Internal Revenue Service

JAVA Secure Coding Guidelines for Most Common IRS Vulnerabilities

Final

Java Application Standard Section: OS:IT:AD:TIO:ASQ:JAS
5/25/2017

Table of Content

Introduction

Secure coding is very broad topic. In this document, we mainly focus on 8 most common vulnerabilities in IRS and aim to provide guideline, detection, prevention and references for every vulnerability. There are many general mitigation and prevention methods applicable to all these common vulnerabilities, like top 10 secure coding practices. In this document, we provide more specific guidance for 8 most common vulnerabilities found by AppScan in IRS.

1. Cross-site scripting (XSS)

Cross-site scripting (XSS) is a security vulnerability often found in web applications. XSS flaws occur whenever an application takes untrusted client-supplied data and sends it to a web browser without validating or encoding that content. XSS usually is utilized by attackers to inject malicious client-side scripts into web pages viewed by different end user. XSS are generally classified into three types: reflected XSS, stored XSS and DOM-based XSS.

1.1 How to detect XSS vulnerability

- Code Review: The most efficient way to find XSS vulnerability is to perform code review and thoroughly examine all spots where malicious input can be injected into the HTML output.
- Tools: Nessus, Nikto, and other similar tools can help scan a website for XSS flaws.

1.2 How to prevent XSS vulnerability

Developers can largely eliminate XSS through sanitizing input and encoding all variable output in a web page before sending it to the end user's browser. OWASP provides the following more detail check lists for mitigating XSS vulnerability:

- XSS (Cross Site Scripting) Prevention Cheat Sheet
- DOM based XSS Prevention Cheat Sheet

1.3 Available Tools/JAVA libraries for XSS

- Testing for Reflected XSS
- Testing for Stored XSS
- Testing for DOM-based XSS
- OWASP JAVA Encoder library may help contextual encoding.
- OWASP HTML Sanitizer may help detecting XSS vulnerability introduced by third party.
- OWASP AntiSamy is a library for HTML and CSS encoding.
- OWASP ESAPI is a security control library that makes it easier for programmers to write lower-risk applications.
- Content Security Policy (CSP) is a security mechanism that helps protect against Cross Site Scripting (XSS).

1.4 Examples

- XSS in JSF

Noncompliant Code Example: The only potential XSS attack hole is explicitly set *escape="false"*

```
<h:outputText value="#{user.name}" escape="false" />
```

Compliant Code Example: JSF has built-in XSS prevention. It is safe to display all user input in any JSF component, which are all implicitly escaped.

```
<h:inputText value="#{user.name}" />
<h:outputText value="#{user.name}" />
```

In JSF 2.X, it is also safe to use EL in web page like :

```
<p>Welcome, #{user.name}</p>
```

- User Input HTML tag

If web app needs to allow users to utilize some of the basic html tags to customize their inputs, JBoss Seam provides a *<s:formattedText/>* tag that allows some basic html tags and styles specified by users, which validate and escape user's inputs by default. See Seam Reference Manual for detail.

- Server Side Encoding

Noncompliant Code Example: Direct put user input in response is a potential attack hole:

```
response.write("<b>" + user_input + "</b>");
```

Compliant Code Example: Filter/sanitize and encode user input is essential to prevent XSS attack.

```
response.write("<b>" + escapePlainTextToHtml(user_input) + "</b>")
```

- XSS in JSP

Using *JSTL <c:out>* tag or *fn:escapeXml()* EL function when (re)displaying user-controlled input

```
<p><c:out value="${bean.userControlledValue}"></p>
<p><input name="foo"
```

```
value="${fn:escapeXml(bean.userControlledValue)}"></p>
```

An alternate way to prevent XSS in JSP is to migrate JSP to JSF.

- Content Security Policy (CSP)

Content Security Policy (CSP) is a security mechanism that helps protect against Cross Site Scripting (XSS). CSP defines *Content-Security-Policy* HTTP header that allows web apps instruct the browser a whitelist of where it can load resources from, whether the browser can use inline styles or scripts.

Example of simple usage, only use script from same site:

```
Content-Security-Policy: script-src 'self'
```

2. System Resource Leaks

System Resource Leaks (SRL) vulnerability occurs when an application does not properly release the resources it acquired. This vulnerability can significantly slow down the app performance and cause app instability. SRL is often used by denial of service attack to cause serious computational resource exhaustion such as bandwidth, memory, disk space and processor time.

