Writing Secure ASP Scripts

Chris Anley [chris@ngssoftware.com]

An NGSSoftware Insight Security Research (NISR) Publication

http://www.ngssoftware.com

Table of contents

Introduction

This paper briefly describes several common classes of coding error generally encountered when auditing web applications running on the Active Server Pages (ASP) platform.

The paper is broken down into three broad sections, each of which addresses several common coding problems. The following is a list of the common errors that are discussed in this document, divided into three broad categories. The remainder of the document deals with each of these problems in turn. Any ASP code samples assume that the default language is VBScript, but all of the points apply equally to JavaScript. Equally, all occurrences of the SQL language assume that Microsoft SQL Server is being used as the back – end database.

Input validation

Insufficient validation of fields in SQL queries
Email handling problems
Parent path problems

Predictability and secure management of state

Poor randomness
Predictable session identifiers
Session state manipulation bugs
Poor credential management

Source maintenance problems

Improper source and data file maintenance
Debug code
Hardcoded credentials
Error messages/error handling

Classes of problem

Input validation

Input validation errors are probably the most common form of problem encountered when auditing ASP applications. Three of the most common classes of input validation error are:

Insufficient validation of fields in SQL queries
Email handling problems
Parent path problems

We will address each of these classes of error in turn.

Insufficient validation of fields in SQL queries

ASP applications frequently communicate with back – end databases. Most databases use the Structured Query Language (SQL) to manage and manipulate data. SQL is a textual language that has a rich syntax; it is effectively a programming language in its own right. Here are some examples of SQL statements:

```
select * from users where username = 'fred'

insert into users (username, password) values ('fred', 'sesame')

drop table users
```

Typically, an ASP application will create the SQL query string dynamically, based on data supplied by the user in a query string or form, like this:

```
strSQL = "select * from users where username = '" &_
   request.form("username") & "' and password = '" &_
   request.form("password") & "'"
```

The "request.form" statements refer to the strings that the user types into a HTML form in their web browser. Unfortunately, there is nothing to prevent the user from typing anything they want into the form. This can lead to the user being able to submit arbitrary SQL queries. Here is an example of how this might happen.

Let's consider an example 'login' form processing script. This script handles an attempt by a user to 'log in' to the ASP application using a username and password. The portion of the code that verified whether the user has used a valid username and password might look something like this:

```
username = request.form("username")
password = request.form("password")

strSQL = "select * from users where username = '" & username &_
         "' and password = '" & password & "'"

set objRS = objDB.execute(strSQL)

if ( objRS.EOF ) then
      response.redirect("LoginFailed.asp")
end if

…(the user is now valid, do some stuff)
```

What we are doing here is selecting all fields in a row from the users table, where the username and password match the supplied data. If the returned recordset is empty (EOF) we redirect the user to a 'login failed' page, otherwise, we continue, with the user data that we retrieved from the database.

This has several flaws. These flaws are easily illustrated if we consider what happens when the user inputs a 'normal' username and password:

```
Username: fred
Password: sesame
```

…the string 'strSQL' becomes

```
select * from users where username = 'fred' and password = 'sesame'
```

…which runs fine. If there is a user 'fred' with a password 'sesame', we will get fred's row, and all will be fine.

However, consider the following scenario:

```
Username: fred'--
Password: whatever
```

…the string 'strSQL' becomes

```
select * from users where username = 'fred'--' and password =
'whatever'
```

Given the above logic, this will log us in as 'fred', without knowing fred's password. The '--' character sequence represents a single - line comment in Transact-SQL, the SQL language used by Microsoft SQL Server, so it stops executing the SQL statement when it reaches it. The result? We get fred's row returned to us, without knowing fred's password.

Further, more dangerous uses of the technique are possible. Consider this:

```
Username: fred' drop table users--
Password: whatever
```

The SQL parser terminates the first Transact-SQL query after fred'. If more SQL statements follow it, they are run as well. In this case, the table 'users' will be deleted, effectively denying everyone the ability to log into the database.

