FIO52-J. Do not store unencrypted sensitive information on the client side

When building an application that uses a client-server model, storing sensitive information, such as user credentials, on the client side may result in its unauthorized disclosure if the client is vulnerable to attack.

For web applications, the most common mitigation to this problem is to provide the client with a cookie and store the sensitive information on the server. Cookies are created by a web server and are stored for a period of time on the client. When the client reconnects to the server, it provides the cookie, which identifies the client to the server, and the server then provides the sensitive information.

Cookies do not protect sensitive information against cross-site scripting (XSS) attacks. An attacker who is able to obtain a cookie either through an XSS attack or directly by attacking the client can obtain the sensitive information from the server using the cookie. This risk is timeboxed if the server invalidates the session after a limited time has elapsed, such as 15 minutes.

A cookie is typically a short string. If it contains sensitive information, that information should be encrypted. Sensitive information includes passwords, credit card numbers, social security numbers, and any other personally identifiable information about the user. For more details about managing passwords, see MSC62-J. Store passwords using a hash function. For more information about securing the memory that holds sensitive information, see MSC59-J. Limit the lifetime of sensitive data.

Noncompliant Code Example

In this noncompliant code example, the login servlet stores the user name and password in the cookie to identify the user for subsequent requests:

```
protected void doPost(HttpServletRequest request,
    HttpServletResponse response) {

  // Validate input (omitted)

  String username = request.getParameter("username");
  char[] password = request.getParameter("password").toCharArray();
  boolean rememberMe = Boolean.valueOf(request.getParameter("rememberme"));

  LoginService loginService = new LoginServiceImpl();

  if (rememberMe) {
    if (request.getCookies()[0] != null &&
      request.getCookies()[0].getValue() != null) {
      String[] value = request.getCookies()[0].getValue().split(";");

      if (!loginService.isUserValid(value[0], value[1].toCharArray())) {
        // Set error and return
      } else {
        // Forward to welcome page
      }
    } else {
        boolean validated = loginService.isUserValid(username, password);

        if (validated) {
          Cookie loginCookie = new Cookie("rememberme", username
                             + ";" + new String(password));
          response.addCookie(loginCookie);
          // ... Forward to welcome page
        } else {
          // Set error and return
        }
     }
   } else {
     // No remember-me functionality selected
     // Proceed with regular authentication;
     // if it fails set error and return
   }

  Arrays.fill(password, ' ');
}
```

However, the attempt to implement the remember-me functionality is insecure because an attacker with access to the client machine can obtain this information directly on the client. This code also violates MSC62-J. Store passwords using a hash function. The client may also have transmitted the password in clear unless it encrypted the password or uses HTTPS.

Compliant Solution (Session)

This compliant solution implements the remember-me functionality by storing the user name and a secure random string in the cookie. It also maintains state in the session using `HttpSession`.

```
```

```
protected void doPost(HttpServletRequest request,
    HttpServletResponse response) {

  // Validate input (omitted)

  String username = request.getParameter("username");
  char[] password = request.getParameter("password").toCharArray();
  boolean rememberMe = Boolean.valueOf(request.getParameter("rememberme"));
  LoginService loginService = new LoginServiceImpl();
  boolean validated = false;
  if (rememberMe) {
    if (request.getCookies()[0] != null &&
        request.getCookies()[0].getValue() != null) {

      String[] value = request.getCookies()[0].getValue().split(";");

      if (value.length != 2) {
        // Set error and return
      }

      if (!loginService.mappingExists(value[0], value[1])) {
        // (username, random) pair is checked
        // Set error and return
      }
    } else {
      validated = loginService.isUserValid(username, password);

      if (!validated) {
        // Set error and return
      }
    }

    String newRandom = loginService.getRandomString();
    // Reset the random every time
    loginService.mapUserForRememberMe(username, newRandom);
    HttpSession session = request.getSession();
    session.invalidate();
    session = request.getSession(true);
    // Set session timeout to 15 minutes
    session.setMaxInactiveInterval(60 * 15);
    // Store user attribute and a random attribute in session scope
    session.setAttribute("user", loginService.getUsername());
    Cookie loginCookie =
      new Cookie("rememberme", username + ";" + newRandom);
    loginCookie.setHttpOnly(true);
    loginCookie.setSecure(true);
    response.addCookie(loginCookie);
    // ... Forward to welcome page
  } else {
    // No remember-me functionality selected
    // ... Authenticate using isUserValid() and if failed, set error
  }
```

```
    Arrays.fill(password, ' ');
  }
```

The server maintains a mapping between user names and secure random strings. When a user selects “Remember me,” the `doPost()` method checks whether the supplied cookie contains a valid user name and random string pair. If the mapping contains a matching pair, the server authenticates the user and forwards him or her to the welcome page. If not, the server returns an error to the client. If the user selects “Remember me” but the client fails to supply a valid cookie, the server requires the user to authenticate using his or her credentials. If the authentication is successful, the server issues a new cookie with remember-me characteristics.

This solution avoids session-fixation attacks by invalidating the current session and creating a new session. It also reduces the window during which an attacker could perform a session-hijacking attack by setting the session timeout to 15 minutes between client accesses.

Applicability

Storing unencrypted sensitive information on the client makes this information available to anyone who can attack the client.

Bibliography

[Oracle 2011c]	Package `javax.servlet.http`
[OWASP 2009]	Session Fixation in Java
[OWASP 2011]	Cross-Site Scripting (XSS)
[W3C 2003]	The World Wide Web Security FAQ