Systems, Tools, and Terminal Science

BLOG ARCHIVES

GNU/Linux Crypto: Passwords

Posted on

It's now becoming more widely known that using guessable passwords or using the same password for more than one account is a serious security risk, because an attacker able to control one account (such as an email account) can do a lot of damage. If an attacker gets the hash of your password from some web service, you want to be assured that the hash will be very difficult to reverse, and even if it can be reversed, that it's unique and won't give them access to any of your other accounts.

This growing awareness has contributed to the popularity of **password managers**, tools designed to securely generate, store, and retrieve passwords, encrypted with a master password or passphrase. In some cases these are locally stored, such as , and in others they are stored on a web service, such as . Both are good tools, and work well with GNU/Linux. I personally have some reservations about LastPass as I don't want my passwords stored on a third party service, and I .

Interestingly, because we now have a tidy GnuPG setup to handle the encryption ourselves, another option is the `pass(1)` tool, billing itself as "the standard UNIX password manager". It's little more than a shell script and some `bash(1)` completions wrapped around existing tools like `git(1)`, `gpg2(1)`, `pwgen(1)`, `tree(1)`, and `xclip(1)`, and your choice of `$EDITOR`. If you're

not already invested in an existing password management method, you might find this a good first application of your new cryptography setup, and a great minimal approach to secure password storage accessible from the command line (and therefore SSH).

On Debian-derived systems, it's available as part of the `pass` package:

```
# apt-get install pass
```

This includes a manual:

```
$ man pass
```

Instructions for installing on other operating systems are also available on the site. Releases are also available for download, and a link to the . If you use this, make sure you have the required tools outlined above installed as well, although `xclip(1)` is only needed if you run the X Windows system.

We can get an overview of what `pass(1)` can do by invoking it with no arguments:

```
$ pass
```

To start, we'll initialize our password store. For your own passwords, you will want to do this as your own user rather than `root`. Because `pass(1)` uses GnuPG for its encryption, we also need to tell it the ID of the appropriate key to use. Remember, you can find this eight-digit hex code by typing `gpg --list-secret-keys`. A unique string identifying your private key such as your name or email address may also work.

```
$ pass init 0x77BB8872
mkdir: created directory ‘/home/tom/.password-store’
Password store initialized for 0x77BB8872.
```

Indeed, we note the directory `~/.password-store` has been created, although it's presently empty except for the `.gpg-id` file recording our key ID:

```
$ find .password-store
.password-store
.password-store/.gpg-id
```

We'll insert an existing password of ours with `pass insert`, giving it a descriptive hierarchical name:

```
$ pass insert google.com/gmail/example@gmail.com
mkdir: created directory ‘/home/tom/.password-store/google.com’
mkdir: created directory ‘/home/tom/.password-store/google.com/gmail’
Enter password for google.com/gmail/example@gmail.com:
Retype password for google.com/gmail/example@gmail.com:
```

The password is read from the command line, encrypted, and placed in `~/.password-store`:

```
$ find .password-store
.password-store
.password-store/google.com
.password-store/google.com/gmail
.password-store/google.com/gmail/example@gmail.com.gpg
.password-store/.gpg-id
```

Notice that `pass(1)` creates a directory structure for us automatically. We can get a nice view of the password store with `pass` with no arguments:

```
$ pass
Password Store
```

```
└── google.com
    └── gmail
        └── example@gmail.com
```

If you'd like it to generate a new secure random password for you, you can use `generate` instead, including a password length as the last argument:

```
$ pass generate google.com/gmail/example@gmail.com 16
The generated password to google.com/gmail/example@gmail.com is:
!Q%i$$&q1+JJi-|X
```

If you have some service that doesn't cooperate with symbols in passwords, you can add the `-n` option to this call:

```
$ pass generate -n google.com/gmail/example@gmail.com 16
The generated password to google.com/gmail/example@gmail.com is:
pJeF18CrZEZzI59D
```

`pass(1)` uses `pwgen(1)` for this password generation. In each case, the password is automatically inserted into the password store for you.

If we need to change an existing password, we can either overwrite it with `insert` again, or use the `edit` operation to invoke our choice of `$EDITOR`:

```
$ pass edit google.com/gmail/example@gmail.com
```

If you do this, you may like to be careful that your editor is not configured to keep backups or swap files in plain text of documents it edits in temporary directories or memory filesystems. If you're using Vim, I in an attempt to solve this problem.

Note that adding or overwriting passwords does not require your passphrase; only retrieval and editing does, consistent with how GnuPG normally works.

This password can now be retrieved and echoed onto the command line given the appropriate passphrase:

```
$ pass google.com/gmail/example@gmail.com
(...gpg-agent pinentry prompt...)
Tr0ub4dor&3
```

If you're using X windows and have `xclip(1)` installed, you can put the password on the clipboard temporarily to paste into web forms:

```
$ pass -c google.com/gmail/example@gmail.com
Copied google.com/gmail/example@gmail.com to clipboard. Will clear in 45 seconds.
```

In each case, note that if you have the bash completion installed and working, you should be able to complete the full path to the passwords with `Tab`, just as if you were directly browsing a directory hierarchy.

