Crypt

Rclone `crypt` remotes encrypt and decrypt other remotes.

A remote of type `crypt` does not access a storage system directly, but instead wraps another remote, which in turn accesses the storage system. This is similar to how alias, union, chunker and a few others work. It makes the usage very flexible, as you can add a layer, in this case an encryption layer, on top of any other backend, even in multiple layers. Rclone's functionality can be used as with any other remote, for example you can mount a crypt remote.

Accessing a storage system through a crypt remote realizes client-side encryption, which makes it safe to keep your data in a location you do not trust will not get compromised. When working against the `crypt` remote, rclone will automatically encrypt (before uploading) and decrypt (after downloading) on your local system as needed on the fly, leaving the data encrypted at rest in the wrapped remote. If you access the storage system using an application other than rclone, or access the wrapped remote directly using rclone, there will not be any encryption/decryption: Downloading existing content will just give you the encrypted (scrambled) format, and anything you upload will *not* become encrypted.

The encryption is a secret-key encryption (also called symmetric key encryption) algorithm, where a password (or pass phrase) is used to generate real encryption key. The password can be supplied by user, or you may chose to let rclone generate one. It will be stored in the configuration file, in a lightly obscured form. If you are in an environment where you are not able to keep your configuration secured, you should add configuration encryption as protection. As long as you have this configuration file, you will be able to decrypt your data. Without the configuration file, as long as you remember the password (or keep it in a safe place), you can re-create the configuration and gain access to the existing data. You may also configure a corresponding remote in a different installation to access the same data. See below for guidance to changing password.

Encryption uses cryptographic salt, to permute the encryption key so that the same string may be encrypted in different ways. When configuring the crypt remote it is optional to enter a salt, or to let rclone generate a unique salt. If omitted, rclone uses a built-in unique string. Normally in cryptography, the salt is stored together with the encrypted content, and do not have to be memorized by the user. This is not the case in rclone, because rclone does not store any additional information on the remotes. Use of custom salt is effectively a second password that must be memorized.

File content encryption is performed using NaCl SecretBox, based on XSalsa20 cipher and Poly1305 for integrity. Names (file- and directory names) are also encrypted by default, but this has some implications and is therefore possible to turned off.

Configuration

Here is an example of how to make a remote called `secret`.

To use `crypt`, first set up the underlying remote. Follow the `rclone config` instructions for the specific backend.

Before configuring the crypt remote, check the underlying remote is working. In this example the underlying remote is called `remote`. We will configure a path `path` within this remote to contain the encrypted content. Anything inside `remote:path` will be encrypted and anything outside will not.

Configure `crypt` using `rclone config`. In this example the `crypt` remote is called `secret`, to differentiate it from the underlying `remote`.

