Edit (https://github.com/foundeo/cfdocs/tree/master/guides/en/security-authentication.md)

Authentication Guide

If you missed our Obfuscation (/security-obfuscation) or Encryption (/security-encryption) guides, we strongly recommend that you read those guides first as this guide builds on some of the principles outlined in those guides to build a robust authentication system.

Authentication is the process by which you validate that a particular user is allowed to login to your application. This should not be confused with *access control*, which determines what parts of your application a user has access to. Authentication is merely the process of ensuring a user can login.

Authentication has been around as long as computers have been around. The first widely used version of authentication for web applications was known as HTTP Basic Authentication and was widely used by everyone in the early days of web application development. Unfortunately, HTTP Basic Authentication has some caveats to it that make it a very insecure way of handling authentication in the modern world, not the least of which is the username and password being sent in clear-text on every single request made. This opened the door for MITM (Man In The Middle) attacks in the easrly 1990's and quickly fell out of favor with most web developers.

Since then several 'flavors' of web application authentication have come along that aimed to address the various attack vectors exposed by preceeding authentication mechanisms. We will use many of these authentication mechanisms, layered upon one another, to build a highly secure authentication system that will provide a concise way for you to allow registered users to authenticate with your application.

Federated Authentication

Federated authentication was born to address the need to ensure that a user was logging in from your server and not from someone else's server. Federation helps prevent someone from generating a form on their site that then logs into your site, exposing your authentication to abuse by servers not under your control.

The first step to federation is to ensure that the value of the referrerr HTTP header value matches that of your domain. While this is still a good practice, modern languages can easily fake the referrer in the HTTP header itself, so another method of federation needs to be layered upon this one - federated cookies.

Federated cookies, when layered with the referrer check, provide a more secure login form by requiring that your domain specific cookie is provided to your server on the authentication request. While it is still feasible to provide this cookie from another server, it makes your authentication more robust and can help ward off those scripts that cannot, or does not, provide for these kinds of added protections. This is not a failsafe, it merely adds upon the layers your authentication requires, thus making it more difficult and time consuming to hack.

Disclosure Prevention

One of the drawbacks of authenticating with web applications is that the username and password still need to be sent to the server for authentication. In most cases, this information is still sent in clear-text on the initial request. While this alleviates one of the problems with HTTP Basic Authentication - the need to send the username and password in clear-text on **every** request, it still presents an opportunity for a MITM attack to see the username and password being passed in.

For most web applications being built these days, the 'username' is typically the users email address. Some systems still support, or require, a unique username that is not your email address, but the vast majority of username login's these days are based on your email. Disclosure of a person's email address is commonly considered 'ok' in that a users email address is publicly available information in most cases.

Passwords on the other hand, are always (or should always) be unique to each system a user wishes to authenticate with. Far too often, however, people are lazy or have far too many login's to keep track of manually (and lack any understanding of the availability of applications that manage these passwords for them) and will use the same password... everywhere.

For this reason, steps should be taken to ensure that the plain-text passwords of your users are not sent to the server so that such a password disclosure doesn't lead to a user having all of their accounts exposed and attacked. We will look at one effective way of doing this using hashing in this guide.

Session Cookies

CFML can provide the *cftoken* session cookie management features out of the box and, by default, these are usually turned on in older CFML engines, with CF 10+ allowing for greater control over how those session cookies are managed. You can, and should, however utilize your own session cookie to manage sessions to achieve greater control over the session as well as eliminate a common attack vector against CFML - anyone who has used CFML understands that the *cftoken* cookie is used for session management, and thus understands this is the attack vector to exploit. If you instead utilize your own session cookie, it can help avoid script kiddie utilities designed to attack the specific *cftoken* cookie provided by CFML. We'll start by defining the name of our session cookie within the *application* scope of your *Application.cfc (/application%2Dcfc)*, as follows:

```
// set the name of the cookie to use for session management (*DO NOT USE* cfid, cftoke
n or jsessionid)
application.cookieName = '__some_obfuscated_name';
```

To prevent potential hackers from recognizing your cookie as an authentication cookie, it is best to obfuscate the name of the cookie in some way. For example:

```
__ga_tracking_beacon_
```

Which might confuse the hacker into believing that this is a Google Analytics cookie, and not one specific to authentication. Be creative in naming your cookie and it can, and often will, be readily dismissed by a potential hacker as unrelated to your authentication. You could also simply use the hash() (/hash) algorithm, but this is less likely to confuse a potential hacker than making the cookie appear to be unrelated to your site:

```
__#hash( 'some_cookie_name', 'SHA-256', 'UTF-8', 25 )#
```

The Login View (Page)

We'll start exploring how to build a secure authentication mechanism by examining the login page you should use for your web application. This login page should take advantage of both federation and prevent against password disclosure. Our examples will, again, be using fw/1 to demonstrate ways of handling these tasks, but the same concepts are applicable accross all web applications (even non-CFML ones!).

