코드가 시큐어코딩이 되었는지 단위 테스트 때 충분히 검증하지 않으면, 나중에 누군가 발견해서 다시 수정하기란 쉽지 않습니다. 비용/공수 측면에서도 개발 단계에서 고치는 것보다 프로젝트 완성단계에서 고치는 것이 30 배 더 발생한다는 IBM 에 연구보고서도

있습니다. 따라서 내가 작성한 소스코드가 보안에 취약하지 않은지, 내가 적용한 시큐어코딩이 안전한지 확인하는 '관심'이 필요합니다.

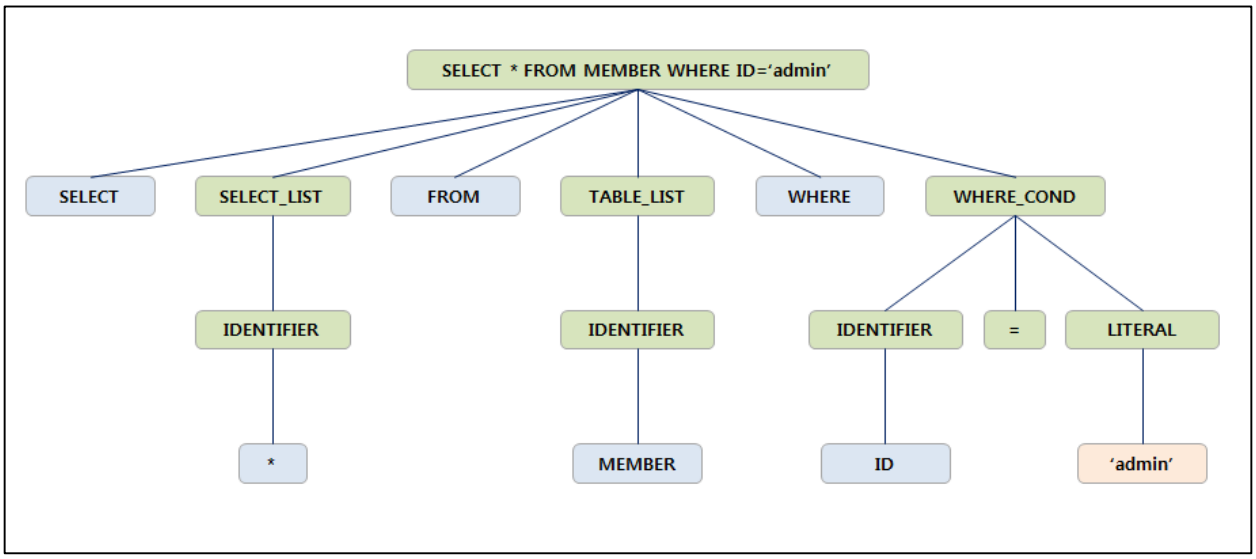
"SQL Injection 을 막으려면 Prepared Statement 를 사용해야 한다." 이건 이 글을 읽고 계신 분이라면 누구나 알고 있는 내용이지만, "Prepared Statement 객체를 쓰더라도 취약할 수 있다." 보안대책을 적용하지 않으면 똑같이 취약한 것이 아니라 "소스코드 구성에 따라 공격/피해 규모가 달라진다." 라는 사실을 아는 개발자는 많지 않습니다. 이번 챕터에서는 이런 관점에서 발생하는 취약점을 기술하고 원리를 설명하여, 시큐어 코딩을 '잘'하는 방법을 알아보겠습니다.