When you are done you can use the crypt remote named `secret` just as you would with any other remote, e.g. `rclone copy D:\docs secret:\docs`, and rclone will encrypt and decrypt as needed on the fly. If you access the wrapped remote `remote:path` directly you will bypass the encryption, and anything you read will be in encrypted form, and anything you write will be undencrypted. To avoid issues it is best to configure a dedicated path for encrypted content, and access it exclusively through a crypt remote.

```
No remotes found - make a new one
n) New remote
s) Set configuration password
q) Quit config
n/s/q> n
name> secret
Type of storage to configure.
Enter a string value. Press Enter for the default ("").
Choose a number from below, or type in your own value
[snip]
XX / Encrypt/Decrypt a remote
   \ "crypt"
[snip]
Storage> crypt
** See help for crypt backend at: https://rclone.org/crypt/ **

Remote to encrypt/decrypt.
Normally should contain a ':' and a path, eg "myremote:path/to/dir",
"myremote:bucket" or maybe "myremote:" (not recommended).
Enter a string value. Press Enter for the default ("").
remote> remote:path
How to encrypt the filenames.
Enter a string value. Press Enter for the default ("standard").
Choose a number from below, or type in your own value
 1 / Encrypt the filenames see the docs for the details.
   \ "standard"
 2 / Very simple filename obfuscation.
   \ "obfuscate"
 3 / Don't encrypt the file names.  Adds a ".bin" extension only.
   \ "off"
filename_encryption>
Option to either encrypt directory names or leave them intact.

NB If filename_encryption is "off" then this option will do nothing.
Enter a boolean value (true or false). Press Enter for the default ("true").
Choose a number from below, or type in your own value
 1 / Encrypt directory names.
   \ "true"
 2 / Don't encrypt directory names, leave them intact.
   \ "false"
directory_name_encryption>
Password or pass phrase for encryption.
y) Yes type in my own password
g) Generate random password
y/g> y
Enter the password:
password:
Confirm the password:
password:
Password or pass phrase for salt. Optional but recommended.
Should be different to the previous password.
```

```
y) Yes type in my own password
g) Generate random password
n) No leave this optional password blank (default)
y/g/n> g
Password strength in bits.
64 is just about memorable
128 is secure
1024 is the maximum
Bits> 128
Your password is: JAsJvRcgR-_veXNfy_sGmQ
Use this password? Please note that an obscured version of this
password (and not the password itself) will be stored under your
configuration file, so keep this generated password in a safe place.
y) Yes (default)
n) No
y/n>
Edit advanced config? (y/n)
y) Yes
n) No (default)
y/n>
Remote config
--------------------
[secret]
type = crypt
remote = remote:path
password = *** ENCRYPTED ***
password2 = *** ENCRYPTED ***
--------------------
y) Yes this is OK (default)
e) Edit this remote
d) Delete this remote
y/e/d>
```

Important The crypt password stored in `rclone.conf` is lightly obscured. That only protects it from cursory inspection. It is not secure unless configuration encryption of `rclone.conf` is specified.

A long passphrase is recommended, or `rclone config` can generate a random one.

The obscured password is created using AES-CTR with a static key. The salt is stored verbatim at the beginning of the obscured password. This static key is shared between all versions of rclone.

If you reconfigure rclone with the same passwords/passphrases elsewhere it will be compatible, but the obscured version will be different due to the different salt.

Rclone does not encrypt

- file length - this can be calculated within 16 bytes
- modification time - used for syncing

Specifying the remote

When configuring the remote to encrypt/decrypt, you may specify any string that rclone accepts as a source/destination of other commands.

The primary use case is to specify the path into an already configured remote (e.g. `remote:path/to/dir` or `remote:bucket`), such that data in a remote untrusted location can be stored encrypted.

You may also specify a local filesystem path, such as `/path/to/dir` on Linux, `C:\path\to\dir` on Windows. By creating a crypt remote pointing to such a local filesystem path, you can use rclone as a utility for pure local file encryption, for example to keep encrypted files on a removable USB drive.

Note: A string which do not contain a `:` will by rclone be treated as a relative path in the local filesystem. For example, if you enter the name `remote` without the trailing `:`, it will be treated as a subdirectory of the current directory with name "remote".

If a path `remote:path/to/dir` is specified, rclone stores encrypted files in `path/to/dir` on the remote. With file name encryption, files saved to `secret:subdir/subfile` are stored in the unencrypted path `path/to/dir` but the `subdir/subpath` element is encrypted.

The path you specify does not have to exist, rclone will create it when needed.

If you intend to use the wrapped remote both directly for keeping unencrypted content, as well as through a crypt remote for encrypted content, it is recommended to point the crypt remote to a separate directory within the wrapped remote. If you use a bucket based storage system (e.g. Swift, S3, Google Compute Storage, B2, Hubic) it is generally advisable to wrap the crypt remote around a specific bucket (`s3:bucket`). If wrapping around the entire root of the storage (`s3:`), and use the optional file name encryption, rclone will encrypt the bucket name.

Changing password

Should the password, or the configuration file containing a lightly obscured form of the password, be compromised, you need to re-encrypt your data with a new password. Since rclone uses secret-key encryption, where the encryption key is generated directly from the password kept on the client, it is not possible to change the password/key of already encrypted content. Just changing the password configured for an existing crypt remote means you will no longer able to decrypt any of the previously encrypted content. The only possibility is to re-upload everything via a crypt remote configured with your new password.