If we no longer need the password, we can remove it with `pass rm`:

```
$ pass rm google.com/gmail/example@gmail.com
Are you sure you would like to delete google.com/gmail/example@gmail.com? [y/N] y

removed ‘/home/tom/.password-store/google.com/gmail/example@gmail.com.gpg’
```

We can delete whole directories of passwords with `pass rm -r`:

```
$ pass rm -r google.com
Are you sure you would like to delete google.com? [y/N] y
removed ‘/home/tom/.password-store/google.com/gmail/example@gmail.com.gpg’
removed directory: ‘/home/tom/.password-store/google.com/gmail’
removed directory: ‘/home/tom/.password-store/google.com’
```

To keep historical passwords, including deleted ones if we find we do need them again one day, we can set up some automatic **version control** on the directory with `pass git init`:

```
$ pass git init
Initialized empty Git repository in /home/tom/.password-store/.git/
[master (root-commit) 0ebb933] Added current contents of password store.
 1 file changed, 1 insertion(+)
 create mode 100644 .gpg-id
```

This will update the repository every time the password store is changed, meaning we can be confident we’ll be able to retrieve old passwords we’ve replaced or deleted:

```
$ pass insert google.com/gmail/newexample@gmail.com
mkdir: created directory ‘/home/tom/.password-store/google.com’
mkdir: created directory ‘/home/tom/.password-store/google.com/gmail’
Enter password for google.com/gmail/newexample@gmail.com:
Retype password for google.com/gmail/newexample@gmail.com:
[master 00971b6] Added given password for google.com/gmail/newexample@gmail.com to store.
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 google.com/gmail/newexample@gmail.com.gpg
```

Because the password files are all encrypted only to your GnuPG key, you can relatively safely back up the store on remote and third-party sites simply by copying the `~/.password-store` directory. If the filenames themselves contain sensitive information, such as private usernames or sites, you might like to back up an encrypted tarball of the store instead:

```
$ tar -cz .password-store \
    | gpg --sign --encrypt -r 0x77BB8872 \
    > password-store-backup.tar.gz.gpg
```

This directory can be restored in a similar way:

```
$ gpg --decrypt \
    < password-store-backup.tar.gz.gpg \
    | tar -xz
```

This entry is part 6 of 10 in the series .

Posted in | Tagged , , , , ,

GNU/Linux Crypto: Email

Posted on

An encrypted password storage is well and good, but now that we have a working GnuPG setup, we should consider using PGP for what it was originally designed: email messages. To do this, we'll be using .

Mutt is a console-based mail user agent, or MUA, designed chiefly for managing and reading mail. Unlike mailer programs like , it was not designed to be a POP3/IMAP client, or an SMTP agent, although versions in recent years do include this functionality; these are tasks done by programs like and .

If like many people, you're using Gmail, this works very well with POP3/IMAP and SMTP, enabling you to compose email messages in plain text with your choice editor in a terminal window, in a highly configurable environment, and doing your own email encryption

for any sensitive communications in such a way that even your email provider can't read it.

General usage of Mutt and setup for Gmail users is not covered in detail here, although it may be the subject of a later article. For now, there are many on the basics of a Mutt setup. If you're interested in the setup for other GNU/Linux mail clients like or Thunderbird, Cory Sadowski has a walking you through that, among other privacy settings relevant to both GNU/Linux and Windows.

All of the below is assuming you already have a ready, with `gpg-agent(1)` running in the background to manage your keys.

Most of the you can find online are quite old, and they usually suggest a lot of lines of `.muttrc` configuration to interface directly with the `gpg` command, with a myriad of options and some byzantine variable substitution:

```
set pgp_clearsign_command="gpg --no-verbose --batch --output - ...
set pgp_decode_command="gpg %?p?--passphrase-fd 0? --no-verbose ...
set pgp_decrypt_command="gpg --passphrase-fd 0 --no-verbose --batch ...
set pgp_encrypt_only_command="pgpewrap gpg --batch --quiet ...
set pgp_encrypt_sign_command="pgpewrap gpg --passphrase-fd 0 ...
set pgp_export_command="gpg --no-verbose --export --armor %r"
set pgp_import_command="gpg --no-verbose --import -v %f"
set pgp_list_pubring_command="gpg --no-verbose --batch --with-colons ...
set pgp_list_secring_command="gpg --no-verbose --batch --with-colons ...
set pgp_sign_command="gpg --no-verbose --batch --output - ...
set pgp_verify_command="gpg --no-verbose --batch --output - --verify %s %f"
set pgp_verify_key_command="gpg --no-verbose --batch --fingerprint ...
```

I'm all for the Unix philosophy of using programs together, but this is just too much. It's a fickle setup that's very hard to work with, and it requires too much understanding of the `gpg(1)` frontend to use and edit sensibly. After all, we want to end up with a setup that we understand reasonably well.

