IDS33-PL. Sanitize untrusted data passed across a trust boundary

Many programs accept untrusted data originating from arbitrary users, network connections, and other untrusted sources and then pass the (modified or unmodified) data across a trust boundary to a different trusted domain. Frequently the data is in the form of a string with some internal syntactic structure, which the subsystem must parse. Such data must be sanitized both because the subsystem may be unprepared to handle the malformed input and because unsanitized input may include an injection attack.

In particular, programs must sanitize all string data that is passed to command interpreters or parsers so that the resulting string is innocuous in the context in which it is parsed or interpreted.

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.

Noncompliant Code Example (XSS)

This noncompliant code example demonstrates an XSS exploit. This code uses the CGI module to display a web form and is adopted from an example from the CGI.pm documentation. The form queries the user for a name and displays the resulting name on the page when the user clicks Submit.

```perl
use CGI qw(:standard);

print header;
print start_html('A Simple Example'),
   h1('A Simple Example'),
   start_form,
   "What's your name? ",textfield('name'),
   submit,
   end_form,
   hr;
if (param()) {
   print "Your name is: ",em(param('name')),
   hr;
}
print end_html;
```

When fed a benign name, such as Larry, this script works well enough:
But this code will happily parse image tags, HTML markup, JavaScript, or any other commands an attacker may wish to send. The following picture demonstrates a remote image being loaded into the page on the request of the attacker:

A Simple Example

What's your name? Larry

Your name is: Larry
In this case, the trust boundary exists between the untrusted data and the CGI script, whereas the trusted domain is the web browser—or rather the HTML parsing and rendering engine within the web browser.

More details about sanitization of this code example can be found in IDS01-PL. Use taint mode while being aware of its limitations.

Noncompliant Code Example (Taint Mode)

Using taint mode will not detect or prevent XSS. Taint mode does not prevent tainted data from being printed to standard output.

Compliant Solution (XSS)

To prevent injection of HTML, JavaScript, or malicious images, any untrusted input must be sanitized. This compliant solution sanitizes the input using the escapeHTML() subroutine from the CGI library.

```perl
# rest of code unchanged
if (param()) {
    print "Your name is: ", em(escapeHTML(param('name'))),
    hr;
} print end_html;
```

When fed the malicious image tag demonstrated previously, the escapeHTML() subroutine sanitizes characters that might be misinterpreted by a web browser, causing the name to appear exactly as it was entered:
SQL Injection

A SQL injection vulnerability arises when the original SQL query can be altered to form an altogether different query. Execution of this altered query may result in information leaks or data modification. The primary means of preventing SQL injection are sanitizing and validating untrusted input and parameterizing queries.

Suppose a database contains user names and passwords used to authenticate users of the system. A SQL command to authenticate a user might take the form:

```
SELECT * FROM Users WHERE userid='<USERID>' AND password='<PASSWORD>'
```

If it returns any records, the user ID and password are valid.

However, if an attacker can substitute arbitrary strings for `<USERID>` and `<PASSWORD>`, he can perform a SQL injection by using the following string for `<USERID>`:

```
validuser' OR '1'='1
```

When injected into the command, the command becomes

```
SELECT * FROM Users WHERE userid='validuser' OR '1'='1' AND password=<PASSWORD>
```

If `validuser` is a valid user name, this SELECT statement selects the `validuser` record in the table. The password is never checked because `userid='validuser'` is true; consequently, the items after the `OR` are not tested. As long as the components after the `OR` generate a syntactically correct SQL expression, the attacker is granted the access of `validuser`.

Likewise, an attacker could supply a string for `<PASSWORD>` such as:
This would yield the following command:

```
SELECT * FROM Users WHERE userid='' AND password='' OR '1'='1'
```

This time, the '1'='1' tautology disables both user ID and password validation, and the attacker is falsely logged in without a correct login ID or password.