Depending on the size of your data, your bandwith, storage quota etc, there are different approaches you can take:

- If you have everything in a different location, for example on your local system, you could remove all of the prior encrypted files, change the password for your configured crypt remote (or delete and re-create the crypt configuration), and then re-upload everything from the alternative location.
- If you have enough space on the storage system you can create a new crypt remote pointing to a separate directory on the same backend, and then use rclone to copy everything from the original crypt remote to the new, effectively decrypting everything on the fly using the old password and re-encrypting using the new password. When done, delete the original crypt remote directory and finally the rclone crypt configuration with the old password. All data will be streamed from the storage system and back, so you will get half the bandwith and be charged twice if you have upload and download quota on the storage system.

Note: A security problem related to the random password generator was fixed in rclone version 1.53.3 (released 2020-11-19). Passwords generated by rclone config in version 1.49.0 (released 2019-08-26) to 1.53.2 (released 2020-10-26) are not considered secure and should be changed. If you made up your own password, or used rclone version older than 1.49.0 or newer than 1.53.2 to generate it, you are *not* affected by this issue. See issue #4783 for more details, and a tool you can use to check if you are affected.

Example

Create the following file structure using "standard" file name encryption.

```
plaintext/
├── file0.txt
├── file1.txt
└── subdir
    ├── file2.txt
    ├── file3.txt
    └── subsubdir
        └── file4.txt
```

Copy these to the remote, and list them

```
$ rclone -q copy plaintext secret:
$ rclone -q ls secret:
        7 file1.txt
        6 file0.txt
        8 subdir/file2.txt
       10 subdir/subsubdir/file4.txt
        9 subdir/file3.txt
```

The crypt remote looks like

```
$ rclone -q ls remote:path
       55 hagjclgavj2mbiqm6u6cnjjqcg
       54 v05749mltvv1tf4onltun46gls
       57 86vhrsv86mpbtd3a0akjuqslj8/dlj7fkq4kdq72emafg7a7s41uo
       58 86vhrsv86mpbtd3a0akjuqslj8/7uu829995du6o42n32otfhjqp4/b9pausrfansjth5ob3jkdqd4lc
       56 86vhrsv86mpbtd3a0akjuqslj8/8njh1sk437gttmep3p70g81aps
```

The directory structure is preserved

```
$ rclone -q ls secret:subdir
        8 file2.txt
        9 file3.txt
       10 subsubdir/file4.txt
```

Without file name encryption `.bin` extensions are added to underlying names. This prevents the cloud provider attempting to interpret file content.

```
$ rclone -q ls remote:path
       54 file0.txt.bin
       57 subdir/file3.txt.bin
       56 subdir/file2.txt.bin
       58 subdir/subsubdir/file4.txt.bin
       55 file1.txt.bin
```

File name encryption modes

Off

- doesn't hide file names or directory structure
- allows for longer file names (~246 characters)
- can use sub paths and copy single files

Standard

- file names encrypted
- file names can't be as long (~143 characters)
- can use sub paths and copy single files
- directory structure visible
- identical files names will have identical uploaded names
- can use shortcuts to shorten the directory recursion

Obfuscation

This is a simple "rotate" of the filename, with each file having a rot distance based on the filename. Rclone stores the distance at the beginning of the filename. A file called "hello" may become "53.jgnnq".

Obfuscation is not a strong encryption of filenames, but hinders automated scanning tools picking up on filename patterns. It is an intermediate between "off" and "standard" which allows for longer path segment names.

There is a possibility with some unicode based filenames that the obfuscation is weak and may map lower case characters to upper case equivalents.

Obfuscation cannot be relied upon for strong protection.

- file names very lightly obfuscated
- file names can be longer than standard encryption
- can use sub paths and copy single files
- directory structure visible
- identical files names will have identical uploaded names

Cloud storage systems have limits on file name length and total path length which rclone is more likely to breach using "Standard" file name encryption. Where file names are less thn 156 characters in length issues should not be encountered, irrespective of cloud storage provider.

An alternative, future rclone file name encryption mode may tolerate backend provider path length limits.

