MSC10-C. Character encoding: UTF8-related issues

UTF-8 is a variable-width encoding for Unicode. UTF-8 uses 1 to 4 bytes per character, depending on the Unicode symbol. UTF-8 has the following properties:

- The classical US-ASCII characters (0 to 0x7f) encode as themselves, so files and strings that are encoded with ASCII values have the same encoding under both ASCII and UTF-8.
- It is easy to convert between UTF-8 and UCS-2 and UCS-4 fixed-width representations of characters.
- The lexicographic sorting order of UCS-4 strings is preserved.
- All possible 2^21 UCS codes can be encoded using UTF-8.

Generally, programs should validate UTF-8 data before performing other checks. The following table lists the well-formed UTF-8 byte sequences.

<table>
<thead>
<tr>
<th>Bits of code point</th>
<th>First code point</th>
<th>Last code point</th>
<th>Bytes in sequence</th>
<th>Byte 1</th>
<th>Byte 2</th>
<th>Byte 3</th>
<th>Byte 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>U+0000</td>
<td>U+007F</td>
<td>1</td>
<td>0xxxxxx</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>U+0080</td>
<td>U+07FF</td>
<td>2</td>
<td>110xxxxx 10xxxxxx</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>U+0800</td>
<td>U+FFFF</td>
<td>3</td>
<td>1110xxxx 10xxxxxx 10xxxxxx</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>U+10000</td>
<td>U+1FFFFF</td>
<td>4</td>
<td>11110xxxx 10xxxxxx 10xxxxxx 10xxxxxx</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Although UTF-8 originated from the Plan 9 developers [Pike 1993], Plan 9's own support covers only the low 16-bit range. In general, many "Unicode" systems support only the low 16-bit range, not the full 21-bit ISO 10646 code space [ISO/IEC 10646:2012].

Security-Related Issues

According to RFC 2279: UTF-8, a transformation format of ISO 10646 [Yergeau 1998], Implementors of UTF-8 need to consider the security aspects of how they handle invalid UTF-8 sequences. It is conceivable that, in some circumstances, an attacker would be able to exploit an incautious UTF-8 parser by sending it an octet sequence that is not permitted by the UTF-8 syntax.

A particularly subtle form of this attack can be carried out against a parser that performs security-critical validity checks against the UTF-8 encoded form of its input, but interprets certain invalid octet sequences as characters. For example, a parser might prohibit the null character when encoded as the single-octet sequence 00, but allow the invalid two-octet sequence C0 80 and interpret it as a null character. Another example might be a parser which prohibits the octet sequence 2F 2E 2E 2F ("../"), yet permits the invalid octet sequence 2F C0 AE 2E 2F.

Following are more specific recommendations.

Accept Only the Shortest Form

Only the "shortest" form of UTF-8 should be permitted. Naive decoders might accept encodings that are longer than necessary, allowing for potentially dangerous input to have multiple representations. For example,

1. Process A performs security checks but does not check for nonshortest UTF-8 forms.
2. Process B accepts the byte sequence from process A and transforms it into UTF-16 while interpreting possible nonshortest forms.
3. The UTF-16 text may contain characters that should have been filtered out by process A and can potentially be dangerous. These "nonshortest" UTF-8 attacks have been used to bypass security validations in high-profile products, such as Microsoft's IIS Web server.

Corrigendum #1: UTF-8 Shortest Form to the Unicode Standard [Unicode 2006] describes modifications made to version 3.0 of the Unicode Standard to forbid the interpretation of the nonshortest forms.

Handling Invalid Inputs

UTF-8 decoders have no uniformly defined behavior upon encountering an invalid input. Following are several ways a UTF-8 decoder might behave in the event of an invalid byte sequence. Note that implementing these behaviors requires careful security considerations.

1. Substitute for the replacement character "U+FFFD" or the wildcard character such as "?” when U+FFFD is not available.
2. Ignore the bytes (for example, delete the invalid byte before the validation process; see "Unicode Technical Report #36, 3.5 Deletion of Code Points" for more information).
3. Interpret the bytes according to a different character encoding (often the ISO-8859-1 character map; other encoding, such as Shift_JIS, is known to trigger self-XSS, and so is potentially dangerous).
4. Fail to notice but decode as if the bytes were some similar bit of UTF-8.
5. Stop decoding and report an error.

The following function from John Viega’s “Protecting Sensitive Data in Memory” [Viega 2003] detects invalid character sequences in a string but does not reject nonminimal forms. It returns 1 if the string is composed only of legitimate sequences; otherwise, it returns 0.
Broken Surrogates

Encoding of individual or out-of-order surrogate halves should not be permitted. Broken surrogates are invalid in Unicode and introduce ambiguity when they appear in Unicode data. Broken surrogates are often signs of bad data transmission. They can also indicate internal bugs in an application or intentional efforts to find security vulnerabilities.

Risk Assessment

Failing to properly handle UTF8-encoded data can result in a data integrity violation or denial-of-service attack.

<table>
<thead>
<tr>
<th>Recommendation</th>
<th>Severity</th>
<th>Likelihood</th>
<th>Remediation Cost</th>
<th>Priority</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSC10-C</td>
<td>Medium</td>
<td>Unlikely</td>
<td>High</td>
<td>P2</td>
<td>L3</td>
</tr>
</tbody>
</table>

Automated Detection

<table>
<thead>
<tr>
<th>Tool</th>
<th>Version</th>
<th>Checker</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDRA tool suite</td>
<td>9.7.1</td>
<td>176 S, 376 S</td>
<td>Partially implemented</td>
</tr>
</tbody>
</table>

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

Related Guidelines

<table>
<thead>
<tr>
<th>SEI CERT C++ Coding Standard</th>
<th>VOID MSC10-CPP. Character encoding: UTF8-related issues</th>
</tr>
</thead>
<tbody>
<tr>
<td>MITRE CWE</td>
<td>CWE-176, Failure to handle Unicode encoding</td>
</tr>
<tr>
<td></td>
<td>CWE-116, Improper encoding or escaping of output</td>
</tr>
</tbody>
</table>

Bibliography

[ISO/IEC 10646-2012]  
[Kuhn 2006] UTF-8 and Unicode FAQ for Unix/Linux  
[Pike 1993] “Hello World”  
[Unicode 2006]  
[Viega 2003] Section 3.12, “Detecting Illegal UTF-8 Characters”  
[Wheeler 2003] Secure Programmer: Call Components Safely  
[Yergeau 1998] RFC 2279