Using this technique, it is possible to run whatever SQL query an attacker wishes. Using the rich capabilities of SQL server and it's range of built in stored procedures and extended stored procedures, it is possible to use the database server as a bridgehead into the back-end network that supports the ASP application. Examples of things an attacker might do:

- use error messages returned by the server to obtain information in fields in the database, or information about the structure of the database (credit card information, user credentials such as passwords)
- use the xp_cmdshell extended stored procedure to run commands as the SQL server user, on the database server
- use the sp_OACreate, sp_OAMethod and sp_OAGetProperty system stored procedures to create Ole Automation (ActiveX) applications that can do everything an ASP script can do

Traditional wisdom has it that if an ASP application uses stored procedures in the database, that SQL injection is not possible. This is a half-truth, and it depends on the manner in which the stored procedure is called from the ASP script.

Good general rules are:

- If the ASP script creates a SQL query string that is submitted to the server, it is vulnerable to SQL injection, *even if* it uses stored procedures
- If the ASP script uses a procedure object that wraps the assignment of parameters to a stored procedure (such as the ADO command object, used with the Parameters collection) then it is generally safe.

To illustrate the stored procedure query injection point, execute the following SQL string:

```
sp_who '1' select * from sysobjects
or
sp_who '1'; select * from sysobjects
```

Either way, the appended query is still run.

There are several ways to fix this problem:

1. Disallow input which is known to be bad
2. Allow only input which is known to be good
3. Attempt to 'escape' delimiting characters

Each scheme has its merits.

If we disallow input which is known to be bad, we would be looking for specific characters or words in the input that we know are dangerous. The code might look like this:

```
function validate( input )

      bad_strings = Array( "'", "select", "union", "insert", "--" )

      for each i in bad_strings

      if ( InStr( input, i ) <> 0 ) then
            validate = false
            exit function
      end if

      next

      validate = true

end function

username = request.form("username")
password = request.form("password")

if ( not validate(username) or not validate(password) ) then
      response.redirect("invalid_input.asp")
end if
```

The disadvantage of this scheme is that we don't necessarily know what bad data looks like. On the basis of this brief tutorial, we could probably guess, but we might pick the wrong items for 'bad_strings'. Also, we would have problems if we want to allow usernames like
O'Brien

The second method, allowing only input which is known to be good, is better, but a little more demanding to code. The idea with this method is that we define several data types, and permit only certain character combinations in each. For example, we might have a 'name', 'password', 'integer' and 'session_id' type. The validate function might look like this:

```
function validate( input, datatype )

      good_name_chars =_
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'- "
      good_password_chars =_
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"
      good_number_chars = "0123456789"
      good_sessionid_chars = "ABCDEF0123456789"

      validate = true

      select case datatype

      case "name"
            for i = 1 to len( input )
                  c = mid( input, i, 1 )

                  if ( InStr( good_name_chars, c ) = 0 ) then
                        validate = false
                  end if

            next
      case "password"
            for i = 1 to len( input )
                  c = mid( input, i, 1 )

                  if ( InStr( good_password_chars, c ) = 0 ) then
                        validate = false
                  end if

            next
      case "number"
            for i = 1 to len( input )
                  c = mid( input, i, 1 )

                  if ( InStr( good_number_chars, c ) = 0 ) then
                        validate = false
                  end if

            next
      case "sessionid"
            for i = 1 to len( input )
                  c = mid( input, i, 1 )

                  if ( InStr( good_sessionid_chars, c ) = 0 ) then
                        validate = false
```

```
                end if

            next
        case else
            validate = false
        end select

end function
```

An important point here is that we allow the character ' in a name, along with '-' and spaces. This would allow an attacker to perform the

```
Username: fred'--
```

attack outlined above, though it would restrict what an attacker could do in terms of arbitrary queries.

The final solution (which is not necessarily better than either of the above) is to attempt to 'escape' delimiting characters. An example function, that 'doubles up' single quote characters, looks like this:

```
function escape( input )
    input = replace(input, "'", "''")
    escape = input
end function
```

The problem with this solution is that not all terms in an SQL query are delimited. For example, if we had a numeric userid, our SQL query might look like this:

```
select * from users where userid = 24 and password = 'whatever'
```

Obviously, the attacker can simply input SQL statements into the field directly, without needing to 'escape' from a data string.

