Android-JAVA
시큐어 코딩 가이드

2011. 9.

행정안전부

CONTENTS

<표 1> Android-JAVA언어 프로그램 분류별 취약점목록

분류	취약점 명칭	위험도	CWE-ID
입력 데이터 검증 및 표현	상대 디렉터리 경로 조작	매우높음	CWE-23
	절대 디렉터리 경로 조작	매우높음	CWE-36
API 악용	null 매개변수 미검사	높음	CWE-398
	equals()와 hashCode() 하나만 정의	높음	CWE-581
보안특성	기밀 정보의 단순한 텍스트 전송	높음	CWE-319
	취약한 암호화 알고리즘의 사용	높음	CWE-327
	적절하지 않은 난수값의 사용	높음	CWE-330
	전역적으로 접근가능한 파일	높음	-
	외부에서 접근하여 활성화 가능한 컴포넌트	높음	-
	공유 아이디에 의한 접근제어 통과	높음	-
시간 및 상태	경쟁 조건: 검사시점과 사용시점	높음	CWE-367
	제대로 제어되지 않은 재귀	높음	CWE-674
에러 처리	오류 메시지 통한 정보 노출	높음	CWE-209
	오류 상황에 대한 처리 부재	높음	CWE-390
코드 품질	널포인터 역참조	높음	CWE-476
캡슐화	공용 메소드로부터 리턴된 private 배열-유형 필드	높음	CWE-495
	private 배열-유형 필드에 공용 데이터 할당	높음	CWE-496
	시스템 데이터 정보 누출	높음	CWE-497

제1장 Android-JAVA 프로그램 보안취약점

제1절 입력 데이터 검증 및 표현

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

1. 상대 디렉터리 경로 조작(Relative Path Traversal)

가. 정의

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

나. 안전한 코딩기법

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

다. 예제

■ 안전하지 않은 코드의 예 - Android-JAVA

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

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

■ 안전한 코드의 예 - Android-JAVA

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

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

라. 참고 문헌

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

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

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

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

- 파일 이름으로부터 replaceAll 메소드를 사용하여 위험한 문자들을 제거하거나, 절대경로 문자열 포함여부를 검사함으로써 임의의 디렉터리에 접근하지 못하도록 프로그램을 작성하는 것이 바람직하다.

다. 예제

■ 안전하지 않은 코드의 예 - Android-JAVA

```
1:  ……
2:  public void onCreate(Bundle savedInstanceState) {
3:      super.onCreate(savedInstanceState);
4:      File file = new File(android.os.Environment.getExternalStorageDirectory(), "inputFile");
5:      try {
6:          InputStream is = new FileInputStream(file);
7:          Properties props = new Properties();
8:          props.load(is);
9:          String name = props.getProperty("filename");
10:         file = new File("/usr/local/tmp/" + name);
11:         file.delete();
12:         is.close();
13:     } catch (IOException e) {
14:         Log.w("Error", "", e);
15:     }
16: }
```

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

■ 안전한 코드의 예 - Android-JAVA

```
1:  ……
2:  public void onCreate(Bundle savedInstanceState) {
3:      super.onCreate(savedInstanceState);
4:
5:      File file = new File(android.os.Environment.getExternalStorageDirectory(), "inputFile");
6:      try {
7:          InputStream is = new FileInputStream(file);
8:          Properties props = new Properties();
9:          props.load(is);
10:          String name = props.getProperty("filename");
11:          if (name.indexOf("/") <0) {
12:              file = new File(name);
13:              file.delete();
14:          }
15:          is.close();
16:      } catch (IOException e) {
17:              Log.w("Error", "", e);
18:      }
19:  }
```

외부의 입력이 파일이름으로 사용될 경우 절대경로명이 사용되지 못하도록, 문자열이 "\" 또는 "/"을 포함하거나 해당 문자열로 시작할 경우 관련 동작 수행을 거부하는 것이 바람직하다.

라. 참고 문헌

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

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

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

제2절 API 악용

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

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

가. 정의

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

나. 안전한 코딩기법

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

다. 예제

■ 안전하지 않은 코드의 예 - Android-JAVA

```
1:  public void onCreate(Bundle savedInstanceState) {
2:      super.onCreate(savedInstanceState);
3:  }
4:
5:  public boolean equals(Object object)
6:  {
7:      return (toString().equals(object.toString()));
8:  }
```

매개 변수가 null인지 검사하지 않았다.

