`regpg`

safely store server secrets

Tony Finch
`<fanf2@cam.ac.uk>`
`<gitmaster@uis.cam.ac.uk>`

Tuesday 21st November 2017

Abstract

The `regpg` program is a thin wrapper around `gpg` for encrypting secrets so they can be stored and shared using `git` and decrypted when Ansible deploys them to servers.

1 Introduction

This talk is in two main sections.

agenda

I will start off by explaining some of the context and thinking behind `regpg` by unpacking its slogan backwards.

- Context
  - secrets?
  - server?
  - store?
  - safely?
  - gpg?
  - re?

Then I'll give a demo of `regpg`'s main features, in roughly the same order as its reference manual.

- Demo
  - keys
  - secrets
  - recrypt
  - X.509 / TLS
  - Ansible
  - conversion

2 Context

In the first part, we'll discuss what `regpg` is and what it is not.

secrets

The secrets we are working with are cryptographic keys

- private keys
- bearer tokens
- shared secrets

We have hundreds of them. They need to be shared with the right people and kept secret from the wrong people.

It's a key distribution problem.

secrets – encryption

We can massively reduce the size of the problem by encrypting the secrets with a small number of master secrets.

For example, before `regpg` I used to encrypt secrets using the root password.

This reduces the key distribution to previously solved problems:

- password distribution
- non-secret file distribution (i.e. git)

secrets – Shamir / Rivest / Adleman

But we can do better with public key cryptography.

Each person keeps their own private key – there's no need to distribute any master secrets. We know how to do this because we already do it for `ssh`keys.

We distribute the public keys of each person who can decrypt using `git`, alongside the encrypted secrets and the unencrypted configuration and source code. This gives us a kind of auditable record of who has access to secrets.

We get better tooling from the `gpg` agent, which saves us from having to repeatedly type our passphase.

We can revoke a person's access if we can destroy all the copies of their private key, without having to replace all the secrets.

You only need the public keys to encrypt a secret, which means an automated system can manage its own keys without having access to all the other secrets in a repository.

server

The specific kinds of secrets we are dealing with are used by servers to authenticate themselves –

- `ssh`host private keys
- TLS private keys
- API keys
- DNS TSIG shared secrets
- etc.

These secrets have to be available unencrypted on the server, so we want it to be convenient to decrypt and install them.

We're not dealing with user passwords.

We're not trying to be a password manager.

server – files

It's often the case that each server secret is in a file by itself – that's true for `ssh`and TLS and DNS keys.

`regpg` works best when each secret is in a file by itself. You can use filenames to identify secrets without having to decrypt them.

Keeping secrets strictly separate from non-secret code and configuration helps `git diff` to works better.

`regpg` does not have any hooks into `git` for automatically decrypting and `diff`ing secrets because secrets are blobs of random data for which `diff` is useless.

store – not share

`regpg` is for encrypting files for storage in version control when they are not in use, and decrypting them for deployment to production.

The other verb that might have fitted in this place is "share", but `regpg` is not directly about sharing.

`regpg` stores secrets in a way that works with `git` or other version control systems, but `regpg` does not get involved with `git`. You use `git` for sharing secrets in the same way you us it for sharing code or configuration.

I have tried a few times to write wrappers that get clever with `git` and they have usually been dismal failures. `regpg` does not get clever with `git`.

safely – hazmat containment

There are a couple of aspects to being safe with `regpg`, and both of them relate to dissatisfaction with `ansible-vault`.

The first is safe cryptography.

`regpg` keeps well away from any low-level primitives. I did a code review of `ansible-vault` and it uses a cryptographic library that literally has "HAZMAT" in its name. And, totally predictably, `ansible-vault` has really bad crypto.

Instead, `regpg` relies on `gpg` for cryptography. `gpg` is terrible software in many ways, but it is widely available, it has reasonably competent crypto, and it is also used by `git` and Debian.

safely – situational awareness

The other aspect to being safe is psychological safety.

`regpg` allows you to make it clear in your Ansible playbook which files should be encrypted, helps you to find out which files actually are or are not encrypted, and tells you when things are inconsistent.

This is unlike `ansible-vault` which does not let you say whether something should be encrypted, and encourages you to encrypt and decrypt in place, and doesn't complain either way, so you can easily expose secrets by mistake.

`regpg` tries to be really easy to understand. It isn't very chatty, but it also does not hide things from you. I want you to feel confident that you know how it works and what it is doing.

gpg

`regpg` is a thin wrapper around `gpg` to adapt it for our purposes.

It's a very thin wrapper. You don't need `regpg` to decrypt secrets – you can still use normal `gpg -d` to decrypt them.

`regpg` simplifies `gpg` in two ways.

Firstly, `regpg` gets rid of `gpg`'s key management and replaces it with Jon Warbrick's scheme.

There are no key servers, no web of trust, no key signing parties. Instead we just use `git` to exchange public keys.