The best method depends upon the circumstances. A very secure method is to use both the first and second methods, in combination. First we verify that the input contains only valid characters, then we look for strings that we know to be specifically bad. The method chosen will, however, depend on the requirements associated with user input to the application.

For more information on this kind of bug, see the other papers I've written on the subject, at:
http://www.ngssoftware.com/papers/advanced_sql_injection.pdf
and
http://www.ngssoftware.com/papers/more_advanced_sql_injection.pdf

Email handling problems

A similarly nasty problem occurs when handling data that is to be passed into an email. The problem arises because most objects that wrap sending an SMTP message send SMTP commands directly to the server, without validating the user's input. For

example, the cdonts.newmail object permits various of it's properties to contain carriage-return and linefeed characters, and periods ('.').

SMTP (Simple Message Transfer Protocol) is a text - based network protocol that is the method that most ASP email objects use to send messages. SMTP has a fairly simple syntax. For example, the commands to send a message might look like this:
('>' indicates client-to-server communication, '<' indicates server-to-client)

```
(client opens connection to server on TCP port 25)
< 220 mail.example.net ESMTP Thu, 3 Jan 2002 18:16:35 +0000
> HELO client
< 250 mail.example.net Hello client [10.1.1.1]
> MAIL FROM: <joe.blow@example.net>
< 250 <joe.blow@example.net> is syntactically correct
> RCPT TO: <foo.bar@example.net>
< 250 <foo.bar@example.net> is syntactically correct
> DATA
< 354 Enter message, ending with "." on a line by itself
> Subject: Test mail
> Date: Thu, 3 Jan 2002 18:14:45 -0000
>
> Got it.
> :o)
>
>      -joe
> .
>
< 250 OK id=12345
> QUIT
< 221 mail.example.net closing connection
(server closes connection)
```

In a similar manner to SQL, you can see that if the user were to enter, (say) in the body of the message, a '.' on a line by itself, the server would terminate the message. In a similar manner to the SQL ';' character, we can then begin a new message, which can have a totally arbitrary sender and recipient, and can contain whatever data the attacker wishes, including potentially harmful MIME encoded file attachments.

The resolution to this problem is to ensure that the user cannot submit a '.' on a line by itself at any point in the message, including the 'from' and 'to' fields. Which of the validation methods is used would depend on circumstances.

David Litchfield of NGSSoftware has written a more lengthy explanation of this type of problem, which can be found at
http://www.ngssoftware.com/papers/aspmail.pdf

Parent path problems

Another exceptionally common problem related to input validation is the 'parent path' problem. We can group in this category all problems having to do with unexpected characters in filenames.

Suppose we have a code snippet that adds some text to a log file corresponding to a user session. The code might look like this:

```
filename = "c:\logs\" & username
set fs = createobject( "scripting.filesystemobject" )
set f = fs.OpenTextFile( filename, 8, True)
f.writeline message
f.close
set f = nothing
set fs = nothing
```

Say we add the text of each web request made by each user to their own log file.

This is fine if the username is, say, 'fred'. But what if the username is
`..\..\..\..\..\..\inetpub\wwwroot\runcmd.asp`

An ASP script will be created in the web root, with contents that the user can control part of. That part might look something like this:

```
<% set o = server.createobject("wscript.shell")
o.run( request.querystring("cmd") ) %>
```

The attacker has created a script that will run an arbitrary command line on the web server. In versions of Microsoft Internet Information Server prior to version 5, this command would run in the context of the local 'system' account, and could thus add local administrative users, or perform any other action that the system itself can perform, such as reading the SAM.

The underlying problem here is the same as the SQL and SMTP problems; the data submitted by the user is interpreted in some context where special characters or character sequences have meaning.

Wherever an application creates filenames, any part of which is controlled by the user, the application must validate the user input very carefully.

Here are some example abuses of poor filename validation:
Reading the source code of any .asp file on the server (revealing ODBC connection strings, for example)
Creating a script file of some sort that can then be run in subsequent HTTP requests.
Overwriting the contents of a critical system file or script with garbage, thereby preventing audit (web server log), or destroying specified parts of the web application (such as an administrative part of the website; simple overwrite the login.asp script with garbage, or null data)

Predictability and secure management of state

Web applications typically requite some way of maintaining the 'state' of a user's interaction with the application. This can manifest itself in a number of ways, and if handled poorly, is open to abuse by attackers.