■ 안전한 코드의 예 - Android-JAVA

```
1:  public void onCreate(Bundle savedInstanceState) {
2:      super.onCreate(savedInstanceState);
3:  }
4:
5:  public boolean equals(Object object)
6:  {
7:      if(object != null)
8:          return (toString().equals(object.toString()));
9:      else return false ;
10: }
```

매개 변수가 null인지 먼저 검사한다.

라. 참고 문헌

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

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

가. 정의

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

나. 안전한 코딩기법

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

다. 예제

■ 안전하지 않은 코드의 예 - Android-JAVA

```
1:      ……
2:   public void onCreate(Bundle savedInstanceState) {
3:       super.onCreate(savedInstanceState);
4:   }
5:
6:   public boolean equals(Object obj) {
7:       if (obj == null)
8:           return false;
9:       int i1 = this.hashCode();
10:       int i2 = obj.hashCode();
11:
12:       if (i1 == i2)
13:           return true;
14:       else
15:           return false;
16:  }
```

equals()와 hashCode() 중 하나만 정의하였다.

■ 안전한 코드의 예 - Android-JAVA

```
1:      ……
2:   public boolean equals(Object obj) {
3:       if (obj == null)
4:           return false;
5:       int i1 = this.hashCode();
6:       int i2 = obj.hashCode();
7:
8:       if (i1 == i2)
9:           return true;
10:       else
11:           return false;
12:  }
13:  public int hashCode() {
14:      return new HashCodeBuilder(17, 37).toHashCode();
15:  }
```

equals()와 hashCode() 모두 정의해야 한다.

라. 참고 문헌

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

제3절 보안특성

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

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

가. 정의

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

나. 안전한 코딩기법

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

다. 예제

■ 안전하지 않은 코드의 예 - Android-JAVA

```
1:  ……
2:  public void onCreate(Bundle savedInstanceState) {
3:    int port = 443;
4:    String hostname = "hostname";
5:    Socket socket = new Socket(hostname, port);
6:    InputStream in = socket.getInputStream();
7:    OutputStream out = socket.getOutputStream();
8:   // Read from in and write to out...
9:   in.close();
10:   out.close();
11:  }
```

일반적인 소켓 통신을 사용하여 크를 통하여 데이터를 외부에 전송하고 있다.
이 경우 패킷 스니핑을 통하여 데이터의 내용이 노출될 수 있다.

■ 안전한 코드의 예 - Android-JAVA

```
1:  ……
2:  public void onCreate(Bundle savedInstanceState) {
3:    int port = 443;
4:    String hostname = "hostname";
5:     SocketFactory socketFactory = SSLSocketFactory.getDefault();
6:     Socket socket = socketFactory.createSocket(hostname, , port);
7:    InputStream in = socket.getInputStream();
8:    OutputStream out = socket.getOutputStream();
9:    // Read from in and write to out...
10:    in.close();
11:    out.close();
12:  }
```

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

라. 참고 문헌

[1] CWE-319 기밀 정보의 단순한 텍스트 전송 -http://cwe.mitre.org/data/definitions/319.html

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

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

가. 정의

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

나. 안전한 코딩기법

- AES처럼 보다 강력한 암호화 알고리즘을 사용하는 것이 바람직하다.

다. 예제

■ 안전하지 않은 코드의 예 - Android-JAVA

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

암호화 알고리즘 중에서 DES 알고리즘을 사용하는 것은 안전하지 않다.

■ 안전한 코드의 예 - Android-JAVA

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

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

라. 참고 문헌

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

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

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

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

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

가. 정의

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

나. 안전한 코딩기법

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

다. 예제

■ 안전하지 않은 코드의 예 - Android-JAVA

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

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

■ 안전한 코드의 예 - Android-JAVA

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

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

라. 참고 문헌

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

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

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

4. 전역적으로 접근 가능한 파일(Files under Global Access)

가. 정의

파일 생성시 다른 응용프로그램이 접근할 수 있는 인자값(MODE_WORLD_READABLE, MODE_WORLD_WRITABLE)을 사용할 경우 보안성이나 무결성이 침해될 수 있다.

