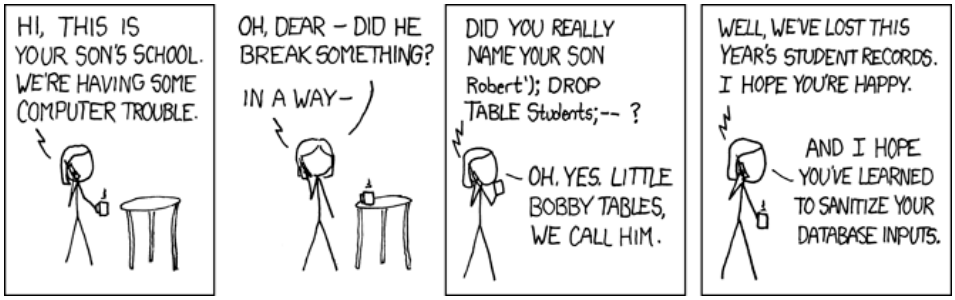
Practice Secure Programming

Information Compiled by Jane Sowerby, Information Security and Privacy
March 28, 2013



From **How to Stop a Hacker - Don't Trust User Input**
- http://www.experimentgarden.com/2009/07/how-to-stop-hacker-dont-trust-user.html

This hilarious comic strip by xkcd illustrates one of the most important rules of system security: never trust user input. All user input to a program or website should be sanitized by checking and processing it to make sure that it will not do damage to the system.

There are several reasons why this is necessary. First of all, programmers have to deal with user ignorance that may lead to user input breaking the system. Second, programmers have to stop deliberate attacks designed to break the system.

How Input Can Be Used to Hack a System

In the xkcd comic strip the computer savvy mother hacked the school database by naming her son a MYSQL statement that causes the database to destroy the student records.

The syntax for storing a new student in the database is probably something similar to:

```
INSERT INTO Students (StudentName) values('Sample Name');
```

MySQL statements end with a semicolon, so when a semicolon is encountered it assumes that is the end of the statement, and it executes the next tokens as a separate statement. So this is what happened when the school secretary entered little Bobby's evil name into the database:

```
INSERT INTO Students (StudentName) values('Robert'); DROP TABLE Students;');
```

Suddenly instead of having one command which stores the student name in the database the MYSQL parser sees three commands:

```
INSERT INTO Students (StudentName) values('Robert');
DROP TABLE Students;
');
```

The first statement stores a new student named Robert. The second command deletes all the student records. The last is just the extra quote and closing parentheses, so it generates an error. However, the damage has already been done and student records are gone.

In this case the input was deliberately designed to break the system. However, consider the case of someone who is registering a new profile on a website. On the server side the programmer has designed the code so that user information is stored in a database. One of the fields is the user's handle.

In this case the user is a pubescent young boy who thinks too much of himself. As his username he enters "Every girl's dream." After completing the rest of the form he hits submit and waits for his new profile. Needless to say, he is mystified when it gives him strange error messages, or perhaps never responds. What happened?

On the server side the code probably did something like this:

```
INSERT INTO user_database (username) values('Every girl's dream');
```

When the MYSQL parser looks at this command it sees that the code wants it to store a new record in the user database. Then it looks at the values. First it sees a string:

```
'Every girl'
```

Then it finds a bunch of garble:

```
s dream'
```

Then it reaches the final closing parentheses. The MYSQL database is going to return an error message and will fail to store the new record because it sees this as a syntax error.

This is a prime example of how user ignorance can also break a system.

So how does the programmer stop this input? There are several simple ways.

Train Human Employees

In the first example, from the xkcd comic strip, the mother's sneaky hacking ploy could have been thwarted by simple employee training. The mother doubtlessly wrote her son's name on a physical paper form, and the data was entered into the computer later by a secretary. Humans are very good at pattern recognition and spotting unusual trends, making them the perfect filter. If the secretary had been trained to show unusual names to the local IT professional or to someone else who knew a lot about computers this hacking attempt could have been stopped before it even started.

In all cases where humans enter data off of physical paper forms they should be trained to determine what type of data they should enter and what they should not enter. "Robert'); DROP TABLE Students;" is obviously not a normal name, so it should be pretty obvious that there is something wrong with it.

Force Form Input to Meet Rigorous Requirements

In cases where information is entered into a database automatically there will be no human checking each entry. Therefore a really easy way to prevent hacking is to limit input to letters or numbers. For example, the name field should only accept letters, and the telephone field should only accept numbers.