**Noncompliant Code Example (SQL Injection)**

This noncompliant code example shows Perl DBI code to authenticate a user to a system. The program connects to a database, prompts the user for a user ID and password, and hashes the password. Unfortunately, this code example permits a SQL injection attack because the string passed to prepare accepts unsanitized input arguments. The attack scenario outlined previously would work as described.

```perl
use DBI;
my $dbfile = "users.db";
my $dbh = DBI->connect("dbi:SQLite:dbname=$dbfile","",""
    or die "Couldn't connect to database: " . DBI->errstr;
sub hash {
    # hash the password
}
print "Enter your id: ";
my $userid = <STDIN>;
chomp $userid;
print "Enter your password: ";
my $password = <STDIN>;
chomp $password;
my $hashed_password = hash( $password);
my $sth = $dbh->prepare("SELECT * FROM Users WHERE userid = '$userid' AND password = '$hashed_password'"
    or die "Couldn't prepare statement: " . $dbh->errstr;
$sth->execute()
    or die "Couldn't execute statement: " . $sth->errstr;
if (my @data = $sth->fetchrow_array()) {
    my $username = $data[1];
    my $id = $data[2];
    print "Access granted to user: $username ($userid)\n";
}
if ($sth->rows == 0) {
    print "Invalid username / password. Access denied\n";
}
$sth->finish;
$dbh->disconnect;
```

**Compliant Solution (Taint Mode)**

One way to find potential injection points quickly is to use Perl's taint mode.

```perl
# ... beginning of code
my $dbh = DBI->connect("dbi:SQLite:dbname=$dbfile","",""
    or die "Couldn't connect to database: " . DBI->errstr;
$dbh->{TaintIn} = 1;
# ... rest of code
```
Perl will refuse to permit tainted data from entering the database via the prepare() method call. It will immediately exit with an error message:

Note that not only must the program be run in taint mode, but the TaintIn attribute must be set on the connection handle, enabling taint checks to be run on the database.

Compliant Solution (Prepared Statement)

Fortunately, Perl's DBI library provides an API for building SQL commands that sanitize untrusted data. The prepare() method properly escapes input strings, preventing SQL injection when used properly. This is an example of component-based sanitization.

```perl
# ... beginning of code
my $sth = $dbh->prepare("SELECT * FROM Users WHERE userid = ? AND password = ?")
  or die "Couldn't prepare statement: ", $dbh->errstr;
$sth->execute($userid, $hashed_password)
  or die "Couldn't execute statement: ", $sth->errstr;

# ... rest of code
```

Risk Assessment

<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>IDS33-PL</td>
<td>High</td>
<td>Likely</td>
<td>High</td>
<td>P9</td>
<td>L2</td>
</tr>
</tbody>
</table>

Automated Detection

<table>
<thead>
<tr>
<th>Tool</th>
<th>Diagnostic</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Taint mode</td>
<td>Insecure dependency in parameter 'id' of DBI::db=.* method call</td>
<td>Catches SQL injection. Requires TaintIn attribute.</td>
</tr>
</tbody>
</table>

Related Guidelines

- SEI CERT C Coding Standard
- SEI CERT C++ Coding Standard
- CERT Oracle Secure Coding Standard for Java

Bibliography

- [Birznieks 1998]: Birznieks, Gunther, "CGI/Perl Taint Mode FAQ, Version 1.0, June 3, 1998"
- [CGI 2005]: CGI.pm: A Perl5 CGI Library, Function-Oriented vs Object-Oriented Use
- [CPAN]: Bunce, Tim, DBI
- [CPAN]: Stosberg, Mark, CGI
- [Lester 2006]: Lester, Andy, "Perl's taint mode to the rescue," O'Reilly OULamp.com, November 17, 2006
- [VU#246409]: Input validation error in quikstore.cgi allows attackers to execute commands
- [VU#282403]: AdCycle does not adequately validate user input thereby allowing for SQL injection
- [Wall 2011]: perlsec