나. 안전한 코딩기법

- 파일에 대한 접근권한은 최소한으로 유지되어야 한다.

다. 예제

■ 안전하지 않은 코드의 예 - Android-JAVA

```
1:  public void onCreate(Bundle savedInstanceState) {
2:      super.onCreate(savedInstanceState);
3:      try {
4:          FileOutputStream fOut = openFileOutput("test", MODE_WORLD_READABLE);
5:          OutputStreamWriter out1 = new OutputStreamWriter(fOut);
6:          out1.write("Hello World");
7:          out1.close();
8:          fOut.close();
9:      } catch (Throwable t) {
10:     }
11: }
```

파일 접근권한을 MODE_WORLD_READABLE이므로 다른 응용프로그램이 접근할 수 있다.

■ 안전한 코드의 예 - Android-JAVA

```
1:  public void onCreate(Bundle savedInstanceState) {
2:      super.onCreate(savedInstanceState);
3:      try {
4:          FileOutputStream fOut = openFileOutput("test", MODE_PRIVATE);
5:          OutputStreamWriter out1 = new OutputStreamWriter(fOut);
6:          out1.write("Hello World");
7:          out1.close();
8:          fOut.close();
9:      } catch (Throwable t) {
10:     }
11: }
```

외부에서 접근할 수 없도록 MODE_PRIVATE로 권한을 설정하였다.

라. 참고 문헌

[1] http://developer.android.com/index.html
[2] 안드로이드 기반 모바일 운영체제 보안기능 분석, 한국인터넷진흥원

5. 외부에서 접근하여 활성화 가능한 컴포넌트 (Exported Access to Components)

가. 정의

안드로이드 애플리케이션에서 manifest.xml 파일에 android:exported="true"로 설정되어 있는 컴포넌트는 외부에서 해당 컴포넌트에 인텐트를 전달하여 활성화 시킬 수 있다. 이 경우 해당 컴포넌트가 원래 의도하지 않았던 상황에서 수행을 시작함으로써 시스템 보안에 침해를 가져올 수 있다. 또한 이러한 작업 요청은 동일한 인텐트 필터를 사용하는 컴포넌트가 여러 개인 경우 인텐트를 라우팅 하는 리졸버(resolver) 액티비티가 동작하게 되고 리졸버 액티비티를 통해 라우팅되는 인텐트는 System 레벨 사용자 권한으로 송신자의 ID가 바뀌어 전송되므로 보안 침해의 위험이 커진다.

나. 안전한 코딩기법

- 컴포넌트에 대한 접근권한을 외부에 제공하지 않은 것이 바람직하다.

다. 예제

■ 안전하지 않은 코드의 예 - Android-JAVA

```
1:  <?xml version="1.0" encoding="utf-8"?>
2:  <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3:      package="com.example.android.samplesync" android:versionCode="1" an-
    droid:versionName="1.0">
4:  ……
5:   <application android:icon="@drawable/icon" android:label="@string/label">
6:       <service android:name=".syncadapter.SyncService" android:exported="true">
7:           <intent-filter>
8:               <action android:name="android.content.SyncAdapter"/>
9:           </intent-filter>
10:           <meta-data android:name="android.content.SyncAdapter"
11:               android:resource="@xml/syncadapter"/>
12:           <meta-data android:name="android.provider.CONTACTS_STRUCTURE"
13:               android:resource="@xml/contacts"/>
14:       </service>
15:   </application>
16:   <uses-sdk android:minSdkVersion="5"/>
17:  </manifest>
```

SyncService 서비스의 속성값이 android:exported="true"로 설정되어 있어 외부에서 해당 컴포넌트를 구동시킴으로서 보안상의취약점이 발생할 수 있게 된다.