There are two ways to do this. Either remove unwanted characters on the server side and then store the new username, or return an error message saying something such as "Only letters may be used in the name field."

With the first technique, when the secretary entered "Robert'); DROP TABLE Students;" into the database the computer would have preprocessed it, removing the quotes, semicolons, and parentheses. When the computer was done all that would be left was "Robert DROP TABLE Students" which, although an unusual name to say the least, would not destroy the database at all because there is no way for it to be interpreted as a command.

Using the second technique the secretary would have gotten an error message saying "Only letters may be entered for the student's name." The secretary would have called Mom up and said "Sorry but we can't enter your son's name in the database. Does he have a nickname or some other name we can use?"

Needless to say neither situation is optimal when you consider that a multicultural environment means that people may have unusual character as part of their name: perhaps dashes or even quotes in a native Hawaiian name. Also, it would be nice to allow people to enter unusually punctuated ASCII art names as forum handles and nicknames.

Another aspect to consider is longer inputs such as the input for entering comments on this post. Users will want to be able to enter quotation marks, semicolons, and other characters that are normally used in text. If you didn't allow these then input would be severely limited.

Clearly forcing input to meet rigorous requirements will not work in these cases. It may work for certain simple fields, but not for longer, more detailed input.

Encode All Special Characters

This simple technique is by far the easiest and most flexible way of stopping users from entering dangerous input. The concept is simple. In HTML each character has a special character code that is used to represent it. For example, the character code for the quotation mark is:

"

If you replace all characters that aren't letters or numbers with a corresponding codes then they will have no effect on the database. There are different ways of doing this. For example, in the xkcd comic the school's system should have **escaped all quotes**. This is a simple technique that replaces all double quotes with:

\"

and all single quotes with

\'

These two replacements are simple codes that MYSQL recognizes as indicating a quote that is part of the text, not the end of the text string. So the code should have turned Bobby's name into:

Robert\'); DROP TABLE Students;

The name would have stored just fine without breaking the end of the string and interpreting the second half as a command.

Most languages, including my favorite server language PHP, have commands for automatically doing character encoding. It is usually best to use both quotation escaping and HTML entity encoding. There is a simple reason for this.

Consider the following scenario. An evil hacker finds Experiment Garden and decides that he is going to destroy it completely, or at the very least sabotage it. So he posts the following comment on the blog:

'); DROP TABLE POSTS; DROP TABLE COMMENTS;

```
<script type="text/javascript">
  var links = document.getElementsByTagName("a");
  for (var i=0; i<links.length; i++) {
     links[i].href = "http://www.youtube.com/watch?v=oHg5SJYRHA0";
  }
</script>
```

This evil comment has two parts. First it includes a MYSQL injection statement that attempts to delete the entire blog if the server side code isn't smart enough to escape quotes. The second part is a little more interesting. If the quotes are escaped, but special characters are not encoded, then when this comment is displayed in the browser, the browser will interpret the second part and run it as JavaScript. The evil JavaScript code modifies every link on the page so that it points to.... well you can probably guess what it points to.

That's right, its an evil rickroll.

However, if HTML entity encoding is turned on the code will instead look like:

```
&lt;script type="text/javascript"&gt;
  var links = document.getElementsByTagName("a");
  for (var i=0; i&lt;links.length; i++) {
     links[i].href = "http://www.youtube.com/watch?v=oHg5SJYRHA0";
  }
&lt;/script&gt;
```

This would not be recognized by the browser as code, so it would not be executed. Instead it would be displayed as is on the screen in the comments.

So to summarize, both types of encoding are needed to prevent malicious or just plain ignorant user input from breaking the system.

Conclusion

This brief discussion covers only a few of the basic ways to stop code injection in user input, and stop special characters in input from breaking the code. However, by combining these techniques depending on the type of input and risk level of your application you can plug one of the major security holes that hackers like to exploit.

Related Notes and Websites about Secure Programming

The cartoon is an example of SQL Injection. Another major programming issue is Cross Site Scripting (XSS).

HTML/Script injection is a popular subject, commonly termed "Cross-Site Scripting", or "XSS". XSS refers to an injection flaw whereby user input to a web script or something along such lines is placed into the output HTML, without being checked for HTML code or scripting.

In our case last week the input was not placed into output HTMS, but transferred to output email messages.