2. 취약한 Prepared Statement

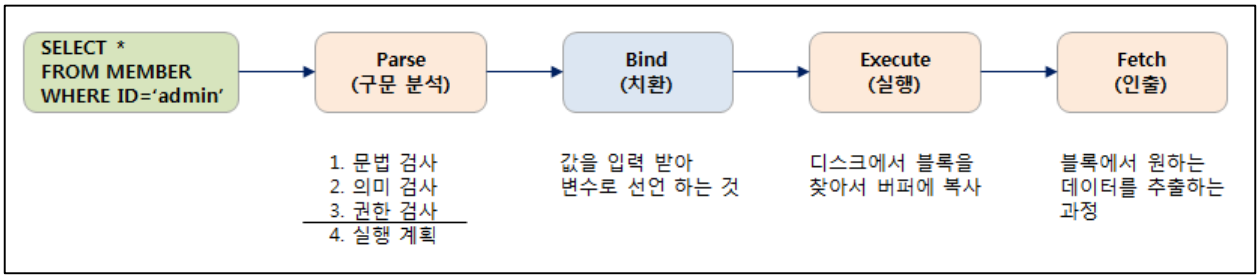
SQL Injection 정오탐 분류 작업을 하다 보면 의외로 흔히 취약하게 나오는 유형이 바인딩 처리의 부재입니다. Prepared Statement 는 바인딩 처리된 부분을 제외하고 나머지 쿼리 문법을 미리 실행해둬서 속도, 보안성을 높이는 방법인데, 원리를 정확하게 이해하기 위해서는 SELECT 쿼리가 어떻게 실행되는지 알아야 합니다.
차근차근 하나씩 알아보겠습니다.

2.1. SELECT문 실행 과정

우리가 웹 상에서 입력한 쿼리는 DBMS 내부적으로 4 가지 과정(Parse, Bind, Execute, Fetch)을 거쳐 결과를 출력합니다. 특히 쿼리의 문법을 검사하기 위해서는 Parse 과정을 거치게 되는데, 입력한 쿼리의 결과를 다음과 같이 파싱하여 트리를 생성합니다.



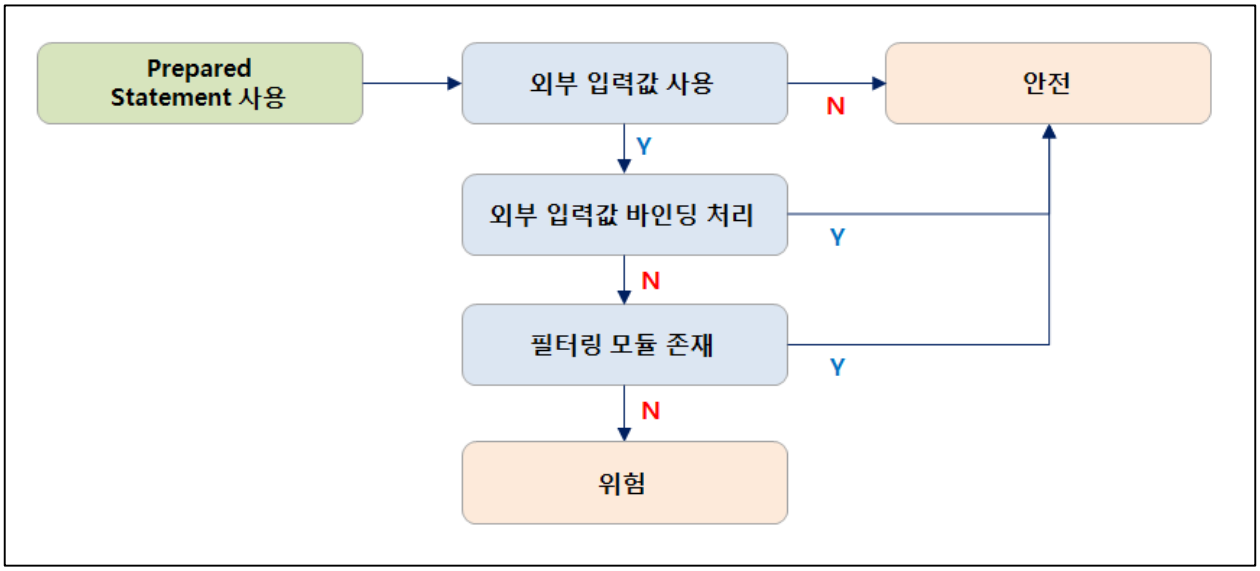
일반적인 Statement 를 사용하여 SELECT 쿼리를 입력했을 때에는 아래 그림과 같이 매번 Parse 부터 Fetch 까지 모든 과정을 수행합니다. 하지만, Prepared Statement 를 사용하는 경우에는 효율을 높이기 위해 Parse 과정을 최초 1 번만 수행하고 이후에는 생략할 수 있는데, Parse 과정을 모두 거친 후에 생성된 결과는 메모리에 저장해두고 필요할 때마다 사용하게 됩니다. 반복적으로 미리 구성된 파싱 트리를 사용하기 위해서는 자주 변경되는 부분을 변수로 선언해 두고, 매번 다른 값을 대입(바인딩)하여 사용합니다.



