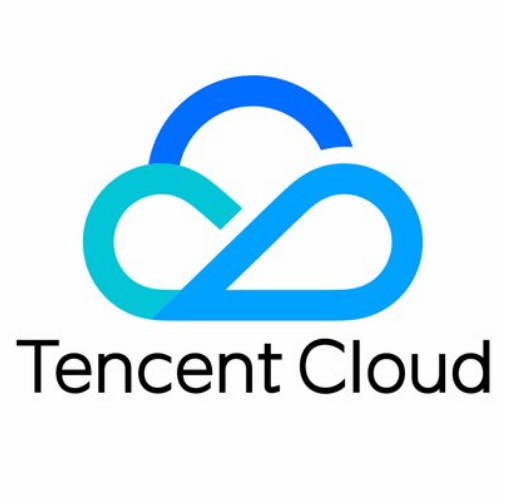
Secrets Manager

Best Practices

Product Documentation



Contents

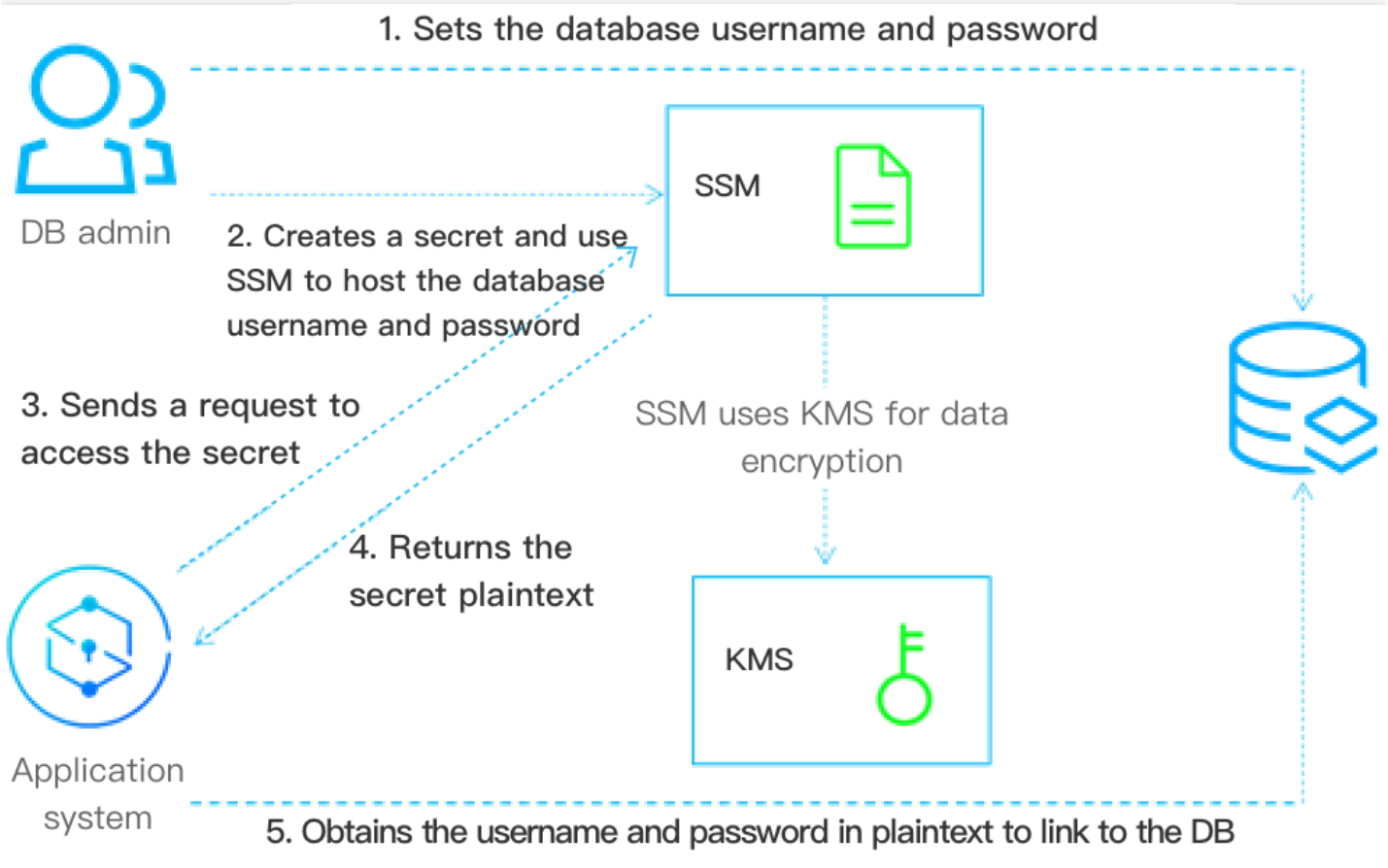
Best Practices
Hosting and Using Secrets

Last updated : 2020-11-16 14:10:11

Normally, various types of verification information (i.e., passwords, tokens, SSH keys, and API keys) for identity verification are embedded in the configuration file of the application as plaintext, which offers lower security. You can use SSM to encrypt and store sensitive information to avoid risks caused by the plaintext coding of sensitive secrets.