Let's start by looking at our login() controller. One of the first things we do is to set a zero-value session cookie when requesting the login form. This will be checked during authentication to ensure that our form is being posted from our own server:

```
// set a zero-value session cookie when hitting the login page (federate the login)
getPageContext().getResponse().addHeader("Set-Cookie", "#application.cookieName#=0;pat
h=/;domain=#listFirst( CGI.HTTP_HOST, ':' )#;HTTPOnly");
```

In the above code, we are simply setting a cookie on the request, utilizing our application scoped cookie name, with a zero value.

The next thing you'll typically want to do is to clear any existing session object (bean) from the system for this session to ensure that there are is no lingering session data, which we'll use during session management (/security-session-management):

```
// lock and clear the sessionObj
lock scope='session' timeout='10' {
    session.sessionObj = createObject( 'component', 'model.beans.Session').init();
}
```

Once this is completed, you would add any other processes you need before displaying your login form and then display the form.

Within the login view (page) itself you want to include two hidden fields, as follows:

```
<input type="hidden" id="heartbeat" name="heartbeat" value="#application.securityServi
ce.getHeartbeat()#">
<input type="hidden" name="f#application.securityService.uberHash( 'token', 'SHA-51
2', 150 )#" value="#CSRFGenerateToken( forceNew = true )#">
```

The first field, which we call 'heartbeat' will contain a random alphanumeric value which we will use later to obfuscate the users password **before** it is sent to the server. We'll cover this more in a moment, but it is important to note that this field has been given a DOM id value which we'll need to later process this field.

It is also important to note that we did not assign this to an rc. value within the controller itself because if an attacker knows this value is passed from the controller to the view, it is feasible for them to pass the rc. value in with the request itself and override that value in the controller. It is unlikely, but feasible, and as such we make the call to our random alphanumeric code generator directly within the form itself.

The second field, which is a hashed 'token' utilizes CFML's CSRFGenerateToken() (/csrfgeneratetoken) function, forcing a new key, which we will use along with the built-in CSRFVerifyToken() (/csrfverifytoken) function to help protect the login form from Cross-Site Request Forgery (CSRF) attacks.

We then proceed to provide the username and password fields as we normally would, giving an id to our password field, something along the lines of:

```
<input name="password" id="password" type="password" value="" autocomplete="off" requi
red>
```

We can now use these two form fields, along with a bit of JavaScript, to protect the user's password from disclosure - by not sending the password from the browser to the server in plain-text, we ensure that the users password is protected from that disclosure. By using the randomly generated heartbeat value in connection with our hashing algorithm, which we'll explore below, we ensure that the password cannot be matched to a rainbow table with relative ease.

We'll be using jQuery in our example, along with an SHA-384 JavaScript algorithm from the CryptoJS (http://bit.ly/1tqrgeu) library:

```
<script type="text/javascript">

    $( document ).ready(function() {

        $( '#btnLogin').on( 'click', function( e ) {
            if( $('#password').val().length ) {

                e.preventDefault();

                var hp = $('#password').val();
                var hb = $('#heartbeat').val();

                hp = CryptoJS.SHA384( hp );
                hp = CryptoJS.SHA384( hp + hb );

                $('#password').val( hp );

                $('#loginForm').submit();

            }

        });

    });

</script>
```

In the code above, we add an onClick() listener to the login form's submit button, check to ensure that the password has length, prevent the default action (submitting the form) and then take the following steps:

We hash (/hash) the user's password field value with SHA-384. We then append that hashed value with the value of the heartbeat field, and hash (/hash) them both together. We then replace (/replace) the value of the user's password field with the hashed password + heartbeat value. Then we submit the form.

In doing so, we have obfuscated the users password and prevented disclosure from the browser to the server in plain-text. This has a drawback with some older browsers which will save the hashed password when choosing to 'save my password for this site', but then users really ought not be letting the browser save their passwords anyway. Modern browsers understand what is going on with this code, however, and will rightly save the actual password typed in by the user.

In just a few simple lines of code (which are spelled out here to show the process more thoroughly and could be truncated to further obscure what the JavaScript is intended to do), we have a) federated our login form with a cookie, b) eliminated any lingering session values for that session, c) ensured that the users password is protected from disclosure when being sent to the server from the browser and d) helped protect against CSRF attacks by generating a token.

Authentication

Once the user has entered their username and password, and clicked your 'login' button, their data is sent to your server to be processed. The first thing we want to do is ensure that the page is being submitted to your server from your server, which we do both by checking the referrer, as follows:

```
// check if the host and referrer match (federate the login)
if( !findNoCase( CGI.HTTP_HOST, CGI.HTTP_REFERER ) ) {
    // they don't, redirect to the login page
    variables.fw.redirect( action = 'main.login', queryString = 'msg=503' );
}
```

And next by checking the cookie is provided, as follows:

```
// check if the session cookie exists (federate the login)
if( !structKeyExists( cookie, application.cookieName ) ) {
    // it doesn't, redirect to the login page
    variables.fw.redirect( action = 'main.login', queryString = 'msg=504' );
}
```

In both cases, if either of these checks fails then the user is redirected back to the login view, thus preventing anyone from submitting a form to your server from another location (/location).

