cfPassphrase v0.2 Documentation

cfPassphrase is a library for securely hashing and checking passphrases using proven industry standard algorithms.

See the file **readme.md** for project information, requirements, licensing and credits.

This documentation aims to provide everything you need to know to use cfPassphrase - please raise an issue if you find any bugs/errors/omissions, or have any questions.

Hyperlinks that are internal to the PDF are blue and dashed, those to external websites are red and underlined.

Installation

How to install cfPassphrase varies between CFML engines. The latest download packages for each CFML engine can be found on the project's homepage. Once you have installed cfPassphrase, see Usage section for details on how to use it.

With Lucee and OpenBD a single file can be placed in the relevant directory - this is the same process as for any other extension/plugin, but please briefly check the relevant notes for additional information.

For simplicity the filenames are referred to here without a version - i.e. as `cfpassphrase.jar` - but the actual filenames do include versions, (e.g. `cfpassphrase-v0.1.jar`). It is *not* necessary to rename the JAR file (though you can if you prefer).

When upgrading, any previous JAR files should be removed or disabled to prevent conflicts.

ColdFusion

Adobe ColdFusion does not provide an extension API, so it is not possible to provide the built-in functions, but it is possible to make custom tags act as built-in tags, and to provide a global component (with CF9 and above) for use in cfscript.

1. Put `cfpassphrase.jar` in `{cfusion}/lib` directory.
2. Put `passphrase.cfc` in `{cfusion}/CustomTag` directory to enable the global component.
3. Put `passphrase.cfm` in `{cfusion}/wwwroot/WEB-INF/cftags` directory to enable the tag.
4. Restart the server.

Notes:

- For ColdFusion 9 and below, `{cfusion}` is simply `{install-dir}`.
- For ColdFusion 10 and above, `{cfusion}` is `{install-dir}/cfusion`.
- The correct lib directory is the one containing `cfusion.jar`.

OpenBlueDragon

1. Put `cfpassphrase.jar` in `{openbd}/webapps/openbd/WEB-INF/lib` directory.
2. Rename the file to `openbdplugin-cfpassphrase.jar`
3. Restart the server.

Notes:

- The correct lib directory is the one containing `OpenBlueDragon.jar`.
- Older versions of OpenBD may require editing `bluedragon.xml` to identify the plugin. Check the log file in `{openbd}/webapps/openbd/WEB-INF/bluedragon/work` to see if the PluginManager is picking up the plugin.

Lucee Extension Bundle

1. Upload `cfpassphrase.lex` via the "Applications" section in Lucee Administrator, or place it in `{lucee-server}/deploy` directory and wait for Lucee to deploy it.

Notes:

- If you have non-LEX cfPassphrase v0.1 installed, it may take precedence - remove or disable the previous JAR to avoid this.
- The `{lucee-server}` directory is configurable, but is probably in the base directory of your Lucee setup. If not, check the servlet configuration for a `lucee-server-root` parameter to see where it is located.

Lucee / Railo

1. Put `cfpassphrase.jar` in `{railo-lib}` directory.
2. Put `cfpassphrase.fld` in `{railo-server}/context/library/fld` directory to enable functions.
3. Put `cfpassphrase.tld` in `{railo-server}/context/library/tld` directory to enable the tag.
4. Restart the server.

Notes:

- Depending on your install method, `{railo-lib}` is either `{install-dir}/lib/ext` or just `{install-dir}/lib` - look for

`railo.jar` to find the right place.

- The `{railo-server}` directory defaults to `{railo-lib}/railo` but can be configured to another location. Check the servlet configuration for a `railo-server-root` parameter to see if it has moved.

Coldbox Plugin

The cfPassphrase JAR can be used in the Coldbox Framework, where it works the same way on all CFML engines. It does not need separate installation, and will not conflict if you do have it installed. See the Coldbox Plugin page for details on using it.

1. Put `cfpassphrase.jar` in your javaloader lib directory. If you do not have one, you can create a `lib` directory in the application root.
2. Configure the setting `javaloader_libpath` to point to the `lib` directory. This setting is defined in the config file (which defaults to `conf/Coldbox.cfc`).
3. Put `Passphrase.cfc` in your `plugins` directory - again, this defaults to being in the application root, but can be re-configured in the config file.
4. Perform an `fwreinit` to pick-up the new config.

Usage

(Before using cfPassphrase, please make sure you have installed it on each server that it will be used.)

The easiest way to use cfPassphrase is via its functions.

