

Twenty Don'ts for ASP Developers

Mark Burnett 2002-07-03

Twenty Don'ts for ASP Developers

by Mark Burnett

last updated July 3, 2002

Firewalls block hackers from directly connecting to your network shares. Windows administrators keep their systems up-to-date with the latest software patches to thwart worms such as Nimda and Code Red. And user passwords are stronger than ever. But are we secure yet? While the situation is much better than it was just a couple years ago, many companies are still quite vulnerable to a number of attacks. Blocking ports and installing patches has not stopped hackers, it has just forced them to find new ways to break in. And chances are, the first place they are going to look is your Web application.

The problem is that while you may have a team of experts to secure your network, you are still dependent on your developers to secure your Web application. Are they properly trained to take on the most sophisticated hackers in the world? Are they at least good enough to defend themselves from a script kiddie who just read a tutorial on SQL injection? Many companies are now realizing that their code is not as secure as it should be.

This article will offer twenty tips for ASP programmers. These are not tips on how to secure a Web application, they are twenty things that ASP developers should avoid doing in order to develop secure Web applications. Unfortunately they address twenty common mistakes that we see over and over again on Web applications.

1. Do Not Write Unfiltered User Input to the Web Client

One of the most common programming mistakes, writing unfiltered content to the Web client opens up a Web application for cross-site scripting attacks. Cross-site scripting, often referred to as XSS, allows someone to exploit the trust of your Web site to take advantage of other Web visitors. By not filtering user input, you allow others to inject HTML, JavaScript, Java Applets, VBScript, etc. into your Web application.

You should never use the `Request` object on a `Response.Write` statement, as in the following code:

```
Response.Write "You have entered " & Request("UserInput")
```

The proper way to handle user input is to assign it to a variable, sanitize it to ensure that it does not contain HTML-related tags, and then write to output. At a minimum, pass the string through the `Server.HtmlEncode` function. In higher risk situations, you may wish to take more aggressive steps by only allowing certain alphanumeric characters.

2. Do Not Trust Client or Session Variables.

Although normally considered safe, if session variables are set based on user input, they should also be sanitized before they can be trusted. In fact, anything dirtied by user input absolutely must be sanitized before using.

3. Do Not Forget to Specify a Specific Character Set

A simple technique to mitigate exposure to cross-site scripting attacks is to explicitly declare which character set should be used on your HTML page. If a character set is not explicitly defined, any character encoding may be used. The reason for this is that due to the many variations of character encoding, it becomes very difficult to filter out unwanted characters.

The following HTML code explicitly sets the character set to ISO-8859-1:

```
<META http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
```

4. Do Not Access Files Based On User Input Without Checking the Path.

Its not uncommon to see a URL such as this:

```
http://www.example.net/article.asp?file=new.htm
```

Whenever I see something like this, I wonder what would happen if I modified the URL to something like one of the following examples:

```
http://www.example.net/article.asp?file=/global.asa  
http://www.example.net/article.asp?file=../../../boot.ini  
http://www.example.net/article.asp?file=LPT1  
http://www.example.net/article.asp?file=%2e%2e\global.asa
```

As you can see, there is much to consider when accessing a file based on user input. For this, rather than trying to consider all the possible ways someone could access a file, I prefer to use a

technique called "round-tripping". To do this, you first pass the file path to a command, API, or component you trust, and then check the path it returns to make sure it is valid.

For example, consider the following code:

```
<%  
Set fso = CreateObject("Scripting.FileSystemObject")  
On Error Resume Next  
Set f = fso.GetFile(request("file"))  
If err then  
    Response.Write "Error"  
Else  
    Response.Write f.Path  
End If  
>
```

In this code, rather than directly using the user input, we first pass it to the `FileSystemObject` using the `GetFile` method and then read the `FileSystemObject`'s interpretation of the path by checking the `Path` property. The `FileSystemObject` will return a normalized, absolute path as it sees it, regardless of encoding or relative references. Now this path is considered much more trusted and we can now write code to be sure the file being accessed is in an authorized directory.

