ColdFusion 11 Developer Security Guide

Pete Freitag

Contents

Overview

In this guide, I will discuss several vulnerabilities that pertain to web applications. Most of the vulnerabilities discussed in this guide are not unique to ColdFusion. However, the mitigation techniques discussed are ColdFusion-specific.

Following best practices such as validation, testing, code reviews, defense in depth, and the principle of least privilege can provide a broad increase in security across your applications.

Testing

It is always important to test code before it reaches a production server. Setting up a local or dedicated development environment is essential. You can improve code quality even more by writing software test cases. MXUnit and TestBox are two popular testing frameworks you can leverage that were built by the ColdFusion community.

Code Reviews

Another way to improve overall security is by performing code reviews. Take time to specifically look for security vulnerabilities in your code. Utilize peer reviews or engage a third party to perform the review. Having an independent party look at your code can sometimes reveal scenarios that the original developer didn't consider.

Defense in Depth

When it comes to security, it never hurts to perform what may appear to be redundant security checks. Never assume that protections in one layer of the application are sufficient when additional defenses can be added. Different types of defenses may catch different classes of vulnerabilities.

A Web Application Firewall (WAF) is a good example of defense in depth. A WAF sits in front of your application code and then logs or blocks malicious requests.

Principle of Least Privilege

The principle of least privilege is a foundational security pillar. The concept is to give the user the minimal permission to do only the job at hand, and nothing more. It is important for the developer to fully understand the different roles of the system so that they can adhere to this principle.

SQL Injection

Take special care whenever a variable is used to derive a SQL query.

Vulnerable Code Example

```
<cfquery>
   SELECT headline, story
   FROM news
   WHERE id = #url.id#
</cfquery>
```

In this example, the attacker can create arbitrary SQL statements to execute against the database by passing values into the url.id variable. For example, the attacker can pass a value of `1;DELETE FROM news` to delete all news articles in the table or 0 UNION SELECT username, password FROM users to extract username and password values from the database.

Secure Code Example

```
<cfquery>
   SELECT headline, story
   FROM news
   WHERE id = <cfqueryparam value="#url.id#"
                            cfsqltype="cf_sql_integer">
</cfquery>
```

By using the cfqueryparam tag, you have *parameterized* the variable in your SQL query, which separates the variables (or parameters) in your SQL query from the SQL statements. In database terms, this is called a prepared statement, and can sometimes yield performance improvements in addition to security.

The cfqueryparam tag will validate the value based on the type specified in the cfsqltype attribute. If the value does not match the cfsqltype specified, an exception will be thrown.

The cfqueryparam tag's use isn't limited to WHERE clauses or SELECT statements. You can and should use it when passing values to your INSERT, UPDATE and DELETE statements as well.

Depending on the database type, there may be some parts of the SQL statement in which cfqueryparam cannot be used. One example is the TOP statement in SQL Server. In such cases, you need to make sure that the variable has been strictly validated such that only expected values are allowed to pass to the SQL statement.

```
<cfif NOT IsValid("integer", url.max) OR url.max LTE 0>
   <cfset url.max = 10>
</cfif>
<cfquery>
   SELECT TOP #Int(url.max)# headline, story
```

```
   FROM news
</cfquery>
```

When in doubt, try the cfqueryparam in the query, if it is not allowed in a given part of a SQL statement, an exception will be thrown.

Beyond the cfquery tag, you also need to ensure that instances of QueryExecute or OrmExecuteQuery do not allow for SQL Injection. To prevent SQL injection in queryExecute pass variables as parameters in the second argument. For example:

```
<cfset result =
    QueryExecute("SELECT headline, story FROM news WHERE id = :id",
{id=url.id})>
```

```
<cfset result =
  OrmExecuteQuery("FROM news WHERE id = ?", [url.id] )>
```

Stored procedures should be invoked using the cfstoredproc tag and the cfprocparam tag can be used to pass variables into the procedure. Do not invoke stored procedures using cfquery and unchecked variables.

Finally, make sure that the database user specified in your datasource has minimal privileges. The greater the privileges of the database user, the greater the impact of a possible SQL injection attack.

Cross Site Scripting (XSS)

Before outputting any variable, be sure that it is properly encoded to prevent client-side execution (such as JavaScript).

Vulnerable Code Example

```
<cfoutput>Hello #url.name#</cfoutput>
```

Using the vulnerable code example, the attacker can pass JavaScript into the url.name variable that will be executed in the browser of anyone visiting the URL. Attackers may also try to post XSS code that will be stored in a database then output later (for example posting a comment, that is then displayed for all visitors on the page).

If the attacker is successful in finding a victim to visit the page executing the script, the script can do any number of harmful things such as:

- Changing the content on the page.
- Creating a login form on the page that submits the credentials to a malicious URL.
- Posting to a URL using the credentials of the currently authenticated user
- Stealing cookies
- Logging keystrokes
- And more.