Creating a hash is done like so:

```
<cfset UserHash = PassphraseHash( PassphraseToHash ) />
```

This should be stored as appropriate and when the time comes to check it, retrieve the previously calculated hash and do:

```
<cfif PassphraseCheck( PassphraseToCheck , UserHash ) >
```

This is the simplest way to use cfPassphrase, and is suitable for most cases. It uses the default algorithm (BCrypt) with the default 16 rounds (65536 iterations).

Whilst this is likely *good enough*, for optimal usage you can specify algorithm params based on where it will be used. For details on why and how to do this, see Tuning Algorithm Params.

Using Another Algorithm

cfPassphrase comes with two alternatives to BCrypt, the NIST-approved PBKDF2 and the more modern SCrypt.

There are arguments in favour of all three of these algorithms. To find out more about them and determine which might be right for you, read the Algorithms page.

To use a different algorithm, specify it as the second argument to PassphraseHash:

```
<cfset UserHash = PassphraseHash( PassphraseToHash , 'pbkdf2' ) />

<cfif PassphraseCheck( PassphraseToCheck , UserHash ) >
```

Note the algorithm does not need to be specified for PassphraseCheck since all the algorithms produce mutually exclusive hash signatures.

To discover the algorithm used, as well as parameter values (iterations, etc), use PassphraseInfo:

```
<cfdump var=#PassphraseInfo( UserHash )# />
```

Further details on all the functions are provided in the documentation pages for PassphraseHash, PassphraseCheck and PassphraseInfo.

Using cfPassphrase as a tag.

There is a cfpassphrase tag for compatibility reasons: Adobe does not provide an API for adding functions, so it is the simplest option to work the same way on all CFML engines. (Applications using Coldbox framework can use the Coldbox Plugin.)

To create a hash with the tag:

```
<cfpassphrase variable="UserHash" action="hash" passphrase=#PassphraseToHash# />
```

To check against a stored hash:

```
<cfpassphrase variable="ValidHash" action="check" passphrase=#PassphraseToCheck# hash=#UserHash# />

<cfif ValidHash >
...
```

To get information about a hash:

```
<cfpassphrase variable="HashInfo" action="info" hash=#UserHash# />

<cfdump var=#hashInfo# />
```

All functions have a respective tag action, and the arguments available to the functions are also available as attributes to the tag.

Using cfPassphrase on ColdFusion - CFC method

Since ColdFusion does not have a way to provide custom built-in functions, the next best thing is a component with equivalent methods, allowing cfPassphrase to work in script syntax on CF9 and above.

Once installed, this is used as follows:

```
UserHash = new Passphrase().hash( PassphraseToHash )
```

And similarly to check:

```
if ( new Passphrase().check( PassphraseToCheck , UserHash ) )
```

Whilst this global CFC method can also work on Lucee, Railo and OpenBD, it is unnecessary and not setup with the standard install.

If you are writing code to work across multiple CFML engines, you can do:

```
if ( Server.ColdFusion.ProductName EQ 'ColdFusion Server' )
{
    PassphraseHash  = new Passphrase().hash;
    PassphraseCheck = new Passphrase().check;
    PassphraseInfo  = new Passphrase().info;
}
```

This is basically creating UDFs with the same name as the built-in functions the plugin provides.

A downside of this is that it must be done as part of the template/cfc you will be using the functions in - putting the above code in Application.cfc wont work, (unless using a framework that includes templates as part of onRequest).

Coldbox Plugin

cfPassphrase is available as a Coldbox plugin. This works the same way on all CFML engines and does not need separate installation - i.e. you only need to follow the "Coldbox Plugin" instructions on the installation page and it will be available for use by in the same way as any plugin, e.g. calling `getPlugin("Passphrase",true)` from a controller, or via Wirebox injection.

If you are unsure about using plugins in Coldbox, please see the relevant Coldbox documentation.

For creating a hash use:

```
<cfset UserHash = getPlugin("Passphrase",true).hash( PassphraseToHash ) />
```

When it comes to checking a passphrase, simply retrive the previously calculated hash and do:

```
<cfif getPlugin("Passphrase",true).check( PassphraseToCheck , UserHash ) >
```

As with any Coldbox plugin, you can inject it with Wirebox, for example inside a handler or service component:

```
property name="Passphrase" inject="coldbox:myplugin:Passphrase";
...
UserHash = Passphrase.hash( PassphraseToHash );
```

or, in a function:

```
<cfargument name="Passphrase" inject="coldbox:myplugin:Passphrase" />
...
<cfif Passphrase.check( PassphraseToCheck , UserHash )>
```

For further usage details, read the general Usage page. Documentation is also available for the individual functions: PassphraseHash, PassphraseCheck and PassphraseInfo; note that the plugin methods do not require the "Passphrase" prefix.