예를 들어, 전체 사원이 1000 명인 회사의 평균 연봉와 인사고과 점수를 조회하려고 하면, 같은 테이블의 동일한 칼럼을 조회하지만 사번과 이름만 다른 실행계획이 1000 개 생길 것입니다. 사번과 이름을 바인드 변수로 지정하면 실행계획은 최초로 1 번만 생성하고 1000 번 반복하면 실행계획의 생성 시간을 절약할 수 있습니다. 이렇게 실행계획을 줄이기 위해 다른 변수나 값을 넣는 작업을 바인드라고 합니다.

이때, 바인딩 데이터는 SQL 문법이 아닌 컴파일 언어로 처리가 됩니다. 그렇기 때문에, 문법적인 의미를 가질 수 없고, 바인딩 변수에 SQL 공격 쿼리를 입력할지라도 의미있는 쿼리로 동작하지 않습니다.

Prepared Statement 를 사용할 때 해킹으로부터 안전한지 확인하는 로직은 다음과 같습니다.



2.2. 바인딩, Prepared Statement

이와 같이 Prepared Statement 를 쓰더라도 바인딩처리를 하지 않으면 SQL Injection 공격이 가능한 것을 알아봤는데, 예제 소스코드를 CASE 별로 확인하면서 다시 정리해보겠습니다.

<취약 - 일반 Statement> .

```
1  String param_id=request.getParameter("id");
2  String param_passwd=request.getParameter("passwd");
3
4  Connection conn = null;
5  Statement stmt = null;
6  ResultSet rs = null;
7
8  try{
9     Context init = new InitialContext();
10    DataSource ds = (DataSource)init.lookup("java:comp/env/jdbc/shanks123");
11    conn = ds.getConnection();
12
13    String sql = "SELECT * FROM MEMBER WHERE ID = '"+param_id+"' AND PW ='"+param_passwd+"'";
14    stmt = conn.createStatement();
15    rs = stmt.executeQuery(sql);
16
17    if(rs.next()){
18            session.setAttribute("s_id", rs.getString("ID"));
19            out.println("<script>location.href='../main.jsp'</script>");
20    ...
```

결과해설 : 일반 Statement 로 선언하여서 외부입력값 (param_id, param_passwd)에 SQL Injection 공격이 가능합니다

<취약 - Prepared Statement, No binding>

```
1  String param_id=request.getParameter("id");
2  String param_passwd=request.getParameter("passwd");
3
4  Connection conn = null;
5  PreparedStatement pstmt = null;
6  ResultSet rs = null;
7
8  try{
9     Context init = new InitialContext();
10    DataSource ds = (DataSource)init.lookup("java:comp/env/jdbc/shanks123");
11    conn = ds.getConnection();
12
13    pstmt = conn.prepareStatement("SELECT * FROM LHSMEMBER3 WHERE ID = '"+param_id+"' AND PW
      ='"+param_passwd+"'");
14    rs = pstmt.executeQuery();
```

```
15    ...
```

결과 해설 : Prepared Statement 로 선언하였지만, 외부 입력값을 "?"로 바인딩처리 하지 않아서 param_id, param_passwd 파라미터에 SQL Injection 공격이 가능합니다.

