SOPS: Secrets OPerationS

SOPS is an editor of encrypted files that supports YAML, JSON, ENV, INI and BINARY formats and encrypts with AWS KMS, GCP KMS, Azure Key Vault, age, and PGP.
(demo)

Usage

If you're using AWS KMS, create one or multiple master keys in the IAM console and export them, comma separated, in the **SOPS_KMS_ARN** env variable. It is recommended to use at least two master keys in different regions.

```
export SOPS_KMS_ARN="arn:aws:kms:us-east-1:9A9ACD6D8CAF:key/6DF5002e-c5f1-4040-943a-FB805C784D81,ar
```

SOPS uses aws-sdk-go-v2 to communicate with AWS KMS. It will automatically read the credentials from the `~/.aws/credentials` file which can be created with the `aws configure` command.

An example of the `~/.aws/credentials` file is shown below:

```
$ cat ~/.aws/credentials
[default]
aws_access_key_id = AKI.....
aws_secret_access_key = mw......
```

In addition to the `~/.aws/credentials` file, you can also use the `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` environment variables to specify your credentials:

```
export AWS_ACCESS_KEY_ID="AKI......"
export AWS_SECRET_ACCESS_KEY="mw......"
```

For more information and additional environment variables, see specifying credentials.

If you want to use PGP, export the fingerprints of the public keys, comma separated, in the **SOPS_PGP_FP** env variable.

```
export SOPS_PGP_FP="7A288ABC4C29DB49C315619243E8CFE4B6E43F21,19F76D446427659608A65E5F5C29ADE8C4893E
```

Note: you can use both PGP and KMS simultaneously.

Then simply call `sops edit` with a file path as argument. It will handle the encryption/decryption transparently and open the cleartext file in an editor

```
$ sops edit mynewtestfile.yaml
mynewtestfile.yaml doesn't exist, creating it.
please wait while an encryption key is being generated and stored in a secure fashion
file written to mynewtestfile.yaml
```

Editing will happen in whatever `$EDITOR` is set to, or, if it's not set, in vim. Keep in mind that SOPS will wait for the editor to exit, and then try to reencrypt the file. Some GUI editors (atom, sublime) spawn a child process and then exit immediately. They usually have an option to wait for the main editor window to be closed before exiting. See #127 for more information.

The resulting encrypted file looks like this:

```
myapp1: ENC[AES256_GCM,data:Tr7o=,iv:1=,aad:No=,tag:k=]
app2:
    db:
        user: ENC[AES256_GCM,data:CwE4O1s=,iv:2k=,aad:o=,tag:w==]
        password: ENC[AES256_GCM,data:p673w==,iv:YY=,aad:UQ=,tag:A=]
    # private key for secret operations in app2
    key: |-
        ENC[AES256_GCM,data:Ea3kL5O5U8=,iv:DM=,aad:FKA=,tag:EA==]
an_array:
    - ENC[AES256_GCM,data:v8jQ=,iv:HBE=,aad:21c=,tag:gA==]
    - ENC[AES256_GCM,data:X10=,iv:o8=,aad:CQ=,tag:Hw==]
    - ENC[AES256_GCM,data:KN=,iv:160=,aad:fI4=,tag:tNw==]
sops:
    kms:
        - created_at: EBBEA80389.88AC89
          enc: CiC....Pm1Hm
          arn: arn:aws:kms:us-east-1:9A9ACD6D8CAF:key/6DF5002e-c5f1-4040-943a-FB805C784D81
        - created_at: EBBEA80391.6DA8CB
          enc: Ci...awNx
          arn: arn:aws:kms:ap-southeast-1:9A9ACD6D8CAF:key/6FF957aa-0fa6-4c14-930e-5D2046E921E2
    pgp:
        - fp: 7A288ABC4C29DB49C315619243E8CFE4B6E43F21
          created_at: EBBEA80391.6CFFBD
          enc: |
              -----BEGIN PGP MESSAGE-----
              hQIMA0t4uZHfl9qgAQ//UvGAwGePyHuf2/zayWcloGaDs0MzI+zw6CmXvMRNPUsA
              ...=oJgS
              -----END PGP MESSAGE-----
```

A copy of the encryption/decryption key is stored securely in each KMS and PGP block. As long as one of the KMS or PGP method is still usable, you will be able to access your data.

To decrypt a file in a `cat` fashion, use the `-d` flag:

```
$ sops decrypt mynewtestfile.yaml
```

SOPS encrypted files contain the necessary information to decrypt their content. All a user of SOPS needs is valid AWS credentials and the necessary permissions on KMS keys.

Given that, the only command a SOPS user needs is:

```
$ sops edit <file>
```

[<file>]{.title-ref} will be opened, decrypted, passed to a text editor (vim by default), encrypted if modified, and saved back to its original location. All of these steps, apart from the actual editing, are transparent to the user.

The order in which available decryption methods are tried can be specified with `--decryption-order` option or **SOPS_DECRYPTION_ORDER** environment variable as a comma separated list. The default order is `age,pgp`. Offline methods are tried first and then the remaining ones.

Test with the dev PGP key

If you want to test **SOPS** without having to do a bunch of setup, you can use the example files and pgp key provided with the repository:

```
$ git clone https://github.com/getsops/sops.git
$ cd sops
$ gpg --import pgp/sops_functional_tests_key.asc
$ sops edit example.yaml
```

This last step will decrypt `example.yaml` using the test private key.

Encrypting with GnuPG subkeys

If you want to encrypt with specific GnuPG subkeys, it does not suffice to provide the exact key ID of the subkey to SOPS, since GnuPG might use *another* subkey instead to encrypt the file key with. To force GnuPG to use a specific subkey, you need to append `!` to the key's fingerprint.

```
creation_rules:
    - pgp: >-
        7A288ABC4C29DB49C315619243E8CFE4B6E43F21!,
        19F76D446427659608A65E5F5C29ADE8C4893E8F!
```

Please note that this is only passed on correctly to GnuPG since SOPS 3.9.3.

Encrypting using age

age is a simple, modern, and secure tool for encrypting files. It's recommended to use age over PGP, if possible.

You can encrypt a file for one or more age recipients (comma separated) using the `--age` option or the **SOPS_AGE_RECIPIENTS** environment variable:

```
$ sops encrypt --age age1yt3tfqlfrwdwx0z0ynwplcr6qxcxfaqycuprpmy89nr83ltx74tqdpszlw test.yaml > tes
```

When decrypting a file with the corresponding identity, SOPS will look for a text file name `keys.txt` located in a `sops` subdirectory of your user configuration directory. On Linux, this would be `$XDG_CONFIG_HOME/sops/age/keys.txt`. On macOS, this would be `$HOME/Library/Application Support/sops/age/keys.txt`. On Windows, this would be `%AppData%\sops\age\keys.txt`. You can specify the location of this file manually by setting the environment variable **SOPS_AGE_KEY_FILE**. Alternatively, you can provide the key(s) directly by setting the **SOPS_AGE_KEY** environment variable.

The contents of this key file should be a list of age X25519 identities, one per line. Lines beginning with `#` are considered comments and ignored. Each identity will be tried in sequence until one is able to decrypt the data.

Encrypting with SSH keys via age is not yet supported by SOPS.