Frequently Asked Questions

This FAQ attempts to cover any questions that you might have about cfPassphrase.

If you have a question not covered in this FAQ or elsewhere in the documentation, please use the issue tracker.

What does this do that I can't already do?

The aim of this project is to make passphrase hashing as easy as possible for CFML developers to implement.

This project doesn't do much which can't already be done with createObject Java calls and appropriate CFML code, however it collects everything into a single package which aims to work consistently for every CFML engine.

It is not required to know anything about the algorithms involved, however it is recommend to read at least the Tuning Algorithm Params page.

I already hash my passwords. Why should I use this?

If you already have a working implementation of an industry best practise key derivation function (i.e. BCrypt/PBKDF2/SCrypt), there is no direct additional benefit. However, the cfPassphrase project aims to provide a standardised implementation across all CFML engines and thus will hopefully become something developers are familiar with.

If you simply use the Hash or Encrypt functions it is possible you are storing hashes which are vulnerable to brute force attacks - even though you might be salting them - and it is recommended you use cfPassphrase instead.

What is a passphrase? Can I use this for passwords?

A password is a passphrase. As their names suggest, a password is generally a single word (or jumble of symbols), whilst a passphrase is *multiple* words, making up a secret phrase.

Using passphrases is recommended for two reasons:

A passphrase is longer and usually contains higher entropy, making them harder to guess (and thus more resistant to brute-force attack).

A passphrase can be easier to remember, and it is also possible to write down a single word from a passphrase as a reminder, without compromising the whole phrase.

As with regular passwords, do not use common or predictable phrases. The use of at least one non-dictionary word is recommended; mixing up punctuation and substituting letters can also help.

What algorithms does cfPassphrase use?

The default algorithm used is BCrypt. PBKDF2 and SCrypt are also available.

The Algorithms page goes into more detail.

What algorithm is the best to use?

Ask three different people and you will get four different answers.

If in doubt, stick to the default algorithm (currently BCrypt) - it may or not be the "best" choice, but in almost all cases it will be *good enough*.

There are some people that claim you should not use BCrypt, and will recommend either PBKDF2 or SCrypt instead, but equally there are those that point out why BCrypt is better than PBKDF2, and that SCrypt is currently too new.

Check the Further Reading section on the Algorithms page for assorted views.

Remember: Whilst a strong algorithm is important, it will not protect against badly chosen passwords - educating people to use *at least* eight characters and not to use common words is more important than deciding between these three algorithms.

What about SHA-3 / Keccak / Skein / others?

At the time of writing, I am not aware of any key derivation functions which implement these new hash functions. As and when that occurs they will be likely be added.

However, given that they are new it may be some time before they are recommended for use - a general rule of thumb is to wait at least five years before using a cryptographic algorithm, to see if any weaknesses in it are discovered.

Why is cfPassphrase running so slow?

It is designed to be slow! The purpose of all the algorithms used is to combat brute force attacks, by using operations that are slow and cannot be bypassed.

The default parameters used by cfPassphrase are intended to make it too costly (in terms of time and equipment) for an attacker.

As always, there is a balance between security and usability. If you find cfPassphrase is running too slowly, you can adjust the parameters used to speed it up - but should take care to ensure you don't reduce them too far.

See Tuning Algorithm Params for further information about this.

PassphraseHash

Description

Performs a one-way salted hash on the provided passphrase, using a slow key derivation function, specified by the algorithm and algorithmparams arguments (or the default values).

The resulting hash can then be stored for future authentication using the PassphraseCheck function, or by any other tool which supports the algorithm used.

Returns

A string. A salted hash, either base64 or hex encoded (depending on the algorithm used).

Function syntax

```
PassphraseHash( Passphrase [, Algorithm [, AlgorithmParams ]])
```

Arguments

Name	Type	Default	Description
Passphrase	String	Required	The text to be hashed.
Algorithm	String	bcrypt	The algorithm to use. bcrypt, pbkdf2, scrypt. See Algorithms for details.
AlgorithmParams	Struct	Optional	Params to provide to the algorithm. See Tuning Algorithm Params for details.

Example

```
<cfset User.create( Form.Username , PassphraseHash( Form.Passphrase ) ) />
```

PassphraseCheck

Description

Checks whether the passphrase provided will produce the hash provided, according to the original algorithm and parameters. (These details are encoded within the hash.)

The hashes can be created with the PassphraseHash function, or by any other source which has used one of the supported algorithms.

Returns

A boolean. True if the passphrase passes, false otherwise.