2.1 How to detect SRL vulnerability

SRL errors are generally caused by programming. Good code preview may mitigate the risks. Typical resource leaks are memory leak and file handle leak. A good sign for memory leaking is the app runs fast at first and slows down over time.

2.2 How to prevent system resource leaks

- Check each resource app acquired and release it explicitly after acquisition.
- Allocate enough resources to meet demand traffic flow
- Classes implementing the interface *java.io.Closeable* (since JDK 1.5) and *java.lang.AutoCloseable* (since JDK 1.7) should be closed explicitly using method *close()* when they are no longer needed.
- Input into a system should be validated/checked to avoid excessive resource consumption. Recommend reading Secure Coding Guideline: Denial of Service.

2.3 Available Tools

Memory analyzer plugin(MAT) from Eclipse helps identify memory leak suspects
VirtualVM helps monitor and find memory leaks.
JRockit from Oracle help analyze runtime performance and detect memory leaks.

2.4 Examples

- Since JAVA SE 7, use try-with-resources statement to ensure closing resources

```
public static Object[] fromFile(String filePath) throws
FileNotFoundException, IOException
{
   int sz = /* get buffer size somehow */
   try (FileReader file = new FileReader(filePath);
       BufferedReader br = new BufferedReader(file, sz))
   {
      return read(br);
   }
}
```

In the above example, the resource declared in the try-with-resources are FileReader and BufferedReader, which will be closed whether the try statement completes normally or abruptly.

- Prior to JAVA SE 7, Deallocation of resources is preferably done in a finally{} instead of in finalize{}

```
public static Object[] fromFile(String filePath) throws FileNotFoundException, IOException
{
   BufferedReader br = null;

   try
   {
      br = new BufferedReader(new FileReader(filePath));
      return read(br);
   }
   catch (Exception ex)
   {
      throw ex;
   }
   finally
   {
      try
      {
         if (br != null) br.close();
      }
      catch(Exception ex)
      {
      }
   }

   return null;
}
```

3. Sensitive Data Exposure

Sensitive date exposure (SDE) vulnerabilities can occur when an application does not properly protect sensitive information from being disclosed to attackers.

3.1 How to detect SDE vulnerability

Extra protection is needed for sensitive data like credit card numbers, personally identifiable information, medical records, tax information etc. The following checklist may help to find out if the application is vulnerable to SDE attack:

- Is this data stored in plain text?
- Is this data transmitted in clear text, internally or externally?
- Are any old / weak or in-house developed cryptographic algorithms used?
- Are weak crypto keys generated, or is proper key management or rotation missing?
- Are any browser security directives or headers missing when sensitive data is provided by / sent to the browser?
- And more …

3.2 How to prevent Sensitive Data Exposure

Following OWASP guideline, for all sensitive data, do all of the following, at a minimum:

- Encrypt all sensitive data and passwords
- Providing transport layer protection with TLS, refer to NIST 800-52
- Utilize public trusted algorithms/libraries, avoid in-house developed code for cryptographic functionality, refer to NIST 800-52 .
- Choose strong algorithms and strong keys, and proper key management. Consider using FIPS 140 validated cryptographic modules.
- Ensure passwords are stored with an algorithm specifically designed for password protection, such as bcrypt, PBKDF2, or scrypt.
- Turn off autocomplete fields on forms collecting sensitive data and disable caching for pages that contain sensitive data.
- Provide strong cryptographic functionality, follow OWASP cheat sheet to insure proper protection.

3.3 Available JAVA libraries for Cryptography

- Java security package provides comprehensive coverage for essential cryptographic functionality.
- Spring security module provides bcrypt and scrypt algorithms.

3.4 Examples

- Generate a secure password

```
String password=”password”;
MessageDigest md =
MessageDigest.getInstance("PBKDF2WithHmacSHA1");

SecureRandom sr = SecureRandom.getInstance("SHA1PRNG");
byte[] salt = new byte[16];
sr.nextBytes(salt);
md.update(salt);
byte[] newpassword=md.digest(password);
```

- Key generation

```
KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA”);
SecureRandom random = SecureRandom.getInstance("SHA1PRNG");
keyGen.initialize(1024, random);
KeyPair keypair = keyGen.generateKeyPair();
PrivateKey private_key = pair.getPrivate();
PublicKey public_key = pair.getPublic();
```