Secure Code Example

```
<cfoutput>Hello #EncodeForHTML(url.name)#</cfoutput>
```

The EncodeForHTML function (requires ColdFusion 10 or higher) will encode the variable to escape special characters using their HTML entity. For example, the < character will be replaced with <

Depending on the context of where the variable is output, you may need to use an encoding function other than EncodeForHTML. For example if you are outputting a variable within JavaScript, or within a JavaScript event attribute (such as inside of an onclick HTML attribute) then you should use the EncodeForJavaScript function instead.

There are four additional encoding functions that can be used in different output contexts, EncodeForCSS, EncodeForHTMLAttribute, EncodeForURL and EncodeForXML.

The encoding functions work very well when you do not want any HTML in the variable you are outputting. However, if you need to allow HTML in an output variable, filtering out harmful HTML is much more difficult. ColdFusion 11 introduced the GetSafeHTML and IsSafeHTML functions, which utilize an XML policy file to determine what HTML tags and attributes are *safe*.

The GetSafeHTML and IsSafeHTML functions are powered by the AntiSamy Java library. ColdFusion 11 ships with a default AntiSamy XML policy file (located {cf.instance.root}/lib/antisamy-basic.xml), you can also create your own policy (consult the *AntiSamy Developer Guide* for documentation, under *Additional XSS Resources*). The policy file controls the definition of what is considered *safe* with respect to GetSafeHTML and IsSafeHTML. You can pass the file path for the AntiSamy XML policy file into the second argument of GetSafeHTML or IsSafeHTML, or you can define a variable in Application.cfc called this.security.antisamypolicy, which allows you to specify a default policy for your application.

Content Security Policy

Some modern browsers have added features to defend against cross site scripting attacks. The Content-Security-Policy HTTP response header is a powerful XSS defense mechanism. By sending Content-Security-Policy response headers you can restrict resources to certain URIs or require same origin, block inline CSS and JavaScript from executing, and more.

Here is an example setting a Content-Security-Policy header with CFML:

```
<cfheader name="Content-Security-Policy" value="default-src 'self'">
```

The above example tells the browser to block loading any resource (images, JS files, CSS files, etc.) that is not on the same origin (self). It also blocks execution of any JavaScript or CSS that is not defined in an external file (also on the same origin), including inline JavaScript such as unload or onmouseover HTML attributes.

While the Content-Security-Policy browser feature is very powerful, it is not supported on all browsers and is not an excuse to avoid fixing XSS vulnerabilities in your code.

Additional XSS Resources

- ScrubHTML - a pure CFML alternative to GetSafeHTML: https://github.com/foundeo/cfml-security
- Using AntiSamy with ColdFusion 10 & Below: http://www.petefreitag.com/item/760.cfm
- AntiSamy Developer Guide: https://owaspantisamy.googlecode.com/files/Developer%20Guide.pdf
- Content Security Policy Quick Reference: http://content-security-policy.com/

Uploading Files

Whenever files are uploaded to the server, take extreme care to ensure that you have properly validated the file path and file type.

Vulnerable Code

```
<cffile action="upload"
      filefield="photo"
      accept="image/gif,image/png,image/jpg"
      destination="#ExpandPath("./photos/")#"
      nameconflict="overwrite">
```

The above vulnerable code example relies on the accept attribute of cffile to validate the uploaded file, which is insufficient. ColdFusion 10 changed the behavior of the accept attribute, and also added a strict attribute.

Prior to ColdFusion 10, the accept attribute only allows a list of mime types and is validated using the mime type sent by the client, this can easily be changed.

In ColdFusion 10, and later the strict attribute was added, which controls how the accept attribute is handled when it contains a mime type list. When strict=true (which is the default when omitted) the file content is validated internally using the FileGetMimeType function. When strict=false, it uses the file extension of the uploaded file to match the mime type.

ColdFusion 10 also added the ability to specify a list of file extensions in the accept attribute, instead of a list of mime types.

Suggestions for secure file uploads:

- Always upload to a destination outside of the web root, even if you plan on putting the file under the web root. The file must be validated outside of the web root first.
- Always validate the file extension against a list of allowed file types.
- Never validate based on mime type alone.
- Never validate based on file type checking alone.
- If possible validate the file contents match the allowed type. Functions such as IsImageFile, IsPDFFile, IsSpreadsheetFile and FileGetMimeType can help.
- Configure your web server to limit the size of files to be uploaded.
- If uploaded files are placed in a directory under the web root, ensure that the web server has been configured to serve only static files from the directory.
- Do not use file names supplied by the client, generate the final file name dynamically.

Secure Code Example

```
<cffile action="upload"
      filefield="photo"
      accept="*.png,*.jpg"
      destination="#getTempDirectory()#"
```

```
      result="cffile"
      nameconflict="makeunique">
<cfset tempFile = cffile.ServerDirectory & "/" & cffile.ServerFile>
<cfif NOT ListFindNoCase("jpg,png", cffile.ServerFileExt)>
      <cfset FileDelete(tempFile)>
      Sorry invalid file type.
<cfelseif NOT ListFindNoCase("image/png,image/jpg",
            FileGetMimeType(tempFile, true))>
      <cfset FileDelete(tempFile)>
      Sorry invalid file type.
<cfelseif NOT IsImageFile(tempFile)>
      <cfset FileDelete(tempFile)>
      Sorry invalid file type.
<cfelse>
      <!--- process file --->
</cfif>
```

File Path Injection

Avoid using untrusted inputs in file operations.