Function syntax

```
PassphraseCheck( Passphrase , Hash [, Algorithm ])
```

Arguments

Name	Type	Default	Description
Passphrase	String	Required	The text to be checked against the hash.
Hash	String	Required	A hash in the format of a supported algorithm.
Algorithm	String	Optional	If unspecified, the algorithm is auto-detected from the hash.

Example

A simplified example showing how PassphraseCheck might be used for logging in:

```
<cfquery name="UserQuery" datasource="UserAuth">
    SELECT Id , Hash
    FROM User
    WHERE Username = <cfqueryparam value="#Form.Username#" />
</cfquery>

<cfif PassphraseCheck( Form.Passphrase , UserQuery.Hash )>
    <cfset User.login(UserQuery.Id) />
<cfelse>
    <cfset User.logFailedLoginAttempt(UserQuery.Id) />
    <cfset Errors.append("Incorrect authentication details.") />
</cfif>
```

PassphraseInfo

Description

Examines the hash provided and returns the information which is encoded within it, including the algorithm, version and iterations.

These details can be used to determine whether a passphrase might need to be re-hashed using a newer algorithm or increased number of iterations.

In addition to the supported algorithms, this function will also identify hashes from common crypt implementations.

Returns

A struct containing information about the hash provided.

Different algorithms provide different keys. All algorithms provide at least the key "Algorithm" containing its name, and "Status" providing a guide to whether this algorithm can/should be used:

- Supported - implemented and no known issues.
- Obsolete - no longer recommended for use.
- Unsupported - not implemented, but not obsolete.

(Note that this value does *not* consider algorithm parameters.)

Function syntax

```
PassphraseInfo( Hash [, Algorithm ])
```

Arguments

Name	Type	Default	Description
Hash	String	Required	A hash in the format of a supported algorithm.
Algorithm	String	Optional	If unspecified, the algorithm is auto-detected from the hash.

Example

The PassphraseInfo function might be used inside a scheduled task that runs occasionally to check for accounts that need to be refreshed:

```
<cfquery name="OldUsers" datasource="UserAuth">
    SELECT Id , Hash
    FROM User
    WHERE LastModified < <cfqueryparam value=#Now()-60# cfsqltype="cf_sql_date" />
    AND   IsStale = 0
</cfquery>

<cfset StaleHashes = [] />

<cfloop query="OldUsers">
    <cfset HashInfo = PassphraseInfo ( OldUsers.Hash ) />

    <cfif HashInfo.Algorithm NEQ CurrentAlgorithm
        OR HashInfo.Iterations < MinCurrentIterations
        >
        <cfset ArrayAppend( StaleHashes , OldUsers.Id ) />
    </cfif>
</cfloop>

<cfif ArrayLen(StaleHashes)>
    <cfquery datasource="UserAuth">
        UPDATE User
        SET IsStale = 1
        WHERE Id IN (<cfqueryparam list value=#ArrayToList(StaleHashes)# cfsqltype="cf_sql_integer />)
    </cfquery>
</cfif>
```

It is not possible to calculate a newer hash without the passphrase, thus you cannot simply update a hash to increase the iterations, and instead need to set a flag to indicate the action must be performed at the next login.

Algorithms

This page gives an overview of the algorithms supported by cfPassphrase, to help you determine if any particular algorithm suits you better.

For information on parameters the different algorithms support and what values you should set them to, see the the Tuning Algorithm Params page.

BCrypt

BCrypt was originally written for use in OpenBSD, and released in 1999 at USENIX, as a replacement for the traditional DES crypt and MD5 crypt hashing functions, both of which have a fixed computation cost.

BCrypt is based on an extension of the Blowfish block cipher named Eksblowfish, and was designed to have an expensive and parametrised computation cost so that it can be incrementally slowed down as hardware speeds increase.

A downside to BCrypt is that it's underlying functions only use the first 72 characters of a passphrase - anything longer is truncated/ignored. To workaround this limitation it is recommended to pass the SHA256 hash of a passphrase to BCrypt, which will ensure its entire length is used.

- Original BCrypt Publication: A Future-Adaptable Password Scheme.
- Wikipedia BCrypt article.
- cfPassphrase uses the jBCrypt implementation.

PBKDF2

PBKDF2 is a standard published by RSA Laboratories, written in 2000 to replace the earlier PBKDF1, as PKCS #5 of their Public-Key Cryptography Standard. (It has also been published as RFC 2898.)

PBKDF2 uses the algorithm HMAC-SHA1, an extension of the SHA-1 hashing function, and, similarly to BCrypt, has an increasable number of iterations to combat advances in hardware speed.