A list of age recipients can be added to the `.sops.yaml`:

```
creation_rules:
    - age: >-
        age1s3cqcks5genc6ru8chl0hkkd04zmxvczsvdxq99ekffe4gmvjpzsedk23c,
        age1qe5lxzzeppw5k79vxnC78DD82sgy224g2nzqlzy3uljs84say3yqgvd0sw
```

It is also possible to use `updatekeys`, when adding or removing age recipients. For example:

```
$ sops updatekeys secret.enc.yaml
2022/02/09 16:32:02 Syncing keys for file /iac/solution1/secret.enc.yaml
The following changes will be made to the file's groups:
Group 1
    age1s3cqcks5genc6ru8chl0hkkd04zmxvczsvdxq99ekffe4gmvjpzsedk23c
+++ age1qe5lxzzeppw5k79vxnC78DD82sgy224g2nzqlzy3uljs84say3yqgvd0sw
Is this okay? (y/n):y
2022/02/09 16:32:04 File /iac/solution1/secret.enc.yaml synced with new keys
```

Encrypting using GCP KMS

GCP KMS uses Application Default Credentials. If you already logged in using

```
$ gcloud auth login
```

you can enable application default credentials using the sdk:

```
$ gcloud auth application-default login
```

Encrypting/decrypting with GCP KMS requires a KMS ResourceID. You can use the cloud console the get the ResourceID or you can create one using the gcloud sdk:

```
$ gcloud kms keyrings create sops --location global
$ gcloud kms keys create sops-key --location global --keyring sops --purpose encryption
$ gcloud kms keys list --location global --keyring sops

# you should see
NAME                                                                PURPOSE          PRIMARY_STA
projects/my-project/locations/global/keyRings/sops/cryptoKeys/sops-key ENCRYPT_DECRYPT  ENABLED
```

Now you can encrypt a file using:

```
$ sops encrypt --gcp-kms projects/my-project/locations/global/keyRings/sops/cryptoKeys/sops-key tes
```

And decrypt it using:

```
$ sops decrypt test.enc.yaml
```

Encrypting using Azure Key Vault

The Azure Key Vault integration uses the default credential chain which tries several authentication methods, in this order:

1. Environment credentials i. Service Principal with Client Secret ii. Service Principal with Certificate iii. User with username and password iv. Configuration for multi-tenant applications
2. Workload Identity credentials
3. Managed Identity credentials
4. Azure CLI credentials

For example, you can use a Service Principal with the following environment

variables:

```
AZURE_TENANT_ID
AZURE_CLIENT_ID
AZURE_CLIENT_SECRET
```

You can create a Service Principal using the CLI like this:

```
$ az ad sp create-for-rbac -n my-keyvault-sp

{
    "appId": "<some-uuid>",
    "displayName": "my-keyvault-sp",
    "name": "http://my-keyvault-sp",
    "password": "<random-string>",
    "tenant": "<tenant-uuid>"
}
```

The [appId]{.title-ref} is the client ID, and the [password]{.title-ref} is the client secret.

Encrypting/decrypting with Azure Key Vault requires the resource identifier for a key. This has the following form:

```
https://${VAULT_URL}/keys/${KEY_NAME}/${KEY_VERSION}
```

To create a Key Vault and assign your service principal permissions on it from the commandline:

```
# Create a resource group if you do not have one:
$ az group create --name sops-rg --location westeurope
# Key Vault names are globally unique, so generate one:
$ keyvault_name=sops-$(uuidgen | tr -d - | head -c 16)
# Create a Vault, a key, and give the service principal access:
$ az keyvault create --name $keyvault_name --resource-group sops-rg --location westeurope
$ az keyvault key create --name sops-key --vault-name $keyvault_name --protection software --ops er
$ az keyvault set-policy --name $keyvault_name --resource-group sops-rg --spn $AZURE_CLIENT_ID \
    --key-permissions encrypt decrypt
# Read the key id:
$ az keyvault key show --name sops-key --vault-name $keyvault_name --query key.kid

https://sops.vault.azure.net/keys/sops-key/some-string
```

Now you can encrypt a file using:

```
$ sops encrypt --azure-kv https://sops.vault.azure.net/keys/sops-key/some-string test.yaml > test.e
```

And decrypt it using:

```
$ sops decrypt test.enc.yaml
```

Encrypting and decrypting from other programs

When using `sops` in scripts or from other programs, there are often situations where you do not want to write encrypted or decrypted data to disk. The best way to avoid this is to pass data to SOPS via stdin, and to let SOPS write data to stdout. By default, the encrypt and decrypt operations write data to stdout already. To pass data via stdin, you need to pass `/dev/stdin` as the input filename. Please note that this only works on Unix-like operating systems such as macOS and Linux. On Windows, you have to use named pipes.

To decrypt data, you can simply do:

```
$ cat encrypted-data | sops decrypt /dev/stdin > decrypted-data
```

To control the input and output format, pass `--input-type` and `--output-type` as appropriate. By default, `sops` determines the input and output format from the provided filename, which is `/dev/stdin` here, and thus will use the binary store which expects JSON input and outputs binary data on decryption.

For example, to decrypt YAML data and obtain the decrypted result as YAML, use:

```
$ cat encrypted-data | sops decrypt --input-type yaml --output-type yaml /dev/stdin > decrypted-dat
```

To encrypt, it is important to note that SOPS also uses the filename to look up the correct creation rule from `.sops.yaml`. Likely `/dev/stdin` will not match a creation rule, or only match the fallback rule without `path_regex`, which is usually not what you want. For that, `sops` provides the `--filename-override` parameter which allows you to tell SOPS which filename to use to match creation rules:

```
$ echo 'foo: bar' | sops encrypt --filename-override path/filename.sops.yaml /dev/stdin > encrypted
```

SOPS will find a matching creation rule for `path/filename.sops.yaml` in `.sops.yaml` and use that one to encrypt the data from stdin. This filename will also be used to determine the input and output store. As always, the input store type can be adjusted by passing `--input-type`, and the output store type by passing `--output-type`:

```
$ echo foo=bar | sops encrypt --filename-override path/filename.sops.yaml --input-type dotenv /dev/
```

Encrypting using Hashicorp Vault

We assume you have an instance (or more) of Vault running and you have privileged access to it. For instructions on how to deploy a secure instance of Vault, refer to Hashicorp's official documentation.

To easily deploy Vault locally: (DO NOT DO THIS FOR PRODUCTION!!!)

```
$ docker run -d -p8200:8200 vault:1.2.0 server -dev -dev-root-token-id=toor
```

```
$ # Substitute this with the address Vault is running on
$ export VAULT_ADDR=http://127.0.0.1:8200

$ # this may not be necessary in case you previously used `vault login` for production use
$ export VAULT_TOKEN=toor

$ # to check if Vault started and is configured correctly
$ vault status
Key             Value
---             -----
Seal Type       shamir
Initialized     true
Sealed          false
Total Shares    1
Threshold       1
Version         1.2.0
```

```
Cluster Name    vault-cluster-9E733602
Cluster ID      1ACD1B61-e8f0-1352-8a41-0383EEF696F7
HA Enabled      false

$ # It is required to enable a transit engine if not already done (It is suggested to create a tran
$ vault secrets enable -path=sops transit
Success! Enabled the transit secrets engine at: sops/

$ # Then create one or more keys
$ vault write sops/keys/firstkey type=rsa-4096
Success! Data written to: sops/keys/firstkey

$ vault write sops/keys/secondkey type=rsa-2048
Success! Data written to: sops/keys/secondkey

$ vault write sops/keys/thirdkey type=chacha20-poly1305
Success! Data written to: sops/keys/thirdkey

$ sops encrypt --hc-vault-transit $VAULT_ADDR/v1/sops/keys/firstkey vault_example.yml

$ cat <<EOF > .sops.yaml
creation_rules:
    - path_regex: \.dev\.yaml$
      hc_vault_transit_uri: "$VAULT_ADDR/v1/sops/keys/secondkey"
    - path_regex: \.prod\.yaml$
      hc_vault_transit_uri: "$VAULT_ADDR/v1/sops/keys/thirdkey"
EOF

$ sops encrypt --verbose prod/raw.yaml > prod/encrypted.yaml
```

Adding and removing keys

When creating new files, `sops` uses the PGP, KMS and GCP KMS defined in the command line arguments `--kms`, `--pgp`, `--gcp-kms` or `--azure-kv`, or from the environment variables `SOPS_KMS_ARN`, `SOPS_PGP_FP`, `SOPS_GCP_KMS_IDS`, `SOPS_AZURE_KEYVAULT_URLS`. That information is stored in the file under the `sops` section, such that decrypting files does not require providing those parameters again.