Vulnerable Code

```
<cfinclude template="views/#header#">
```

The above vulnerable code sample does not validate the value of the #header# variable before using it in a file path. An attacker can use the vulnerable code to read any file on the server that ColdFusion has access to. For example by requesting ?header=../../server-config.txt the attacker may read a configuration file that is not meant to be public.

In addition to reading files with this vulnerability, the attacker may also be able to execute code because the cfinclude tag will execute CFML code when processed. ColdFusion 11 mitigates some of this risk by only executing files with the cfm or cfml file extension in a cfinclude tag, all other files are included statically. You can control the list of file extensions that are executed with cfinclude by setting the this.compileextforinclude variable in Application.cfc or by configuring it in ColdFusion Administrator. Adding file extensions to this list or specifying the * wildcard will increase the impact of a file path injection vulnerability in your source code.

While thus far we have discussed the cfinclude tag, the same care and precautions must be taken on any tag or function that uses file paths. Here's a partial list of tags and functions that use file paths: cffile, cfdirectory, cffileupload, cfzip, cfftp, ExpandPath, FileOpen, FileCopy, FileDelete, FileExists, FileMove, FileRead, FileWrite, DirectoryExists, DirectoryDelete, DirectoryList, DirectoryCreate, DirectoryRename, etc.

Mitigation Steps

- Avoid variables derived from untrusted input in file paths.
- Always fully scope variables used in file paths.
- Always validate variables used in file paths directly before use.
- Strip out .. and other unexpected characters which may be used for traversal.

Encryption & Cryptography

When working with encryption, some of the important aspects to consider are key storage, encryption algorithms, and encryption key size. Enterprise editions of ColdFusion ship with the RSA BSAFE Crypto-J Java Cryptography Extension (JCE) to provide FIPS compliant crypto algorithm implementations. Any edition of ColdFusion 7 or greater can be configured to use the JCE of your choice to provide algorithm support for the Encrypt, Decrypt, Hash, HMac, GenerateSecretKey, GeneratePBKDFKey functions and more.

Cryptography Suggestions

- When possible use a Hardware Security Module (HSM) for key storage.
- Don't hard code encryption keys into source code.
- Avoid storing encryption keys in plain text on the server, leverage a HSM, a Java KeyStore, or an operating system facility such as Microsoft Data Protection API (DPAPI)
- Do not store keys adjacent to encrypted data.
- Rotate encryption keys on a periodic basis.
- Do not use the CFMX_COMPAT encryption algorithm, it is not a cryptographically secure algorithm. When the algorithm argument of Encrypt or Decrypt function is omitted CFMX_COMPAT is used by default. This algorithm is default only for the purpose of backwards compatibility with ColdFusion 6 and below, it is the weakest option.
- To use certain encryption key sizes greater than 128 you may need to enable the Java Unlimited Strength Jurisdiction Policy (see Additional Resources below).
- When passing the algorithm to Encrypt or Decrypt you can specify feedback mode block cipher algorithms and padding algorithms. For example, when you pass "AES" it is a shortcut for "AES/ECB/PKCS5Padding". Cipher Block Chaining (CBC) mode is considered the "fastest and most secure feedback mode" according to Adobe documentation. To use CBC instead of ECB specify "AES/CBC/PKCS5Padding" as the algorithm. Additional feedback modes and padding methods are also supported.
- When working with random numbers (such as with Rand, RandRange or Randomize functions), use a SecureRandom number generator. On Sun JVMs SHA1PRNG can be used, on IBM JVMs IBMSecureRandom can be used. The RSA BSAFE JCE provider provides additional algorithms to consider. Avoid using ECDRBG and Dual_EC_DRBG, the Dual Elliptic Curve Deterministic Random Bit Generator due to potential weaknesses.

Additional Resources

- Enabling Unlimited Strength Crypto in ColdFusion: http://www.petefreitag.com/item/803.cfm
- Strong Encryption in ColdFusion MX 7: http://helpx.adobe.com/coldfusion/kb/strong-encryption-coldfusion-mx-7.html

Error Handling

Improper or insufficient error handling can lead to security vulnerabilities and information disclosure. Good error logging and alerting can help thwart, or provide useful forensic information in the event of an attack.

ColdFusion provides several mechanisms for handling errors, the most basic level is the cftry and cfcatch tags. For example:

```
<cftry>
  <cfset d = 5/0>
  <cfcatch>
      <cfset d = 0>
  </cfcatch>
</cftry>

Or using the cfscript equivalent:

try {
  d = 5/0;
} catch {
  d = 0;
}
```

If a runtime exception is not caught by a try / catch, it propagates up to the application level event handler, defined in Application.cfc by implementing the onError Function.

The onError handler will process all runtime exceptions thrown within the application, however it will not process syntax errors that prevent your code from compiling. To handle syntax errors on an application specific basis you can place a cferror tag inside your Application.cfc or Application.cfm file. Alternatively you can define a *Site-wide Error Handler* in the ColdFusion Administrator, which takes precedence over cferror for syntax errors.

