Crypt

Originalartikel

Backup

<html> The

```
crypt
```

remote encrypts and decrypts another remote. To use it first set up the underlying remote following the config instructions for that remote. You can also use a local pathname instead of a remote which will encrypt and decrypt from that directory which might be useful for encrypting onto a USB stick for example. First check your chosen remote is working - we’ll call it

```
remote:path
```

in these docs. Note that anything inside

```
remote:path
```

will be encrypted and anything outside won’t. This means that if you are using a bucket based remote (eg S3, B2, swift) then you should probably put the bucket in the remote

```
s3:bucket
```

. If you just use

```
s3:
```

then rclone will make encrypted bucket names too (if using file name encryption) which may or may not be what you want. Now configure

```
crypt
```

using

```
rclone config
```

. We will call this one

```
secret
```

to differentiate it from the

```
remote
```

. <pre>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. Choose a number from below,

or type in your own value 1 / Amazon Drive

```
\ "amazon cloud drive"
```

2 / Amazon S3 (also Dreamhost, Ceph, Minio)

```
\ "s3"
```

3 / Backblaze B2

```
\ "b2"
```

4 / Dropbox

```
\ "dropbox"
```

5 / Encrypt/Decrypt a remote

```
\ "crypt"
```

6 / Google Cloud Storage (this is not Google Drive)

```
\ "google cloud storage"
```

7 / Google Drive

```
\ "drive"
```

8 / Hubic

```
\ "hubic"
```

9 / Local Disk

```
\ "local"
```

10 / Microsoft OneDrive

```
\ "onedrive"
```

11 / Openstack Swift (Rackspace Cloud Files, Memset Memstore, OVH)

```
\ "swift"
```

12 / SSH/SFTP Connection

```
\ "sftp"
```

13 / Yandex Disk

```
\ "yandex"
```

Storage> 5 Remote to encrypt/decrypt. Normally should contain a ':' and a path, eg „myremote:path/to/dir“, „myremote:bucket“ or maybe „myremote:“ (not recommended). remote> remote:path How to encrypt the filenames. Choose a number from below, or type in your own value 1 / Don't encrypt the file names. Adds a „.bin“ extension only.

```
\ "off"
```

2 / Encrypt the filenames see the docs for the details.

```
\ "standard"
```

3 / Very simple filename obfuscation.

```
\ "obfuscate"
```

filename_encryption> 2 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 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? y) Yes n) No y/n> y Remote config

---

[secret] remote = remote:path filename_encryption = standard password = ***** **ENCRYPTED** ***** password2 = ***** **ENCRYPTED** *****

---

y) Yes this is OK e) Edit this remote d) Delete this remote y/e/d> y </pre> Important The password is stored in the config file is lightly obscured so it isn’t immediately obvious what it is. It is in no way secure unless you use config file encryption. A long passphrase is recommended, or you can use a random one. Note that if you reconfigure rclone with the same passwords/passphrases elsewhere it will be compatible - all the secrets used are derived from those two passwords/passphrases. Note that rclone does not encrypt file length - this can be calcuated within 16 bytes modification time - used for syncing <h2 id=„specifying-the-remote“>Specifying the remote</h2> In normal use, make sure the remote has a

```
:
```

in. If you specify the remote without a

```
:
```

then rclone will use a local directory of that name. So if you use a remote of

```
/path/to/secret/files
```

then rclone will encrypt stuff to that directory. If you use a remote of

```
name
```

then rclone will put files in a directory called

```
name
```

in the current directory. If you specify the remote as

```
remote:path/to/dir
```

then rclone will store encrypted files in

```
path/to/dir
```

on the remote. If you are using file name encryption, then when you save files to

```
secret:subdir/subfile
```

this will store them in the unencrypted path

```
path/to/dir
```

but the

```
subdir/subpath
```

bit will be encrypted. Note that unless you want encrypted bucket names (which are difficult to manage because you won’t know what directory they represent in web interfaces etc), you should probably specify a bucket, eg

```
remote:secretbucket
```

when using bucket based remotes such as S3, Swift, Hubic, B2, GCS. <h2 id=„example“>Example</h2> To test I made a little directory of files using “standard” file name encryption. <pre>plaintext/ ├── file0.txt ├── file1.txt └── subdir

```
  &#9500;&#9472;&#9472; file2.txt
  &#9500;&#9472;&#9472; file3.txt
  &#9492;&#9472;&#9472; subsubdir
      &#9492;&#9472;&#9472; file4.txt
```