Directions

The following uses the hosted username and password of a database as an example to introduce the basic use cases of secret hosting.



1. DB admin sets the database username and password.

2. DB admin creates a secret object in SSM. The secret object is used to store the encrypted username and password obtained in Step 1.
3. When the application needs to access the database, it needs to send a request to SSM to access the secret.
4. After SSM receives the secret plaintext, it decrypts the secret and sends the secret plaintext to the application over HTTPS.
5. The application reads and parses the secret plaintext returned by SSM to obtain the username and password for accessing the target database.
6. DB admin can create multiple versions for a secret. Also, it can update the version content to implement configuration synchronization, version management, and secret rotation.

Application Effects

The application system can call the SSM APIs or SDK to obtain the sensitive secret plaintext, avoiding leakage risks caused by coding secret as plaintext in the application or configuration file. The calling comparison is as follows:

- The following are examples of storing the username and password of a database as plaintext in the local configuration or the code file, which brings a higher risk of sensitive secret leakage.
  - Sample code of obtaining the secret plaintext:

```
func GetDBConfig() string {
dbConnStr := "user:password@tcp(127.0.0.1:3306)/test"
return dbConnStr
}
```

  - Sample code of using the secret plaintext:

```
conn, err := sql.Open("mysql", GetDBConfig())
if err != nil {
// error handler
}
```

- The following are examples of using SSM to store the username and password for connecting to the database. It avoids storing the username and password as plaintext in the code or local configuration file.

- Sample code of obtaining the secret plaintext:

```
func GetDBConfig(secretName, version *string) string {
credential := common.NewCredential(
secretId,
secretKey,
)
cpf := profile.NewClientProfile()
cpf.HttpProfile.Endpoint = endpoint
client, _ := ssm.NewClient(credential, region, cpf)

request := ssm.NewGetSecretValueRequest()
request.SecretName = secretName
request.VersionId = version

resp, err := client.GetSecretValue(request)
if err != nil {
// error handler
}
return *resp.Response.SecretString
}
```

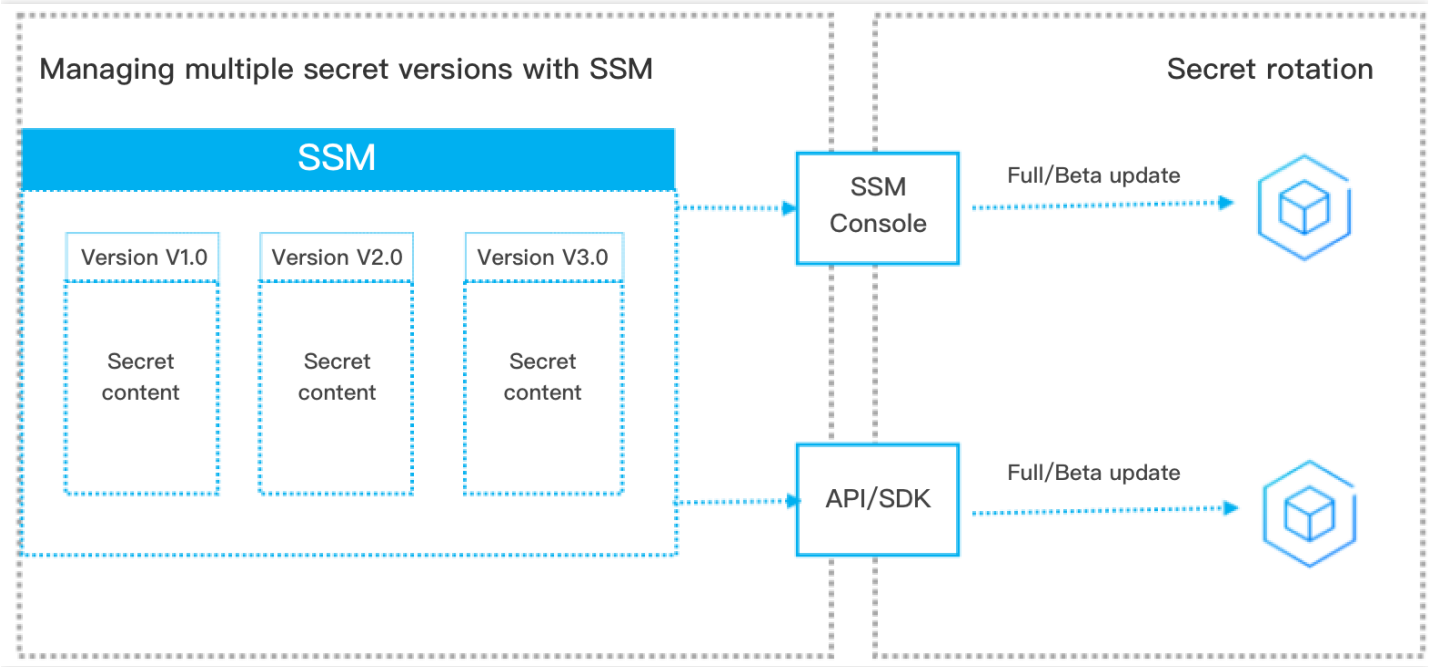
  - Sample code of using the secret plaintext:

```
secretName := "MySecret1"
version := "MyVersion1"
conn, err := sql.Open("mysql", GetDBConfig(&secretName, &version))
if err != nil {
// error handler
}
```