Secondly, `regpg` provides several little helpers to make it easier to use `gpg`-encrypted secrets with other tools such as OpenSSL, OpenSSH, and Ansible.

regpg

Why is it called "`regpg`"? Where does the "re" come from?

Partly named after its `recrypt` subcommand which we will see shortly.

In `gpg`, the term "recipients" means those who can decrypt a message. `regpg` is all about managing a list of recipients and repeatably and reliably encrypting files to those recipients.

3 Demo

dependencies

- prerequisites
  - perl
  - gnupg
  - gnupg-agent
  - pinentry-*
- helpers
  - ansible
  - git
  - openssl

  - openssh-client
  - xclip

The prerequisites are required for `regpg`'s core functionality; the helpers are optional but some `regpg` features won't work without them.

These are Debian package names. If anyone is able to help with installation instructions on other systems, please let me know!

The `pinentry` program is used by `gpg-agent` to prompt you for your passphrase. There are multiple versions – I use `pinentry-gtk2` but there are also `-gnome3` and `-qt` and `-curses` versions.

check `gpg-agent`

- `echo $GPG_AGENT_INFO`
- `eval $(gpg-agent --daemon)`

You should find that `gnupg-agent` is started automatically when you log in - use the first command to check this.

You can start it manually using the second command.

install

- quick
  - `cd ~/bin`
  - `curl -O https://dotat.at/prog/regpg/regpg`
- home page `https://dotat.at/prog/regpg/`
  - supporting documentation
  - distribution tar balls
  - test suite

generate key

- Generate a key just for `regpg`
- Separate from your other `gpg` keys (if any)
- `gpg --gen-key`
- Answer the quiz

generate key – demo

manage keys

- addkey
- addself $\Longleftarrow$
- delkey $\Longleftarrow$
- exportkey
- importkey
- lskeys $\Longleftarrow$

manage keys – demo

- mkdir demo
- cd demo
- regpg addself
- ls

`regpg` has made a public key ring (and a backup file, because `gpg` loves backup files)

The "addself" subcommand adds keys which match your login name and for which you have the private key.

This is the only configuration file for `regpg`

Normally you would put this at the top of your Ansible setup next to your `ansible.cfg` and inventory etc.

- regpg lskeys
- regpg ls
- regpg del fanf9
- regpg ls
- regpg add fanf9
- regpg ls

One bit of magic going on here is that `regpg` ensures that `gpg` uses the backwards compatible keyring format, even if you are using `gpg` 2.1.

secrets

- encrypt $\Leftarrow$
- decrypt $\Leftarrow$
- recrypt
- edit $\Leftarrow$
- pbcopy
- pbpaste
- shred $\Leftarrow$
- check $\Leftarrow$

The pasteboard commands use the names from Mac OS X, but if you use them on Linux it will use `xclip` instead.

secrets – demo

- echo secret one ¿foo
- regpg encrypt foo foo.asc

Note `regpg` does not need a passphrase to encrypt, just the public keys.

The `.asc` extension is the ugly but conventional name for a PGP-encrypted ASCII-armored file. (ASCII armoring is like Base64.)

- regpg check

The "check" subcommand looks for encrypted files by recursively grepping for the "BEGIN PGP MESSAGE" ASCII-armoring.

`regpg` warns us that we have left behind an unencrypted file. (It uses a simple heuristic based on filenames.)

- regpg shred foo
- regpg ck
- regpg decrypt foo.asc

`regpg` asks for your passphrase to decrypt the first time

- regpg decrypt foo.asc

The `gpg-agent` has stashed the passphrase so we don't need to keep typing it.

- regpg edit foo.asc

You should not normally need to edit an encrypted file by hand, but if you do, `regpg` tries to make it safer by keeping temporary files in a RAM disk (at least on Linux – there's no ramfs on Mac OS) and shredding them afterwards.

- echo secret three — regpg en bar.asc

`regpg` is friendly to pipelines.

- regpg ck
- touch foo bar
- regpg ck
- regpg shred -r
- regpg ck

Several `regpg` subcommands take a `-r` option which recurses over all the files found by `regpg check`

recrypt

- delkey $\Leftarrow$
- importkey $\Leftarrow$
- lskeys $\Leftarrow$
- recrypt $\Leftarrow$
- check $\Leftarrow$

recrypt – demo

- curl https://dotat.at/fanf.gpg — regpg importkey
- regpg ls
- regpg ck

`regpg` says a key has been added to the keyring and lists which files need to be decrypted and re-encrypted so that every recipient listed in the keyring can decrypt them

- regpg recrypt -r
- regpg ck

Again the `-r` option means recursively apply the command to all the files listed by `regpg check`

- regpg del fanf2
- regpg ls
- regpg ck
- regpg re -r

The `-r` option can be applied to the key management subcommands to combine them with the "recrypt" subcommand

- curl https://dotat.at/fanf.gpg — regpg importkey -r
- regpg ls

- regpg ck
- regpg del -r fanf2
- regpg ls
- regpg ck

generate TLS / ssh

- gencsrconf $\Longleftarrow$
- gencsr $\Longleftarrow$
- genkey $\Longleftarrow$
- genpwd

It turns out that OpenSSL and OpenSSH have mostly the same key format, so `regpg` uses the same “genkey” command for both of them.

generate TLS / ssh – demo

- regpg genkey rsa id˙rsa.asc id˙rsa.pub

For ssh keypairs, give it a key algorithm, and private and public key files.