- Encrypt Data with public key

```
Cipher cipher = Cipher.getInstance(“RSA”);
cipher.init(Cipher.PUBLIC_KEY, public_key);
byte[] encryptedBytes = cipher.doFinal(Your_DATA_NEED_BE_ENCRYPTED);
```

- Decrypt Data with private key

```
Cipher cipher = Cipher.getInstance(“RSA”);
cipher.init(Cipher.PRIVATE_KEY, private_key);
byte[] decryptedBytes = cipher.doFinal(Your_DATA_NEED_BE_DECRYPTED);
```

4. Insecure Direct Object Reference

An insecure direct object reference (IDOR) occurs when unauthorized users/attackers can access a direct object reference, which could be an internal implementation object, such as files, directories or database records. This may result in significant impact on an organization information leakage depending on the type of exposed data during the attack.

4.1 How to detect IDOR vulnerability

Questions to ask developers:

- Does each field, where a user can supply input and may point directly to reference objects, have sufficient authorization checks?
- Is it possible to access resources belonging to other users or bypass authorization using different URL parameters or different login credentials?

4.2 How to prevent insecure direct object reference

- To prevent these vulnerabilities, authentication/authorization policies must be in place and thoroughly enforced.
- According to OWASP, the first checkpoint of this vulnerability is to "map out all locations in the application where user input is used to reference objects directly".
- Applications need validate all user inputs.
- Verify user authorization each time sensitive objects/files/contents are requested
- If reference is indirect, the mapping to the direct reference must be restricted to the authorized current user.
- Indirect reference map may add an extra layer to the app's security.
    - minimize user ability to predict object IDs/Names
    - Don't expose the actual ID/name of objects
- Code review and website walkthrough may be done to prevent this vulnerability.

4.3 Available Tools/Libraries

ESAPI (The OWASP Enterprise Security API) is a free, open source, web application security control library that makes it easier for programmers to write lower-risk applications.

4.4 Examples

- Exploiting URL parameters

  The attacker simply modifies URL parameters in the browser to access unauthorized data. The following example shows attacker accesses another user account by changing parameter *account* in URL string.

```
http://demo.com/account?account=not-my-account
```

  If the corresponding server-side code (like below) doesn't perform user verification, the attacker an access any user's account

```
String query = "SELECT * FROM accts WHERE account = ?";
PreparedStatement  pstmt = connection.prepareStatement(query);

pstmt.setString( 1, request.getParameter("account"));
ResultSet results = pstmt.executeQuery();
….
//return results to UI
```

- Indirect Reference Map

Indirect reference map is used for converting from a set of internal direct object references (i.e. user IDs, filenames, database keys) to a set of indirect reference that can be safely exposed to the public. Using a random generated string instead of straight object property makes it hard for hackers to guess real identifiers.

```
MyFileObject  myFile ;

 // generate my own object
myFile =…..;

AccessReferenceMap map = new RandomAccessReferenceMap();

 // store the map  somewhere safe - like the session!
 String indRef = map.getIndirectReference( myFile );

 String href = "http://www.demo.com/accessFile?file=" + indRef );
 ...
//retrieve
String indref = request.getParameter( "file" );

 MyFileObject  file = (MyFileObject )map.getDirectReference( indref );
```

5. SQL Injection

SQL (Structured Query Language) injection is a common application security flaw that allows an attacker to inject (execute) SQL commands in an application. It is on OWASP top 10 list and one of the most widespread and dangerous application vulnerability. In general, the input can come from any HTML based form that submits parameters that are processed by a backend application that relies on a database .

5.1 How to detect SQL injection

Comprehensive detection of SQL injection attacks is a difficult task. Web server log auditing combined with network Intrusion Detection Systems (IDS) can help finding SQL injection. Various software tools exist that allow for the rapid search of webserver logs for specified keywords or regular expressions. such as "EXEC", "POST", "UNION", "CAST", or a single quotation mark, which should not normally be provided by the end users. Besides these, attackers also can exploit the whitespace between commands, encode using HEX, BASE 64 etc. in order to bypass IDS detection or log-based analysis.

5.2 How to prevent SQL injection

- Disable potentially harmful SQL stored procedure calls
- Use parametrized SQL query
- Document all database accounts, stored procedures/functions and prepared statements along with their use
- Harden network setting, implementing firewall rules to block or restrict external and internal database access.
- Secure the operating system and applications, using principles of least privilege
- More detail see OWASP SQL Injection Prevention Cheat Sheet

5.3 Recommendations & Samples

• Use of prepared statement with bind variables

Vulnerable Usage

```
Statement stmt = connection.createStatement ();
String query = "SELECT users FROM user_table WHERE user_name =?"
```

Malicious user input

```
Joe or '1'='1'
```

Exploit

SELECT users FROM user_table WHERE user_name = *Joe or '1'='1'* will return/divulge list of all users from the table.

