Secure DotNetNuke modules

Writing secure DotNetNuke modules

Cathal Connolly

Revision 1.0.0
Last Updated: May 24, 2006
Applies to: DotNetNuke ver. 3.0+

Information in this document, including URL and other Internet Web site references, is subject to change without notice. The entire risk of the use or the results of the use of this document remains with the user.

The example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious. No association with any real company, organization, product, domain name, email address, logo, person, places, or events is intended or should be inferred.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Perpetual Motion Interactive Systems, Inc. Perpetual Motion may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Perpetual Motion, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

DotNetNuke® and the DotNetNuke logo are either registered trademarks or trademarks of Perpetual Motion Interactive Systems, Inc. in the United States and/or other countries.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

Abstract

The purpose of this guide to assist the developer in developing secure DotNetNuke (DNN) modules that plug into the core framework of DNN.

Contents

Chapter 1: Introduction

Introduction

The purpose of this guide to assist the developer in ensuring the DotNetNuke (DNN) modules they write are written with security in mind.

In this guide we will highlight common issues, and offer guidance on how to identify and counter potential security issues that may occur whilst developing DotNetNuke modules.

In addition this document also covers general web security and recommended practices such as defense in depth. “Defense in depth” is the principle of adding layers of protection to defend against attempts by potential hackers. Whilst a hacker may be skilled in certain areas such as SQL injection, they may not have the skills necessary to break through a number of layers. If you code your modules so that they do not make any assumptions, and validate their functionality at a number of levels, your modules will be much more secure e.g. if you intend to write a module that allows file uploads, then it’s a good idea to add code to validate the user and limit the type of files to a safe subset of extensions, rather than assuming any call to this code can be trusted.

What Are Modules?

Modules provide developers with the ability to extend the functionality of DotNetNuke. DotNetNuke provides a pluggable framework that can be expanded by the development of modules. Modules can be developed in any .NET language that support the Code Dom (e.g. Managed C++ cannot be used, but most of the other .Net languages can), even though DNN is developed in VB.NET, a C# developer can still create a module that plugs into the core framework provided by DNN. This pluggable framework is accomplished by creating compiled private assemblies that expose and utilize interfaces specific to DNN. Once you compile the assembly, and then just create a user interface (UI) in the form of ascx files that allow your user to interact with your module.

Modules provide you with maximum code reusability; you can key off the module ID value (provided by the DNN framework) which allows you to display unique data for each implementation of your module.

Why would modules be insecure?

Web applications are popular targets for potential attackers. Whilst the DotNetNuke framework itself has a number of mechanisms that close common attack vectors, poorly written modules may open up holes that would allow attackers to alter content, gain access to data or even take over entire portals.

What do you mean by user input?

Variations on the phrase “User input” are used a number of times in this document, so it’s worth considering what we mean by that. Whilst this may appear at first glance to refer only to values entered by the user, and retrieved from either the forms or querystring collection, really it covers any data that is passed between client and server.

HTTP requests and responses consist of both a head and a body, both of which can contain data. With a little effort, or utilizing one of many available tools, an attacker can alter any of this data to attempt to circumvent your web application protection. Whilst writing your modules, it is worth keeping in mind any time you read data from the following, that you should not implicitly trust it, but should make an effort to sanitise the input so it cannot do any harm. The key areas to watch for are:

- Forms collection (including text passed in hidden form fields)
- Cookies
- Querystring
- Items collection
- Data retrieved from HTTP Headers (e.g. it is possible to alter the user-agent string of a browser request to act as an XSS string)

Chapter 2: Types of issue

Core filtering functions

Before looking at the various types of issues, it's worthwhile looking at one of the core functions that can be used to provide protection against a number of issues.

DotNetNuke contains a routine called InputFilter which has a number of potential values, defined by the following enumeration.

```
Enum FilterFlag
   MultiLine = 1
   NoMarkup = 2
   NoScripting = 4
   NoSQL = 8
End Enum
```

- MultiLine is not really a security function, it was used in the past by some core functions to replace CRLF (carriage-return line feed) characters with the html equivalent
 tags.

- NoMarkup – when this value is used, the contents of the passed string are htmlencoded. This is a very safe mechanism particularly against cross-site scripting attacks, as it converts potential HTML to plain text converting characters such as < to their html encoding equivalent ("<"). This is the safest filter as it will stop arbitrary html being interpreted as javascript. In some cases this may not be suitable e.g. forums/bulletin boards that allow the user to use some HTML markup.

- NoScripting – this value is used to search and strip any suspect HTML from strings passed to it. Typically this includes script or html that a hacker would try to use during a cross-site scripting attack.