Directory name encryption

Crypt offers the option of encrypting dir names or leaving them intact. There are two options:

True

Encrypts the whole file path including directory names Example: `1/12/123.txt` is encrypted to`p0e52nreeaj0a5ea7s64m4j72s/l42g6771hnv3an9cgc8cr2n1ng/qgm4avr35m5loi1th53ato71v0`

False

Only encrypts file names, skips directory names Example: `1/12/123.txt` is encrypted to `1/12/qgm4avr35m5loi1th53ato71v0`

Modified time and hashes

Crypt stores modification times using the underlying remote so support depends on that.

Hashes are not stored for crypt. However the data integrity is protected by an extremely strong crypto authenticator.

Use the `rclone cryptcheck` command to check the integrity of a crypted remote instead of `rclone check` which can't check the checksums properly.

Standard Options

Here are the standard options specific to crypt (Encrypt/Decrypt a remote).

--crypt-remote

Remote to encrypt/decrypt. Normally should contain a ':' and a path, e.g. "myremote:path/to/dir", "myremote:bucket" or maybe "myremote:" (not recommended).

- Config: remote
- Env Var: RCLONE_CRYPT_REMOTE
- Type: string
- Default: ""

--crypt-filename-encryption

How to encrypt the filenames.

- Config: filename_encryption
- Env Var: RCLONE_CRYPT_FILENAME_ENCRYPTION
- Type: string
- Default: "standard"
- Examples:
    - "standard"
        - Encrypt the filenames see the docs for the details.

    - "obfuscate"
        - Very simple filename obfuscation.
    - "off"
        - Don't encrypt the file names. Adds a ".bin" extension only.

--crypt-directory-name-encryption

Option to either encrypt directory names or leave them intact.

NB If filename_encryption is "off" then this option will do nothing.

- Config: directory_name_encryption
- Env Var: RCLONE_CRYPT_DIRECTORY_NAME_ENCRYPTION
- Type: bool
- Default: true
- Examples:
    - "true"
        - Encrypt directory names.
    - "false"
        - Don't encrypt directory names, leave them intact.

--crypt-password

Password or pass phrase for encryption.

NB Input to this must be obscured - see rclone obscure.

- Config: password
- Env Var: RCLONE_CRYPT_PASSWORD
- Type: string
- Default: ""

--crypt-password2

Password or pass phrase for salt. Optional but recommended. Should be different to the previous password.

NB Input to this must be obscured - see rclone obscure.

- Config: password2
- Env Var: RCLONE_CRYPT_PASSWORD2
- Type: string
- Default: ""

Advanced Options

Here are the advanced options specific to crypt (Encrypt/Decrypt a remote).

--crypt-server-side-across-configs

Allow server-side operations (e.g. copy) to work across different crypt configs.

Normally this option is not what you want, but if you have two crypts pointing to the same backend you can use it.

This can be used, for example, to change file name encryption type without re-uploading all the data. Just make two crypt backends pointing to two different directories with the single changed parameter and use rclone move to move the files between the crypt remotes.

- Config: server_side_across_configs
- Env Var: RCLONE_CRYPT_SERVER_SIDE_ACROSS_CONFIGS
- Type: bool
- Default: false

--crypt-show-mapping

For all files listed show how the names encrypt.

If this flag is set then for each file that the remote is asked to list, it will log (at level INFO) a line stating the decrypted file name and the encrypted file name.

This is so you can work out which encrypted names are which decrypted names just in case you need to do something with the encrypted file names, or for debugging purposes.

- Config: show_mapping
- Env Var: RCLONE_CRYPT_SHOW_MAPPING
- Type: bool
- Default: false

Backend commands

Here are the commands specific to the crypt backend.

Run them with

```
rclone backend COMMAND remote:
```

The help below will explain what arguments each command takes.

See the "rclone backend" command for more info on how to pass options and arguments.

These can be run on a running backend using the rc command backend/command.

encode

Encode the given filename(s)

```
rclone backend encode remote: [options] [<arguments>+]
```

This encodes the filenames given as arguments returning a list of strings of the encoded results.