Master PGP and KMS keys can be added and removed from a `sops` file in one of three ways:

1. By using a `.sops.yaml` file and the `updatekeys` command.
2. By using command line flags.
3. By editing the file directly.

The SOPS team recommends the `updatekeys` approach.

`updatekeys` command

The `updatekeys` command uses the .sops.yaml configuration file to update (add or remove) the corresponding secrets in the encrypted file. Note that the example below uses the Block Scalar yaml construct to build a space separated list.

```
creation_rules:
    - pgp: >-
        7A288ABC4C29DB49C315619243E8CFE4B6E43F21,
        0438461D5B06D7653F3E2B7BC2E9311B5D8C81B4
```

```
$ sops updatekeys test.enc.yaml
```

SOPS will prompt you with the changes to be made. This interactivity can be disabled by supplying the `-y` flag.

`rotate` command

The `rotate` command generates a new data encryption key and reencrypt all values with the new key. At the same time, the command line flag `--add-kms`, `--add-pgp`, `--add-gcp-kms`, `--add-azure-kv`, `--rm-kms`, `--rm-pgp`, `--rm-gcp-kms` and `--rm-azure-kv` can be used to add and remove keys from a file. These flags use the comma separated syntax as the `--kms`, `--pgp`, `--gcp-kms` and `--azure-kv` arguments when creating new files.

Use `updatekeys` if you want to add a key without rotating the data key.

```
# add a new pgp key to the file and rotate the data key
$ sops rotate -i --add-pgp 7A288ABC4C29DB49C315619243E8CFE4B6E43F21 example.yaml

# remove a pgp key from the file and rotate the data key
$ sops rotate -i --rm-pgp 7A288ABC4C29DB49C315619243E8CFE4B6E43F21 example.yaml
```

Direct Editing

Alternatively, invoking `sops edit` with the flag **-s** will display the master keys while editing. This method can be used to add or remove `kms` or `pgp` keys under the `sops` section.

For example, to add a KMS master key to a file, add the following entry while editing:

```
sops:
    kms:
        - arn: arn:aws:kms:us-east-1:9A9ACD6D8CAF:key/6DF5002e-c5f1-4040-943a-FB805C784D81
```

And, similarly, to add a PGP master key, we add its fingerprint:

```
sops:
    pgp:
        - fp: 7A288ABC4C29DB49C315619243E8CFE4B6E43F21
```

When the file is saved, SOPS will update its metadata and encrypt the data key with the freshly added master keys. The removed entries are simply deleted from the file.

When removing keys, it is recommended to rotate the data key using `-r`, otherwise, owners of the removed key may have add access to the data key in the past.

KMS AWS Profiles

If you want to use a specific profile, you can do so with `aws_profile`:

```
sops:
    kms:
        - arn: arn:aws:kms:us-east-1:9A9ACD6D8CAF:key/6DF5002e-c5f1-4040-943a-FB805C784D81
          aws_profile: foo
```

If no AWS profile is set, default credentials will be used.

Similarly the [--aws-profile]{.title-ref} flag can be set with the command line with any of the KMS commands.

Assuming roles and using KMS in various AWS accounts

SOPS has the ability to use KMS in multiple AWS accounts by assuming roles in each account. Being able to assume roles is a nice feature of AWS that allows administrators to establish trust relationships between accounts, typically from the most secure account to the least secure one. In our use-case, we use roles to indicate that a user of the Master AWS account is allowed to make use of KMS master keys in development and staging AWS accounts. Using roles, a single file can be encrypted with KMS keys in multiple accounts, thus increasing reliability and ease of use.

You can use keys in various accounts by tying each KMS master key to a role that the user is allowed to assume in each account. The IAM roles documentation has full details on how this needs to be configured on AWS's side.

From the point of view of SOPS, you only need to specify the role a KMS key must assume alongside its ARN, as follows:

```
sops:
    kms:
        - arn: arn:aws:kms:us-east-1:9A9ACD6D8CAF:key/6DF5002e-c5f1-4040-943a-FB805C784D81
          role: arn:aws:iam::6D8FCB797D8C:role/sops-dev-xyz
```

The role must have permission to call Encrypt and Decrypt using KMS. An example policy is shown below.

```
{
  "Sid": "Allow use of the key",
  "Effect": "Allow",
  "Action": [
    "kms:Encrypt",
    "kms:Decrypt",
    "kms:ReEncrypt*",
    "kms:GenerateDataKey*",
    "kms:DescribeKey"
  ],
  "Resource": "*",
  "Principal": {
    "AWS": [
      "arn:aws:iam::6D8FCB797D8C:role/sops-dev-xyz"
    ]
  }
}
```

You can specify a role in the `--kms` flag and `SOPS_KMS_ARN` variable by appending it to the ARN of the master key, separated by a **+** sign:

```
<KMS ARN>+<ROLE ARN>
arn:aws:kms:us-west-2:6D8FCB797D8C:key/01792269-4132-404c-ab86-BD966A94BAFF+arn:aws:iam::6D8FCB797D
```

AWS KMS Encryption Context

SOPS has the ability to use AWS KMS key policy and encryption context to refine the access control of a given KMS master key.

When creating a new file, you can specify the encryption context in the `--encryption-context` flag by comma separated list of key-value pairs:

```
$ sops edit --encryption-context Environment:production,Role:web-server test.dev.yaml
```

The format of the Encrypt Context string is `<EncryptionContext Key>:<EncryptionContext Value>,<EncryptionContext Key>:<EncryptionContext Value>,...`

The encryption context will be stored in the file metadata and does not need to be provided at decryption.

Encryption contexts can be used in conjunction with KMS Key Policies to define roles that can only access a given context. An example policy is shown below:

```
{
  "Effect": "Allow",
  "Principal": {
    "AWS": "arn:aws:iam::EEEEDDDDCCCC:role/RoleForExampleApp"
  },
  "Action": "kms:Decrypt",
  "Resource": "*",
  "Condition": {
    "StringEquals": {
      "kms:EncryptionContext:AppName": "ExampleApp",
      "kms:EncryptionContext:FilePath": "/var/opt/secrets/"
    }
  }
}
```

Key Rotation

It is recommended to renew the data key on a regular basis. `sops` supports key rotation via the `rotate` command. Invoking it on an existing file causes `sops` to reencrypt the file with a new data key, which is then encrypted with the various KMS and PGP master keys defined in the file.

Add the `-i` option to write the rotated file back, instead of printing it to stdout.

```
$ sops rotate example.yaml
```

Using .sops.yaml conf to select KMS, PGP and age for new files

It is often tedious to specify the `--kms --gcp-kms --pgp` and `--age` parameters for creation of all new files. If your secrets are stored under a specific directory, like a `git` repository, you can create a `.sops.yaml` configuration file at the root directory to define which keys are used for which filename.

Note: The file needs to be named `.sops.yaml`. Other names (i.e. `.sops.yml`) won't be automatically discovered by sops. You'll need to pass the `--config .sops.yml` option for it to be picked up.