Poor randomness

Applications generally have some requirement for randomness. The application may have to generate its own session identifiers, for example, or it might have to create some kind of random password.

Most 'random' number generators build into languages and libraries are based upon arithmetic 'pseudo' random number generators. A problem frequently exhibited by these generators is that they issue repeating sequences. Another common problem is that of seeding with predictable data, such as a tick count, an IP address or hostname.

Some applications pick data that is not at all random for supposedly 'random' numbers. For example, it is common to use the current time, measured in seconds, as the 'secret', combine it with (say) the userid and then pass it through a hashing function such as MD5 or SHA1. The problem here is that there are only 3600 seconds in an hour. It is within the realms of possibility that an attacker can generate 3600 requests in the time that a user's session exists on the server, given that most web servers can comfortably handle several thousand requests per second. A crucial point is that hashing the data doesn't help, or more precisely, hashing a value doesn't change the amount of entropy it contains. If there is only a small amount of 'randomness' input, only a small amount of 'randomness' will be output. Competent attackers have access to session id generation code for a variety of platforms, and there are only a small number of cryptographically 'strong' hashing functions in existence. In essence, hashing doesn't win you much.

An attacker is very likely to be able to guess the time at the server, even to millisecond resolution; often it is contained in web responses. The ICMP timestamp request is another method of obtaining the time at the server.

Predictable session identifiers

Some applications use monotonically increasing session identifiers (i.e. the first id is 1, the second 2 and so on). Some applications use the primary key of a table in a database; again, this is extremely weak and quite easy to guess, given a single valid identifier.

The reason why predictable session identifiers must be avoided is that knowledge of the session id typically grants access to the application. Once a user has passed the 'authentication' phase of an application, the session identifier is the only way the application has of verifying who is who.

Consequently, if an attacker can guess the session id of a user who is currently authenticated with the application, they will be able to interact with the application as though they were that user.

Session state manipulation bugs

This is a subtle class of problem that exploits the manipulation of the state of an application.

Most ASP applications maintain some kind of state using the built-in 'session' object. For example, an application might do this (hopefully after the 'authentication' process has been successfully concluded):

```
session("username") = request.form("username")
```

The application would then use the `session("username")` variable whenever it wishes to retrieve the user's username.

The state of the 'session' object is maintained by ASP session id cookies.

The problem with maintaining a complex session state is that whenever the code manipulates the state, it might be laying itself open to malicious changes by an attacker.

Applications will use 'include' files to structure scripts. Every .asp script might include a 'header', 'footer' or 'menu', or similar. These 'include' scripts are generally executed in the context of the main 'body' scripts, but there is nothing stopping an attacker from requesting the script in isolation. If the 'include' script manipulates the session state, an attacker can take advantage of this.

Typical attacks using this technique might be:

Becoming another user by manipulating the 'authentication' states.
Obtaining free 'downloads' by bypassing state transitions that the application goes through to 'verify' valid users.
Obtaining administrative access to an application