Usage Example:

```
rclone backend encode crypt: file1 [file2...]
rclone rc backend/command command=encode fs=crypt: file1 [file2...]
```

decode

Decode the given filename(s)

```
rclone backend decode remote: [options] [<arguments>+]
```

This decodes the filenames given as arguments returning a list of strings of the decoded results. It will return an error if any of the inputs are invalid.

Usage Example:

```
rclone backend decode crypt: encryptedfile1 [encryptedfile2...]
rclone rc backend/command command=decode fs=crypt: encryptedfile1 [encryptedfile2...]
```

Backing up a crypted remote

If you wish to backup a crypted remote, it is recommended that you use `rclone sync` on the encrypted files, and make sure the passwords are the same in the new encrypted remote.

This will have the following advantages

- `rclone sync` will check the checksums while copying
- you can use `rclone check` between the encrypted remotes
- you don't decrypt and encrypt unnecessarily

For example, let's say you have your original remote at `remote:` with the encrypted version at `eremote:` with path `remote:crypt`. You would then set up the new remote `remote2:` and then the encrypted version `eremote2:` with path `remote2:crypt` using the same passwords as `eremote:`.

To sync the two remotes you would do

```
rclone sync -i remote:crypt remote2:crypt
```

And to check the integrity you would do

```
rclone check remote:crypt remote2:crypt
```

File formats

File encryption

Files are encrypted 1:1 source file to destination object. The file has a header and is divided into chunks.

Header

- 8 bytes magic string `RCLONE\x00\x00`
- 24 bytes Nonce (IV)

The initial nonce is generated from the operating systems crypto strong random number generator. The nonce is incremented for each chunk read making sure each nonce is unique for each block written. The chance of a nonce being re-used is minuscule. If you wrote an exabyte of data (10^{18} bytes) you would have a probability of approximately 2×10^{-32} of re-using a nonce.

Chunk

Each chunk will contain 64kB of data, except for the last one which may have less data. The data chunk is in standard NaCl SecretBox format. SecretBox uses XSalsa20 and Poly1305 to encrypt and authenticate messages.

Each chunk contains:

- 16 Bytes of Poly1305 authenticator
- 1 - 65536 bytes XSalsa20 encrypted data

64k chunk size was chosen as the best performing chunk size (the authenticator takes too much time below this and the performance drops off due to cache effects above this). Note that these chunks are buffered in memory so they can't be too big.

This uses a 32 byte (256 bit key) key derived from the user password.

Examples

1 byte file will encrypt to

- 32 bytes header
- 17 bytes data chunk

49 bytes total

1MB (1048576 bytes) file will encrypt to

- 32 bytes header
- 16 chunks of 65568 bytes

1049120 bytes total (a 0.05% overhead). This is the overhead for big files.

Name encryption

File names are encrypted segment by segment - the path is broken up into `/` separated strings and these are encrypted individually.

File segments are padded using PKCS#7 to a multiple of 16 bytes before encryption.

They are then encrypted with EME using AES with 256 bit key. EME (ECB-Mix-ECB) is a wide-block encryption mode presented in the 2003 paper "A Parallelizable Enciphering Mode" by Halevi and Rogaway.

This makes for deterministic encryption which is what we want - the same filename must encrypt to the same thing otherwise we can't find it on the cloud storage system.

This means that

- filenames with the same name will encrypt the same
- filenames which start the same won't have a common prefix

This uses a 32 byte key (256 bits) and a 16 byte (128 bits) IV both of which are derived from the user password.

After encryption they are written out using a modified version of standard `base32` encoding as described in RFC4648. The standard encoding is modified in two ways:

- it becomes lower case (no-one likes upper case filenames!)
- we strip the padding character `=`

`base32` is used rather than the more efficient `base64` so rclone can be used on case insensitive remotes (e.g. Windows, Amazon Drive).

Key derivation

Rclone uses `scrypt` with parameters `N=16384, r=8, p=1` with an optional user supplied salt (password2) to derive the 32+32+16 = 80 bytes of key material required. If the user doesn't supply a salt then rclone uses an internal one.

`scrypt` makes it impractical to mount a dictionary attack on rclone encrypted data. For full protection against this you should always use a salt.

SEE ALSO

- rclone cryptdecode - Show forward/reverse mapping of encrypted filenames