Let's take an example:

- file named **something.dev.yaml** should use one set of KMS A, PGP and age
- file named **something.prod.yaml** should use another set of KMS B, PGP and age
- other files use a third set of KMS C and PGP
- all live under **mysecretrepo/something.{dev,prod,gcp}.yaml**

Under those circumstances, a file placed at **mysecretrepo/.sops.yaml** can manage the three sets of configurations for the three types of files:

```
# creation rules are evaluated sequentially, the first match wins
creation_rules:
    # upon creation of a file that matches the pattern *.dev.yaml,
    # KMS set A as well as PGP and age is used
    - path_regex: \.dev\.yaml$
      kms: 'arn:aws:kms:us-west-2:6D8FCB797D8C:key/01792269-4132-404c-ab86-BD966A94BAFF,arn:aws:kms
      pgp: '0438461D5B06D7653F3E2B7BC2E9311B5D8C81B4'
      age: 'age129h70qwx39k7h5x6l9hg566nwm53527zvamre8vep9e3plsm44uqgy8gla'

    # prod files use KMS set B in the PROD IAM, PGP and age
    - path_regex: \.prod\.yaml$
      kms: 'arn:aws:kms:us-west-2:C9EAD8F89ADC:key/AFAD0F6a-5d3f-489e-b86c-A8DFE1F90CE1+arn:aws:iam
      pgp: '0438461D5B06D7653F3E2B7BC2E9311B5D8C81B4'
      age: 'age129h70qwx39k7h5x6l9hg566nwm53527zvamre8vep9e3plsm44uqgy8gla'
      hc_vault_uris: "http://localhost:8200/v1/sops/keys/thirdkey"

    # gcp files using GCP KMS
    - path_regex: \.gcp\.yaml$
      gcp_kms: projects/mygcproject/locations/global/keyRings/mykeyring/cryptoKeys/thekey

    # Finally, if the rules above have not matched, this one is a
    # catchall that will encrypt the file using KMS set C as well as PGP
    # The absence of a path_regex means it will match everything
    - kms: 'arn:aws:kms:us-west-2:6D8FCB797D8C:key/01792269-4132-404c-ab86-BD966A94BAFF,arn:aws:kms
      pgp: '0438461D5B06D7653F3E2B7BC2E9311B5D8C81B4'
```

When creating any file under **mysecretrepo**, whether at the root or under a subdirectory, SOPS will recursively look for a `.sops.yaml` file. If one is found, the filename of the file being created is compared with the filename regexes of the configuration file. The first regex that matches is selected, and its KMS and PGP keys are used to encrypt the file. It should be noted that the looking up of `.sops.yaml` is from the working directory (CWD) instead of the directory of the encrypting file (see Issue 242).

The `path_regex` checks the path of the encrypting file relative to the `.sops.yaml` config file. Here is another example:

- files located under directory **development** should use one set of KMS A
- files located under directory **production** should use another set of KMS B
- other files use a third set of KMS C

```
creation_rules:
    # upon creation of a file under development,
    # KMS set A is used
    - path_regex: .*/development/.*
      kms: 'arn:aws:kms:us-west-2:6D8FCB797D8C:key/01792269-4132-404c-ab86-BD966A94BAFF,arn:aws:kms
      pgp: '0438461D5B06D7653F3E2B7BC2E9311B5D8C81B4'

    # prod files use KMS set B in the PROD IAM
    - path_regex: .*/production/.*
      kms: 'arn:aws:kms:us-west-2:C9EAD8F89ADC:key/AFAD0F6a-5d3f-489e-b86c-A8DFE1F90CE1+arn:aws:iam
      pgp: '0438461D5B06D7653F3E2B7BC2E9311B5D8C81B4'

    # other files use KMS set C
    - kms: 'arn:aws:kms:us-west-2:6D8FCB797D8C:key/01792269-4132-404c-ab86-BD966A94BAFF,arn:aws:kms
      pgp: '0438461D5B06D7653F3E2B7BC2E9311B5D8C81B4'
```

Creating a new file with the right keys is now as simple as

```
$ sops edit <newfile>.prod.yaml
```

Note that the configuration file is ignored when KMS or PGP parameters are passed on the SOPS command line or in environment variables.

Specify a different GPG executable

SOPS checks for the `SOPS_GPG_EXEC` environment variable. If specified, it will attempt to use the executable set there instead of the default of `gpg`.

Example: place the following in your `~/.bashrc`

```
SOPS_GPG_EXEC = 'your_gpg_client_wrapper'
```

Specify a different GPG key server

By default, SOPS uses the key server `keys.openpgp.org` to retrieve the GPG keys that are not present in the local keyring. This is no longer configurable. You can learn more about why from this write-up: SKS Keyserver Network Under Attack.

Key groups

By default, SOPS encrypts the data key for a file with each of the master keys, such that if any of the master keys is available, the file can be decrypted. However, it is sometimes desirable to require access to multiple master keys in order to decrypt files. This can be achieved with key groups.

When using key groups in SOPS, data keys are split into parts such that keys from multiple groups are required to decrypt a file. SOPS uses Shamir's Secret Sharing to split the data key such that each key group has a fragment, each key in the key group can decrypt that fragment, and a configurable number of fragments (threshold) are needed to decrypt and piece together the complete data key. When decrypting a file using multiple key groups, SOPS goes through key groups in order, and in each group, tries to recover the fragment of the data key using a master key from that group. Once the fragment is recovered, SOPS moves on to the next group, until enough fragments have been recovered to obtain the complete data key.

By default, the threshold is set to the number of key groups. For example, if you have three key groups configured in your SOPS file and you don't override the default threshold, then one master key from each of the three groups will be required to decrypt the file.

Management of key groups is done with the `sops groups` command.

For example, you can add a new key group with 3 PGP keys and 3 KMS keys to the file `my_file.yaml`:

```
$ sops groups add --file my_file.yaml --pgp fingerprint1 --pgp fingerprint2 --pgp fingerprint3 --km
```

Or you can delete the 1st group (group number 0, as groups are zero-indexed) from

my_file.yaml:

```
$ sops groups delete --file my_file.yaml 0
```

Key groups can also be specified in the `.sops.yaml` config file, like so:

```
creation_rules:
    - path_regex: .*keygroups.*
      key_groups:
          # First key group
          - pgp:
                - fingerprint1
                - fingerprint2
            kms:
                - arn: arn1
                  role: role1
                  context:
                      foo: bar
                - arn: arn2
                  aws_profile: myprofile
          # Second key group
          - pgp:
                - fingerprint3
                - fingerprint4
            kms:
                - arn: arn3
                - arn: arn4
          # Third key group
          - pgp:
                - fingerprint5
```

Given this configuration, we can create a new encrypted file like we normally would, and optionally provide the `--shamir-secret-sharing-threshold` command line flag if we want to override the default threshold. SOPS will then split the data key into three parts (from the number of key groups) and encrypt each fragment with the master keys found in each group.

For example:

```
$ sops edit --shamir-secret-sharing-threshold 2 example.json
```

Alternatively, you can configure the Shamir threshold for each creation rule in the `.sops.yaml` config with `shamir_threshold`:

```
creation_rules:
    - path_regex: .*keygroups.*
      shamir_threshold: 2
      key_groups:
          # First key group
          - pgp:
                - fingerprint1
                - fingerprint2
            kms:
                - arn: arn1
                  role: role1
                  context:
                      foo: bar
                - arn: arn2
                  aws_profile: myprofile
          # Second key group
          - pgp:
                - fingerprint3
                - fingerprint4
            kms:
                - arn: arn3
                - arn: arn4
          # Third key group
          - pgp:
                - fingerprint5
```

And then run `sops edit example.json`.

The threshold (`shamir_threshold`) is set to 2, so this configuration will require master keys from two of the three different key groups in order to decrypt the file. You can then decrypt the file the same way as with any other SOPS file:

```
$ sops decrypt example.json
```

Key service

There are situations where you might want to run SOPS on a machine that doesn't have direct access to encryption keys such as PGP keys. The `sops` key service allows you to forward a socket so that SOPS can access encryption keys stored on a remote machine. This is similar to GPG Agent, but more portable.

SOPS uses a client-server approach to encrypting and decrypting the data key. By default, SOPS runs a local key service in-process. SOPS uses a key service client to send an encrypt or decrypt request to a key service, which then performs the operation. The requests are sent using gRPC and Protocol Buffers. The requests contain an identifier for the key they should perform the operation with, and the plaintext or encrypted data key. The requests do not contain any cryptographic keys, public or private.