As an example of this class of error, here is a login page drawn from sample code: (some lines are wrapped)

```
<%@LANGUAGE = "VBSCRIPT"%>

<%session("username") = request("username")%>

<%
strErr = ""
if request("action") = "login" then

      Set Conn = Server.CreateObject("ADODB.Connection")
      Conn.Open "DRIVER={Microsoft Access Driver (*.mdb)}; DBQ=" &_
      Server.MapPath("forum.mdb")

      SQL = "select username, password from users where username = '"
            & request("username") & "'"
      set RSuser = Conn.execute (SQL)

      if RSuser("password") = request("password") then

            Response.Redirect("list.asp")
      else
            strErr = strErr & "Invalid password"
      end if

elseif request("action") = "register" then
```

```
      if request("username") = "" then
            strErr = strErr & "Username is a required field"
      end if
      if request("password") = "" then
            strErr = strErr & "Password is a required field"
      end if
      if strErr = "" then
            if request("password") <> request("confirm") then
                  strErr = strErr & "Password must match the
confirmation"
            else

                  Set Conn = Server.CreateObject("ADODB.Connection")
                  Conn.Open "DRIVER={Microsoft Access Driver
(*.mdb)}; DBQ=" &_
                  Server.MapPath("forum.mdb")

                  SQL = "select username from users where username =
'" & request("username") & "'"
                  set RSuser = Conn.execute (SQL)

                  if not RSuser.BOF and not RSuser.EOF then
                        strErr = strErr & "This username already
exists"
                  else
                        SQL = "insert into users (username, password,
date_created, date_modified)  VALUES ('" & request("username") & "',
'" & request("password") & "', '" & now & "', '" & now & "')"
                        Conn.execute (SQL)
                  end if
            end if
      end if


else
      strErr = strErr & "Unrecognised Action, please resubmit"
end if
%>

<%
if strErr <> "" then
%>
<html>
<head>
      <title>Error</title>
</head>
<body>
<table border="0" cellpadding="0" cellspacing="0" width="400">
<tr>
      <td><b>The following errors were detected:</b></td>
</tr>
<tr>
      <td><%=strErr%></td>
</tr>
<tr>
      <td>Please go <a href="default.asp">back</a> and correct
them.</td>
</tr>
</table>
</body>
```

```
</html>

<%else%>
<html>
<head>
      <title>Registration</title>
</head>

<body>

<table border="0" cellpadding="0" cellspacing="0" width="400">
<tr>
      <td><b>Registration completed</b></td>
      <td>Click <a href="list.asp">here</a> to enter the forum.</td>
</tr>
</table>
</body>
</html>
<%end if%>
<%Conn.close%>
```

The point to note here is the second line of code:

```
<%session("username") = request("username")%>
```

The page then goes on to perform various operations based on the value of the 'action' field in the request. The point to note is that if the attacker specifies a username and an invalid action, the session("username") variable remains set to whatever the attacker chose. In this sample site, it turns out that the session("username") field is all that is used for authentication.

The resolution to these issues is to take care in the use of session() variables; ensure that if an error is encountered, that the modifications to the session state are erased, or otherwise rendered 'safe'.

Poor credential management

Another issue related to state and session maintenance is that of credential management.

Problems in this area include plaintext usernames and passwords in querystrings, form 'post' data or cookies. The fewer times credentials are transmitted, the better.

It is generally bad practice to have any form of credential (username, password, session id) in the querystring of a HTTP request. This is because of the likelihood that an attacker will gain the ability to read the web server's log file. Cookies and POST data are typically not held in web server logs, and thus are a slightly safer place to transfer credentials.

Another common issue is that of 'incremental verification'. This occurs where an application requests multiple items of data in order to authenticate, verifying each item in sequence and returning a different error message if each item is incorrect.

For example, input of the username 'wibble123eqr' might result in the error message "error: the user does not exist"
Whereas the username 'admin' might result in the error message
"error: the password was incorrect"

Other examples might be 'business unit', 'username', 'password', or even 'domain name', 'username', 'password'.

Many applications allow an attacker to verify whether a given userid exists by simply registering for an id themselves, then guessing names based on the format of the userid they were given. For example, if the attacker 'John Smith' receives the username 'jsmith', it is obvious what username he should guess if he suspects that his arch-rival 'John Doe' also uses the system. The resolution to this problem is to return the minimum of error information when incorrect credentials are entered.

Source maintenance problems

This section discusses problems that arise through the development process itself. It is often the case that management insistence on aggressive deadlines leads to the general quality of the codebase deteriorating. The cause of this problem is typically the pressure that the development team is under; the best resolution to the problem is to make the quality of the codebase, in terms of error handling, file locations, and credential handling, as important a part of the development process as the code itself. If an application is lacking in this area, management must allow the development team the time to 'tidy up'.

Improper source and data file maintenance

If an attacker can access the source code to an application, they will have an advantage in attacking it. Backup files are often created when files are modified in the source tree. Typically these files might be named .bak, .old or similar. Occasionally the file is copied in windows explorer, which results in a 'copy of' filename.