I like this overarching statement

Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use a whitelist of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does.

When performing input validation, consider all potentially relevant properties, including length, type of input, the full range of acceptable values, missing or extra inputs, syntax, consistency across related fields, and conformance to business rules. As an example of business rule logic, "boat" may be syntactically valid because it only contains alphanumeric characters, but it is not valid if the input is only expected to contain colors such as "red" or "blue."

Do not rely exclusively on looking for malicious or malformed inputs (i.e., do not rely on a blacklist). A blacklist is likely to miss at least one undesirable input, especially if the code's environment changes. This can give attackers enough room to bypass the intended validation. However, blacklists can be useful for detecting potential attacks or determining which inputs are so malformed that they should be rejected outright.

Here are some good web sources of secure programming practices and after these are two excerpts with details on sanitizing input.

Bobby Tables: A guide to preventing SQL injection
http://bobby-tables.com/

Good Power Point on SQL Injection
http://www.slideshare.net/billkarwin/sql-injection-myths-and-fallacies

This is a free Oracle Tutuorial on Defending Against SQL Injection (not XSS however)
http://download.oracle.com/oll/tutorials/SQLInjection/index.htm
It has a lesson on filtering input with DBMS-ASSERT functions

This site – CWE Common Weakness Enumeration -- has some good information
http://cwe.mitre.org/data/definitions/20.html

How to Protect Your Website from Hacking Attacks
http://www.wikihow.com/Protect-Your-Website-from-Hacking-Attacks

From OWASP -- *Open Web Application Security Project* (*OWASP*)

https://www.owasp.org/index.php/OWASP_Proactive_Controls

3: Validation

One of the most important ways to build a secure web application is to limit what input a user is allowed to submit to your web application. Limiting user input is a technique called “input validation”. Input validation is most often built into web applications in server-side code using regular expressions. Regular expressions are a kind of code syntax that can help tell if a string matches a certain pattern. Secure programmers can use regular expressions to help define what good user input should look like.
There are two types on input validation: “White list validation and blacklist validation”. White list validation seeks to define what good input should look like. Any input that does not meet this “good input” definition should be rejected. “Black list” validation seeks to detect known attacks and only reject those attacks or bad characters. “Black list” validation is much more difficult to build into your applications effectively and is not often suggested when initially building a secure web application. The following examples will focus on whitelist validation examples.
When a user first registers for an account with our web application, some of the first things we ask a user to provide for us would be a username, password and email address. If this input came from a malicious user, the input could contain dangerous attacks that could harm our web application! One of the ways we can make attacking this web application more difficult is to use regular expressions to validate the user input from this form. Also, it's critical to treat all client side data as untrusted. (we need to expand on this)
Let’s start with the following regular expression for the username.
^[a-z0-9_]{3,16}$
This regular expression input validation whitelist of good characters only allows lowercase letters, numbers and the underscore character. The size of the username is also being limited to 3-16 characters in this example.
Here is an example regular expression for the password field.
^(?=.*[a-z])(?=.*[A-Z]) (?=.*\d) (?=.*[@#$%]).{10,64}$
This regular expression ensures that a password is 10 to 64 characters in length and includes a uppercase letter, a lowercase letter, a number and a special character (one or more uses of @, #, $, or %).
Here is an example regular expression for an email address (per the HTML5 specification http://www.w3.org/TR/html5/forms.html#valid-e-mail-address).
^[a-zA-Z0-9.!#$%&'*+/=?^_`{|}~-]+@[a-zA-Z0-9-]+(?:\.[a-zA-Z0-9-]+)*$
There are special cases for validation where regular expressions are not enough. If your application handles markup -- untrusted input that is supposed to contain HTML -- it can be very difficult to validate. Encoding is also difficult, since it would break all the tags that are supposed to be in the input. Therefore, you need a library that can parse and clean HTML formatted text such as the OWASP Java HTML Sanitizer. A regular expression is not the right tool to parse and sanitize untrusted HTML.
Here we illustrate one of the unfortunate truisms about input validation: input validation does not necessarily make untrusted input “safe” especially when dealing with “open text input” where complete sentences from users need to be accepted.

https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet

Defense Option 3: Escaping All User Supplied Input

This third technique is to escape user input before putting it in a query. If you are concerned that rewriting your dynamic queries as prepared statements or stored procedures might break your application or adversely affect performance, then this might be the best approach for you. However, this methodology is frail compared to using parameterized queries and we cannot guarantee it will prevent all SQL Injection in all situations. This technique should only be used, with caution, to retrofit legacy code in a cost effective way. Applications built from scratch, or applications requiring low risk tolerance should be built or re-written using parameterized queries.

