IDS51-J. Properly encode or escape output

Proper input sanitization can prevent insertion of malicious data into a subsystem such as a database. However, different subsystems require different types of sanitization. Fortunately, it is usually obvious which subsystems will eventually receive which inputs, and consequently what type of sanitization is required.

Several subsystems exist for the purpose of outputting data. An HTML renderer is one common subsystem for displaying output. Data sent to an output subsystem may appear to originate from a trusted source. However, it is dangerous to assume that output sanitization is unnecessary because such data may indirectly originate from an *un*trusted source and may include malicious content. Failure to properly sanitize data passed to an output subsystem can allow several types of attacks. For example, HTML renderers are prone to HTML injection and cross-site scripting (XSS) attacks [OWASP 2011]. Output sanitization to prevent such attacks is as vital as input sanitization.

As with input validation, data should be normalized before sanitizing it for malicious characters. Properly encode all output characters other than those known to be safe to avoid vulnerabilities caused by data that bypasses validation. See IDS01-J. Normalize strings before validating them for more information.

Noncompliant Code Example

This noncompliant code example uses the model-view-controller (MVC) concept of the Java EE–based Spring Framework to display data to the user without encoding or escaping it. Because the data is sent to a web browser, the code is subject to both HTML injection and XSS attacks.

```
@RequestMapping("/getnotifications.htm")
public ModelAndView getNotifications(
  HttpServletRequest request, HttpServletResponse response) {
  ModelAndView mv = new ModelAndView();
  try {
    UserInfo userDetails = getUserInfo();
    List<Map<String,Object>> list = new ArrayList<Map<String, Object>>();
    List<Notification> notificationList =
        NotificationService.getNotificationsForUserId(userDetails.getPersonId());

    for (Notification notification: notificationList) {
      Map<String,Object> map = new HashMap<String, Object>();
      map.put("id", notification.getId());
      map.put("message", notification.getMessage());
      list.add(map);
    }

     mv.addObject("Notifications", list);
  }
    catch(Throwable t) {
    // Log to file and handle
  }

  return mv;
}
```

Compliant Solution

This compliant solution defines a `ValidateOutput` class that normalizes the output to a known character set, performs output sanitization using a whitelist, and encodes any unspecified data values to enforce a double-checking mechanism. Note that the required whitelisting patterns can vary according to the specific needs of different fields [OWASP 2013].

```
public class ValidateOutput {
  // Allows only alphanumeric characters and spaces
  private static final Pattern pattern = Pattern.compile("^[a-zA-Z0-9\\s]{0,20}$");

  // Validates and encodes the input field based on a whitelist
  public String validate(String name, String input) throws ValidationException {
    String canonical = normalize(input);

    if (!pattern.matcher(canonical).matches()) {
      throw new ValidationException("Improper format in " + name + " field");
    }

    // Performs output encoding for nonvalid characters
    canonical = HTMLEntityEncode(canonical);
    return canonical;
  }

  // Normalizes to known instances
  private String normalize(String input) {
    String canonical =
      java.text.Normalizer.normalize(input, Normalizer.Form.NFKC);
    return canonical;
  }

  // Encodes nonvalid data
  private static String HTMLEntityEncode(String input) {
    StringBuffer sb = new StringBuffer();

    for (int i = 0; i < input.length(); i++) {
      char ch = input.charAt(i);
      if (Character.isLetterOrDigit(ch) || Character.isWhitespace(ch)) {
        sb.append(ch);
      } else {
        sb.append("&#" + (int)ch + ";");
      }
    }
    return sb.toString();
  }
}

// ...

@RequestMapping("/getnotifications.htm")
public ModelAndView getNotifications(HttpServletRequest request, HttpServletResponse response) {
  ValidateOutput vo = new ValidateOutput();

  ModelAndView mv = new ModelAndView();
  try {
    UserInfo userDetails = getUserInfo();
    List<Map<String,Object>> list = new ArrayList<Map<String,Object>>();
    List<Notification> notificationList =
        NotificationService.getNotificationsForUserId(userDetails.getPersonId());

    for (Notification notification: notificationList) {
      Map<String,Object> map = new HashMap<String,Object>();
      map.put("id", vo.validate("id", notification.getId()));
      map.put("message", vo.validate("message", notification.getMessage()));
      list.add(map);
    }

     mv.addObject("Notifications", list);
  }
  catch(Throwable t) {
    // Log to file and handle
  }

  return mv;
}
```

Output encoding and escaping is mandatory when accepting dangerous characters such as double quotes and angle braces. Even when input is whitelisted to disallow such characters, output escaping is recommended because it provides a second level of defense. Note that the exact escape sequence can vary depending on where the output is embedded. For example, untrusted output may occur in an HTML value attribute, CSS, URL, or script; output encoding routine will be different in each case. It is also impossible to securely use untrusted data in some contexts. Consult the OWASP XSS (Cross-Site Scripting) Prevention Cheat Sheet for more information on preventing XSS attacks.

Noncompliant Code Example

This noncompliant code example takes a user input query string and build a URL. Because the URL is not properly encoded, the URL returned may not be valid if it contains non-URL-safe characters, as per RFC 1738.

```
String buildUrl(String q) {
  String url = "https://example.com?query=" + q;

  return url;
}
```

For example, if the user supplies the input string "<#catgifs>", the `url` returned is `"https://example.com?query=<#catgifs>"` which is not a valid URL.

Compliant Solution (Java 8)

Use `java.util.Base64` to encode and decode data when transferring binary data over mediums that only allow printable characters like URLs, filenames, and MIME.

```
String buildEncodedUrl(String q) {
    String encodedUrl = "https://example.com?query=" + Base64.getUrlEncoder().encodeToString(q.getBytes());

    return encodedUrl;
}
```

If the user supplies the input string "<#catgifs>", the `url` returned is `"https://example.com?query=%3C%23catgifs%3E"` which is a valid URL.

Applicability

Failure to encode or escape output before it is displayed or passed across a trust boundary can result in the execution of arbitrary code.

Automated Detection

Tool	Version	Checker	Description
The Checker Framework	2.1.3	**Tainting Checker**	Trust and security errors (see Chapter 8)

Related Vulnerabilities

The Apache GERONIMO-1474 vulnerability, reported in January 2006, allowed attackers to submit URLs containing JavaScript. The Web Access Log Viewer failed to sanitize the data it forwarded to the administrator console, thereby enabling a classic XSS attack.

Bibliography

[OWASP 2011]	Cross-site Scripting (XSS)
[OWASP 2014]	How to Add Validation Logic to HttpServletRequest XSS (Cross-Site Scripting) Prevention Cheat Sheet