Search Medium

Published in Level Up Coding

Richard Fan Follow
Mar 16, 2020 · 7 min read · · Listen

Save

Using SOPS and git hooks to share secrets — Part 1

DevOps drives everything into code (including secrets)

DevOps is a doctrine, not a framework. If you ask 10 peoples what is DevOps, you will get 10 different answers. But among those answers, Automation and Infrastructure as code would somewhat be part of them. Thanks to the tools available, we can now hand off those infrastructure configs and manual deployment commands to the computer and share it with everyone. However, what should we do with our secrets, like access key and password? Should we share them with our team? Where should we put them?

Scenario: Sharing deployment config

Let's say I am developing a serverless application using AWS SAM. I have created the following `Makefile` so that I can deploy the app with one simple `make` command.

```
deploy:
    sam build
    sam package --output-template packaged.yaml --s3-bucket
$(BUCKET_NAME)
    sam deploy--template-file packaged.yaml --stack-name $(STACK_NAME)
--capabilities CAPABILITY_IAM
```

I also created a `.env` file to store the S3 bucket name and CloudFormation Stack name

```
export BUCKET_NAME=my-dummy-bucket
export STACK_NAME=dummy-stack
```

In order to let my teammate (or tomorrow's me) know which bucket and which CloudFormation stack I am using, I have to save `.env` somewhere. Although it is not top-secret, I don't want people outside my team to know which S3 bucket I am using, or what is our naming convention. So instead of pushing this file directly to the repository, I have to store it in another way.

Method 1: Using parameter store / secret manager

This should be the most commonly used method. Instead of storing the information in a text file, we can make use of hosted parameter stores or secret managers (e.g. AWS Parameter Store, Secret Manager / Azure Key Vault / HashiCorp Vault).

There are many secret manager options

For example, I can change `Makefile` into this:

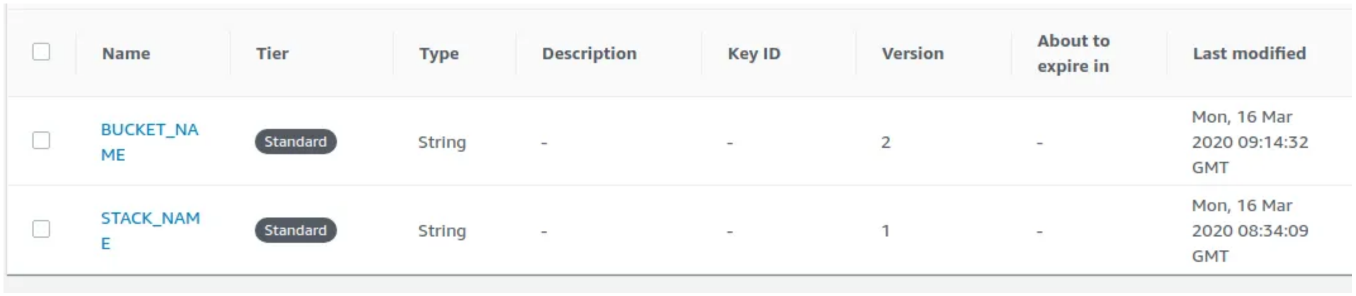
```
BUCKET_NAME=$(shell aws ssm get-parameter --name BUCKET_NAME --query
'Parameter.Value' --output text)
STACK_NAME=$(shell aws ssm get-parameter --name STACK_NAME --query
'Parameter.Value' --output text)

deploy:
    sam build
    sam package --output-template packaged.yaml --s3-bucket
$(BUCKET_NAME)
    sam deploy --template-file packaged.yaml --stack-name $(STACK_NAME)
--capabilities CAPABILITY_IAM
```

Now, I use awscli to grab `BUCKET_NAME` and `STACK_NAME` from AWS Parameter Store. I can grant my teammate access permissions so that they can run the `make` command as I did on my computer. If one of my teammates leave the company, I can simply revoke his access and don't have to worry about leaving any secret on his computer.

However, as the secrets are saving outside the repository, we have to manage them separately. How do we keep track of which parameters belong to which project? How do we do version control on those parameters?

Many parameter stores support versioning features, but it is still difficult to manage two sets of version tree (code repository and parameter store). When issues emerge, we have to match the timeline of both version tree to get the full picture of state changes.