■ 안전한 코드의 예 - Android-JAVA

```
1:  <?xml version="1.0" encoding="utf-8"?>
2:  <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3:      package="com.example.android.samplesync" android:versionCode="1" an-
    droid:versionName="1.0">
4:  ……
5:   <application android:icon="@drawable/icon" android:label="@string/label">
6:       <service android:name=".syncadapter.SyncService" android:exported="false">
7:           <intent-filter>
8:               <action android:name="android.content.SyncAdapter"/>
9:           </intent-filter>
10:           <meta-data android:name="android.content.SyncAdapter"
11:               android:resource="@xml/syncadapter"/>
12:           <meta-data android:name="android.provider.CONTACTS_STRUCTURE"
13:               android:resource="@xml/contacts"/>
14:       </service>
15:   </application>
16:   <uses-sdk android:minSdkVersion="5"/>
17:  </manifest>
```

android:exported 속성을 "false"로 설정하거나 설정을 제거하면 해당 속성이 "false"가 되어 외부로부터의 구동이 차단된다.

라. 참고 문헌

[1] http://developer.android.com/index.html

[2] 안드로이드 기반 모바일 운영체제 보안기능 분석, 한국인터넷진흥원

6. 공유 아이디에 의한 접근제어 통과 (Access Control Bypass using Share User ID)

가. 정의

Manifest.xml 파일의 manifest 태그에 android:sharedUserId 속성을 설정할 경우 같은 아이디와 서명을 사용함으로써 다른 응용프로그램이 해당 프로그램의 정보를 접근할 수 있게 된다. 이를 통하여 의도적 및 비의도적으로 해당 프로그램의 무결성과 보안성이 침해될 수 있다.

나. 안전한 코딩기법

- 공유 아이디 설정을 하지 않는 것이 바람직하다.

다. 예제

■ 안전하지 않은 코드의 예 - Android-JAVA

```
1:  ……
2:  <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3:          package="com.example.android.apis"
4:      android:versionCode="1"
5:      android:versionName="1.0"
6:      android:sharedUserId="android.uid.developer1">
```

Manifest.xml 파일의 manifest 태그에 android:sharedUserId 속성을 설정하고 있어 같은 sharedUserId 태그값과 응용프로그램 서명을 가진 다른 응용프로그램이 이 프로그램의 모든 데이터에 접근할 수 있다.

■ 안전한 코드의 예 - Android-JAVA

```
1:  ……
2:  <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3:          package="com.example.android.apis"
4:      android:versionCode="1"
5:      android:versionName="1.0">
6:  <!-- android:sharedUserId="android.uid.developer1" 삭제한다. -->
```

Manifest.xml 파일의 manifest 태그에 android:sharedUserId 속성을 설정하지 않아야, 아이디 공유로 인한 데이터의 유출이나 부적절한 접근 위험을 방지할 수 있다.

라. 참고 문헌

[1] http://developer.android.com/index.html
[2] 안드로이드 기반 모바일 운영체제 보안기능 분석, 한국인터넷진흥원

제4절 시간 및 상태

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

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

가. 정의

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

나. 안전한 코딩기법

- 공유자원(예: 파일)을 여러 스레드가 접근하여 사용할 경우, 동기화 구문을 이용하여 한 번에 하나의 스레드만 접근 가능하도록 프로그램을 작성하여야 한다.

다. 예제

■ 안전하지 않은 코드의 예 - Android-JAVA

```
1:  public class UA367 extends Activity {
2:      @override
3:      public void onCreate(Bundle savedInstanceState) {
4:          super.onCreate(savedInstanceState);
5:              FileAccessThread fileAccessThread = new FileAccessThread();
6:              FileDeleteThread fileDeleteThread = new FileDeleteThread();
7:              fileAccessThread.start();
8:              fileDeleteThread.start();
9:      }
10: }
11:
12: class FileAccessThread extends Thread {
13:     public void run() {
14:       try {
15:           File f = new File("Test_367.txt");
16:           if (f.exists()) {  // 만약 파일이 존재하면 파일내용을 읽음
17:             BufferedReader br = new BufferedReader(new FileReader(f));
18:             br.close();
19:           }
20:       } catch(FileNotFoundException e) {
21:             System.out.println("Exception Occurred") ; //예외처리
```

```
22:             } catch(IOException e) {
23:                     System.out.println("Exception Occurred") ; //예외처리
24:     }
25:   }
26:
27:   class FileDeleteThread extends Thread {
28:       public void run() {
29:         try {
30:                 File f = new File("Test_367.txt");
31:                 if (f.exists()) {  // 만약 파일이 존재하면 파일을 삭제함
32:                   f.delete();
33:                 }
34:         } catch(FileNotFoundException e) {
35:                     System.out.println("Exception Occurred") ; //예외처리
36:         } catch(IOException e) {
37:                     System.out.println("Exception Occurred") ; //예외처리
38:         }
39:       }
40:   }
```