- NoSQL – when used, this value calls a function that searches the passed text and strip's anything that might be arbitrary SQL i.e. to protect against SQL Injection attacks.

All of these values are designed to be used on inbound requests i.e. you pass the text/value that you have accepted from the user, so they can be converted/scrubbed accordingly. Doing this inbound offers more protection than having to remember to apply it to all outbound instances.

The values in the enumeration can be used individually or we can use the bitwise OR operator to combine the values e.g. this code will both apply the NoScripting filter, and the NoMarkup filter.

Example of two InputFilter functions being applied

```
Dim objSecurity As New PortalSecurity

Dim myFilteredText as string=objSecurity.InputFilter(request.form("txtSearch"),
PortalSecurity.FilterFlag.NoScripting Or PortalSecurity.FilterFlag.NoMarkup)
lblSearchtext.text=”Searching for :” & myFilteredText
```

SQL Injection

SQL Injection is a common issue, where the attacker alters input which eventually forms part of a database query. Commonly, additional SQL statements are concatenated to an existing query, to allow these secondary statements to run, accessing alternative data, or granting permissions, and in certain scenarios where the database user has sufficient permissions, gaining access to the database server itself. This guide focuses on databases that use the Transact-SQL Dialect (SQL2000/MSDE2000/SQL2005/SQL Express), though SQL injection affects many database applications such as MySQL and Oracle.

Identifying the issue

There are a number of variations, but the most common one is where user input is trusted and forms part of either a concatenation or string replacement within a query, which is then executed e.g.

```
Dim myQuery as string="SELECT username from usertable where password='" &
request.querystring("txtUsername") & "'"
Dim myCommand As New SqlCommand(myQuery, myConnection)
```

In this code snippet, a hacker could alters the URL to alter the query e.g. default.aspx?txtUsername=admin could be changed to default.aspx?txtUsername=admin';delete from sometable -- . As SQL Server uses the semicolon (;) as a delimiter for statements, this querystring alteration will cause 2 statements to run

Using parameterized stored procedures would help guard against this type of issue, but stored procedures that contain concatenation routines themselves could be an issue e.g.

```
create proc GetSearchResults(@searchTerm nvarchar(50))
as
  declare @sql nvarchar(300)
  set @sql = 'select * from searchResults where SearchTerm like ''%'
    + @searchTerm + '%'''
  exec sp_executesql @sql

go
```

All the DotNetNuke core code uses parameterized SQL Stored procedures, and you are highly recommended to follow this approach.

What to check your code for

- Search your code for any dynamic SQL strings
- Check stored procedures for code that uses the sp_executesql stored procedure or the T-SQL function exec/execute to run queries

Protecting against the issue

First and foremost, in an ideal world you would not allow user input to build a portion of a database query. If your module requires you to do this, then where possible ensure that all database access is conducted via stored procedures. In addition to this ensure that you allow for the smallest possible amount of text, validate any user input that is passed to it, and strongly type any variables used i.e. if accepting a parameter from a querystring that should be numeric, check for this.

If your module still needs to accept user input, then you should make use of the NoSQL enumeration of InputFilter. This filter is a blacklist filter (i.e. it pattern matches for known terms), and therefore like all blacklist filters may be fooled by carefully encoded content (e.g. if a hacker disguises their text via an encoding such as hex – please take a look at this guide for more details and examples of this), but does offer an additional layer of protection.

Original unsafe code

```
Dim myQuery as string="SELECT username from usertable where password='" &
request.querystring("txtUsername") & "'"
Dim myCommand As New SqlCommand(myQuery, myConnection)
```

Safer version

```
Dim objSecurity As New PortalSecurity

Dim myFilteredText as string=objSecurity.InputFilter(request.querystring("txtUsername"),
PortalSecurity.FilterFlag.NoSQL)

Dim myQuery as string="SELECT username from usertable where password='" & myFilteredText
& "'"
Dim myCommand As New SqlCommand(myQuery, myConnection)
```

Cross site scripting

Cross site scripting (XSS), is another vulnerability that relies on un-sanitized user input. In this case, rather than being sent to the database server, the malicious script that's sent to the application as input, is eventually echoed back to a users browser where it is executed. Commonly, this script will attempt to gain access to the user's cookies, so an attacker can use them to access the application with the user's credentials i.e. an impersonation attack. Other less common XSS attacks, will execute javascript in the users context, perhaps to redirect to another page that bears a close resemblance to the original ("a phishing attack")

Identifying the issue

Anywhere you return user input to the screen, or store if to be later returned, perhaps in a file or database store, or even in a application level object such as session, cookie, application etc., the code should be inspected to check if the values have not been sanitized e.g. this code snippet shows an example of unsafe code.

```
Dim mySearchTerm as string= request.querystring("txtSearch")

lblSearchtext.text="Searching for :" & mySearchTerm
```

When it runs, it trusts the input it retrieves from the request.form("txtSearch") value, and renders it, but this input could be unsafe e.g. if you receive an email with a link that suggests you go to http://somesite.com/default.aspx?txtSearch=<script>alert('unsafe input');</script>

The code above would render then render this javascript. Whilst this example is little more than an annoyance, it is not difficult to alter it so that it is more dangerous.