<양호 - Prepared Statement>

```
1   String param_id=request.getParameter("id");
2   String param_passwd=request.getParameter("passwd");
3
4   Connection conn = null;
5   PreparedStatement pstmt = null;
6   ResultSet rs = null;
7
8   try{
9      Context init = new InitialContext();
10     DataSource ds = (DataSource)init.lookup("java:comp/env/jdbc/shanks123");
11     conn = ds.getConnection();
12
13     pstmt = conn.prepareStatement("SELECT * FROM LHSMEMBER3 WHERE ID = ? AND PW = ?");
14     pstmt.setString(1,param_id);
15     pstmt.setString(1,param_passwd);
16     rs = pstmt.executeQuery();
17     ...
```

결과 해설 : Prepared Statement 로 선언하였고, 외부 입력값을 "?"로 바인딩처리 하였으므로 SQL Injection 공격이 불가능합니다.

결론 : 이와 같이 Prepared Statement 를 사용하는 것도 중요하지만, Binding 처리를 하여 외부 입력값이 파싱단계로 처리되지 않도록(문법적인 의미를 가질 수 없도록) 구성하는 게 중요합니다.

3. 식별과 인증 - 로그인 페이지

이번 세션에서는 같은 비즈니스 로직이라도 코드 구성에 따라 어떤 공격이 가능한지, 얼마나 더 취약해지는지 알아보겠습니다.

로그인 페이지에서 ID, PW 를 이용하여 인가된 사용자인지 확인하게 되는데, ID 로 누구인지 확인하는 작업은 **식별**에 해당하고, PW 가 맞는지 확인하는 작업은 **인증**에 해당합니다. 이 식별과 인증을 이용해 로그인 페이지를 구성하는 것은 크게 두가지 형태로 나눌 수 있는데 이는 다음과 같습니다.

- 식별과 인증을 동시(SQL)에 하는 형태
- 식별을 먼저(SQL)하고 인증은 나중(JAVA)에 하는 형태

3.1. 식별과 인증을 동시에 하는 경우

아래 소스코드를 보면 파라미터로 받은 ID와 PW를 SQL문 안에서 동시에 처리해 주고 있습니다.

```
1  String param_id=request.getParameter("id");
2  String param_passwd=request.getParameter("passwd");
3
4  try{
5     Context init = new InitialContext();
6     DataSource ds = (DataSource)init.lookup("java:comp/env/jdbc/shanks123");
7     conn = ds.getConnection();
8
9     String sql = "SELECT * FROM MEMBER WHERE ID = '"+param_id+"' AND PW ='"+param_passwd+"'";
10    stmt = conn.createStatement();
11    rs = stmt.executeQuery(sql);
12
13    if(rs.next()){
14          session.setAttribute("s_id", rs.getString("ID"));
15          out.println("<script>location.href='../main.jsp'</script>");
16    ...
```

식별과 인증을 동시에 처리하면 아래 표와 같이 공격이 가능합니다. (빨간색이 입력값)

입력 ID	입력 PW	공격 구문	해석
' or '1'='1'--		WHERE ID=''or '1'='1'--	**<Member 테이블 내 첫번째 계정 권한 탈취>** WHERE 절 뒤에 1=1 이 무조건 참이라서 WHERE 이 없는 것과 같이 MEMBER 테이블 전체가 조회가 됨. rs.next()가 가장 먼저 등록된 상위에 있는 데이터를 읽는데 보통 TEST 나 ADMIN 계정으로 높은 권한을 탈취할 수 있음
admin'--		WHERE ID='admin'--	**<원하는 계정 권한 탈취>** 원하는 사용자의 비밀번호를 입력하지 않고 로그인 함
asdf' AND '1'='1		WHERE ID='asdf' AND '1'='1' WHERE ID='asdf' AND '1'='2'	**<Blind SQL Injection>** 존재하는 ID 를 참, 거짓여부에 따라 로그인이 되는지 확인함. 이후 원하는 DB 데이터를 획득