☐	Name	Tier	Type	Description	Key ID	Version	About to expire in	Last modified
☐	BUCKET_NAME	Standard	String	-	-	2	-	Mon, 16 Mar 2020 09:14:32 GMT
☐	STACK_NAME	Standard	String	-	-	1	-	Mon, 16 Mar 2020 08:34:09 GMT

AWS Parameter Store support versioning

Pros:
1. Full control of the secret
2. No plaintext copy in local machine

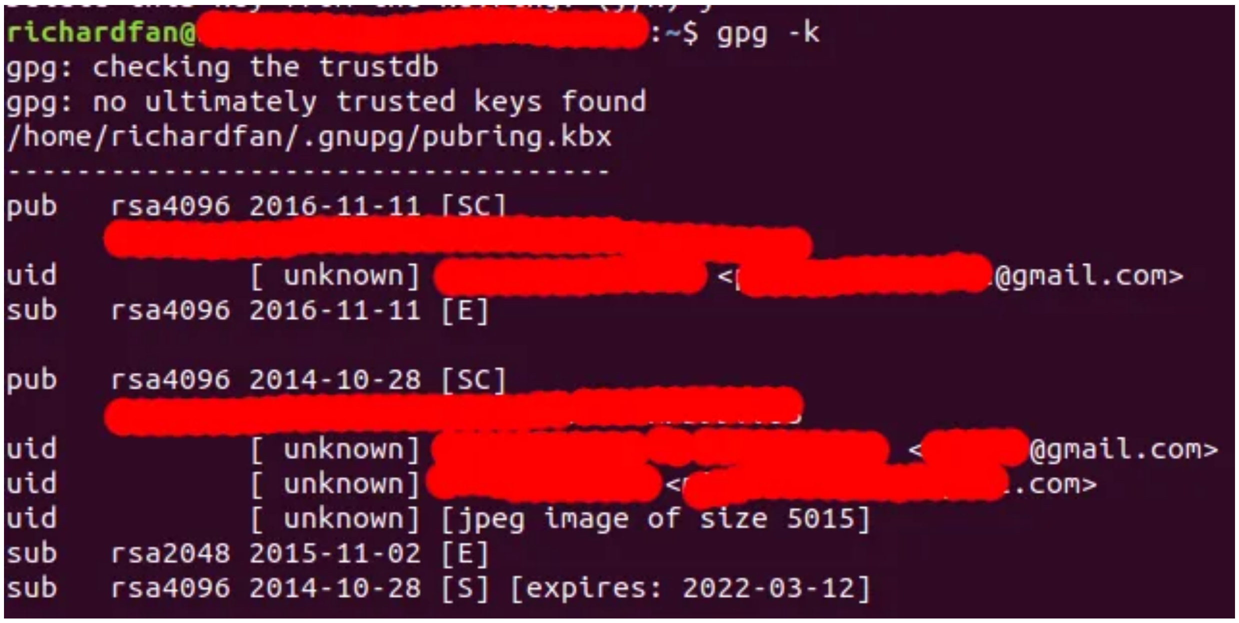
Cons:
1. Need to manage 2 sets of resources

Method 2: Encrypting secrets

To avoid the overhead of managing 2 separate resources, we can put the secrets inside the same repository as the code. But to prevent any data breach, we have to ensure we don't put them online in plaintext.

If you search for some related keywords on Google, you may find git-secret. This is a really good tool, it allows you to encrypt any files using GPG. Basically, you add the GPG public keys from your teammates, and git-secret will use them to encrypt files you specify.

However, git-secret assumes developers should have their own GPG keypair but I think this is not the case in many organisations. Also, because it uses everybody's public keys to encrypt files. When new members join, you have to gather their public keys and encrypt every file again so that they can decrypt it. Likewise, if anybody leaves the team, you have to remove his public keys and re-encrypt all the files so that he no longer have access.



```
richardfan@[redacted]:~$ gpg -k
gpg: checking the trustdb
gpg: no ultimately trusted keys found
/home/richardfan/.gnupg/pubring.kbx
-----------------------------------
pub   rsa4096 2016-11-11 [SC]
      [redacted]
uid           [ unknown] [redacted] <[redacted]@gmail.com>
sub   rsa4096 2016-11-11 [E]

pub   rsa4096 2014-10-28 [SC]
      [redacted]
uid           [ unknown] [redacted] <[redacted]@gmail.com>
uid           [ unknown] [redacted] <[redacted].com>
uid           [ unknown] [jpeg image of size 5015]
sub   rsa2048 2015-11-02 [E]
sub   rsa4096 2014-10-28 [S] [expires: 2022-03-12]
```

Manage GPG keys is another hell

Another problem with this method is the process of pushing and pulling code. Developers may accidentally push secrets in plaintext. Or they may forget to encrypt and push secrets after modifying it.