Secure Usage

```
PreparedStatement pstmt = connection.prepareStatement (query);
pstmt.setString (1, user_name);
```

• Use parameterized query instead of dynamic SQL inside the stored procedures

Vulnerable Usage

```
CREATE PROCEDURE getUserID
  @name VARCHAR (50)
AS
  EXEC ('SELECT user_id FROM users WHERE user_ name = '''+@name+ ''')
RETURN
```

Malicious user input

```
xxx' AND 'a'='b' UNION SELECT password FROM users WHERE
user_name='admin
```

Exploit

```
'SELECT user_id FROM users WHERE user_name = '''xxx' AND 'a'='b' UNION
SELECT password FROM users WHERE user_name='admin''''
```

The above SQL will return password for administrator

Secure Usage

```
CREATE PROCEDURE getUserID
  @name VARCHAR (50)
AS
  EXEC('SELECT user_id FROM users WHERE user_ name = :input';)
RETURN
```

• For Hibernate HQL, avoid using deprecated session.find () method

Vulnerable Usage

```
Salary salary = (Salary) session.find ("from org.example.Salary as salary where
user.id=" +
userIds.get (i));
```

The above HQL will allow injection attacks from userId's

Malicious user input

```
= Joe_id or '1'='1'
```

Exploit

Joe_id or '1'='1' will return salaries of all users from salary table.

Secure Usage

```
int userId = userIds.get (i);
Salary salary = (Salary) session.find ("from org.example.Salary as salary where
user.id=?", userId, StringType);
```

References

OWASP: https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet

MITRE: https://capec.mitre.org/data/definitions/66.html

Secure Coding Guidelines 3-2 -- http://www.oracle.com/technetwork/java/seccodeguide-139067.html#3

CERT guidelines -- https://www.securecoding.cert.org/confluence/display/java/IDS00-J.+Prevent+SQL+injection

6. Broken Authentication and Session Management

Application functions related to authentication and session management, if not implemented correctly, can allow visibility to unauthorized information. This can result in compromise of passwords, tokens, cookies or other information that can allow the attacker to impersonate the user. Types of broken authentication and session management attacks are:

Session Fixation Attack
Session Hijacking
Brute Force Attack (Password Management)
Replay Attack
Insufficient Session Expiration

• Session Fixation Attack

In Session Fixation attack, the user is "fixated" or tagged with a session ID that is already known by attacker. The known session ID (generated by attacker) could be distributed to the victim(s) using a phishing link in email. Once the user logs in successfully to the trusted site, the attacker is able to impersonate the user using the known session ID.

• Session Hijacking

The Session Hijacking Attack is an attack in which the attacker tries to get the user's session ID and then use that session ID for his own session impersonating the victim. Session Hijacking is almost identical to Session Fixation except that in Session Hijacking the user's session ID is not known before the attack.

• Brute Force Attack (Password Management)

Brute Force Attacks (Password Management) Brute Force attacks occur when an attacker is able to repeatedly guess or identify unauthorized information such as usernames, passwords, operations, etc. from an application.

• Replay Attack

The Replay Attack is an attack in which the user's browser communication is being listened to by an attacker. This communication is captured in some manner and then played again or "replayed" so that the attacker masquerades as the legitimate user. The Replay attack is also known as a man-in-the-middle attack. There are numerous techniques that an attacker can use to eavesdrop on users. Such techniques include IP packet sniffing and substitution, keystroke capturing even if the traffic is encrypted, and cookie-stealing via proxy

• Insufficient Session Expiration

Insufficient Session Expiration occurs when a web site allows an attacker to reuse old session data (credentials, session ID's etc.) for access. This increases a web site's exposure to attacks that steal or impersonate other user. A long expiration time increases an attacker's chance of successfully guessing a valid session ID.