What to check your code for

Examine your code for anywhere you reference values from HttpRequest, and check all outputting code, such as use of HttpResponse and places where you explicitly set variables values via .value or .text. If using inline code, examine your use of <%= %>

Protecting against the issue

Once again the key point is to not trust user input . In this case the recommended practice is to utilize InputFilter with “NoMarkup”. If your input must allow HTML, then you can use “NoScripting” instead, but please think carefully whether or not your module really needs to support HTML text, as the “NoScripting” filter is a blacklist filter, and therefore like all blacklist filters may be fooled by carefully encoded content (e.g. please take a look at this guide for more details and examples of cross-site scripting(XSS) encoding).

Original unsafe code

```
Dim mySearch as string= request.querystring("txtSearch")

lblSearchtext.text=”Searching for :” & mySearch
```

Better version that strips common XSS variants

```
Dim mySearch as string= request.querystring("txtSearch")

Dim objSecurity As New PortalSecurity

mySearch =objSecurity.InputFilter(mySearch, PortalSecurity.FilterFlag.NoScripting)
lblSearchtext.text=”Searching for :” & mySearchTerm
```

Best version that HTML-Encodes the string

```
Dim mySearch as string= request.querystring("txtSearch")

Dim objSecurity As New PortalSecurity

mySearch =objSecurity.InputFilter(mySearch, PortalSecurity.FilterFlag.NoMarkup)
lblSearchtext.text=”Searching for :” & mySearchTerm
```

Unnecessary attack points

DotNetNuke uses a modular architecture. These modules are usually a collection of user controls (.ascx files), that contain custom code. An .ascx control cannot be viewed directly in a browser, so this allows you to make sure that people navigate your modules in the way you intended. From time to time, module developers use .aspx pages, either to allow them to be directly called (e.g. just as DotNetNuke does with the "install.aspx" page that ships with the core-code) or perhaps to provide content that is of a different disposition to the content in a page (e.g. a page that creates thumbnails/rss feeds). These pages should be carefully examined as they are directly callable, potential attackers will look closely at them. Ideally 3rd party module developers should try to write their modules utilizing only ascx control's, as then DotNetNuke provides protection automatically, whereas if a module developer uses aspx pages, they must remember to validate the user within their code.

Identifying the issue

Locate any callable entities such as .aspx pages, and check that the code does not make assumptions. You should repeat whatever authentication methods your module has used e.g. user/role permissions, filetypes etc.

Filtering for allowed file types

Identifying the issue

Web applications such as DotNetNuke commonly support file upload capabilities so that users can post documents and graphic files to application folders, so they can be made available for download or rendered. If your code allows users to upload content that can be executed on the server such as .aspx or .asp pages, it could allow users of your portal to take over the application/server or elevate their rights.

What to check your code for

Anywhere you've code that allows file uploads, either through using existing DotNetNuke code, or via custom code.

Protecting against the issue

Ideally, you should utilize the existing core functions, such as UploadFile that provide a number of levels of checking, but if you decide to implement your own code, DotNetNuke allows you to specify allowed file types at the portal level, under Site Settings. By default this is a restrictive list of 'safe' file types, so you're recommended to utilize the existing code in the following fashion:

```
Dim _portalSettings As PortalSettings = PortalController.GetCurrentPortalSettings
Dim objPortalController As New PortalController
Dim strFileName As String =  Path.GetFileName(objHtmlInputFile.FileName)
Dim strExtension As String = Replace(Path.GetExtension(strFileName), ".", "")


If InStr(1, "," & _portalSettings.HostSettings("FileExtensions").ToString.ToUpper, "," &
strExtension.ToUpper) <>0 then
'do not allow upload
end if
```

In addition to this, it's a good idea to check that your user both has sufficient rights to upload files, and that the file they're attempting to upload will not exceed the maximum size of the portal. This adds an additional level of protection against users attempted to cause a denial of service attack via resource exhaustion through posting large files.