Rotating Hosted Secrets

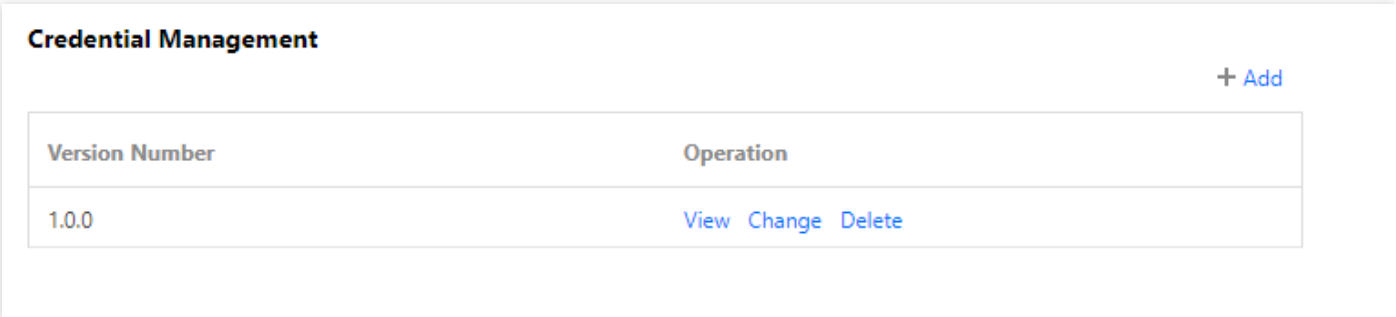
Last updated : 2021-08-18 14:21:42

To improve system security, the update of a secret needs to be synced across multiple applications and configurations. For a multi-application scenario, if secrets are stored in local files, an application might be missed, running a risk of application interruptions. SSM enables you to apply a secret update to all dependent applications. You can also create multiple versions for a secret to implement beta updates and rotation.



You can rotate secrets using either of the following ways:

- **Method 1**: add a secret version. The server can implement beta rotation by updating the secret version.



- **Method 2**: update the content of the current secret. When the server calls an API to obtain the secret, the secret content is updated automatically. For more information, please see Examples of Secret API Calls.

Secret API Calling Example

Last updated : 2020-11-16 14:11:25

Hosting and Protecting Secrets

Example: DB admin can create a secret (MySecret) and specify the version (MyVersion1). The database username and password are encrypted and stored using SSM. If no KMS key is specified, SSM will automatically create a default key.

```
var (
secretName = "MySecret1"
version = "MyVersion1"
plainText = "user:password@tcp(127.0.0.1:3306)/test"
)

func ExampleCreateSecret() {
credential := common.NewCredential(
secretId,
secretKey,
)
cpf := profile.NewClientProfile()
cpf.HttpProfile.Endpoint = endpoint
client, _ := ssm.NewClient(credential, region, cpf)

request := ssm.NewCreateSecretRequest()
request.SecretName = &secretName
request.VersionId = &version
request.SecretString = &plainText

resp, err := client.CreateSecret(request)
if err != nil {
// error handler
}
fmt.Println(*resp.Response.SecretName)

// create ok
}
```