6.1 How to detect Broken Authentication and Session Management

- Code Review and Pen Testing: Both code review and penetration testing can be used to diagnose authentication and session management problems. Carefully review each aspect of your authentication mechanisms to ensure that user's credentials are protected at all times, while they are at rest (e.g., on disk), and while they are in transit (e.g., during login). Review every available mechanism for changing a user's credentials to ensure that only an authorized user can change them. Review your session management mechanism to ensure that session identifiers are always protected and are used in such a way as to minimize the likelihood of accidental or hostile exposure.
- Tools: AppScan

6.2 How to prevent Broken Authentication and Session Management

- User Logins
  - Always use encrypted forms for user login.
  - Validate form input.
  - Avoid using remember me functionality with mission critical applications.
  - Provide users with a logout button to manually terminate a session
- Password Policies
  - Don’t use verbose error messages (e.g. invalid username, account does not exist)
  - Use strong passwords: length and complexity (alphanumeric with special characters)
  - Use password aging: passwords expire after a specific time forcing users to change it
  - Don’t allow use of previous passwords when renewing the password.
  - Lockout the account after a minimum login threshold is reached
  - Store hashed passwords (versus plaintext) using a salt to prevent rainbow table attacks.
  - Provide two-step authentication features
  - Implement strong complex password requirements
  - Notify users for any modifications to password.
- Session Control
  - Expire a session token after a predefined period of time
  - Expire a session token after a predefined period of activity
  - Expire a session token when users logs out. The browser cookies should be deleted and user’s session object on the server should be destroyed.
  - Use strong random number generators to ensure you have a sufficiently long session Id
  - Store session ids in cookies and never pass them via URL parameters, hidden form fields, or custom HTTP headers

Use standard frameworks to handle security rather than writing your own custom session management systems

Set absolute time limits on session identifiers to ensure proper session expiration

- Cookie Security

  Store session identifiers in session cookies rather than persistent cookies.

  Set Secure cookie attribute to ensure such are transmitted over secure connections.

  Set HttpOnly cookie to ensure scripts cannot access these cookies via DOM object.

6.3 Available tools/Libraries

http://docs.oracle.com/javase/6/docs/api/java/security/SecureRandom.html API to generate cryptographically strong random and unpredictable session ID.

6.4 Examples

Vulnerable Usage

Cookies remain the same (are not modified) after login is successful (e.g. JSESSIONID)

Exploit

Session cookie can be used by attacker to impersonate the real user

Secure Usage

Change JSESSIONID cookie before and after login:

```
//Issue is same session object is being used so get current session
HttpSession beforeloginsession = request.getSession();
//invalidate that session
 beforeloginsession.invalidate();
//Generate a new session, new JSESSIONID
HttpSession afterloginsession = request.getSession(true);
```

Vulnerable Usage

Cookie value is sent unencrypted in clear text

Exploit

Session cookie can be used by attacker to impersonate the real user

Secure Usage

Use HTTPOnly and Secure flags

Configurations for the setting of the HTTPOnly and Secure flags are generally handled in the configuration of the web application, application server as well as the client browser. Be sure your browser is up-to-date as well.

Setting of HTTPOnly Flag in web application configuration file

```
<web-app>
<session-config>
<cookie-config>
<!--
session tracking cookies created by this application will be marked as HttpOnly
-->
        <http-only>true</http-only>
</cookie-config>
 </session-config>
</web-app>
```

HTTPOnly Flag setting using Java API

```
Cookie cookie = getClientCookie("clientCookieName");
cookie.setHttpOnly(true);
```

Setting of Secure Flag in web application configuration file

```
<web-app>
<session-config>
<cookie-config>
<!--
Any session tracking cookies created by this application will be marked as
HttpOnly
-->
        <secure>true</secure>
</cookie-config>
 </session-config>
</web-app>
```

Vulnerable Usage

Cookie value is predictable (e.g. using incremental counter)

Exploit

Session cookie can be used by attacker to impersonate the real user

Secure Usage

Ensure the Session ID that your web application creates is user-unique and unpredictable. This is usually accomplished with the use of the SecureRandom class in Java(*java.security.SecureRandom*).

See http://docs.oracle.com/javase/6/docs/api/java/security/SecureRandom.html for more details. Do not rely on using a timestamp as this has proven to be predictable. Also, do not use the Random class(*java.util.Random*)in Java for the creation of your session tokens. The API clearly states to not do this:"Instances of *java.util.Random* are not cryptographically secure."See http://docs.oracle.com/javase/7/docs/api/java/util/Random.html for more details.

Vulnerable Usage

Cookie value/Session ID value is visible

Exploit

Session cookie can be used by attacker to impersonate the real user

Secure Usage

Don’t put session IDs in the URL.