For instance in the next snippet, we check to make sure the file is within limits (or the portal has unlimited space set [hostpace=0], but we allow menu items under the host menu to ignore this check.

```
If ((objPortalController.GetPortalSpaceUsed(_portalSettings.PortalId) +
objHtmlInputFile.ContentLength) / 1000000) <= _portalSettings.HostSpace) Or
_portalSettings.HostSpace = 0) Or (_portalSettings.ActiveTab.ParentId =
_portalSettings.SuperTabId) Then
```

File redirection

Identifying the issue

Many modules will allow the user to download a file from a list. Often the code that is used to force the download the file can be forced to download a file of the hackers choice e.g. the web.config file. This is usually accomplished by altering the path passed to the code that downloads the file, usually through using multiple periods to cause the choice of folder/file to be altered (known as a parent-paths based attack) e.g. default.aspx?download=safefile.jpg is changed to default.aspx?download=../../web.config . This is a common failure in lots of code, where the path variable is trusted, and you are advised to take particular care if you write modules using this functionality.

What to check your code for

Anywhere you provide the ability to download files, in particular code that uses response.append to force downloads. For instance, the following code is vulnerable to attacks using the URL shown above.

```
strFilename=request(“download”)
Response.AppendHeader("content-disposition", "attachment; filename=" + strFileName)
Response.ContentType = Request.QueryString("contenttype").ToString
Response.WriteFile(strLink)
Response.End()
```

Protecting against the issue

In general, do not allow folders to form part of the path. You should always calculate paths based on the values generated by DotNetNuke. In addition, utilize the core function that checks for attempts to use cross-application mapping and parent path traversal

```
Dim objPortal As PortalInfo = objPortalController.GetPortal(intPortalId)
txtHomeDirectory.Text = objPortal.HomeDirectory
txtFileLocation.text=txtHomeDirectory.text & QueryStringDecode(Request("download")
```

Protecting sensitive output

Identifying the issue

Often attackers will write programs that operate in a similar fashion to search engines, indexing your site content. However, they're usually looking to harvest data such as email address, to add to spam mailing lists.

What to check your code for

Anywhere you write out email address's, in particular code that contains that uses the hyperlinks that contain the mailto: protocol to generate clickable email links.

```
Response.Write ("<a href='mailto:someemail@somewhere.com'>someemail@somewhere.com</a>")
```

Protecting against the issue

DotNetNuke provides a function, CloakText, that hides sensitive data by generating javascript that spiders pass over, but that still correctly renders output. The core function, FormatEmail utilizes this and concatenates the output with a the html to create a hyperlink

```
Response.Write (FormatEmail("somemail@somewhere.com"))
```

Chapter 3 Best practices

It is recommended that you utilize a few common best practices when developing your modules. Not only will this help to create modules with a good blend of defensive layers, but they will increase the overall quality of your final product. Where possible try to utilize the following in your development practices.

Variable typing

In web applications data is often passed in the header/body of the request. This takes the form of a string, but often you intend this string to only contain values from a particular type. If you know this type in advance then, you should check for it e.g

In asp.net 1.1

```
Try
Int32.Parse(request("moduleID")).
Catch …
```

In asp.net 2.0

```
If Int32.TryParse(request("moduleID"))=true then …
```

Use attributes

Many elements, both html and server controls, contain attributes that can help constrain input. Where applicable try to utilize the ReadOnly and Maxlength attributes, and always ensure that textbox's obscure sensitive data by using password functions e,g

```
<asp:textbox id="myPassword" TextMode="Password" Maxlength="10"
runat=server></asp:textbox>
```

Validators

Asp.net ships with a number of different validators, that help you check the validity of your data both client and server side. An explanation of these is beyond the scope of this document, but the following links should help illustrate the use.

Asp.net 1.1

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnaspp/html/aspplusvalid.asp

asp.net 2.0

http://www.asp.net/QuickStart/aspnet/doc/validation/default.aspx

Additional Information

The DotNetNuke Portal Application Framework is constantly being revised and improved. To ensure that you have the most recent version of the software and this document, please visit the DotNetNuke website at:

http://www.dotnetnuke.com

The following additional websites provide helpful information about technologies and concepts related to DotNetNuke:

DotNetNuke Community Forums
http://www.dotnetnuke.com/tabid/795/Default.aspx

Microsoft® ASP.Net
http://www.asp.net

Open Source
http://www.opensource.org/

W3C Cascading Style Sheets, level 1
http://www.w3.org/TR/CSS1

Errors and Omissions

If you discover any errors or omissions in this document, please email marketing@dotnetnuke.com. Please provide the title of the document, the page number of the error and the corrected content along with any additional information that will help us in correcting the error.

Appendix A: Document History

Version	Last Update	Author(s)	Changes