WARNING: the key service connection currently does not use any sort of authentication or encryption. Therefore, it is recommended that you make sure the connection is authenticated and encrypted in some other way, for example through an SSH tunnel.

Whenever we try to encrypt or decrypt a data key, SOPS will try to do so first with the local key service (unless it's disabled), and if that fails, it will try all other remote key services until one succeeds.

You can start a key service server by running `sops keyservice`.

You can specify the key services the `sops` binary uses with `--keyservice`. This flag can be specified more than once, so you can use multiple key services. The local key service can be disabled with `enable-local-keyservice=false`.

For example, to decrypt a file using both the local key service and the key service exposed on the unix socket located in `/tmp/sops.sock`, you can run:

```
$ sops decrypt --keyservice unix:///tmp/sops.sock file.yaml`
```

And if you only want to use the key service exposed on the unix socket located in `/tmp/sops.sock` and not the local key service, you can run:

```
$ sops decrypt --enable-local-keyservice=false --keyservice unix:///tmp/sops.sock file.yaml
```

Auditing

Sometimes, users want to be able to tell what files were accessed by whom in an environment they control. For this reason, SOPS can generate audit logs to record activity on encrypted files. When enabled, SOPS will write a log entry into a pre-configured PostgreSQL database when a file is decrypted. The log includes a timestamp, the username SOPS is running as, and the file that was decrypted.

In order to enable auditing, you must first create the database and credentials using the schema found in `audit/schema.sql`. This schema defines the tables that store the audit events and a role named `sops` that only has permission to add entries to the audit event tables. The default password for the role `sops` is `sops`. You should change this password.

Once you have created the database, you have to tell SOPS how to connect to it. Because we don't want users of SOPS to be able to control auditing, the audit configuration file location is not configurable, and must be at `/etc/sops/audit.yaml`. This file should have strict permissions such that only the root user can modify it.

For example, to enable auditing to a PostgreSQL database named `sops` running on localhost, using the user `sops` and the password `sops`, `/etc/sops/audit.yaml` should have the following contents:

```
backends:
    postgres:
        - connection_string: "postgres://sops:sops@localhost/sops?sslmode=verify-full"
```

You can find more information on the `connection_string` format in the PostgreSQL docs.

Under the `postgres` map entry in the above YAML is a list, so one can provide more than one backend, and SOPS will log to all of them:

```
backends:
    postgres:
        - connection_string: "postgres://sops:sops@localhost/sops?sslmode=verify-full"
        - connection_string: "postgres://sops:sops@remotehost/sops?sslmode=verify-full"
```

Saving Output to a File

By default SOPS just dumps all the output to the standard output. We can use the `--output` flag followed by a filename to save the output to the file specified. Beware using both `--in-place` and `--output` flags will result in an error.

Passing Secrets to Other Processes

In addition to writing secrets to standard output and to files on disk, SOPS has two commands for passing decrypted secrets to a new process: `exec-env` and `exec-file`. These commands will place all output into the environment of a child process and into a temporary file, respectively. For example, if a program looks for credentials in its environment, `exec-env` can be used to ensure that the decrypted contents are available only to this process and never written to disk.

```
# print secrets to stdout to confirm values
$ sops decrypt out.json
{
        "database_password": "jf48t9wfw094gf4nhdf023r",
        "AWS_ACCESS_KEY_ID": "AKIAIOSFODNN7EXAMPLE",
        "AWS_SECRET_KEY": "wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY"
}

# decrypt out.json and run a command
# the command prints the environment variable and runs a script that uses it
$ sops exec-env out.json 'echo secret: $database_password; ./database-import'
secret: jf48t9wfw094gf4nhdf023r

# launch a shell with the secrets available in its environment
$ sops exec-env out.json 'sh'
sh-3.2# echo $database_password
jf48t9wfw094gf4nhdf023r

# the secret is not accessible anywhere else
sh-3.2$ exit
$ echo your password: $database_password
your password:
```

If the command you want to run only operates on files, you can use `exec-file` instead. By default, SOPS will use a FIFO to pass the contents of the decrypted file to the new program. Using a FIFO, secrets are only passed in memory which has two benefits: the plaintext secrets never touch the disk, and the child process can only read the secrets once. In contexts where this won't work, eg platforms like Windows where FIFOs unavailable or secret files that need to be available to the child process longer term, the `--no-fifo` flag can be used to instruct SOPS to use a traditional temporary file that will get cleaned up once the process is finished executing. `exec-file` behaves similar to `find(1)` in that `{}` is used as a placeholder in the command which will be substituted with the temporary file path (whether a FIFO or an actual file).

```
# operating on the same file as before, but as a file this time
$ sops exec-file out.json 'echo your temporary file: {}; cat {}'
your temporary file: /tmp/.sops76B9AF499/tmp-file
{
        "database_password": "jf48t9wfw094gf4nhdf023r",
        "AWS_ACCESS_KEY_ID": "AKIAIOSFODNN7EXAMPLE",
        "AWS_SECRET_KEY": "wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY"
}

# launch a shell with a variable TMPFILE pointing to the temporary file
$ sops exec-file --no-fifo out.json 'TMPFILE={} sh'
sh-3.2$ echo $TMPFILE
/tmp/.sopsAF9FAA069/tmp-fil1D6EEC8648
sh-3.2$ cat $TMPFILE
{
        "database_password": "jf48t9wfw094gf4nhdf023r",
        "AWS_ACCESS_KEY_ID": "AKIAIOSFODNN7EXAMPLE",
        "AWS_SECRET_KEY": "wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY"
}
sh-3.2$ ./program --config $TMPFILE
sh-3.2$ exit

# try to open the temporary file from earlier
$ cat /tmp/.sopsAF9FAA069/tmp-fil1D6EEC8648
cat: /tmp/.sopsAF9FAA069/tmp-fil1D6EEC8648: No such file or directory
```

Additionally, on unix-like platforms, both `exec-env` and `exec-file` support dropping

privileges before executing the new program via the `--user <username>` flag. This is particularly useful in cases where the encrypted file is only readable by root, but the target program does not need root privileges to function. This flag should be used where possible for added security.

To overwrite the default file name (`tmp-file`) in `exec-file` use the `--filename <filename>` parameter.

```
# the encrypted file can't be read by the current user
$ cat out.json
cat: out.json: Permission denied

# execute sops as root, decrypt secrets, then drop privileges
$ sudo sops exec-env --user nobody out.json 'sh'
sh-3.2$ echo $database_password
jf48t9wfw094gf4nhdf023r

# dropped privileges, still can't load the original file
sh-3.2$ id
uid=BD6B697294(nobody) gid=BD6B697294(nobody) groups=BD6B697294(nobody)
sh-3.2$ cat out.json
cat: out.json: Permission denied
```

Using the publish command

`sops publish $file` publishes a file to a pre-configured destination (this lives in the SOPS config file). Additionally, support re-encryption rules that work just like the creation rules.

This command requires a `.sops.yaml` configuration file. Below is an example:

```
destination_rules:
    - s3_bucket: "sops-secrets"
      path_regex: s3/*
      recreation_rule:
          pgp: 0961B6FE12452D2E8AC0739859BACA3BE9C0B307
    - gcs_bucket: "sops-secrets"
      path_regex: gcs/*
      recreation_rule:
          pgp: 0961B6FE12452D2E8AC0739859BACA3BE9C0B307
    - vault_path: "sops/"
      vault_kv_mount_name: "secret/" # default
      vault_kv_version: 2 # default
      path_regex: vault/*
      omit_extensions: true
```

The above configuration will place all files under `s3/*` into the S3 bucket `sops-secrets`, all files under `gcs/*` into the GCS bucket `sops-secrets`, and the contents of all files under `vault/*` into Vault's KV store under the path `secrets/sops/`. For the files that will be published to S3 and GCS, it will decrypt them and re-encrypt them using the `0961B6FE12452D2E8AC0739859BACA3BE9C0B307` pgp key.