</pre> Copy these to the remote and list them back <pre>$ 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
```

</pre> Now see what that looked like when encrypted <pre>$ rclone -q ls remote:path

```
     55 hagjclgavj2mbiqm6u6cnjjqcg
     54 v05749mltvv1tf4onltun46gls
     57 86vhrsv86mpbtd3a0akjuqslj8/dlj7fkq4kdq72emafg7a7s41uo
     58
86vhrsv86mpbtd3a0akjuqslj8/7uu829995du6o42n32otfhjqp4/b9pausrfansjth5ob3jkdq
d4lc
     56 86vhrsv86mpbtd3a0akjuqslj8/8njh1sk437gttmep3p70g81aps
```

</pre> Note that this retains the directory structure which means you can do this <pre>$ rclone -q ls secret:subdir

```
      8 file2.txt
      9 file3.txt
     10 subsubdir/file4.txt
```

</pre> If don’t use file name encryption then the remote will look like this - note the

```
.bin
```

extensions added to prevent the cloud provider attempting to interpret the data. <pre>$ 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
```

</pre> <h3 id=„file-name-encryption-modes“>File name encryption modes</h3> Here are some of the features of the 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 (~156 characters) can use sub paths and copy single files directory structure visibile 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. We store the distance at the beginning of the filename. So a file called “hello” may become “53.jgnnq” This is not a strong encryption of filenames, but it may stop automated scanning tools from picking up on filename patterns. As such it’s an intermediate between “off” and “standard”. The advantage is that it 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. You can not rely on this 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 visibile identical files names will have identical uploaded names Cloud storage systems have various limits on file name length and total path length which you are more likely to hit using “Standard” file name encryption. If you keep your file names to below 156 characters in length then you should be OK on all providers. There may be an even more secure file name encryption mode in the future which will address the long file name problem. 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. Note that you should use the

```
rclone cryptcheck
```

command to check the integrity of a crypted remote instead of

```
rclone check
```

which can’t check the checksums properly. <h3 id=„specific-options“>Specific options</h3> Here are the command line options specific to this cloud storage system. <h4 id=„crypt-show-mapping“>–crypt-show-mapping</h4> 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. <h2 id=„backing-up-a-crypted-remote“>Backing up a crypted remote</h2> If you wish to backup a crypted remote, it it 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 unecessarily 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 <pre>rclone sync remote:crypt remote2:crypt </pre> And to check the integrity you would do <pre>rclone check remote:crypt remote2:crypt </pre> <h2 id=„file-formats“>File formats</h2> <h3 id=„file-encryption“>File encryption</h3> Files are encrypted 1:1 source file to destination object. The file has a header and is divided into chunks. 8 bytes magic string

```
RCLONE\x00\x00
```

 24 bytes Nonce (IV) The initial nonce is generated from the operating systems crypto strong random number genrator. 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 miniscule. If you wrote an exabyte of data (10¹⁸ bytes) you would have a probability of approximately 2×10⁻³² of re-using a nonce. <h4 id=„chunk“>Chunk</h4> 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. <h4 id=„examples“>Examples</h4> 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. <h3 id=„name-encryption“>Name encryption</h3> 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 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 determinstic 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 (eg Windows, Amazon Drive). <h3 id=„key-derivation“>Key derivation</h3> Rclone uses

```
scrypt
```

with parameters

```
N=16384, r=8, p=1
```

with a 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 agains this you should always use a salt. </html>