Developers should also implement the onMissingTemplate function in Application.cfc. The onMissingTemplate function is invoked when a request is made for a cfm file that does not exist. Rather than showing the unfriendly default error developers can log and display a simple message.

When writing an error handler, the message sent back to the browser for a generic unexpected exception should not divulge any details of the exception. If an error message contains file paths, code, SQL table or column names, an attacker may be able to use that information to craft an attack.

When exceptions or errors occur you should log the details of the exception and have a mechanism to be alerted when unexpected exceptions occur. Take care not to cause a denial of service when alerting yourself (don't send an email for each error). You must also take care to ensure that you do not log or transmit sensitive information such as passwords or credit card numbers in your error handling code.

Additional Resources

- Application.cfc onError Documentation: https://wikidocs.adobe.com/wiki/display/coldfusionen/onError
- LogBox - an open source enterprise logging framework for CFML: http://wiki.coldbox.org/wiki/LogBox.cfm
- BugLogHQ - an open source CFML bug logging application: http://www.bugloghq.com/

Validation

By checking that variables match the format that you expect, you can avoid a lot of potential security issues. You may have noticed by now that many of the mitigation steps for resolving security issues involve validating a variable before using it. For example, in the *Vulnerable Code Example* in the *SQL Injection* section, if we had validated that url.id was an integer before using it in the SQL statement we would have been immune to the SQL Injection vulnerability in the code.

The bottom line is that adding validation to your application often makes it more secure and less vulnerable to attack.

Canonicalization

Before validating a string, it should be canonicalized to a its most basic form, this typically involves reversing all encoding or escaping that might occur in the string. ColdFusion 10 added the Canonicalize function which will reverse HTML entities or URL encoding into their canonical form (for example < becomes <).

The Canonicalize function takes up to 4 arguments:

- **inputString**: The string you are canonicalizing is passed into the first argument (required)
- **restrictMultiple**: When true the function checks for multiple encoding attempts. For example %2526 may be used to attempt to represent an ampersand if decoded twice. URL Decoding %2526 results in %26, if %26 is URL decoded again it results in an ampersand &. When throwOnError is false the function will return an empty string if multiple encoding is detected. (required)
- **restrictMixed**: When true the function checks for mixed encoding attempts, such as %26lt; which attempts to mix URL encoding with HTML entity encoding. When throwOnError is false the function returns an empty string if mixed encoding is found in the inputString.
- **throwOnError**: When true an exception is thrown when mixed or multiple encoding is found in the string if restrictMultiple or restrictMixed are also true, if false returns an empty string instead of throwing an exception. Default is false, ColdFusion 11 added this argument, the ColdFusion 10 behavior is equivalent to setting throwOnError true.

The Canonicalize function does not deal with character set issues, which can open XSS attack vectors. Be sure that your Content-Type HTTP response header specifies a character set to prevent issues related to automatic character set detection and conversion in browsers. For example:

```
<cfcontent type="text/html; charset=utf-8">
```

ColdFusion Validation

ColdFusion provides several functions for validating basic types, the most flexible is the IsValid function which can be used to validate that a value is a valid integer, email, zipcode, us date, telephone number, url, or several other types.

ColdFusion implements several additional functions that are useful for validation, many of them start with is (consult the CFML Language Reference), for example IsDate, IsJSON, IsXML, IsSafeHTML, etc.

The cfparam tag can also be very useful for validating input variables and specifying default values when those variables are not defined, for example:

```
<cfparam name="url.id" default="0" type="integer">
```

When a variable does not match the type specified in cfparam it will throw an exception. When using cfparam be sure you have error handling to catch the possible exceptions thrown by it.

The cfparam tag supports many of the same types as the IsValid function.

Client Side Validation

While client side validation, such as JavaScript form validation does provide a good user experience, it should not be depended on for security. All validation must be performed server side in addition to any client side validation.

Additional Resources

- IsValid function documentation: https://wikidocs.adobe.com/wiki/display/coldfusionen/IsValid
- The cfparam tag documentation: https://wikidocs.adobe.com/wiki/display/coldfusionen/cfparam
- ValidateThis CFML Object validation framework: http://www.validatethis.org/
- Double Encoding: https://www.owasp.org/index.php/Double_Encoding

Cross Site Request Forgeries

Cross Site Request Forgeries (CSRF) vulnerabilities are exploited when an attacker can trick an authenticated user into clicking a URL, or embedding a URL in a page that will be requested by their authenticated browser.

CSRF is best illustrated by an example. Consider the following code that is used to upgrade a user's permissions to become an administrator when they click on a link:

Vulnerable Code

```
<cfif session.isAdministrator>
    <cfparam name="url.userID" type="integer" default="0">
    <cfquery>
        UPDATE users
        SET isAdministrator = 1
        WHERE userID =
            <cfqueryparam value="#url.userID#"
                          cfsqltype="cf_sql_integer">
    </cfquery>
<cfelse>
    You must be an administrator to perform this action.
</cfif>
```

The above code resides in a template accessible here: https://example.com/admin/make-administrator.cfm

The user Johnny is an Administrator (session.isAdministrator is true), and he is currently logged in to example.com. Mary is a user (her userID=1337) at example.com as well, but she is not an administrator at example.com.