You would deploy a file to S3 with a command like: `sops publish s3/app.yaml`

To publish all files in selected directory recursively, you need to specify `--recursive` flag.

If you don't want file extension to appear in destination secret path, use `--omit-extensions` flag or `omit_extensions: true` in the destination rule in `.sops.yaml`.

Publishing to Vault

There are a few settings for Vault that you can place in your destination rules. The first is `vault_path`, which is required. The others are optional, and they are `vault_address`, `vault_kv_mount_name`, `vault_kv_version`.

SOPS uses the official Vault API provided by Hashicorp, which makes use of environment variables for configuring the client.

`vault_kv_mount_name` is used if your Vault KV is mounted somewhere other than `secret/`. `vault_kv_version` supports `1` and `2`, with `2` being the default.

If the destination secret path already exists in Vault and contains the same data as the source file, it will be skipped.

Below is an example of publishing to Vault (using token auth with a local dev instance of Vault).

```
$ export VAULT_TOKEN=...
$ export VAULT_ADDR='http://127.0.0.1:8200'
$ sops decrypt vault/test.yaml
example_string: bar
example_number: 42
example_map:
    key: value
$ sops publish vault/test.yaml
uploading /home/user/sops_directory/vault/test.yaml to http://127.0.0.1:8200/v1/secret/data/sops/te
$ vault kv get secret/sops/test.yaml
====== Metadata ======
Key              Value
---              -----
created_time     2019-07-11T03:32:17.F8B86D017Z
deletion_time    n/a
destroyed        false
version          3

========= Data =========
Key               Value
---               -----
example_map       map[key:value]
example_number    42
example_string    bar
```

Important information on types

YAML, JSON, ENV and INI type extensions

SOPS uses the file extension to decide which encryption method to use on the file content. `YAML`, `JSON`, `ENV`, and `INI` files are treated as trees of data, and key/values are extracted from the files to only encrypt the leaf values. The tree structure is also used to check the integrity of the file.

Therefore, if a file is encrypted using a specific format, it needs to be decrypted in the same format. The easiest way to achieve this is to conserve the original file extension after encrypting a file. For example:

```
$ sops encrypt -i myfile.json
$ sops decrypt myfile.json
```

If you want to change the extension of the file once encrypted, you need to provide `sops` with the `--input-type` flag upon decryption. For example:

```
$ sops encrypt myfile.json > myfile.json.enc

$ sops decrypt --input-type json myfile.json.enc
```

When operating on stdin, use the `--input-type` and `--output-type` flags as follows:

```
$ cat myfile.json | sops decrypt --input-type json --output-type json /dev/stdin
```

JSON and JSON_binary indentation

SOPS indents `JSON` files by default using one `tab`. However, you can change this default behaviour to use `spaces` by either using the additional `--indent=2` CLI option or by configuring `.sops.yaml` with the code below.

The special value `0` disables indentation, and `-1` uses a single tab.

```
stores:
    json:
        indent: 2
    json_binary:
        indent: 2
```

YAML indentation

SOPS indents `YAML` files by default using 4 spaces. However, you can change this default behaviour by either using the additional `--indent=2` CLI option or by configuring `.sops.yaml` with:

```
stores:
    yaml:
        indent: 2
```

:::: note ::: title Note :::

The YAML emitter used by sops only supports values between 2 and 9. If you specify 1, or 10 and larger, the indent will be 2. ::::

YAML anchors

SOPS only supports a subset of `YAML`'s many types. Encrypting YAML files that contain strings, numbers and booleans will work fine, but files that contain anchors will not work, because the anchors redefine the structure of the file at load time.

This file will not work in SOPS:

```
bill-to:  &id001
    street: |
        123 Tornado Alley
        Suite 16
    city:   East Centerville
    state:  KS

ship-to:  *id001
```

SOPS uses the path to a value as additional data in the AEAD encryption, and thus dynamic paths generated by anchors break the authentication step.

JSON and TEXT file types do not support anchors and thus have no such limitation.

YAML Streams

`YAML` supports having more than one "document" in a single file, while formats like `JSON` do not. SOPS is able to handle both. This means the following multi-document will be encrypted as expected:

```
---
data: foo
---
data: bar
```

Note that the `sops` metadata, i.e. the hash, etc, is computed for the physical file rather than each internal "document".

Top-level arrays

`YAML` and `JSON` top-level arrays are not supported, because SOPS needs a top-level `sops` key to store its metadata.

This file will not work in SOPS:

```
---
  - some
  - array
  - elements
```

But this one will work because the `sops` key can be added at the same level as the `data` key.

```
data:
    - some
    - array
    - elements
```

Similarly, with `JSON` arrays, this document will not work:

```
[
  "some",
  "array",
  "elements"
]
```

But this one will work just fine:

```
{
  "data": [
    "some",
    "array",
    "elements"
  ]
```

```
}
```

Examples

Take a look into the examples folder for detailed use cases of SOPS in a CI environment. The section below describes specific tips for common use cases.

Creating a new file

The command below creates a new file with a data key encrypted by KMS and PGP.

```
$ sops edit --kms "arn:aws:kms:us-west-2:6D8FCB797D8C:key/01792269-4132-404c-ab86-BD966A94BAFF" --p
```

Encrypting an existing file

Similar to the previous command, we tell SOPS to use one KMS and one PGP key. The path points to an existing cleartext file, so we give `sops` the flag `-e` to encrypt the file, and redirect the output to a destination file.

```
$ export SOPS_KMS_ARN="arn:aws:kms:us-west-2:6D8FCB797D8C:key/01792269-4132-404c-ab86-BD966A94BAFF"
$ export SOPS_PGP_FP="36354F50EE9AF9F24A729294D9AC49DB29DF786D"
$ sops encrypt /path/to/existing/file.yaml > /path/to/new/encrypted/file.yaml
```

Decrypt the file with `-d`.

```
$ sops decrypt /path/to/new/encrypted/file.yaml
```

Encrypt or decrypt a file in place

Rather than redirecting the output of `-e` or `-d`, `sops` can replace the original file after encrypting or decrypting it.

```
# file.yaml is in cleartext
$ sops encrypt -i /path/to/existing/file.yaml
# file.yaml is now encrypted
$ sops decrypt -i /path/to/existing/file.yaml
# file.yaml is back in cleartext
```

Encrypting binary files

SOPS primary use case is encrypting YAML and JSON configuration files, but it also has the ability to manage binary files. When encrypting a binary, SOPS will read the data as bytes, encrypt it, store the encrypted base64 under `tree['data']` and write the result as JSON.

Note that the base64 encoding of encrypted data can actually make the encrypted file larger than the cleartext one.

In-place encryption/decryption also works on binary files.

```
$ dd if=/dev/urandom of=/tmp/somerandom bs=1024
count=512
512+0 records in
512+0 records out
ADBD77 bytes (524 kB) copied, 0.FB99EA8 s, 11.2 MB/s

$ sha512sum /tmp/somerandom
6A7644DFD7F162C7E085E6DFFFB67366B16DE4C324EE2D10A5679A8723DDC65CBF4DA10CF96E4538D424EBFD7D8F7D72528

$ sops encrypt -i /tmp/somerandom
please wait while a data encryption key is being generated and stored securely

$ sops decrypt -i /tmp/somerandom

$ sha512sum /tmp/somerandom
6A7644DFD7F162C7E085E6DFFFB67366B16DE4C324EE2D10A5679A8723DDC65CBF4DA10CF96E4538D424EBFD7D8F7D72528
```