5. Do Not Send SQL Queries Without Filtering User Input

SQL Injection is the process of exploiting a Web application, usually through a Web form, tricking it to pass malicious SQL statements to the database server. With Microsoft's SQL Server, this is often done by entering a single quote in the Web form, followed by the correctly formed SQL. For example, consider the following code to authenticate a user from a Web form:

```
strSQL="SELECT * FROM Customers WHERE Username = '" &  
Request("Username") & "' & Password = '" & strPassword & "'"
```

Now this code is quite typical of what you would see in a Web application. However, consider what would happen if the user entered the following:

```
Username: Test  
Password: ' or True
```

When the `strSQL` string is built, the resulting SQL will be as follows:

```
strSQL="SELECT * FROM Customers WHERE Username = 'ValidUser' & Password  
= ' ' or True --'"
```

This statement will essentially return the `ValidUser` customer, regardless of what password is set for that account; the `True` condition will always cause the `WHERE` condition to match. Note that the double dash ("--") at the end of the statement acts as a comment character, ignoring the remaining characters.

To sanitize form input for sending to a database, always be sure to escape the single quote by searching and replacing it with two single quotes. This will cause the database to send the quote string as a literal character rather than interpreting it as the closing of a string. Be aware, however, that since numeric input does not require quotes, this technique will not be effective. In the case of numeric input, simply check that the form input is indeed numeric.

6. Do Not Trust Database Content

Like session variables, programmers have a tendency to trust the data coming from their own database. However, if that data at one time came from user input, it cannot be trusted. Database input must be filtered before using just like any user input.

7. Do Not Store Passwords or Other Sensitive Information in ASP Pages

As much as this has already been said, it is still very common to see passwords, file paths, database locations, or other sensitive information in ASP code. IIS has been plagued with file-viewing exploits and you can expect there will be plenty more to come. Although normally your ASP code is safe from viewing, you really should not put anything there you don't want someone else to know. And this goes for SQL statements too, as they can reveal your entire database structure.

As an alternative, store this information in COM components, Registry keys, File DSNs, or any place other than your Web root.

8. Do Not Rely on Weak Security Checks.

It is all too easy to fall into the trap of using weak security measures such as relying on the `HTTP_REFERER` variable, enforcing security with client-side script, or limiting form input by setting the `maxlength` property on a form field. While these security measures may seem secure on the surface, they are the equivalent of putting a cheap padlock on a door to keep intruders out: sure,

it may keep some people out but all it takes is a lock cutter or a swift blow with a sledgehammer and you're in.

9. Do Not Leave Comments in Client-Side HTML

Again, this is a recommendation that security experts have been making for years, but I still see developer notes, bits of server-side code, and other sensitive information in HTML source comments. All it takes is a right-click of the mouse to view the source of a Web page. Be sure to always check your Web page HTML source to make certain there is no sensitive information stored in HTML comments.

10. Do Not Volunteer Too Much Information

I recently visited my bank's Web site, curious to see what kind of security measures they took to protect my account information. On the first page I was asked for my account number, the one printed on my checks, and a password. I entered my account number, but considering all the possibilities for a password, I entered the ever-popular "asdfg." I clicked on OK and was given a message that my password must be exactly four characters long and must only contain numbers. Suddenly, that password field went from trillions of possibilities to a mere ten thousand possibilities. The moral here is that there was no need for them to tell me the exact specifications of my password - either I know my password or I don't. And thanks to my bank, it probably wouldn't be too hard for someone else to know it too.

11. Do Not Write Where You Are Reading

Many Web applications at some point require that something be written to disk. It may be a log file, a user transaction, or even a Microsoft Access Database. The problem is that you should never write to a Web directory that has read, or much worse, execute permissions. In fact, if you can avoid it, you should never write to any directory of your Web application. Instead, create a partition or directory outside the Web root and write all you want to that location.

Remember, in Internet Services Manager, if you check that `write` box, you are saying that any anonymous user can write files to that directory.