asdf	1234' AND '1'='1	WHERE ID='asdf' AND PW='1234' AND '1'='1' WHERE ID='asdf' AND PW='1234' AND '1'='2'	**<Blind SQL Injection>** 존재하는 ID, PW 를 참, 거짓여부에 따라 로그인이 되는지 확인함. 이후 원하는 DB 데이터를 획득

3.2. 식별과 인증을 별도로 하는 경우

다음은 식별과 인증을 별도로 처리하는 소스코드 입니다.

```
1  String param_id=request.getParameter("id");
2  String param_passwd=request.getParameter("passwd");
3
4  try{
5     Context init = new InitialContext();
6     DataSource ds = (DataSource)init.lookup("java:comp/env/jdbc/shanks123");
7     conn = ds.getConnection();
8
9     String sql = "SELECT * FROM MEMBER WHERE ID = '"+param_id+"'";
10    stmt = conn.createStatement();
11    rs = stmt.executeQuery(sql);
12
13    if(rs.next()){
14    if(param_passwd.equals(rs.getString("PW")) {
15          session.setAttribute("s_id",rs.getString("ID"));
16          out.println("<script>location.href='../main.jsp'</script>");
17 ...
```

PW 가 맞는지 검증하는 부분을 JSP 문법(14 번 라인)에서 처리하기 때문에 SQL 문에서 PW 에 대한 우회가 불가능합니다. 따라서 식별과 인증을 동시에 하는 경우와 같이 타 ID 로 로그인이 가능하다거나 비밀번호 로직을 우회하는 등에 공격이 불가능합니다.

식별과 인증을 별도로 처리하는 경우 아래와 같은 공격만 가능합니다. (빨간색이 입력값)

공격 대상	공격 구문	해석
param_id,	WHERE ID='asdf' and '1'='1' WHERE ID='asdf' and '1'='2'	**<Blind SQL Injection>** 존재하는 ID 를 참, 거짓여부에 따라 로그인이 되는지 확인함. 이후 원하는 DB 데이터를 획득 (식별과 인증을 동시에 하는 경우와 다르게 PW 는 asdf 계정에 맞는 값을 입력해야 함)

보신바와 같이 식별과 인증을 동시에 SQL 문에서 처리하는 경우 공격할 수 있는 포인트나 위협이 더 많습니다. 거꾸로 말하면 Prepared Statement, 입력값 필터링을 못하는 상황이더라도 위협을 줄일 수 있는 방법이 있다는 것입니다.

가장 간단한 로그인 소스코드이기 때문에 PW Hash 처리, 임계값 설정, 고객사 요구사항 반영등이 추가되면 상황이 조금 달라지겠지만 어떻게 구성해야 논리적으로 안전한지 생각할 수 있는 설명은 되었을 거라 생각합니다. 예시를 로그인 페이지로 들었지만 소스코드에 SQL 쿼리가 존재하는 곳은 모두 동일합니다.

결론 : SQL 문법을 부정 사용하는 것과 JAVA 문법을 우회하는 것은 별개이기 때문에, SQL 문에서는 최소한의 식별만, JAVA 문법에서 보안과 관련된 코딩을 한다면 위협이 줄어든다는게 keypoint 입니다.
개발 일정이 타이트해서 설계에 신경 쓸 시간이 없고, 여건상 코딩을 마음대로 할 수 없을 순 있지만, 코딩하기 전에 10 초만 생각해보고 안전하게 구성 하는게 습관화 된다면 유지보수 하거나 소스코드/모의해킹 점검시 소스코드 수정을 위해 투자해야하는 시간이 몇배는 줄어들 거라고 생각합니다.