Extract a sub-part of a document tree

SOPS can extract a specific part of a YAML or JSON document, by provided the path in the `--extract` command line flag. This is useful to extract specific values, like keys, without needing an extra parser.

```
$ sops decrypt --extract '["app2"]["key"]' ~/git/svc/sops/example.yaml
-----BEGIN RSA PRIVATE KEY-----
MIIBPAIBAAJBAPTMNIyHuZtpLYc7VsHQtwOkWYobkUblmHWRmbXzlAX6K8tMf3Wf
ImcbNkqAKnELzFAPSBeEMhrBN0PyOC9lYlMCAwEAAQJBALXD4sjuBn1E7Y9aGiMz
bJEBuZJ4wbhYxomVoQKfaCu+kH80uLFZKoSz85/ySauWE8LgZcMLIBoiXNhDKfQL
vHECIQD6tCG9NMFWor69kgbX8vK5Y+QL+kRq+9HK6yZ9a+hsLQIhAPn4Ie6HGTjw
fHSTXWZpGSan7NwTkIu4U5q2SlLjcZh/AiEA78NYRRBwGwAYNUqzutGBqyXKUl4u
Erb0xAEyVV7e8J0CIQC8VBY8f8yg+Y7Kxbw4zDYGyb3KkXL10YorpeuZR4LuQQIg
bKGPkMM4w5blyE1tqGN0T7sJwEx+EUOgacRNqM2ljVA=
-----END RSA PRIVATE KEY-----
```

The tree path syntax uses regular python dictionary syntax, without the variable name. Extract keys by naming them, and array elements by numbering them.

```
$ sops decrypt --extract '["an_array"][1]' ~/git/svc/sops/example.yaml
secretuser2
```

Set a sub-part in a document tree

SOPS can set a specific part of a YAML or JSON document, by providing the path and value in the `set` command. This is useful to set specific values, like keys, without needing an editor.

```
$ sops set ~/git/svc/sops/example.yaml '["app2"]["key"]' '"app2keystringvalue"'
```

The tree path syntax uses regular python dictionary syntax, without the variable name. Set to keys by naming them, and array elements by numbering them.

```
$ sops set ~/git/svc/sops/example.yaml '["an_array"][1]' '"secretuser2"'
```

The value must be formatted as json.

```
$ sops set ~/git/svc/sops/example.yaml '["an_array"][1]' '{"uid1":null,"uid2":1000,"uid3":["bob"]}'
```

Unset a sub-part in a document tree

Symmetrically, SOPS can unset a specific part of a YAML or JSON document, by providing the path in the `unset` command. This is useful to unset specific values, like keys, without needing an editor.

```
$ sops unset ~/git/svc/sops/example.yaml '["app2"]["key"]'
```

The tree path syntax uses regular python dictionary syntax, without the variable name. Set to keys by naming them, and array elements by numbering them.

```
$ sops unset ~/git/svc/sops/example.yaml '["an_array"][1]'
```

Showing diffs in cleartext in git

You most likely want to store encrypted files in a version controlled repository. SOPS can be used with git to decrypt files when showing diffs between versions. This is very handy for reviewing changes or visualizing history.

To configure SOPS to decrypt files during diff, create a `.gitattributes` file at the root of your repository that contains a filter and a command.

```
*.yaml diff=sopsdiffer
```

Here we only care about YAML files. `sopsdiffer` is an arbitrary name that we map to a SOPS command in the git configuration file of the repository.

```
$ git config diff.sopsdiffer.textconv "sops decrypt"

$ grep -A 1 sopsdiffer .git/config
[diff "sopsdiffer"]
    textconv = "sops decrypt"
```

With this in place, calls to `git diff` will decrypt both previous and current versions of the target file prior to displaying the diff. And it even works with git client interfaces, because they call git diff under the hood!

Encrypting only parts of a file

Note: this only works on YAML and JSON files, not on BINARY files.

By default, SOPS encrypts all the values of a YAML or JSON file and leaves the keys in cleartext. In some instances, you may want to exclude some values from being encrypted. This can be accomplished by adding the suffix **_unencrypted** to any key of a file. When set, all values underneath the key that set the **_unencrypted** suffix will be left in cleartext.

Note that, while in cleartext, unencrypted content is still added to the checksum of the file, and thus cannot be modified outside of SOPS without breaking the file integrity check. This behavior can be modified using `--mac-only-encrypted` flag or `.sops.yaml` config file which makes SOPS compute a MAC only over values it encrypted and not all values.

The unencrypted suffix can be set to a different value using the `--unencrypted-suffix` option.

Conversely, you can opt in to only encrypt some values in a YAML or JSON file, by adding a chosen suffix to those keys and passing it to the `--encrypted-suffix` option.

A third method is to use the `--encrypted-regex` which will only encrypt values under keys that match the supplied regular expression. For example, this command:

```
$ sops encrypt --encrypted-regex '^(data|stringData)$' k8s-secrets.yaml
```

will encrypt the values under the `data` and `stringData` keys in a YAML file containing kubernetes secrets. It will not encrypt other values that help you to navigate the file, like `metadata` which contains the secrets' names.

Conversely, you can opt in to only leave certain keys without encrypting by using the `--unencrypted-regex` option, which will leave the values unencrypted of those keys that match the supplied regular expression. For example, this command:

```
$ sops encrypt --unencrypted-regex '^(description|metadata)$' k8s-secrets.yaml
```

will not encrypt the values under the `description` and `metadata` keys in a YAML file containing kubernetes secrets, while encrypting everything else.

For YAML files, another method is to use `--encrypted-comment-regex` which will only encrypt comments and values which have a preceding comment matching the supplied regular expression.

Conversely, you can opt in to only left certain keys without encrypting by using the `--unencrypted-comment-regex` option, which will leave the values and comments unencrypted when they have a preeceding comment, or a trailing comment on the same line, that matches the supplied regular expression.

You can also specify these options in the `.sops.yaml` config file.

Note: these six options `--unencrypted-suffix`, `--encrypted-suffix`, `--encrypted-regex`, `--unencrypted-regex`, `--encrypted-comment-regex`, and `--unencrypted-comment-regex` are mutually exclusive and cannot all be used in the same file.

Encryption Protocol

When SOPS creates a file, it generates a random 256 bit data key and asks each KMS and PGP master key to encrypt the data key. The encrypted version of the data key is stored in the `sops` metadata under `sops.kms` and `sops.pgp`.

For KMS:

```
sops:
    kms:
        - enc: CiC6yCOtzsnFhkfdIslYZ0bAf//gYLYCmIu87B3sy/5yYxKnAQEBAQB4usgjrc7JxYZH3SLJWGdGwH//4GC2
          enc_ts: EBC6A98549.DBA66A
          arn: arn:aws:kms:us-east-1:9A9ACD6D8CAF:key/6DF5002e-c5f1-4040-943a-FB805C784D81
```

For PGP:

```
sops:
    pgp:
        - fp: 7A288ABC4C29DB49C315619243E8CFE4B6E43F21
          created_at: EBBEA80391.6CFFBD
          enc: |
              -----BEGIN PGP MESSAGE-----
              Version: GnuPG v1

              hQIMA0t4uZHfl9qgAQ//UvGAwGePyHuf2/zayWcloGaDs0MzI+zw6CmXvMRNPUsA
```

```
pAgRKczJmDu4+XzN+cxX5Iq9xEWIbny9B5rOjwTXT3qcUYZ4Gkzbq4MWkjuPp/Iv
qO4MJaYzoH5YxC4YORQ2LvzhA2YGsCzYnljmatGEUNg01yJ6r5mwFwDxl4Nc80Cn
RwnHuGExK8j1jYJZu/juK1qRbuBOAuruIPPWV2047BAPA7waacG1IdUW3ZtBkOy3
O0BIfG2ekRg0Nik6sTOhDUA+l2bewCcECI8FYCEjwHm9Sg5cxmP2V5m1mby+uKAm
kewaoOyjbmV1Mh3iI1b/AQMr+/6ZE9MT2KnsoWosYamFyjxV5r1ZZM7cWKnOT+tu
KOvGhTV1TeOfVpajNTNwtV/Oyh3mMLQ0F0HgCTqomQVqw5+sj7OWAASuD3CU/dyo
pcmY5Qe0TNL1JsMNEH8LJDqSh+E0hsUxdY1ouVsg3ysf6mdM8ciWb3WRGxih1Vmf
unfLy8Ly3V7ZIC8EHV8aLJqh32jIZV4i2zXIoO4ZBKrudKcECY1C2+zb/TziVAL8
qyPe47q8gi1rIyEv5uirLZjgpP+JkDUgoMnzlX334FZ9pWtQMYW4Y67urAI4xUq6
/q1zBAeHoeeeQK+YKDB7Ak/Y22YsiqQbNp2n4CKSKAE4erZLWVtDvSp+49SWmS/S
XgGi+13MaXIp0ecPKyNTBjF+NOw/I3muyKr8EbDHrd2XgIT06QXqjYLsCb1TZ0zm
xgXsOTY3b+ONQ2zjhcovanDp7/k77B+gFitLYKg4BLZsl7gJB12T8MQnpfSmRT4=
=oJgS
-----END PGP MESSAGE-----
```

SOPS then opens a text editor on the newly created file. The user adds data to the file and saves it when done.