Mary can become an administrator if she can get Johnny to visit the URL https://example.com/admin/make-administrator.cfm?userID=1337. There are many ways that Mary could trick Johnny into visiting that url with his authenticated browser session without him realizing it. For example she might send him an email or message with the url inside an <img> tag, or she might create a webpage with the url embedded in an <img> tag, or an iframe, or any other tag that will fetch the url. Mary might also be able to exploit a cross site scripting hole in example.com and get him to request the url.

Preventing CSRF Attacks

The best way to prevent CSRF attacks is to require password reentry when performing sensitive operations. Reentering your password every time you perform an action is not always possible from a usability perspective, other techniques can be used.

Another solution that works well in preventing CSRF attacks, but is not very usable, is requiring the completion of a CAPTCHA. ColdFusion's cfimage tag can be used to create CAPTCHA images, if this is an

appropriate solution for your requirements. Keep in mind that CAPTCHAs often do not meet accessibility standards because they are impossible or difficult to complete by people with disabilities.

Another technique requires you to pass a random secret token string in the request (valid for a single user, often called a CSRF token) whenever authenticated actions are performed. This makes it virtually impossible for the attacker to know the url or form data required to perform the action ahead of time. The attacker may however be able to find the valid CSRF token if a cross site scripting vulnerability exists on the site.

ColdFusion 10 added two functions for working with CSRF tokens, CSRFGenerateToken and CSRFVerifyToken.

The CSRFGenerateToken function takes two optional arguments. The first argument is a key, which can be used to separate tokens for different purposes (different forms). Keep in mind that each token you generate with a unique key is stored in the session, so it will utilize memory for the duration of the session. The second argument for CSRFGenerateToken, forceNew will cause a new token to be generated for the given key. It is a good idea to regenerate the token after it has been used successfully in many cases. In some cases you may want to generate a new token every time you display the form, this provides the best security, but could cause an issue if the form is loaded multiple times on the page, or if the form is loaded in multiple tabs in the users browser.

Here is an example of using CSRFGenerateToken in a form:

```
<form action="make-administrator.cfm" method="POST">
    <input type="hidden"
      name="token" value="#CSRFGenerateToken("make-admin")#" />
    <input type="hidden"
      name="userID" value="#encodeForHTMLAttribute(user.userID)#" />
    <input type="submit" value="Make Administrator" />
</form>
```

Now on our form action page (make-administrator.cfm in this example) you can use the CSRFVerifyToken function to check that the token is valid for the current user. The first argument accepts the token, and the second argument is optional to pass the key used in CSRFGenerateToken. For example:

```
<cfif CSRFVerifyToken(form.token, "make-admin")>
    <!--- perform operation --->
</cfif>
```

Require HTTP POST

It is also a good idea to ensure that certain operations only occur over the HTTP POST method, to prevent attack vectors that implicitly use HTTP GET (such as img tags or iframes). You can use the following code to ensure that the HTTP method is POST:

```
<cfif UCase(cgi.request_method) IS "POST">
    <!--- perform operations --->
</cfif>
```

Check Referer and Origin Headers

Finally it is also useful to check the HTTP Referer header and Origin header if they exist to make sure that they match a whitelist of allowed domains and or URIs. For example:

```
<cfif Len(cgi.http_referer)
      AND LCase(cgi.http_referer) IS NOT
"https://example.com/admin/user.cfm">
    Sorry the referrer you sent is not allowed to post here.
    <!--- log attempt here --->
    <cfabort>
</cfif>

<cfset httpRequest = GetHttpRequestData()>
<cfif StructKeyExists(httpRequest.headers, "Origin")>
    <cfif httpRequest.headers["Origin"] IS NOT "https://example.com">
        Sorry invalid origin.
        <!--- log attempt here --->
        <cfabort>
    </cfif>
</cfif>
```

Additional CSRF Resources

- OWASP CSRF Prevention Cheat Sheet: https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet

Cookies

Since cookies are often used to maintain state or sessions, understanding practices to limit their transmission and readability can improve security.

Cookies are set using a HTTP response header Set-Cookie, an example response header may look like this:

```
Set-Cookie: favorite=oatmeal;
```

The browser then sends the cookie name and value back to the server in the Cookie HTTP request header on subsequent requests. In CFML we can reference that cookie using the cookie scope, cookie.favorite in the above case would be set to *oatmeal*. Cookies in CFML are typically set using the cfcookie tag, but they can also be set using the cfheader tag.

Cookies can have additional attributes that instruct the browser when to send them, how long to keep them, and if they should be hidden from JavaScript or other non HTTP APIs.

The secure attribute

The first attribute we will discuss is the most obviously important to security is the **secure** attribute. When the secure attribute is enabled on a cookie, the browser will only send the cookie back to the server over a secure transport mechanism (such as HTTPS). If a cookie contains sensitive information (such as session identifiers) it should only be sent over a secure transport mechanism (to prevent eavesdropping), and thus should have the secure attribute enabled.