Don’t hardcode credentials in configuration files or inside the code.

7. Cross-Site Request Forgery (CSRF)

CSRF is the use of a forged session data stolen by an attacker. The victim is tricked into issuing a request to an attacker-controlled website without the victim's knowledge. The authentic website has a CSRF vulnerability (lack of CSRF token), the attacker is able to use for forged session to masquerade as legitimate user.

7.1 How to Detect Cross-Site Request Forgery (CSRF)

- Code Review: The easiest way to check whether an application is vulnerable is to see if each link and form contains an unpredictable token for each user. Without such an unpredictable token, attackers can forge malicious requests. Focus on the links and forms that invoke state-changing functions, since those are the most important CSRF targets.
- Tools: AppScan
- Monitoring Events: Mismatch of token value or non-existent of a Session object should abort request; log message as potential CSRF attack and Alert or notify production support

7.2 How to Prevent Cross-Site Request Forgery (CSRF)

- **Anti-CSRF Tokens**
  The Enterprise Security API (ESAPI) from OWASP offers anti-CSRF tokens in their library. Using this API, the developer can generate anti-CSRF tokens on the server-side and append the token to each response given to an authenticated user. The user's session stores the token and, upon each subsequent request, sends that token as a hidden field. Upon receiving the request, there is a comparison of the anti-CSRF token value from the client to the expected value stored on the server-side. As long as the client's token matches what the server expects, the session remains active. However, if there is a mismatch, the session can, immediately, be invalidated

  Generate token on server-side

```
// Generate a random string
SecureRandom random = SecureRandom.getInstance("SHA1PRNG");

Place into hidden field of web page within Response and in Session object as an attribute:
<input type='hidden' name='"+ CSRFTokenUtil.SESSION_ATTR_KEY +"' value='" + getToken() + "'/>";
```

After receipt of token from subsequent request, validate on server-side:

```
IF [gen_token_in_req_matches]
THEN
[continue]
 ELSE
[log_error_unacceptable_token_value_received]
 END-IF.
request.getParameter(SESSION_ATTR_KEY) == getToken()
```

- Require users to re-authenticate when performing critical sensitive operations
- Have strong measures for re-authentication such as Captcha, 2 step verification processes
- **Multiple Browser Tabs**
  Having multiple browser tabs open at the same time allows the availability of authenticated session information in those tab from one browser. End users should always click "Logout" to invalidate their authenticated cookies as well as close their browsers once done.

7.3 Available Tools/Libraries

CSRF Guard: https://www.owasp.org/index.php/Category:OWASP_CSRFGuard_Project
ESAPI: https://www.owasp.org/index.php/Category:OWASP_Enterprise_Security_API

7.4 Examples

• Use of secure random token per request

Vulnerable Usage

Creation of a new user

```
<form method="POST" action="http://www.example.com/create_new_user">
    <input type="hidden" name="user_id" value="my_user">
   <input type="hidden" name="password" value="my_password">
</form>
```

Transfer of funds

http://bank.com/transferfunds?account=real_user&Amount=10000

Malicious user input

Both of above examples are lacking identification from the client. This allows submission of a form (to create users) or submission of a GET request (to transfer funds) without any unique identifier to qualify the sender of the request; meaning, there is no evidence to support that the sender is the intended party instead of an impostor. The application on server does not validate the requester of the action (form submission/GET request) in any way.

Exploit

Creation of a new user allows hacker to create user accounts on website

```
<form method="POST" action="http://www.example.com/create_new_user">
<input type="hidden" name="user_id" value="hacker_userid">
<input type="hidden" name="password" value="hacker_password">
</form>
<script>   document.usr_form.submit(); </script>
```

Transfer of funds allows hacker to transfer funds to a different account
http://bank.com/transferfunds?account=hacker_user&Amount=10000

8. Defective third party libraries

- Never trust third party libraries.
- All third party libraries must be scanned, sanitized and approved by IRS before use.

9. Other Vulnerabilities

Java application vulnerabilities are moving targets and will be long-term battle. As more and more free and commercial tools have been developed to combat these security issues, a widespread vulnerability may drop into an uncommon one and new one may emerge any time. One OWASP released 2017 Top 10 vulnerabilities, two new vulnerabilities are reported: Under-protected APIs and Insufficient Attack Protection.

This guideline provides basic information for current most common vulnerabilities in IRS and related mitigation strategies. Every development team/section should keep coding guideline in mind to build safe and solid IRS JAVA applications.