- regpg genkey rsa tls.pem.asc

For TLS give it the algorithm and private key file name.

I always find it difficult to remember how to make a certificate signing request, so `regpg` provides some help.

First, get a configuration file from an existing certificate, either from a file or a web server:

- regpg gencsrconf cam.ac.uk tls.csr.conf
- vi tls.csr.conf
- regpg gencsr tls.pem.asc tls.csr.conf tls.csr

You should commit your CSR so you can re-use it next time if none of the details of yur certificate have changed.

You should commit your CSR configuration file to keep a convenient record of changes to your CSR.

set up hooks

- init $\Leftarrow$
- init git $\Leftarrow$
- init ansible $\Leftarrow$
- init ansible-vault

All these commands are itempotent, and unlike other `regpg` commands, they are quite verbose.

set up hooks – demo

- regpg init

Does nothing if there is a `pubring.gpg` file

- rm pubring.gpg*
- regpg init

If there is no keyring, it does "addself" verbosely.

There is a tiny hook for git, which lets you see the history of `pubring.gpg` more easily.

(It does not do anything for secret files since it isn't useful to diff cryptographic secrets.)

- git init
- git add .
- git commit -m 'initial commit'
- git log –patch pubring.gpg
- regpg init git
- git log –patch pubring.gpg
- git status

There are two parts to this hook, a `.gitattributes` file which you should commit, and some local repository configuration which cannot be committed.

- git add .gitattributes
- git commit -m ’regpg init git’

Whenever you newly clone a repository, you should run `regpg init git` inside it to set up the local configuration.

- curl https://dotat.at/fanf.gpg — regpg importkey
- git diff

This feature is really important for auditing changes to your `pubring.gpg` file, because that’s your access control list.

OK, let’s try out Ansible

- echo [defaults] >ansible.cfg
- echo hostfile = inventory >>ansible.cfg
- echo localhost ansible_connection=local >inventory
- ansible -m debug -a msg=hi localhost
- git add .
- git commit -m ’start ansible’

OK, we have a basic Ansible setup.

- regpg init ansible
- cat ansible.cfg

`regpg` added a plugin for decrypting secrets. This is 20 lines of python that just invokes `gpg --decrypt` – the plugin does not use `regpg`.

- cat gpg-preload.yml

`regpg` added a plugin for preloading `gpg-agent`. You can use this to make gpg ask you for your passphrase just once – there is a race condition which can make it asking for every host.

- ansible-playbook gpg-preload.yml

- git add .

- git commit -m ’regpg init ansible’

There’s an example of how to use this setup at the end of the `regpg` man page.

- regpg help

The thing to note here is that I am using `with_fileglob:` to make ansible search for files using its usual search path. The `gpg_d` plugin does no searching.

I am using this setup at the moment for my systems. It’s optimized for simplicity of implementation, though it can be a bit annoying. I’m interested in feedback if you think something more elaborate would be worth the effort.

converters

- conv ansible-gpg $\Leftarrow$
- conv ansible-vault $\Leftarrow$
- conv stgza

converters – demo

I’m going to set up `regpg` for use with `ansible-vault` which is only recommended if you want to convert from `ansible-vault` to `regpg`.

- regpg init ansible-vault
- git status
- cat ansible.cfg

Now we have a setup similar to Jon Warbrick’s `ansible-gpg`

- echo hello wombats ¿wombat
- ansible-vault encrypt wombat
- cat wombat
- vi echidna.yml
- ansible-playbook echidna.yml

Ansible is automatically decrypting things for us.

Let's convert this file to a normal `regpg` setup.

- regpg conv ansible-vault
- regpg conv ansible-vault wombat wombat.asc
- vi echidna.yml

- regpg ck
- regpg shred -r

There's another conversion command which helps with conversion from `ansible-gpg`

- git clone git://git.uis.cam.ac.uk/uis/u/jw35/ansible-gpg.git
- cd ansible-gpg
- ls -la
- file .ansible-gpg/pubring.gpg

The `ansible-gpg` repo comes with a demo setup. Note that the pubring is in incompatible `gpg` 2.1 keybox format.

- regpg conv ansible-gpg
- git status
- file pubring.gpg
- regpg conv ansible-vault

This is now the setup that you get from `regpg init ansible-vault` and it allows you to convert your setup one file at a time.

4 Done!

Questions?