12. Do Not Put Sensitive Information on URLs

Be considerate of those using your Web application; do not put their sensitive information on your URLs. URLs are passed in plain text, are stored in your Web cache, are auto-completed in the

address bar, and show up as referrer variables. You are exposing any information you put on the URL to others. Instead of passing the sensitive information on the URL, use session variables, POST data, and by all means, encrypt everything.

13. Do Not Use .INC Files

Using `include` files to centralize code is good programming practice, but this common implementation is not so good for security. On a default IIS installation, files with the `.INC` extension are not mapped to anything, so if a request is made for one of those files, IIS will happily return the source as if it were a plain text file.

When centralizing your code, give all include files an `.ASP` extension and put them in the same directory. And what IIS permissions do you need on include directories? The answer is none. No read, write, script, or execute access is needed for the directory that contains your include files.

14. Do Not Send E-Mails Without Validating User Input

As I said before, user input cannot be trusted. And that goes for the E-Mail addresses too. Consider how your user registration form would handle multiple E-Mail addresses, embedded SMTP commands, command-line arguments, etc.

15. Do Not Put Sensitive Data in Hidden Form Fields

This is a common mistake made on shopping cart applications. When a user goes from the process of selecting items, viewing their shopping cart, entering credit card information, then checking out, it is easy to pass purchase information from page to page by using hidden form fields. But consider what would happen if someone changed their cached copy of a form to modify that information? Do you pass the purchase price as a hidden form field? Can someone change to another user's context after they have been authenticated in their own? Can they skip the payment part by changing a hidden form field? These and many other errors still exist in thousands of Web applications using hidden form fields.

16. Do Not Let IIS Handle Errors For You

Error messages usually give up way too information about your code, your database, and your network. You should make sure that IIS does not send detailed error messages to the client, but you should also include robust error-handling in your code. The more errors you handle, the less

likely an attacker is going to be able to discover information.

17. Do Not Lose Control of Your Code

The best way to keep out hackers is to maintain control over your code. That means you should practice tight change control and know what code belongs there and what code does not. You should watch for dead code and eliminate remnants of the programming process such as debug code, temporary files, backup files, and the all-too-common `test.asp`.

18. Do Not Publish Code Without an Audit

When publishing new code on your Web site, you should always put it through a testing process. A carefully-written test plan can ensure that insecure code does not creep into your Web application. Further, you should not update the code in the site itself; instead replace all Web files from a trusted master copy.

19. Do Not Unnecessarily Store Sensitive Information in Your Database

If a hacker broke into your database, what would they find? Do you store sensitive customer information like Social Security numbers, home phone numbers, or credit card numbers from old transactions? Is this information necessary for your Web application? Who would you have to answer to if this information was ever stolen?

Quite frankly, much of what is stored really does not need to be stored. And if it does, it probably does not need to be stored on your web application's database. Further, when it comes to credit card numbers, there is little reason to keep the entire number in your database after a transaction is finished. If your database is full of this type of information, ask yourself if you want to take on the responsibility that comes with holding this data. Here's a headline to consider: "<YOUR COMPANY> Web Site Hacked, 50,000 Customer Credit Cards Cancelled." I can guarantee you that if someone's credit card is cancelled because your site was hacked, chances are they won't be rushing to your Web site to make any more purchases.

20. Do Not Think That This is all There is to Consider

Of course, both hacking and security are constantly evolving. It is essential that you are constantly aware of the newest exploits as well as the newest protection strategies. Keep up with news at sites such as [SecurityFocus](http://www.securityfocus.com) and subscribe to security-related mailing lists, such as

SecurityFocus's [Web Application Security](#) list.

Thinking securely is often an unnatural transition for programmers. After years of learning how to make things easy for users, you must now consider how to make things hard for hackers. As you balance features, schedule, and budget, you must also keep hackers from using your code against you. While there is much to do when building a secure Web application, you can at least start with these twenty things you shouldn't do. So take this list and with it take hard look at your ASP source. You might be surprised what you find.

[Privacy Statement](#)

Copyright 2006, SecurityFocus