Viewing the Metadata of Secrets

- **Example 1**: obtaining the secret list and metadata of secrets.

```
func ExampleListSecrets() {
credential := common.NewCredential(
secretId,
secretKey,
)
cpf := profile.NewClientProfile()
cpf.HttpProfile.Endpoint = endpoint
client, _ := ssm.NewClient(credential, region, cpf)

request := ssm.NewListSecretsRequest()

resp, err := client.ListSecrets(request)
if err != nil {
// error handler
}
fmt.Println(resp.Response.SecretMetadatas)
// get secrets metadata
// ...
}
```

- **Example 2**: obtaining the version information using the secret name (MySecret1).

```
var (
secretName = "MySecret1"
)
func ExampleListSecretVersionIds() {
credential := common.NewCredential(
secretId,
secretKey,
)
cpf := profile.NewClientProfile()
cpf.HttpProfile.Endpoint = endpoint
client, _ := ssm.NewClient(credential, region, cpf)

request := ssm.NewListSecretVersionIdsRequest()
request.SecretName = &secretName

resp, err := client.ListSecretVersionIds(request)
if err != nil {
// error handler
}
fmt.Println(resp.Response.Versions)

// get version list
```

```
// ...
}
```

Obtaining the Sensitive Data Plaintext Stored in SSM

Example: the caller obtains the database username and password plaintext using the secret name (MySecret1) and secret version (MyVersion1).

```
var (
secretName = "MySecret1"
version = "MyVersion1"
)

func ExampleGetSecretValue() {
credential := common.NewCredential(
secretId,
secretKey,
)
cpf := profile.NewClientProfile()
cpf.HttpProfile.Endpoint = endpoint
client, _ := ssm.NewClient(credential, region, cpf)

request := ssm.NewGetSecretValueRequest()
request.VersionId = &version
request.SecretName = &secretName

resp, err := client.GetSecretValue(request)
if err != nil {
// error handler
}
fmt.Println(*resp.Response.SecretString)

// get plain text, connect db
// ...
}
```

Updating the Content of a Secret

Example: updating the database username and password plaintext using the secret name (MySecret1) and version (MyVersion1).

```
var (
secretName = "MySecret1"
version = "MyVersion1"
newSecretValue = "user2:password2@tcp(127.0.0.1:3306)/test"
)

func ExamplePutSecretValue() {
credential := common.NewCredential(
secretId,
secretKey,
)
cpf := profile.NewClientProfile()
cpf.HttpProfile.Endpoint = endpoint
client, _ := ssm.NewClient(credential, region, cpf)

request := ssm.NewPutSecretValueRequest()
request.SecretName = &secretName
request.VersionId = &version
request.SecretString = &newSecretValue

resp, err := client.PutSecretValue(request)
if err != nil {
// error handler
}
fmt.Println(*resp.Response.SecretName)
// secret updated
// ...
}
```

Disabling, Deleting, and Restoring a Secret

- **Example 1**: disabling a secret. After a secret is disabled, the server can no longer obtain any content of the secret.

```
var (
secretName = "MySecret1"
)
func ExampleDisableSecret() {
credential := common.NewCredential(
secretId,
secretKey,
)
cpf := profile.NewClientProfile()
```

```
cpf.HttpProfile.Endpoint = endpoint
client, _ := ssm.NewClient(credential, region, cpf)

request := ssm.NewDisableSecretRequest()
request.SecretName = &secretName
resp, err := client.DisableSecret(request)

if err != nil {
// error handler
}
fmt.Println(*resp.Response.SecretName)
// secret disabled
// ...
}
```

- **Example 2**: deleting a secret. You can set the schedule delete time, before which the secret can be restored.

> ⚠ **Note :**
>
> Only disabled secrets can be deleted.

```
func ExampleDeleteSecret() {
credential := common.NewCredential(
secretId,
secretKey,
)
cpf := profile.NewClientProfile()
cpf.HttpProfile.Endpoint = endpoint
client, _ := ssm.NewClient(credential, region, cpf)

request := ssm.NewDeleteSecretRequest()
request.SecretName = &secretName
request.RecoveryWindowInDays = &recoverWindowsInDays

resp, err := client.DeleteSecret(request)
if err != nil {
// error handler
}
fmt.Println(*resp.Response.DeleteTime)
// secret deleted
// ...
}
```

- **Example 3**: restoring a secret. You can restore and enable the `PendingDelete` secret.

```
func ExampleRestoreSecret() {
credential := common.NewCredential(
secretId,
secretKey,
)
cpf := profile.NewClientProfile()
cpf.HttpProfile.Endpoint = endpoint
client, _ := ssm.NewClient(credential, region, cpf)

request := ssm.NewRestoreSecretRequest()
request.SecretName = &secretName

resp, err := client.RestoreSecret(request)
if err != nil {
// error handler
}
fmt.Println(*resp.Response.SecretName)
// secret restored
// ...
}
```