The HttpOnly attribute

If an attacker is able to exploit a XSS vulnerability, they can read the visitors cookies using JavaScripts document.cookie variable. In 2002 browser vendors began supporting a new cookie attribute called HttpOnly which prevents the cookie from being read by non HTTP APIs, such as JavaScript. So when a cookie is set with the HttpOnly attribute, its value is not available in document.cookie.

Use the HttpOnly attribute on all cookies that do not need to be read by JavaScript. Note that XMLHttpRequests or AJAX requests will still send the cookies when they are marked HttpOnly, they are simply not available to the JavaScript API.

The path attribute

When you set the path attribute on a cookie the browser will only send the cookie on requests under the path. So for example if you have set a cookie with path=/admin the browser will only send the cookie when your URI begins with /admin.

You can use the path attribute to reduce transmission of cookies to only certain portions of your site that needs them.

The domain attribute

The domain attribute of a cookie can be used to control what domains have access to a cookie. For example if you wish to set a cookie that can be read by all subdomains of example.com you can specify domain=.example.com

According to RFC 6265: If the server omits the Domain attribute, the user agent will return the cookie only to the origin server. Internet explorer however does not follow this standard and will make the cookie available to all subdomains off the current domain.

It is important to consider cookie domain attributes, especially if any third party services are hosted on subdomains of your primary domain. For this reason it is best to use a different domain name for hosting potentially untrusted content, or third party services.

The expires attribute

The expires attribute is used to give the cookie an end of life date. When you omit the expires attribute the cookie this creates what is sometimes called a browser session cookie (or non-persistent cookie) that lives for the duration of the browser process.

Additional Cookie Resources

- RFC6265 HTTP State Management Mechanism: http://tools.ietf.org/html/rfc6265
- IE Cookie Internals (IE does not follow the RFC): http://blogs.msdn.com/b/ieinternals/archive/2009/08/20/wininet-ie-cookie-internals-faq.aspx
- Securing your application using HttpOnly cookies with ColdFusion http://www.adobe.com/devnet/coldfusion/articles/coldfusion-securing-apps.html

Sessions

ColdFusion supports two types of session types: CFID/CFTOKEN ColdFusion sessions and JEE jsessionid sessions. The decision to use ColdFusion sessions or JEE sessions must be made at a server wide level (in ColdFusion Administrator).

There are advantages and disadvantages to each type of session, here are some of the differences between the two types, as of ColdFusion 11:

ColdFusion Session (CFID/CFTOKEN) Pro's:

- Session cookie options can be configured on a per Application basis in Application.cfc using the this.sessioncookie variable.
- Support for SessionRotate and SessionInvalidate functions.

ColdFusion Session (CFID/CFTOKEN) Con's:

- Cannot configure the length of the session identifier
- The CFID cookie is often flagged by security scanning tools as a non-random session identifier because such tools fail to look at the CFTOKEN cookie.
- Not accessible from other JEE resources.

JEE Session (jsessionid) Pro's:

- Uses a single variable jsessionid to identify the session, instead of two.
- Session data can be accessed by JEE Servlets, JSP or Servlet Filters.
- Configurable session identifier length
- You can configure a custom class for random session id generation.
- Automatically adds the secure flag when https is used.

JEE Session (jsessionid) Con's:

- SessionRotate and SessionInvalidate functions do not rotate the or invalidate the underlying JEE session identifier (because a single J2EE session may hold multiple CF application sessions on the same domain). You can still invalidate and rotate JEE sessions it just takes additional code.
- Settings are server wide for the most part, you cannot configure application specific settings in Application.cfc as you can with CF sessions.

Preventing Session Hijacking

A valid session identifier (CFID/CFTOKEN or jsessionid value) is almost as good as a valid username and password. If the attacker can figure out a current valid session identifier they can impersonate that user.

Because the session identifier is so valuable, it is important to take as many measures as possible to protect it. Here are some steps you can take to protect the session id(s):

- Always use SSL / HTTPS when sessions are used to encrypt the session identifiers in transit. Whenever SSL is used, ensure that the secure attribute is set on the session identifier cookies.
- Avoid appending the session identifiers to the URL query string. End users will email, or publish URLs without realizing their session identifier is in the url. When you use the cflocation tag it will append the session identifier by default, unless you specify addtoken="false".

- Ensure that session identifier cookies use the HttpOnly cookie attribute.
- Set the session timeout as low as possible.
- Call SessionRotate() after a successful login to avoid session fixation attacks.
- Invoke SessionInvalidate() or if using JEE session call getPageContext().getRequest().getSession().invalidate() to invalidate the session upon logout, or upon a potentially malicious request.

Authentication & Authorization

Several steps can be taken to improve security of code that handles authentication of user credentials.

Password Storage

Passwords should not be stored in plain text, instead they should be salted with a random unique string and stored using a *one way function*. ColdFusion provides the Hash function, a one way function which supports multiple algorithms implemented by the Java Cryptography Extension (JCE). The SHA-512 algorithm for example always produces a 128 character hexadecimal result string.