So, throw all that away; we're going to use instead. The above is exactly the problem that this library is designed to solve; it's a library to which applications can link to streamline the usage of GnuPG functions, including interfacing with agents. We can

replace all of the above with this:

```
set crypt_use_gpgme = yes
```

If you have Mutt installed, odds are it already has a GPGME interface. You can check if your current version of Mutt has GPGME powers by looking at the `mutt -v` version output. Here's the output of mine, using the packaged Mutt from Debian GNU/Linux, which does have GPGME support:

```
$ mutt -v | grep -i gpgme
+CRYPT_BACKEND_CLASSIC_PGP  +CRYPT_BACKEND_CLASSIC_SMIME  +CRYPT_BACKEND_GPGME

upstream/548577-gpgme-1.2.patch
```

If you don't have a version of Mutt with GPGME, you can by downloading the source and building it with `--enable-gpgme`:

```
$ ./configure --enable-gpgme
$ make
# make install
```

You may need to make sure you have the GPGME library and headers installed first:

```
# apt-get install libgpgme11 libgpgme11-dev
```

Add the following lines to your `.muttrc` file; remove anything else beginning with `crypt_*` or `pgp_*`:

```
# Use GPGME
set crypt_use_gpgme = yes

# Sign replies to signed email
set crypt_replysign = yes

# Encrypt replies to encrypted email
set crypt_replyencrypt = yes

# Encrypt and sign replies to encrypted and signed email
set crypt_replysignencrypted = yes

# Attempt to verify email signatures automatically
set crypt_verify_sig = yes
```

Restart Mutt, and you should be ready to go.

First of all, check that you have the public key for your intended recipient available in your GnuPG keychain:

```
$ gpg --list-keys joe@example.com
```

If you're able to download it from somewhere, a useful formula is to download it with `curl(1)` and import it directly into `gpg(1)`:

```
$ curl http://www.example.com/joe-somebody.asc | gpg --import
gpg: key 1234ABCD: public key "Joe Somebody <joe@example.com>" imported
gpg: Total number processed: 1
gpg:               imported: 1  (RSA: 1)
gpg: no ultimately trusted keys found
```

Remember, it's your responsibility to decide how much you trust this public key; normally it's best practice that you actually meet someone in person to exchange written key fingerprints in order to be completely sure that the key corresponds to that user.

If you don't have anyone else you know using PGP to communicate with, you can send me a message encrypted with `0xC14286EA77BB8872` to `tom@sanctum.geek.nz`. If you send or link me your public key in your message, then I'll reply to you with a message encrypted with your public key so you can check everything's working.

```
$ curl https://sanctum.geek.nz/keys/pgp/tom@sanctum.geek.nz.pub.asc | gpg --import
```

EDIT 2017: Yes, this offer is still good; you can still email me if you want to test your new setup; I will still reply to you!

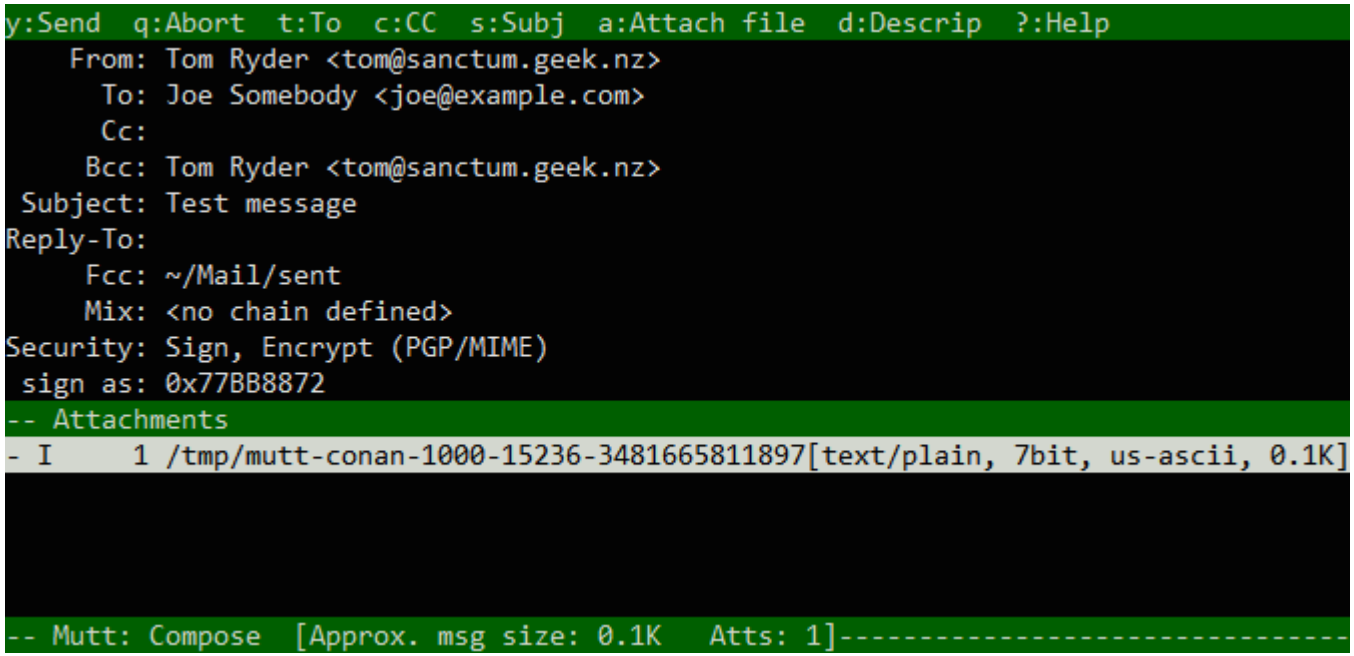
Back in Mutt, begin composing a message with `m`. Enter the recipient and subject as normal, and compose your message. When you've finished writing and save and quit `$EDITOR`, and your message is in the Compose screen waiting to send, press `p` to bring up the PGP menu at the bottom:

```
PGP (e)ncrypt, (s)ign, sign (a)s, (b)oth, s/(m)ime or (c)lear?
```

We'll press `b` to both sign and encrypt the message.