Pros:
1. You can centrally manage everything of the project in 1 repository
2. Easy for development as everything on local is shown in plaintext

Cons:
1. Overhead of managing keys

2. It makes push/pull process complicated

SOPS: Encrypting using central key stores

After some research, I finally found Mozilla SOPS. The principle of SOPS is similar to git-secret, it helps you encrypt files before pushing them to the repository. What makes it different is that it supports many different key stores besides GPG, like AWS KMS. Using SOPS, we can now save secrets inside code repositories while centrally control access at the same time.

I will walkthrough you on how I integrate SOPS in my project

Step 1: Install SOPS

Goto SOPS release page and download the latest version. It has Windows, macOS, Ubuntu/Debian (.deb) and CentOS (.rpm) versions.

Step 2: Configure IAM user

SOPS needs our credential to access KMS key, I will create IAM user for each of my teammates (if they don't already have) and configure access key in my computer

1. Create IAM users with programmatic access

2. Install AWS CLI

3. Run `aws configure` to provide AWS CLI with the IAM user's access key

Step 3: Setup KMS custom key

We have to create a key that can be shared with all our teammates so that we can all use it to encrypt/decrypt secrets. In my case, I use AWS KMS, you can use other key stores that are supported.

1. Goto https://console.aws.amazon.com/kms/home#/kms/keys

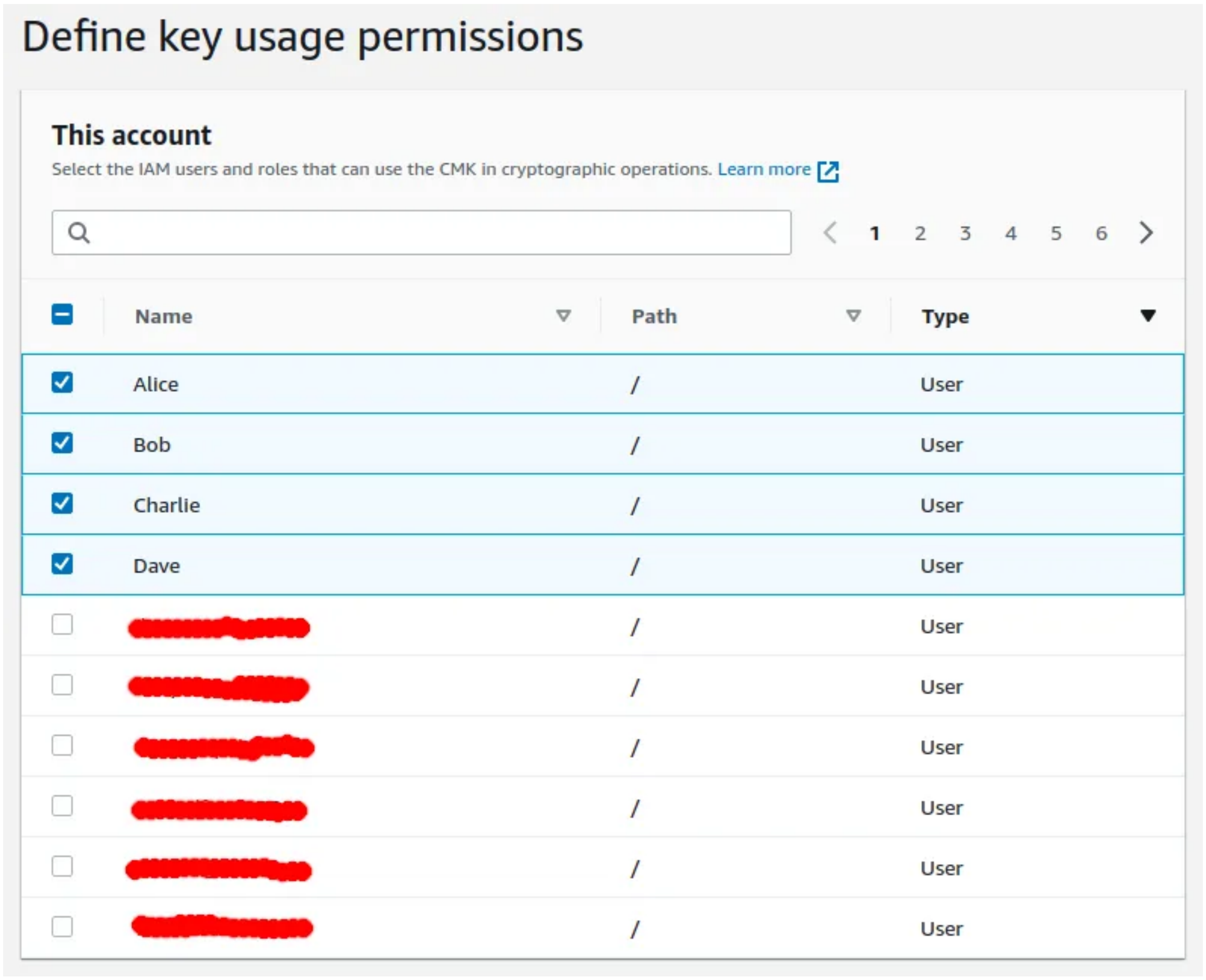
2. Click "Create key", keep everything as default