This technique works like this. Each DBMS supports one or more character escaping schemes specific to certain kinds of queries. If you then escape all user supplied input using the proper escaping scheme for the database you are using, the DBMS will not confuse that input with SQL code written by the developer, thus avoiding any possible SQL injection vulnerabilities.
Full details on ESAPI are available here on OWASP.
The javadoc for ESAPI is available here at its Google Code repository.
You can also directly browse the source at Google, which is frequently helpful if the javadoc isn't perfectly clear.

To find the javadoc specifically for the database encoders, click on the 'Codec' class on the left hand side. There are lots of Codecs implemented. The two Database specific codecs are OracleCodec, and MySQLCodec.

Just click on their names in the 'All Known Implementing Classes:' at the top of the Interface Codec page.

At this time, ESAPI currently has database encoders for:
Oracle
MySQL (Both ANSI and native modes are supported)

Database encoders for:
SQL Server
PostgreSQL

Are forthcoming. If your database encoder is missing, please let us know.

Database Specific Escaping Details

If you want to build your own escaping routines, here are the escaping details for each of the databases that we have developed ESAPI Encoders for:

Oracle Escaping

This information is based on the Oracle Escape character information found here:
http://www.orafaq.com/wiki/SQL_FAQ#How_does_one_escape_special_characters_when_writing_SQL_queries.3F

Escaping Dynamic Queries

To use an ESAPI database codec is pretty simple. An Oracle example looks something like:
```
ESAPI.encoder().encodeForSQL( new OracleCodec(), queryparam );
```

So, if you had an existing Dynamic query being generated in your code that was going to Oracle that looked like this:
```
String query = "SELECT user_id FROM user_data WHERE user_name = '" + req.getParameter("userID")
+ "' and user_password = '" + req.getParameter("pwd") +"'";
try {
    Statement statement = connection.createStatement( … );
    ResultSet results = statement.executeQuery( query );
}
```

You would rewrite the first line to look like this:
```
Codec ORACLE_CODEC = new OracleCodec();
String query = "SELECT user_id FROM user_data WHERE user_name = '" +
  ESAPI.encoder().encodeForSQL( ORACLE_CODEC, req.getParameter("userID")) + "' and user_password = '"
  + ESAPI.encoder().encodeForSQL( ORACLE_CODEC, req.getParameter("pwd")) +"'";
```

And it would now be safe from SQL injection, regardless of the input supplied.

For maximum code readability, you could also construct your own OracleEncoder.
```
Encoder oe = new OracleEncoder();
String query = "SELECT user_id FROM user_data WHERE user_name = '"
  + oe.encode( req.getParameter("userID")) + "' and user_password = '"
  + oe.encode( req.getParameter("pwd")) +"'";
```

With this type of solution, all your developers would have to do is wrap each user supplied parameter being passed in into an ESAPI.encoder().encodeForOracle( ) call or whatever you named it, and you would be done.

Turn off character replacement

Use SET DEFINE OFF or SET SCAN OFF to ensure that automatic character replacement is turned off. If this character replacement is turned on, the & character will be treated like a SQLPlus variable prefix that could allow an attacker to retrieve private data.

See http://download.oracle.com/docs/cd/B19306_01/server.102/b14357/ch12040.htm#i2698854 and http://stackoverflow.com/questions/152837/how-to-insert-a-string-which-contains-an for more information

Escaping Wildcard characters in Like Clauses

The LIKE keyword allows for text scanning searches. In Oracle, the underscore '_' character matches only one character, while the ampersand '%' is used to match zero or more occurrences of any characters. These characters must be escaped in LIKE clause criteria. For example:

```
SELECT name FROM emp
WHERE id LIKE '%/_%' ESCAPE '/';

SELECT name FROM emp
WHERE id LIKE '%\%%' ESCAPE '\';
```

Oracle 10g escaping

An alternative for Oracle 10g and later is to place { and } around the string to escape the entire string. However, you have to be careful that there isn't a } character already in the string. You must search for these and if there is one, then you must replace it with }}. Otherwise that character will end the escaping early, and may introduce a vulnerability.