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

■ 안전한 코드의 예 - Android-JAVA

```
1:   public class SA367 extends Activity  {
2:       public void onCreate(Bundle savedInstanceState) {
3:       super.onCreate(savedInstanceState);
4:
5:       FileAccessThread fileAccess = new FileAccessThread();
6:       Thread first = new Thread(fileAccess);
7:       Thread second = new Thread(fileAccess);
8:       Thread third = new Thread(fileAccess);
9:       Thread fourth = new Thread(fileAccess);
10:       first.start();
11:       second.start();
12:       third.start();
13:       fourth.start();
14:       }
15:   }
16:
17:   class FileAccessThread implements Runnable {
18:       public synchronized void run() {
```

```
19:             try {
20:               File f = new File("Test.txt");
21:               if (f.exists()) { // 만약 파일이 존재하면 파일 내용을 읽음
22:                   Thread.sleep(100);        // 시간이 소요되는 작업을 가정함
23:                   BufferedReader br = new BufferedReader(new FileReader(f));
24:                   System.out.println(br.readLine());
25:                   br.close();                  // 파일 내용을 모두 읽은 후 삭제
26:                   f.delete();
27:               }
28:             } catch (IOException e) {    // 예외처리
29:                    System.err.println("IOException occured");
30:             }
31:         }
32:     }
```

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

라. 참고 문헌

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

2. 제대로 제어되지 않은 재귀(Uncontrolled Recursion)

가. 정의

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

나. 안전한 코딩기법

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

다. 예제

■ 안전하지 않은 코드의 예 - Android-JAVA

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

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

■ 안전한 코드의 예 - Android-JAVA

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

모든 재귀 호출은 조건문이나 반복문 블럭 안에서 수행하고 적절한 귀납조건 설정과 인수의 수렴여부를 확인해야 한다.

라. 참고 문헌

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

제5절 에러 처리

정상적인 에러는 사전에 정의된 예외사항이 특정 조건에서 발생하는 에러이며, 비정상적인 에러는 사전에 정의되지 않은 상황에서 발생하는 에러이다. 개발자는 정상적이거나 비정상적인 에러 발생에 대비한 안전한 에러처리 루틴을 사전에 정의하고 구현함으로써 에러처리 과정 중에 발생 할 수 있는 보안위협을 사전에 방지할 수 있다. 에러를 충분하게(또는 전혀) 처리하지 않을 때 혹은 에러메시지에 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

SW의 오류 메시지를 통해 환경, 사용자, 관련 데이터 등 프로그램 내부정보가 유출될 수 있다. 예로, 예외발생 시 예외이름이나 스택트레이스를 출력하면 프로그램 내부구조를 쉽게 파악할 수 있다.

나. 안전한 코딩기법

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

다. 예제

■ 안전하지 않은 코드의 예 - Android-JAVA

```
1:  ……
2:  public void onCreate(Bundle savedInstanceState) {
3:      super.onCreate(savedInstanceState);
4:      try{ throw new IOException(); }
5:      catch (IOException e) {    e.printStackTrace();    }
6:  }
```

예외 이름이나 스택 트레이스를 출력하면 프로그램 내부 정보가 유출된다.

■ 안전한 코드의 예 - Android-JAVA

```
1:  ……
2:  public void onCreate(Bundle savedInstanceState) {
3:      super.onCreate(savedInstanceState);
4:      try{
5:          throw new IOException();
6:      }
7:      catch (IOException e) {    System.out.println("예외발생");    }
8:  }
```

예외 이름이나 스택 트레이스를 출력하지 않는다.

라. 참고 문헌

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

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

가. 정의

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

나. 안전한 코딩기법

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

다. 예제

■ 안전하지 않은 코드의 예 - Android-JAVA

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

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

■ 안전한 코드의 예 - Android-JAVA

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

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

라. 참고 문헌

[1] CWE-390 오류 상황에 대한 처리 부재 - http://cwe.mitre.org/data/definitions/390.html
[2] OWASP Top Ten 2004 Category A7 - Improper Error Handling

제6절 코드 품질

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

1. 널포인터 역참조(NULL Pointer Dereference)

가. 정의

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

나. 안전한 코딩기법

- 레퍼런스(reference)에 대한 null값 여부를 검사하여 안전한 경우에만 사용해야 한다.

