Input Validation and Data Sanitization

The Myth of Trust

Software programs often contain multiple components that act as subsystems wherein each component operates in one or more trusted domains. For example, one component may have access to the file system but lack access to the network, while another component has access to the network but lacks access to the file system. *Distrustful decomposition and privilege separation* [Dougherty 2009] are examples of secure design patterns that reduce the amount of code that runs with special privileges by designing the system using mutually untrusting components.

Although software components can obey policies that allow them to transmit data across trust boundaries, they cannot specify the level of trust given to any component. The deployer of the application must define the trust boundaries with the help of a systemwide security policy. A security auditor can use that definition to determine whether the software adequately supports the security objectives of the application.

A Java program can contain both internally developed and third-party code. Java was designed to allow the execution of untrusted code; consequently, third-party code can operate in its own trusted domain. The public API of such third-party code can be considered to be a trust boundary. Data that crosses a trust boundary should be validated unless the code that produces this data provides guarantees of validity. A subscriber or client may omit validation when the data flowing into its trust boundary is appropriate for use as is. In all other cases, inbound data must be validated.

Injection Attacks

Data received by a component from a source outside the component’s trust boundary can be malicious and can result in an injection attack, as shown in the scenario in Figure 1.

![Figure 1. Injection attack](image)

Programs must take steps to ensure that data received across a trust boundary is appropriate and not malicious. These steps can include the following:

**Validation:** Validation is the process of ensuring that input data falls within the expected domain of valid program input. This requires that inputs conform to type and numeric range requirements as well as to input invariants for the class or subsystem.

**Sanitization:** In many cases, the data is passed directly to a component in a different trusted domain. Data sanitization is the process of ensuring that data conforms to the requirements of the subsystem to which it is passed. Sanitization also involves ensuring that data conforms to security-related requirements regarding leaking or exposure of sensitive data when output across a trust boundary. Sanitization may include the elimination of unwanted characters from the input by means of removing, replacing, encoding, or escaping the characters. Sanitization may occur following input (input sanitization) or before the data is passed across a trust boundary (output sanitization). Data sanitization and input validation may coexist and complement each other. Many command interpreters and parsers provide their own sanitization and validation methods. When available, their use is preferred over custom sanitization techniques because custom-developed sanitization can often neglect special cases or hidden complexities in the parser. Another problem with custom sanitization code is that it may not be adequately maintained when new capabilities are added to the command interpreter or parser software.

**Canonicalization and Normalization:** Canonicalization is the process of lossless reduction of the input to its equivalent simplest known form. Normalization is the process of lossy conversion of input data to the simplest known (and anticipated) form. Canonicalization and normalization must occur before validation to prevent attackers from exploiting the validation routine to strip away invalid characters and, as a result, constructing an invalid (and potentially malicious) character sequence. See *FIO16-J. Canonicalize path names before validating them* for more information. Normalization should be performed only on fully assembled user input. Never normalize partial input or combine normalized input with nonnormalized input.

Complex subsystems that accept string data that specify commands or instructions are a special concern. String data passed to these components may contain special characters that can trigger commands or actions, resulting in a software vulnerability.

These are examples of components that can interpret commands or instructions:

- Operating system command interpreter (see *IDS07-J. Sanitize untrusted data passed to the Runtime.exec() method*)
- A data repository with a SQL-compliant interface (see *IDS00-J. Prevent SQL Injection*)
- XML parser (see *IDS16-J. Prevent XML Injection and IDS17-J. XML External Entity Attacks*)
- Regular expression engines (see *IDS08-J. Sanitize untrusted data included in a regular expression*)
- Formatted output methods (see *IDS06-J. Exclude unsanitized user input from format strings*)
- XPath evaluators
- Lightweight Directory Access Protocol (LDAP) directory service
- Script engines
<table>
<thead>
<tr>
<th>Bibliography</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Seacord 2015]</td>
</tr>
</tbody>
</table>