As an example, let us say that the file 'login.asp' has been patched several times, in a hurried manner. The following files are quite likely to be in the same directory:

```
login.asp
login.bak
login.old
login.asp.bak
login.asp.old
login.asp.2
Copy of login.asp
Copy (2) of login.asp
```

This has quite a serious impact on the security of the application. For example, since the 'login.bak' file will not be interpreted by Active Server Pages; it's source code will be returned in 'raw' form. This will give an attacker knowledge of the exact authentication scheme in use. It might include an ODBC connection string, or other credentials.

Old copies of the file with a '.asp' extension are in some ways a more serious issue. If an old version of the file behaved differently, the attacker may be able to use this to his advantage. For example, if a previous version of the script was vulnerable to SQL injection (or one of the other data validation issues outlined in this document) the attacker will be able to use those issues.

More problems can be caused by version control systems. Improperly used, CVS can leave files lying around the website that are as good as a directory listing - the './CVS/Entries' file contains a list of all files and directories in the current directory. Also, if a file has been placed 'in conflict' by having two people make mutually exclusive changes, a 'conflict' file is left in the directory, named '.#<filename>.<version>' and the version is normally easy to determine.

It is fairly obvious that an attacker, even with no knowledge of the system, will be able to obtain a fair amount of source code by just guessing filenames corresponding to old versions of existing scripts. Occasionally whole directories will be copied in windows explorer, resulting in a 'copy of' filename corresponding to an existing directory on the server.

Good source control, and the time to 'tidy up' are the best resolution to this problem.

Another security problem in this general area relates to 'including' files in ASP scripts. Developers will often call the 'include' files 'database.inc' (for example) - the problem with this is that the '.inc' extension is not interpreted by ASP, and so anyone requesting

http://www.example.com/scripts/database.inc

(say) gets the source code of 'database.inc'. It is unfortunately common practice for folks to leave hardcoded credentials in 'include' files of this kind, which can lead to an almost immediate compromise of the database server or even the network. As soon as the attacker can see source code, they can begin to pick apart the structure of the application. It is best to ensure that include files have a '.asp' extension, so that the source code remains hidden.

Debug code

As part of the development process, developers often add code for use when debugging the application. This frequently allows rich error information to be returned to the developer, or possibly allows the full details of a backend transaction to be viewed. In the worst cases, backdoor or 'test' authentication methods are included, which allow the developers to bypass aspects of the authentication process in order to more easily test other functionality.

A 'blind' attackers strategies in this case will be to submit likely 'debug' flags with most requests. Examples might be

```
debug=true
debug=1
trace=on
log=on
```

and so on. Of course, if any of these mechanisms exist in the source code, the attacker may perhaps learn of them by other means; a source code disclosure issue from a 'parent path' problem in another script, or possibly a flaw in the web server software itself (IIS has historically had many issues of this kind and more are regularly discovered).

Hardcoded credentials

An issue related to code quality is the practice of hardcoding usernames and passwords in the source code. This is more prevalent in 'scripted' environments like ASP than it is in 'compiled' environments like Visual Basic or C++, perhaps because of the perception that a script is easier to change.

Whatever the reason, hardcoded credentials are bad practice. Long experience of compromising ASP applications has taught that the best place to store credentials on a windows server is in the registry. Preferably, these credentials should be stored in an encrypted or even 'obfuscated' form.

Error messages/error handling

One of the most useful forms of feedback a web server gives an attacker is its error messages. As a trivial example of this, it is frequently possible to determine whether a directory exists on the web server by requesting a number of 'likely' known directories - containing sample scripts, or just common names - and observing the return codes. Often (almost exclusively) the server will return a '404 file not found' error for directories which do not exist, and some other error - 'directory listing not allowed', 'access denied' or 'authentication required' if the directory does exist.

Research papers have been written on methods of obtaining the structure of a SQL server database using SQL injection and ASP error messages. It is possible to determine the value held in any field with a known name, the names of all fields of tables, and the names of parameters for stored procedures fairly easily, using only error messages.

When the 'filesystem' object is used in a script, it will return a different error if it is directed to a nonexistent path, than when it is directed to a nonexistent file within an existing directory. Using this fact alone, it is possible to narrow down the version of the web server, operating system and database software to the level of individual patches and service packs.

The resolution to this problem is to ensure that the errors returned by the production application contain the absolute minimum of information.