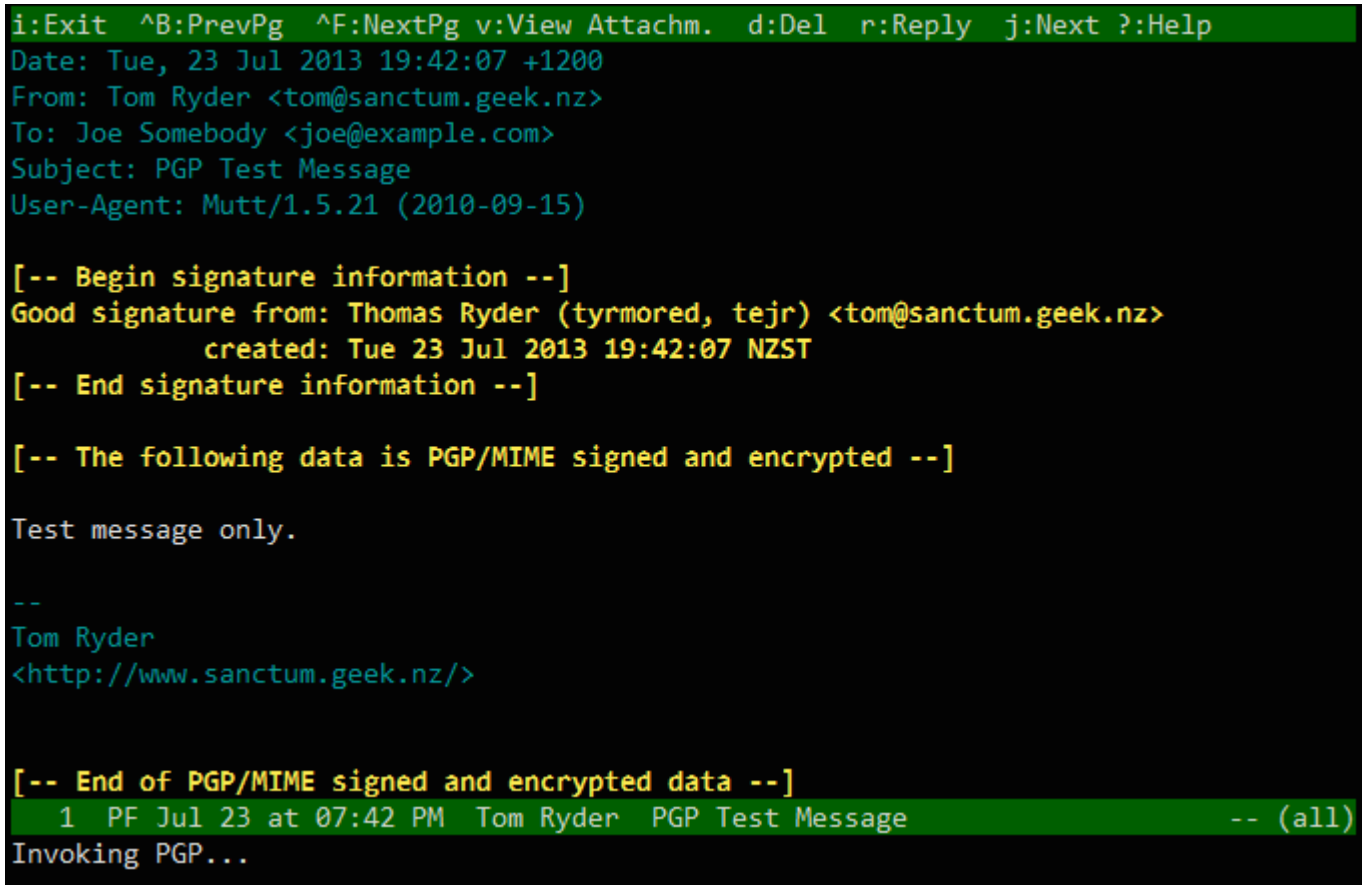
If you want to be able to read the message after sending, then you'll need to arrange to encrypt it with your key as well as the recipient's. I find the cleanest way to do this is to add your address to the `Bcc:` header with `b`. You can also set this as a default with the following line in `~/.gnupg/gpg.conf`, where `0x1234ABCD` is the short ID of your own key:

```
encrypt-to 0x1234ABCD
```



When you send the message with `y`, you might need to specify which key you want to use for each recipient, if you don't have a unique key on your keychain with your recipient's email address.

When you send, you should be prompted for your passphrase by your PIN entry program, unless your agent is already holding the key for you. This is needed in order to sign the message. When you've provided this, the message will be sent, and if you included yourself in the `Bcc:` field, you should be able to read it in your sent mail, with some headers showing the PGP information (whether the message was signed, encrypted, or both):



Your recipient will be able to decrypt the message in their mail user agent with their private key, and nobody else but the two of you will be able to read it. Note that this works for any number of recipients, as long as you have a public key for each of them.

Keep in mind that the metadata of the message, such as the sender and recipient name and address, date and time it was sent, and (importantly) the subject, are sent in plain text. Only the body of the message (including attachments) is encrypted.