The algorithms used by PBKDF2 require CPU only, with very limited memory, making it possible to use fast GPUs to perform brute force attacks, though this is only an issue for short passwords with limited iterations.

- Wikipedia PBKDF2 article.
- cfPassphrase uses Java's PBKDF2WithHmacSHA1 (PKCS #5 v2.0) implementation.

SCrypt

SCrypt was written for Colin Percival's Tarsnap online backup service and released at BSDCan in 2009. It was created as a replacement for both PBKDF2 and BCrypt, which both have low/fixed memory cost.

SCrypt allows CPU cost and memory cost to be increased, making it more resilient to attacks using GPUs and integrated circuits.

It is considered by some to currently be too new and as yet unproven, though it has been submitted to the IETF, on its way to becoming a standard.

- Original SCrypt publication: Stronger Key Derivation via Sequential Memory-Hard Functions.
- Wikipedia SCrypt article.
- cfPassphrase uses the Java implementation of SCrypt by Will Glozer.

Further Reading

There are numerous different opinions on the different algorithms that can be used for hashing passphrases, and it would be difficult to summarise them all accurately. Instead, below is a list of links with assorted information which you may find useful if you want more information than is provided on this page.

As always, don't trust the opinion of any one source, but read around the subject and see what different people have to say.

OWASP: Password Storage Cheat Sheet
Recommendations from the Open Web Application Security Project.

IT Security Stack Exchange: How to securely hash passwords?
A thorough overview by cryptographer Thomas Pornin.

Coda Hale: How To Safely Store A Password
Programmer Coda Hale advocates using BCrypt over fast hashing functions.

Unlimited Novelty: Don't use bcrypt
Programmer Tony Arcieri advocates using SCrypt over BCrypt.

IT Security Stack Exchange: Do any security experts recommend bcrypt for password storage?
Cryptographer Thomas Pornin's view on why BCrypt is ok to use.

Coding Horror: Speed Hashing
Programmer Jeff Atwood discusses how quickly hashes can be cracked.

Openwall: Password hashing at scale
Presentation slides by security specialist Alexander Peslyak on considerations when using slow hashing for millions of users.

Tuning Algorithm Params

All the algorithms available with cfPassphrase are designed to be tunable - that is, they have parameters which can be modified to ensure the algorithms are not made out of date as hardware power increases.

There is no single value that will work for everyone, though the general rule of thumb is: as high as possible, but not so high that usability suffers.

If you pick parameter values that result in a ten second time to hash, this will be more secure than if it takes one second - but a lot more people will complain if it takes ten seconds to login!

Remember also that slow hashing is not a substitute for good passphrase practise. The ever increasing speed of computing, and existence of specialised hardware, mean that it is important to educate your users about secure passphrases.

BCrypt

Rounds

The BCrypt rounds parameter is an integer which is raised as a power of two to calculate the iterations.

The default 16 rounds results in 2^16, or 65536 iterations.

Specify as many rounds as you can, and never go below 10.

PBKDF2

Iterations

The PBKDF2 iterations parameter can be any positive whole number (it is not required to be a power of two).

The default is 86,000 iterations - more is better. Never go below 10,000.

SaltBytes

This parameter determines the size of the generated salt, in bytes.

The purpose of a salt is to prevent attackers from pre-computing lookup tables. It is recommended to use a salt which is the same length as the hash.

The default value is 24 bytes. It is not recommended to go below 8 bytes.

HashBytes

This parameter determines the number of bytes output by the hash function. It should be at least as large as the output of the key derivation function.

The PBKDF2 default value is 24 bytes.

Notes

PBKDF2 uses the HmacSHA1 function, which outputs 160 bits (20 bytes).

SCrypt

There is limited information available for SCrypt. The values below come from Colin Percival's original paper/presentation - I haven't found clear information on when the latter two parameters should be increased.

The names here come from the Java implementation used and are not referenced in the original paper - it is likely that they will be renamed to be consistent with other implementations.

The higher level parameters maxtime/maxmemfrac/maxmem have yet to be implemented.

CpuCost

This parameter is also referred to as `N`, and is the number of iterations. It must be a power of 2.

The default value is 65536 (2^16). It is recommended to use at least 16384 (2^14).

MemoryCost

This parameter is also referred to as `r` and controls the blocksize for the underlying hash, which affects the relative memory cost.

The default value is 8.

Parallelization

The parallelization factor can be used to fine-tune the relative CPU cost. It is also referred to as `p`.

The default value is 1.

Notes

SCrypt uses a 128-bit salt and a 256-bit derived key from the HmacSHA256 function.