We then do a little server-side validation (because we cannot always trust that this request is being sent from a form that contained client-side validation, e.g. a script someone wrote to attempt to hack your authentication), so we make sure both the username and password are provided:

```
// ensure a username and password were sent
if( !len( rc.username ) OR !len( rc.password ) ) {
    // they weren't, redirect to the login page
    variables.fw.redirect( action = 'main.login', queryString = 'msg=500' );
}
```

We then want to ensure that the CSRF token has been provided and matches the token we have stored for this users session:

```
// ensure the CSRF token is provided and valid
if( !structKeyExists( rc, 'f' & application.securityService.uberHash( 'token', 'SHA-51
2', 150 ) ) OR !CSRFVerifyToken( rc[ 'f' & application.securityService.uberHash( 'toke
n', 'SHA-512', 150 ) ] ) ) {
    // it doesn't, redirect to the login page
    variables.fw.redirect( action = 'main.login', queryString = 'msg=505' );
}
```

If we've passed all of these checks, our next step is to get the user's information from storage (typically the database) by passing the provided username (email address in many cases), as follows:

```
// get the user from the database by encrypted username
qGetUser = userService.filter( username = application.securityService.dataEnc( rc.user
name, 'repeatable' ) );
```

If you followed along with all our guides, you may recognize us discussing the need to use repeatable encryption when using the CBC\PKCS5Padding option for encryption.

Next, we check if the provided username even exists in the database:

```
// check if there is a record for the passed username
if( !qGetUser.recordCount ) {
    // there isn't, redirect to the login page
    variables.fw.redirect( action = 'main.login', queryString = 'msg=404' );
}
```

Next we'll typically want to ensure that the user has an active account in the system. We do this with an 'isActive' bit flag in our database which we then check for 'false':

```
// check to be sure this user has an active account
if( !qGetUser.isActive ) {
    // they don't, redirect to the login page
    variables.fw.redirect( action = 'main.login', queryString = 'msg=555' );
}
```

If we've gotten this far in the process, then it's time to decrypt (/decrypt) the stored users (hashed) password and compare (/compare) it against the passed in heartbeat and password form field values, like so:

```
// hash (/hash) the users stored password with the passed heartbeat for comparison
hashedPwd = hash( lcase( application.securityService.dataDec( qGetUser.password, 'db'
) ) & rc.heartbeat, 'SHA-384' );

// compare (/compare) the hashed stored password with the passed password
if( !findNoCase( hashedPwd, rc.password ) ) {
    // they don't match, redirect to the login page
    variables.fw.redirect( action = 'main.login', queryString = 'msg=403' );
}
```

And now, assuming the user has provided the proper password upon login, we generate a session object:

```
// lock the session scope and create a sessionObj for this user
lock scope='session' timeout='10' {
    session.sessionObj = application.securityService.createUserSession(
        userId = qGetUser.userId,
        role = qGetUser.role,
        firstName = application.securityService.dataDec( qGetUser.firstName, 'db' ),
        lastName = application.securityService.dataDec( qGetUser.lastName, 'db' )
    );
}
```

The above is one example of storing useful information in the session object, but is by no means complete and by no means required. You are free to come up with your own session object bean and populate it as needed for your particular application.

We then set a cookie for the user with an encrypted value, which we will then use during session management (/security-session-management) to manage the users session after they have successfully authenticated:

```
// set the session cookie with the new encrypted session id
getPageContext().getResponse().addHeader("Set-Cookie", "#application.cookieName#=#appl
ication.securityService.setSessionIdForCookie( session.sessionObj.getSessionId() )#;pa
th=/;domain=#listFirst( CGI.HTTP_HOST, ':' )#;HTTPOnly");
```

And then we redirect the user to whichever page within our system we consider the 'landing' page for authenticated users:

```
// and go to the dashboard view
variables.fw.redirect( 'main.dashboard' );
```

And presto! A secure authentication has been made and we are now ready to serve content to this authenticated user.

This combination of layered security best practices ensures that any hacker wanting to break into our authentication mechanisms would need to spend a fair amount of time reverse-engineering the process and mimicking both the browser and potential server responses. If a hacker really wants to try and break your authentication, they can still do it but it will take considerable effort on their part. Less capable and less interested hackers will simply move on to easier targets, of which there are many.

All of these concepts can be found in convenient functions in my SecurityService.cfc (http://bit.ly/1IkY5zK) service, main.cfc (http://bit.ly/1S1syAp) controller and default.cfm (http://bit.ly/1XhyJrX) view, which are part of a larger example of using these and other security techniques to create a secure (http://bit.ly/1Msdwkt), or two-factor (http://bit.ly/1Yx4hGt), framework one (FW/1) (http://bit.ly/22lB2eu) application.