With GPGME, Mutt tries to use the first secret key available to it in its private keychain. If you want to use some other specific keypair for signing messages, you can specify that with with the `pgp_sign_as` option in `.muttrc`:

```
set pgp_sign_as = 0x9876FEDC
```

If you'd like to automatically sign all of your outgoing mail, you can set the `crypt_autosign` option:

```
set crypt_autosign = yes
```

The first batch of options we set earlier will already automatically sign and/or encrypt messages in responses to messages doing either/both.

If you'd like to include a link to your PGP key in the headers to each message, you can add a custom header with `my_hdr`:

```
my_hdr X-PGP-Key: https://sanctum.geek.nz/keys/pgp/tom@sanctum.geek.nz.pub.asc
```

All of this combines with Mutt's extensive speed and high-powered configuration to make Mutt a very capable and convenient PGP mail client. As always, the more people you know using PGP, and the more public keys you have, the more useful this will be.

This entry is part 7 of 10 in the series .

Posted in | Tagged , , , , ,

GNU/Linux Crypto: Backups

Posted on

While having local backups for quick restores is important, such as on a USB disk or spare hard drive, it's equally important to have a backup offsite from which you can restore your important documents if, for example, your office was burgled or burned down, losing both your workstation and backup media.

The easiest way to do this for most people is with a storage provider, offering convenient access to bulk storage of suitable size maintained on another company's systems for a relatively modest price or even for free, such as the service, or Microsoft's offering, . The best storage providers will also encrypt the data on their own servers, whether or not they have access.

Trusting a company with all your data and the encryption thereof is risky, particularly given recent revelations of , and privacy-conscious users should prefer the security of encrypting the backups *before* they go up onto the provider's servers. The provider may implement closed and/or symmetric encryption mechanisms of their own, which may or may not be trustworthy. For very strong personal encryption, as established, we can use our GnuPG setup to encrypt files before we put them up there:

```
$ tar -cf docsbackup-"$(date +%Y-%m-%d)".tar $HOME/Documents
$ gpg --encrypt docsbackup-2013-07-27.tar
$ scp docsbackup-2013-07-27.tar.gpg user@backup.example.com:docsbackup
```

The problem with encrypting whole files before we put them up for storage is that for even modestly sized data, performing entire backups and uploading all of the files together every time can cost a lot of bandwidth. Similarly, we'd like to be able to restore our personal files as they were on a specific date, in case of bad backups or accidental deletion, but without storing every file on every backup day, which may end up requiring far too much space.

Normally, the solution is to use an incremental backup system, meaning after first uploading your files in their entirety to the backup system, successive backups upload *only the changes*, storing them in a retrievable and space-efficient format. Systems like , a free Perl frontend to `rsync(1)`, allow this.

Unfortunately, Dirvish doesn't encrypt the files or changesets it stores. What's needed is an incremental backup solution that efficiently calculates and stores changes in files on a remote server, and *also* encrypts them. , a Python tool built around `librsync`, excels at this, and can use our GnuPG asymmetric key setup for the file encryption. It's available in Debian-derived systems in the `duplicity` package. Note that, as before, a GnuPG key setup with an agent is required for this to work.

We can get an idea of how `duplicity(1)` works by asking it to start a backup vault on our local machine. It uses much the same `source` `destination` argument as tools like `rsync` or `scp`:

```
$ cd
$ duplicity --encrypt-key tom@sanctum.geek.nz Documents file://docsbackup
```

It's important to specify `--encrypt-key`, because otherwise `duplicity(1)` will use symmetric encryption with a passphrase rather than a public key, which is considerably less secure. Specify the email address corresponding to the public keypair you would like to use for the encryption.

This performs a full encrypted backup of the directory, returning the following output:

```
Local and Remote metadata are synchronized, no sync needed.
Last full backup date: none
No signatures found, switching to full backup.
--------------[ Backup Statistics ]--------------
StartTime 1374903081.74 (Sat Jul 27 17:31:21 2013)
EndTime 1374903081.75 (Sat Jul 27 17:31:21 2013)
ElapsedTime 0.01 (0.01 seconds)
SourceFiles 4
SourceFileSize 142251 (139 KB)
NewFiles 4
NewFileSize 142251 (139 KB)
DeletedFiles 0
ChangedFiles 0
ChangedFileSize 0 (0 bytes)
ChangedDeltaSize 0 (0 bytes)
DeltaEntries 4
RawDeltaSize 138155 (135 KB)
TotalDestinationSizeChange 138461 (135 KB)
Errors 0
-------------------------------------------------
```

You'll note you were not prompted for your passphrase to do this. Remember, encrypting files with your public key does not require a passphrase; the whole idea is that anyone can encrypt using your key without needing your permission.

Checking the created directory `docsbackup`, we find three new files within it, all three of them encrypted:

```
$ ls -1 docsbackup
duplicity-full.20130727T053121Z.manifest.gpg
duplicity-full.20130727T053121Z.vol1.difftar.gpg
duplicity-full-signatures.20130727T053121Z.sigtar.gpg
```

The `vol1.difftar.gpg` file contains the actual data stored; the other two files contain *metadata* about the backup's contents, for use to calculate differences the next time the backup runs.