3. Give it an alias e.g. "my-sam-project-encryption-key"

4. Define who will be key administrators (who can config the key)

5. Define who can use the key (User that will use the key to encrypt/decrypt secrets)

6. Copy the ARN of the key, we will use it later



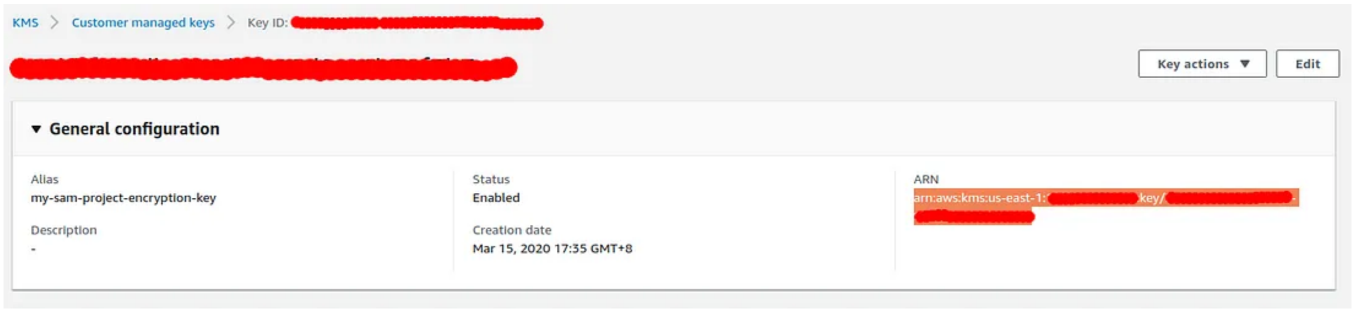
Define key usage permissions

This account

Select the IAM users and roles that can use the CMK in cryptographic operations. Learn more

	Name	Path	Type
✓	Alice	/	User
✓	Bob	/	User
✓	Charlie	/	User
✓	Dave	/	User
	[redacted]	/	User
	[redacted]	/	User
	[redacted]	/	User
	[redacted]	/	User
	[redacted]	/	User
	[redacted]	/	User

Select the users who will access the secrets



KMS > Customer managed keys > Key ID: [redacted]

[redacted]

▾ General configuration

Alias
my-sam-project-encryption-key

Description
-

Status
Enabled

Creation date
Mar 15, 2020 17:35 GMT+8

ARN
arn:aws:kms:us-east-1:[redacted]:key/[redacted]

Copy KMS key ID for later use

Step 4: Configure SOPS

SOPS use `.sops.yaml` file to configure which key to use when encrypting files. In my project, I used the following config:

```
creation_rules:
  - path_regex: \.env
    kms: 'arn:aws:kms:us-east-1:xxxxxxxxxxxx:key/xxxxxxxx-xxxx-xxxx-
xxxx-xxxxxxxxxxxx'
```

This file simply tells SOPS to use the KMS I specify (the ARN I have copied in the last step) to encrypt `.env` file. If you have multiple permission levels, you can create multiple keys in KMS and define more than one creation rules.