MySQL Escaping

MySQL supports two escaping modes:
1. ANSI_QUOTES SQL mode, and a mode with this off, which we call
2. MySQL mode.

ANSI SQL mode: Simply encode all ' (single tick) characters with '' (two single ticks)

MySQL mode, do the following:

```
 NUL (0x00) --> \0  [This is a zero, not the letter O]
 BS  (0x08) --> \b
 TAB (0x09) --> \t
 LF  (0x0a) --> \n
 CR  (0x0d) --> \r
 SUB (0x1a) --> \Z
 "   (0x22) --> \"
 %   (0x25) --> \%
 '   (0x27) --> \'
 \   (0x5c) --> \\
 _   (0x5f) --> \_
 all other non-alphanumeric characters with ASCII values less than 256  --> \c
 where 'c' is the original non-alphanumeric character.
```

This information is based on the MySQL Escape character information found here:
http://mirror.yandex.ru/mirrors/ftp.mysql.com/doc/refman/5.0/en/string-syntax.html

SQL Server Escaping

We have not implemented the SQL Server escaping routine yet, but the following has good pointers to articles describing how to prevent SQL injection attacks on SQL server
http://blogs.msdn.com/raulga/archive/2007/01/04/dynamic-sql-sql-injection.aspx

DB2 Escaping

This information is based on DB2 WebQuery special characters found here: https://www-304.ibm.com/support/docview.wss?uid=nas14488c61e3223e8a78625744f00782983 as well as some information from Oracle's JDBC DB2 driver found here: http://docs.oracle.com/cd/E12840_01/wls/docs103/jdbc_drivers/sqlescape.html

Information in regards to differences between several DB2 Universal drivers can be found here: http://publib.boulder.ibm.com/infocenter/db2luw/v8/index.jsp?topic=/com.ibm.db2.udb.doc/ad/rjvjcsqc.htm

(from OWASP Secure Architecture and Coding Power Point)

Input Validation

Strings should have a maximum length
Overlong strings lead to buffer overflows
Bad math with long strings leads to buffer overflows

Always set a character set
ISO 8859-1 or UTF-8

ASVS: Requires input to made into the simplest possible form before any work is performed
Without ESAPI: UNBELIEVABLY, STUPIDLY, REALLY HARD™
With ESAPI: Relatively easy

```
public class MyInputValidation {

    public String realFileNameFromInput(String inputFileName) {
        ArrayList list = new ArrayList();
        list.add( new WindowsCodec() );
        Encoder encoder = new DefaultEncoder( list );

        // Only allow singly-encoded strings
        String clean = encoder.canonicalize( request.getParameter( "input" ), false);

        return clean;
    }
}
```

ASVS: Requires positive validation over all other mechanisms
Positive validation
Sanitization
Negative validation
No validation

ASVS: Requires a single input validation mechanism

All data passing through a trust boundary should be validated
All callers to use the centralized input validation routines
Hook framework to input validation routines

ASVS: Requires server side input validation

Ajax / JS / RIA input validation is allowed...
Just have to repeat that on the server!

ASVS: Positive input validation routines should reject bad input
DO NOT sanitize bad input!
Assumes you know more than the attacker does

Log all failures
ASVS: Requires access control failures to be logged
Tip: Allow configurable logging for success and other events

```
public class MyInputValidation {

    public boolean validateFileName(String context, String filename) {
        boolean validated = false;

        try {
            if ( ESAPI.Validator().isValidFileName(context, filename, false) ) {
                validated = true;
            }
        }
        catch (Exception ex) {
            // Log result
            validated = false;
            logger.audit("Input validation failure - bad filename", ...);
        }

        return validated;
    }
}
```

Output Encoding

Output encoding is often considered the “fix” for XSS

NO! It’s the last chance before you’re hosed
Consider it defense in depth if input validation fails

Do not store pre-encoded
Tip: Just in time output encode for only the output mechanism (i.e. SMTP, RSS, SOAP, etc)
It may be tempting but:

- Do not store encoded or escaped data!
- HTML Encoded data is not safe for XML or LDAP or ...

ASVS: Requires output encoding mechanism to be enforced server side
Not much point in encoding on the client - can be trivially bypassed

Oracle techniques

Use DBMS_ASSERT package to sanitize user inputs

Don’t use implicit types conversions...
...and don’t rely on defaults
Application logic unintended change besides SQL injections