If we make a small change to a file in the directory being backed up, and run the same command again, we note that the backup has been performed *incrementally*, and only the changes (the new file) have been saved:

```
$ duplicity --encrypt-key tom@sanctum.geek.nz Documents file://docsbackup
Local and Remote metadata are synchronized, no sync needed.
Last full backup date: Sat Jul 27 17:34:33 2013
--------------[ Backup Statistics ]--------------
StartTime 1374903396.52 (Sat Jul 27 17:36:36 2013)
EndTime 1374903396.52 (Sat Jul 27 17:36:36 2013)
ElapsedTime 0.01 (0.01 seconds)
SourceFiles 5
SourceFileSize 142255 (139 KB)
NewFiles 2
NewFileSize 4100 (4.00 KB)
DeletedFiles 0
ChangedFiles 0
ChangedFileSize 0 (0 bytes)
ChangedDeltaSize 0 (0 bytes)
DeltaEntries 2
RawDeltaSize 4 (4 bytes)
TotalDestinationSizeChange 753 (753 bytes)
Errors 0
-------------------------------------------------
```

We also find three new files in the `docsbackup` directory containing the new data:

```
$ ls -1 docsbackup
duplicity-full.20130727T053433Z.manifest.gpg
duplicity-full.20130727T053433Z.vol1.difftar.gpg
duplicity-full-signatures.20130727T053433Z.sigtar.gpg
duplicity-inc.20130727T053433Z.to.20130727T053636Z.manifest.gpg
duplicity-inc.20130727T053433Z.to.20130727T053636Z.vol1.difftar.gpg
duplicity-new-signatures.20130727T053433Z.to.20130727T053636Z.sigtar.gpg
```

Note that the new files have the prefix `duplicity-inc-` or `duplicity-new-`, denoting them as incremental backups and not full ones.

Note that in order to keep track of what files have already been backed up, `duplicity(1)` stores metadata in `~/.cache/duplicity`, as well as storing them along with the backup. This allows us to let our backup processes run unattended, rather than having to put in our passphrase to read the metadata on the remote server before performing an incremental backup. Of course, if we lose our cached files, that's OK; we can read the ones out of the backup vault by supplying our passphrase on request for decryption.

If you have SSH or even just SCP/SFTP access to your storage provider's servers, not much has to change to make `duplicity(1)` store the files up there instead:

```
$ duplicity --encrypt-key tom@sanctum.geek.nz Documents sftp://user@backup.example.com:docsback
```

Your backups will then be sent over an SSH link to the directory `docsbackup` on the system `backup.example.com`, with username `user`. In this way, not only is all the data protected in transmission, it's stored encrypted on the remote server; it never sees your plaintext data. All anyone with access to your backups can see is their approximate size, the dates they were made, and (if you publish your public key) the user ID on the GnuPG key used to encrypt them.

If you're `ssh-agent(1)` to store your decrypted private keys, you won't even have to enter a passphrase for that.

The `duplicity(1)` frontend supports other methods of uploading to different servers, too, including the `boto` backend for S3 Amazon Web Services, the `gdocs` backend for Google Docs, and `httplib2` or `oauthlib` for Ubuntu One.

If you like, you can also sign your backups to make sure they haven't been tampered with at the time of restoration, by changing `--encrypt-key` to `--encrypt-sign-key`. Note that this will require your passphrase.

Restoring from a `duplicity(1)` backup volume is much the same, but with the arguments reversed:

```
$ duplicity sftp://user@backup.example.com:docsbackup docsrestore
Synchronizing remote metadata to local cache...
GnuPG passphrase:
Copying duplicity-full-signatures.20130727T053433Z.sigtar.gpg to local cache.
Copying duplicity-full.20130727T053433Z.manifest.gpg to local cache.
Copying duplicity-inc.20130727T053433Z.to.20130727T053636Z.manifest.gpg to local cache.
Copying duplicity-new-signatures.20130727T053433Z.to.20130727T053636Z.sigtar.gpg to local cache
Last full backup date: Sat Jul 27 17:34:33 2013
```

Note that this time you are asked for your passphrase. This is because restoring the backup requires decrypting the data and possibly the signatures in the backup vault. After doing this, the complete set of documents from the time of your most recent incremental backup will be available in `docsrestore`.

Using this incremental system also allows you to restore your data in the state in the last backup before a given time. For example, to retrieve my `~/Documents` directory as it was three days ago, I might run this:

```
$ duplicity --time 3D \
    sftp://user@backup.example.com:docsbackup \
    docsrestore
```

You can extend this to only restore particular files for large vaults, if you only need a particular file from the vault:

```
$ duplicity --time 3D \
    --file-to-restore private/eff.txt \
    sftp://user@backup.example.com:docsbackup \
    docsrestore
```

You should run your first full backup interactively to make sure it's doing exactly what you need, but once you're confident that everything is working correctly, you can set up a simple Bash script to run incremental backups for you. Here's an example script, saved in `$HOME/.local/bin/backup-remote`:

```
#!/usr/bin/env bash

# Run keychain to recognise any agents holding decrypted keys we might need
# (optional, depending on your SSH key setup)
eval "$(keychain --eval --quiet)"

# Specify directory to back up, GnuPG key ID, and remote username and
# hostname
keyid=tom@sanctum.geek.nz
local=/home/tom/Documents
remote=sftp://user@backup.example.com/docbackups

# Run backup with duplicity
/usr/bin/duplicity --encrypt-key "$keyid" -- "$local" "$remote"
```

The line with `keychain` is optional, but will be necessary if you're using an SSH key with a passphrase on it; you'll also need to have authenticated with `ssh-agent` at least once. See the earlier article on for details on this setup.