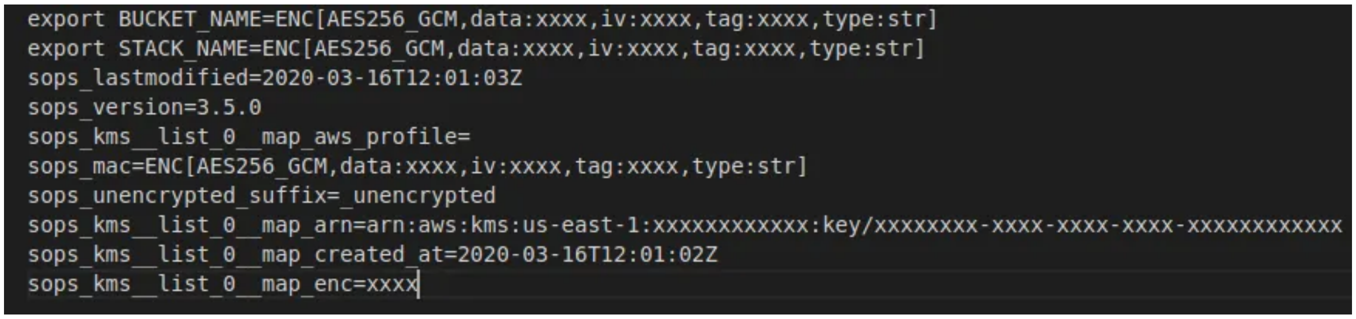
```
creation_rules:
  - path_regex: \.env$
    kms: 'arn:aws:kms:us-east-1:xxxxxxxxxxxx:key/xxxxxxxx-xxxx-xxxx-
xxxx-xxxxxxxxxxxx'
  - path_regex: \.prod\.env$
    kms: 'arn:aws:kms:us-east-1:xxxxxxxxxxxx:key/yyyyyyyy-yyyy-yyyy-
yyyy-yyyyyyyyyyyy
```

In this way, I can ensure `.env` and `.prod.env` will use 2 different keys to encrypt. So that I can ensure development team members will not necessarily gain access to the production environment.

Step 5: Encrypt secret

Before committing the project, we can run `sops -e --output .enc.env .env` This command encrypts the file `.env` into `.enc.env` If you have multiple

After encryption, my `.enc.env` looks like this:



```
export BUCKET_NAME=ENC[AES256_GCM,data:xxxx,iv:xxxx,tag:xxxx,type:str]
export STACK_NAME=ENC[AES256_GCM,data:xxxx,iv:xxxx,tag:xxxx,type:str]
sops_lastmodified=2020-03-16T12:01:03Z
sops_version=3.5.0
sops_kms__list_0__map_aws_profile=
sops_mac=ENC[AES256_GCM,data:xxxx,iv:xxxx,tag:xxxx,type:str]
sops_unencrypted_suffix=_unencrypted
sops_kms__list_0__map_arn=arn:aws:kms:us-east-1:xxxxxxxxxxxx:key/xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
sops_kms__list_0__map_created_at=2020-03-16T12:01:02Z
sops_kms__list_0__map_enc=xxxx
```

You can see that my bucket name and CloudFormation stack name have been encrypted. SOPS also append the file with some metadata, especially the key used for encryption.

Step 6: Add origin secret files into .gitignore

To prevent developers from accidentally commit plaintext secrets into the repository, we should add its filename into `.gitignore` so that git will not track them. In my case, my `.gitignore` should look like this:

```
# Do not commit secret files
.env
```

Step 7: Decrypt secrets after pulling updates

We have done with the encryption part, now my teammates have to decrypt it after pulling the repository. After pulling the update, simply run `sops -d --output .env .enc.env` If my teammate have configured access key correctly and have access to that KMS key, it will decrypt the file into the original `.env`

What's next?

Now, we can safely share the project secrets with teammates through the same repository. We can also centrally manage who can access the secret via AWS KMS. However, there are still human processes (encrypt and decrypt the files) which is not a good practise in the DevOps world. In part 2, we will talk about how to automate those processes.

DevOps Git Security Automation