Avoid the use of hash algorithms such as MD5 and SHA1 which have known weaknesses.

ColdFusion 11 introduced the GeneratePBKDFKey function which utilizes the PBKDF2 (password based key derivation function) algorithm. The PBKDF2 algorithm is considered an *adaptive one way function* because it can adapt to increasing CPU speeds by increasing the iterations. Two other *adaptive one way function* algorithms to consider are bcrypt and scrypt, which are not built-in to ColdFusion but can be easily used in CFML via Java libraries. Each algorithm has unique properties which should be researched and evaluated before deciding which one way function is best for your application.

Here is an example using GeneratePBDKFKey to authenticate a user's password:

```
<cfquery name="auth">
    SELECT userID, salt, password FROM users
    WHERE username = <cfqueryparam value="#form.username#">
</cfquery>
<cfset computedPassword = GeneratePBKDFKey("PBKDF2WithSHA512",
form.password, (auth.recordcount ? auth.salt:"nosalt"), 10000, 512)>
<cfif computedPassword IS auth.password AND auth.recordcount>
     <!--- password is correct --->
</cfif>
```

The code example above avoids a timing attack by invoking the GeneratePBKDFKey function regardless the validity of the username. If the GeneratePBDKFKey function was only invoked on valid usernames, the attacker may notice that authentication requests for valid usernames take much longer than requests for invalid usernames. Such an attack allows the attacker to build a list of potentially valid usernames for the system.

Auditing & Logging

Maintaining an audit log of successful and unsuccessful login attempts can be very useful in preventing bruit force authentication attacks.

With an audit log in place you can add logic to your authentication code such as:

- If *X* authentication attempts from *ipaddress* within *Y* seconds, block for *N* minutes.
- If *X* failed authentication attempts from *username* within *Y* seconds, block for *N* minutes.

Take care not to cause a denial of service on authentication of valid users when implementing such blocking.

Multi-factor Authentication

Authentication that relies on passwords alone can be strengthened by providing a second factor of authentication, often referred to as multi factor authentication or two factor authentication. The factors of authentication are typically defined as *something you know*, *something you have* and *something you are*.

Most web applications rely on a single factor authentication, usually *something you know*, a password. Multi-factor authentication requires at least two different factors, the *something you have* factor could be a phone, smart card, or hardware token. *Something you are* could be a biometric factor such as a fingerprint.

One approach to providing two factor authentication involves using a Time Based One Time Password (TOTP) algorithm. Several smart phone applications can be used to generate TOTP passwords in real-time (Google Authenticator for example) to provide the *something you have (your smartphone)* factor.

Remember Me Cookies

Users often like the ability to skip entering their username and password every time they visit your site. Take caution when implementing such a feature, applications that require maximum security should avoid this type of feature.

Ensure that the cookie used to persist authentication follows the guidance outlined in the Cookies section of this guide.

The token stored in the cookie should not be based on the password, it should be generated randomly using a secure random algorithm and of sufficient length. The token should be stored in the database with salt using a one way function.

Requests containing invalid tokens should be audited.

Forgot Password Features

Code that handles forgot password recovery is just as important as code that handles user name and password authentication. Use similar techniques as you would for authentication, such as audit logging to prevent bruit force attacks.

Authorization Controls

The portion of your code that determines if a specific user can view a resource or perform an action falls under authorization controls. Authorization code should be written cautiously to ensure that controls are tight and cannot be bypassed.

One common authorization failure is an insecure reference to a direct object. Here is an example:

```
show-docs.cfm:
<cfquery name="docs">
SELECT id,title FROM docs
WHERE userID = <cfqueryparam value="#session.userID#">
</cfquery>
<cfoutput query="docs">
    <a href="doc.cfm?id=#docs.id#">#encodeForHTML(docs.title)#</a><br
/>
</cfouptut>

doc.cfm:
<cfquery name="doc">
SELECT title, data FROM docs
WHERE id = <cfqueryparam value="#url.id#">
</cfquery>
<cfoutput>
<h1>#encodeForHTML(doc.title)#</h1>
#encodeForHTML(doc.data)#
</cfoutput>
```

In the above example show-docs.cfm properly restricts the document list to those which the user owns, however doc.cfm does not perform any authentication to verify that the requested user is authorized to view the document. This is an oversimplified example, but vulnerabilities such as this can be found commonly when care is not taken.

Role Based Authorization

ColdFusion supports CFC function level role based authentication. This allows you to specify what roles the current authenticated user must have to invoke a function.

In order to use this feature you must invoke the cfloginuser tag inside a cflogin tag. The cflogin tag is invoked only if a user is not already logged in via cfloginuser. For example:

```
<cflogin allowconcurrent="false">
    <cfif StructKeyExists(form, "pass")
     AND StructKeyExists(form, "user")>
      <cfset auth = myAuth.authenticate(form.user, form.pass)>
      <cfif auth.successful>
            <cfloginuser name="#auth.name#"
                  password="#auth.passwordHash#"
                  roles="#auth.roleList#">
      </cfif>
    </cfif>
</cflogin>
```

When the cfloginuser tag is invoked you can leverage a number of CFML functions to determine login state and roles. The IsUserLoggedIn function can be used to determine if a user has been authenticated with cfloginuser.