Don't forget to make the script executable:

```
$ chmod +x ~/.local/bin/backup-remote
```

You can then have `cron(8)` call this for you every week, running it as your user, by editing your `crontab(5)` :

```
$ crontab -e
```

The following line would run this script every morning, beginning at 6.00am:

```
0 6 * * *   ~/.local/bin/backup-remote
```

A few general best practices apply to this, consistent with the :

- Check that your backups completed; either have the output of the `cron` script mailed to you, or log it to a file that you check at least occasionally to make sure your backups are working. I highly recommend using an email message, and including error output:

  ```
  MAILTO=you@example.com
  0 6 * * *   ~/.local/bin/backup-remote 2>&1
  ```

- Run backups to your local servers too; this might prevent your backup provider from reading your files, but it won't save them from being accidentally deleted.

- Don't forget to occasionally test-restore your backups to make sure they're working correctly. It's also wise to use `duplicity verify` on them occasionally, particularly if you don't back up every day:

  ```
  $ duplicity verify sftp://user@remote.example.com/docbackups Documents
  Local and Remote metadata are synchronized, no sync needed.
  Last full backup date: Sat Jul 27 17:34:33 2013
  ```

```
GnuPG passphrase:
Verify complete: 2195 files compared, 0 differences found.
```

- This incremental system means that you'll likely only have to make full backups once, so you should back up too much data rather than too little; if you can spare the bandwidth and have the space, backing up your entire computer isn't really that extreme.

- Try not to depend too much on your remote backups; see them as a last resort, and work securely and with local backups as much as you can. Certainly, never rely on backups as a version control system; .

This entry is part 8 of 10 in the series .

Posted in | Tagged , , , , , , , , , , ,

GNU/Linux Crypto: Disks

Posted on

GnuPG provides us with a means to securely encrypt individual files on a filesystem, but for really high-security information or environments, it may be appropriate to encrypt an entire disk, to mitigate problems such as caching sensitive files in plaintext. The GNU/Linux kernel includes its own disk encryption solution in the kernel, `dm-crypt`. This can be leveraged with a low-end tool called `cryptsetup`, or more easily with LUKS, the , implementing strong cryptography with passphrases or keyfiles.

In this example, we'll demonstrate how this can work to encrypt a USB drive, which is a good method for securely storing really sensitive data such as PGP master keys that's only needed occasionally, rather than leaving it always mounted on a networked device. Be aware that this erases any existing files on the drive.

The cryptographic tools used by `dm-crypt` and LUKS are built-in to Linux kernels after 2.6, but you may have to install a package to get access to the `cryptsetup` frontend. On Debian-derived systems, it's available in `cryptsetup`:

```
# apt-get install cryptsetup
```

On RPM-based systems like Fedora or CentOS, the package has the same name, `cryptsetup`:

```
# yum install cryptsetup
```

After identifying the block device on which we want the encrypted filesystem, for example `/dev/sdc1`, we can erase any existing contents using a call to `wipefs`:

```
# wipefs -a /dev/sdc1
```

Alternatively, we can zero out the whole disk, if we want to completely overwrite any trace of the data previously on the disk; this can take a long time for large volumes:

```
# cat /dev/zero >/dev/sdc1
```

If you don't have a USB drive to hand, but would still like to try this out, you can use a loop block device in a file. For example, to create a 100MB loop device:

```
# dd if=/dev/zero of=/loopdev bs=1k count=102400
102400+0 records in
102400+0 records out
104857600 bytes (105 MB) copied, 0.331452 s, 316 MB/s
# losetup -f
/dev/loop0
# losetup /dev/loop0 /loopdev
```

You can then follow the rest of this guide using `/dev/loop0` as the raw block device in place of `/dev/sdc1`. In the above output, `losetup -f` returns the first available loop device for use.

Setting up a LUKS container on the block device is then done as follows, providing a passphrase of decent strength; as always, the longer the better. Ideally, you should not use the same passphrase as your GnuPG or SSH keys.

```
# cryptsetup luksFormat /dev/sdc1

WARNING!
========
This will overwrite data on /dev/sdc1 irrevocably.

Are you sure? (Type uppercase yes): YES
Enter passphrase:
Verify passphrase:
```

This creates an abstracted encryption container on the disk, which can be opened by providing the appropriate passphrase. A virtual mapped device is then provided that encrypts all data written to it transparently, with the encrypted data written to the disk.

We can open the mapped device using `cryptsetup luksOpen`, which will prompt us for the passphrase:

```
# cryptsetup luksOpen /dev/sdc1 secret
```

The last argument here is the filename for the block device to appear under `/dev/mapper`; this example provides `/dev/mapper/secret`.