Upon save, SOPS browses the entire file as a key/value tree. Every time SOPS encounters a leaf value (a value that does not have children), it encrypts the value with AES256_GCM using the data key and a 256 bit random initialization vector.

Each file uses a single data key to encrypt all values of a document, but each value receives a unique initialization vector and has unique authentication data.

Additional data is used to guarantee the integrity of the encrypted data and of the tree structure: when encrypting the tree, key names are concatenated into a byte string that is used as AEAD additional data (aad) when encrypting values. We expect that keys do not carry sensitive information, and keeping them in cleartext allows for better diff and overall readability.

Any valid KMS or PGP master key can later decrypt the data key and access the data.

Multiple master keys allow for sharing encrypted files without sharing master keys, and provide a disaster recovery solution. The recommended way to use SOPS is to have two KMS master keys in different regions and one PGP public key with the private key stored offline. If, by any chance, both KMS master keys are lost, you can always recover the encrypted data using the PGP private key.

Message Authentication Code

In addition to authenticating branches of the tree using keys as additional data, SOPS computes a MAC on all the values to ensure that no value has been added or removed fraudulently. The MAC is stored encrypted with AES_GCM and the data key under tree -> `sops` -> `mac`. This behavior can be modified using `--mac-only-encrypted` flag or `.sops.yaml` config file which makes SOPS compute a MAC only over values it encrypted and not all values.

Motivation

Automating the distribution of secrets and credentials to components of an infrastructure is a hard problem. We know how to encrypt secrets and share them between humans, but extending that trust to systems is difficult. Particularly when these systems follow devops principles and are created and destroyed without human intervention. The issue boils down to establishing the initial trust of a system that just joined the infrastructure, and providing it access to the secrets it needs to configure itself.

The initial trust

In many infrastructures, even highly dynamic ones, the initial trust is established by a human. An example is seen in Puppet by the way certificates are issued: when a new system attempts to join a Puppetmaster, an administrator must, by default, manually approve the issuance of the certificate the system needs. This is cumbersome, and many puppetmasters are configured to auto-sign new certificates to work around that issue. This is obviously not recommended and far from ideal.

AWS provides a more flexible approach to trusting new systems. It uses a powerful mechanism of roles and identities. In AWS, it is possible to verify that a new system has been granted a specific role at creation, and it is possible to map that role to specific resources. Instead of trusting new systems directly, the administrator trusts the AWS permission model and its automation infrastructure. As long as AWS keys are safe, and the AWS API is secure, we can assume that trust is maintained and systems are who they say they are.

KMS, Trust and secrets distribution

Using the AWS trust model, we can create fine grained access controls to Amazon's Key Management Service (KMS). KMS is a service that encrypts and decrypts data with AES_GCM, using keys that are never visible to users of the service. Each KMS master key has a set of role-based access controls, and individual roles are permitted to encrypt or decrypt using the master key. KMS helps solve the problem of distributing keys, by shifting it into an access control problem that can be solved using AWS's trust model.

Operational requirements

When Mozilla's Services Operations team started revisiting the issue of distributing secrets to EC2 instances, we set a goal to store these secrets encrypted until the very last moment, when they need to be decrypted on target systems. Not unlike many other organizations that operate sufficiently complex automation, we found this to be a hard problem with a number of prerequisites:

1. Secrets must be stored in YAML files for easy integration into hiera
2. Secrets must be stored in GIT, and when a new CloudFormation stack is built, the current HEAD is pinned to the stack. (This allows secrets to be changed in GIT without impacting the current stack that may autoscale).
3. Entries must be encrypted separately. Encrypting entire files as blobs makes git conflict resolution almost impossible. Encrypting each entry separately is much easier to manage.
4. Secrets must always be encrypted on disk (admin laptop, upstream git repo, jenkins and S3) and only be decrypted on the target systems

SOPS can be used to encrypt YAML, JSON and BINARY files. In BINARY mode, the

content of the file is treated as a blob, the same way PGP would encrypt an entire file. In YAML and JSON modes, however, the content of the file is manipulated as a tree where keys are stored in cleartext, and values are encrypted. hiera-eyaml does something similar, and over the years we learned to appreciate its benefits, namely:

- diffs are meaningful. If a single value of a file is modified, only that value will show up in the diff. The diff is still limited to only showing encrypted data, but that information is already more granular that indicating that an entire file has changed.
- conflicts are easier to resolve. If multiple users are working on the same encrypted files, as long as they don't modify the same values, changes are easy to merge. This is an improvement over the PGP encryption approach where unsolvable conflicts often happen when multiple users work on the same file.

OpenPGP integration

OpenPGP gets a lot of bad press for being an outdated crypto protocol, and while true, what really made us look for alternatives is the difficulty of managing and distributing keys to systems. With KMS, we manage permissions to an API, not keys, and that's a lot easier to do.

But PGP is not dead yet, and we still rely on it heavily as a backup solution: all our files are encrypted with KMS and with one PGP public key, with its private key stored securely for emergency decryption in the event that we lose all our KMS master keys.

SOPS can be used without KMS entirely, the same way you would use an encrypted PGP file: by referencing the pubkeys of each individual who has access to the file. It can easily be done by providing SOPS with a comma-separated list of public keys when creating a new file:

```
$ sops edit --pgp "19F76D446427659608A65E5F5C29ADE8C4893E8F,7BFAF0E29E5083DCF5EDDE8978209AFA610F93D
```

Threat Model

The security of the data stored using SOPS is as strong as the weakest cryptographic mechanism. Values are encrypted using AES256_GCM which is the strongest symmetric encryption algorithm known today. Data keys are encrypted in either KMS, which also uses AES256_GCM, or PGP which uses either RSA or ECDSA keys.

Going from the most likely to the least likely, the threats are as follows:

Compromised AWS credentials grant access to KMS master key

An attacker with access to an AWS console can grant itself access to one of the KMS master keys used to encrypt a `sops` data key. This threat should be mitigated by protecting AWS accesses with strong controls, such as multi-factor authentication, and also by performing regular audits of permissions granted to AWS users.

Compromised PGP key

PGP keys are routinely mishandled, either because owners copy them from machine to machine, or because the key is left forgotten on an unused machine an attacker gains access to. When using PGP encryption, SOPS users should take special care of PGP private keys, and store them on smart cards or offline as often as possible.

Factorized RSA key

SOPS doesn't apply any restriction on the size or type of PGP keys. A weak PGP keys, for example 512 bits RSA, could be factorized by an attacker to gain access to the private key and decrypt the data key. Users of SOPS should rely on strong keys, such as 2048+ bits RSA keys, or 256+ bits ECDSA keys.

Weak AES cryptography

A vulnerability in AES256_GCM could potentially leak the data key or the KMS master key used by a SOPS encrypted file. While no such vulnerability exists today, we recommend that users keep their encrypted files reasonably private.