The name specified in the cfloginuser tag can be retrieved using the GetAuthUser function.

When specifying the password attribute in cfloginuser, don't use the user's actual password, use a one way function.

The list of roles specified in the roles attribute of cfloginuser can be retrieved using the GetUserRoles function. To check if a user is in a single role you can use IsUserInRole(roleName) or to check if a user belongs to any role in a list of roles, the IsUserInAnyRole(roleList) function can be invoked.

ColdFusion 11 added the ability to restrict concurrent logins from the same user by specifying allowconcurrent="false" in the cflogin tag. This feature can be used to discourage sharing user accounts, and may help in the event of a session hijacking attack.

One of the most convenient features of this framework is the ability to specify a role list in a CFC function, for example:

```
<cfcomponent>
    <cffunction name="doAdminStuff" roles="admin,superuser">
         <!--- only runs if user has the admin or superuser role --->
    </cffunction>
</cfcomponent>
```

When using cflogin be sure to set this.loginStorage="session" in Application.cfc, the default loginStorage is a cookie which contains values such as the username and the CF application name.

Upon logout be sure to invoke the cflogout tag, and invalidate the session.

Authentication Resources

- PBKDF2: http://en.wikipedia.org/wiki/PBKDF2
- Bcrypt: http://en.wikipedia.org/wiki/Bcrypt
- Scrypt: http://en.wikipedia.org/wiki/Scrypt
- CFML Implementation of Google Authenticator TOTP for multifactor authentication: https://github.com/marcins/cf-google-authenticator
- CFML Implementation of Duo Security API for multifactor authentication: https://github.com/duosecurity/duo_coldfusion
- OWASP Guide to Authentication: https://www.owasp.org/index.php/Guide_to_Authentication
- OWASP Guide to Authorization: https://www.owasp.org/index.php/Guide_to_Authorization

PDF

The cfhtmltopdf tag, introduced in ColdFusion 11 provides powerful HTML rendering, powered by WebKit to produce PDF files. Because the HTML is rendered by the server, care should be taken when using variables in the PDF document.

All preventative measures that pertain to cross site scripting (see the prior section) also apply to variables written in the cfhtmltopdf tag. JavaScript can be executed during rendering in the cfhtmltopdf tag.

Because the JavaScript would be executed on the server during rendering, the risks are quite different from a client side cross site scripting attack. Some of the risks include denial of service, exploit potential for unknown vulnerabilities in Webkit, and network firewall bypass (because the server may be behind a firewall with network access to other systems.).

Vulnerable Code Example

```
<cfhtmltopdf>
  <h1>Hello #url.name#</h1>
</cfhtmltopdf>
```

Secure Code Example

```
<cfhtmltopdf>
  <h1>Hello #EncodeForHTML(url.name)#</h1>
</cfhtmltopdf>
```

Consider running the PDF Generation service on an isolated dedicated server to mitigate the risks associated with the rendering of dynamic content.

Additional Suggestions to Improve Security

Here are some additional suggestions to improve security of your CFML applications:

- Avoid the use of Evaluate or IIF on untrusted inputs.
- Use access="remote" only when necessary when defining functions in CFCs
- Avoid creating unnamed applications.
- When using client variables with clientStorage=cookie all variables are set inside a cookie and can be modified and read by the user.
- Avoid creating forms with an excessive number of form fields. ColdFusion limits the number of fields to 100 by default to prevent a web application vulnerability known as HashDos. More about HashDos can be found here: http://www.petefreitag.com/item/808.cfm
- When writing variables that output headers in various protocols (HTTP, SMTP, etc.) ensure that CRLF (carriage return \r line feed \n) characters have not been injected. This can lead to response splitting or header injection.
- Filter out <!DOCTYPE> and <!ENTITY> tags when processing XML to avoid XML entity injection attacks.
- When using DeserializeJSON specify strictMapping true (the default) to avoid converting JSON into queries objects.
- Check the number of parameters to avoid HashDos collision vulnerabilities when input data is parsed into a struct whose key names are arbitrary, for example when working with JSON.
- Use the setHTTPMethod JavaScript function in cfajaxproxy calls to force the POST HTTP method when sensitive data is sent in the AJAX request.
- Enable and configure sandbox security in the ColdFusion Administrator using the principle of least privilege.
- Read and follow the recommendations found in the ColdFusion 11 Lockdown Guide.
- Sanitize and validate variables used in XPath expressions to avoid XPath injection
- Sanitize and validate variables used in LDAP queries to avoid LDAP injection.

Additional Resources & References

- ColdFusion 8 Developer Security Guide: http://www.adobe.com/content/dam/Adobe/en/devnet/coldfusion/pdfs/coldfusion_security_cf8.pdf
- Open Web Application Security Project (OWASP): https://www.owasp.org/

Adobe documentation.

Adobe Systems Incorporated, 345 Park Avenue, San Jose, California 95110, USA.