With this done, the block device on `/dev/mapper/secret` can now be used in the same way as any other block device; all of the disk operations are abstracted through encryption operations. You'll probably want to create a filesystem on it; in this case, I'm creating an `ext4` filesystem:

```
# mkfs.ext4 /dev/mapper/secret
mke2fs 1.42.8 (20-Jun-2013)
Filesystem label=
OS type: Linux
Block size=1024 (log=0)
Fragment size=1024 (log=0)
Stride=0 blocks, Stripe width=0 blocks
25168 inodes, 100352 blocks
5017 blocks (5.00%) reserved for the super user
First data block=1
Maximum filesystem blocks=67371008
13 block groups
8192 blocks per group, 8192 fragments per group
1936 inodes per group
Superblock backups stored on blocks:
        8193, 24577, 40961, 57345, 73729

Allocating group tables: done
Writing inode tables: done
Creating journal (4096 blocks): done
Writing superblocks and filesystem accounting information: done
```

We can then mount the device as normal, and data put into the newly created filesytem will be transparently encrypted:

```
# mkdir -p /mnt/secret
# mount /dev/mapper/secret /mnt/secret
```

For example, we could store a private GnuPG key on it:

```
# cp -prv /home/tom/.gnupg/secring.gpg /mnt/secret
```

We can get some information about the LUKS container and the specifics of its encryption using `luksDump` on the underlying block device. This shows us the encryption method used, in this case `aes-xts-plain64`.

```
# cryptsetup luksDump /dev/sdc1
LUKS header information for /dev/sdc1

Version:        1
Cipher name:    aes
Cipher mode:    xts-plain64
Hash spec:      sha1
Payload offset: 4096
MK bits:        256
MK digest:      87 6d 08 59 b2 f0 c6 6e ca ec 5f 72 2c e0 35 33 c2 9e cb 8e
MK salt:        7f a5 38 4c 14 85 61 cb 6c 22 65 48 87 21 60 8f
                fa 40 2a ab ae 7d cc df c9 9b a4 e3 3c 64 b6 bb
MK iterations:  49375
UUID:           f4e5f28c-3b34-4003-9bcd-dbb2352042ba

Key Slot 0: ENABLED
        Iterations:             197530
        Salt:                   2d 57 f6 2b 44 a6 61 ee d6 ee e4 7d 64 f0 71 d6
                                55 16 09 83 b4 f0 94 ca 19 17 11 a9 34 84 02 96
        Key material offset:    8
        AF stripes:             4000
Key Slot 1: DISABLED
Key Slot 2: DISABLED
Key Slot 3: DISABLED
Key Slot 4: DISABLED
Key Slot 5: DISABLED
Key Slot 6: DISABLED
Key Slot 7: DISABLED
```

When finished with the data on the device, we should both unmount any filesystem on it, and also close the mapped device so that the passphrase is required to re-open it:

```
# umount /mnt/secret
# cryptsetup luksClose /dev/mapper/secret
```

If the data is a removable device, you should also consider physically removing the media from the machine and placing it in some secure location.

This post only scratches the surface of LUKS functionality; many more things are possible with the system, including automatic mounting of encrypted filesystems and the use of stored keyfiles instead of typed passphrases. The `cryptsetup` contains a great deal of information, including some treatment of data recovery, and the Arch Wiki has on various ways of using LUKS securely.

This entry is part 9 of 10 in the series .

Posted in | Tagged , , , , ,

GNU/Linux Crypto: Importance

Posted on

While this series was being written, from June 2013, began leaking top-secret documents from the United States National Security Agency, showing that the agency was capable of Internet surveillance on a massive scale with the PRISM surveillance system and with the XKeyscore interface into their amassed data. The fact that covert government surveillance was possible and was taking place does not come as particularly surprising news to network engineers and conspiracy theorists, but the revelations have finally given the general, non-technical public an idea of how badly the proprietary systems around which they have built much of their digital lives can be used to harm them and compromise their privacy.

Concerned people in the United States will be only too aware of how the secret abuse of power to exercise this surveillance and the failed motions to curtail it by the United States Congress has dented their trust in their own government. However, the leaks' implications are international as well. The foreign intelligence agency in my own country of New Zealand, the Government Communications Security Bureau, was earlier this year , and diplomatic cables from WikiLeaks show the GCSB with the NSA. In spite of this, new legislation is set to extend the GCSB's powers, despite independent reviews from both a legal and human rights perspective, even after amendments. The scandal and the anger over surveillance abuse extends to the , , , and many other countries.

I do hold out some hope for the efforts such as the Electronic Frontier Foundation's to curtail the surveillance or at the very least to register the public's anger about this unwarranted intrusion into private lives. However I am concerned not just by the possibility of the rise of a global surveillance state, but by the implications this has for the right to secure communications using cryptography for authentication and encryption.

It's no secret that cryptography and encryption , and that they expend a great deal of effort in attempting to circumvent it, including from businesses for applications like HTTPS. My concern is this: If it becomes publically accepted that governments spy warrantlessly on international networks and that this is justified or necessary, then we may reach a point where the legality of the general public's use of cryptography itself may again be called into question.

Computing professionals of my generation likely did not begin their careers until after the United States' cryptographic export controls were relaxed in 1999, perhaps prompting us to take for granted the availability of algorithms like RSA and AES with high key sizes for cryptographic purposes. A world where a government agency would actively attempt to curtail the use of such technology may seem very far-fetched to us — perhaps less so to those who remember that was a radical new idea that caused its activist creator Phil Zimmermann real legal trouble.

I believe that computing enthusiasts and users of free software operating systems, not just cryptographic experts, are in a special position to assist their concerned friends and family with defending their online privacy and securing their communications, and that if we value both freedom and security of information, then we in fact have a responsibility to do so. I believe that people need to be aware of not just the implications of massive surveillance on a global scale, but also how to exercise their rights to fight against it. If the legality of cryptography is ever called into question again as the result of its impeding warrantless surveillance, then its pervasiveness and the public's insistence on its free availability should make restricting its use not just impractical, but unthinkable.

This entry is part 10 of 10 in the series .

Posted in | Tagged , , , , ,