다. 예제

■ 안전하지 않은 코드의 예 - Android-JAVA

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

위 예제는 "cmd" 속성이 항상 정의되어 있다고 가정하고 있지만, 공격자가 "cmd" 속성을 조작하면, cmd는 null이 되고 trim() 메소드 호출시 널 포인터 예외가 발생하게 된다.

■ 안전한 코드의 예 - Android-JAVA

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

먼저 cmd가 널인지 검사한 후에 사용한다.

라. 참고 문헌

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

제7절 캡슐화

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

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

가. 정의

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

나. 안전한 코딩기법

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

다. 예제

■ 안전하지 않은 코드의 예 - Android-JAVA

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

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

■ 안전한 코드의 예 - Android-JAVA

```
1:  private String[] colors;
2:  // 메소드를 private으로 하거나, 복제본 반환, 수정하는 public 메소드를 별도로 만든다.
3:  public void onCreate(Bundle savedInstanceState) {
4:      super.onCreate(savedInstanceState);
5:      String[] newColors = getColors();
6:  }
7:  public String[] getColors() {
8:      String[] ret = null;
9:      if ( this.colors != null ) {
10:         ret = new String[colors.length];
11:         for (int i = 0; i < colors.length; i++) {    ret[i] = this.colors[i];    }
12:     }
13:     return ret;
14: }
```

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

라. 참고 문헌

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

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

가. 정의

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

나. 안전한 코딩기법

- public으로 선언된 데이터가 private 선언된 배열에 저장되지 않도록 한다.

다. 예제

■ 안전하지 않은 코드의 예 - Android-JAVA

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

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

■ 안전한 코드의 예 - Android-JAVA

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

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

라. 참고 문헌

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

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

가. 정의

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

나. 안전한 코딩기법

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

다. 예제

■ 안전하지 않은 코드의 예 - Android-JAVA

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

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

■ 안전한 코드의 예 - Android-JAVA

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

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

라. 참고 문헌

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

제2장 용어정리 및 약어표

제1절 용어정리

- **Advanced Encryption Standard (AES)** : 미국 정부표준으로 지정된 블록암호 형식으로 이전의 DES를 대체하며, 미국 표준 기술 연구소(NIST)가 5년의 표준화 과정을 거쳐 2001년 11월에 연방정보처리표준(FIPS 197)으로 발표하였다.
- **DES 알고리즘** : DES(Data Encryption Standard)암호는 암호화키와 복호화키가 같은 대칭키 암호로 이 암호는 대칭 블록암호로서 평문의 각 블록의 길이가 64비트이고, 키가 64비트이며, 암호문이 64비트인 암호이다. 전수공격(Brute Force)공격에 의해서 해독되었다.
- **Manifest 파일** : 안드로이드용 어플리케이션의 권한, 리소스 사용 등을 정의한 XML 문서
- **Synchronized** : JAVA에서 임계코드를 동기화하기 위해서 제공하는 구문이다.

제2절 약어표

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

[개정 이력]

순번	제・개정일	변경 내용	비고
1	2011. 6. 21	[제정] SW 개발보안 가이드	V1.0
2	2011. 8. 25	[개정] o '붙임3. Android-JAVA 시큐어코딩 가이드' ・(p.2) '상대디렉터리 경로 조작'에 대한 '안전한 소스코드 예제' 수정 ・(p.15) '외부에서 접근하여 활성화 가능한 컴포넌트'의 '가. 정의' 수정	V1.1

Android-JAVA
시큐어 코딩 가이드

2011년 6월 초판 인쇄
2011년 6월 초판 발행
2011년 9월 2판 인쇄
2011년 9월 2판 발행

발행처 행정안전부 (http://www.mopas.go.kr)

인쇄처 한올 (Tel: 02-2279-8494)

□ 본 보고서의 내용과 관련한 문의는 아래로 해 주시기 바랍니다.

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

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

정보시스템 SW 개발 · 운영자를 위한

소프트웨어 개발보안 가이드

행정안전부