Using Protect/Reveal with CRDP (CM Setup Tutorial)

Overview

This document is a tutorial on CipherTrust Manager (CM) Protect/Reveal using CipherTrust RESTful Data Protection (CRDP). This capability was first introduced with the Data Protection Gateway product and now available with CipherTrust Rest Data Protection (CRDP).

CipherTrust RESTful Data Protection (CRDP) solution provides RESTful webservices to protect sensitive data via wide range of data protection methods. CRDP is designed from the ground up to seamlessly fit with existing cloud-ready applications. It is deployed as a docker container and performs wide range of cryptographic operations. It is easy to deploy, configure, manage, and migrate. CRDP allows enterprises to centrally configure their data-centric cryptographic policies in a reusable, human-readable way and to deploy data protection that fits within their native cloud deployment.

Protect/Reveal is a new theme that will continue to be built on that helps organizations deploy applications in a secure, consistent, compliant manner without being a burden to the developers or DevOps teams.

Assumptions

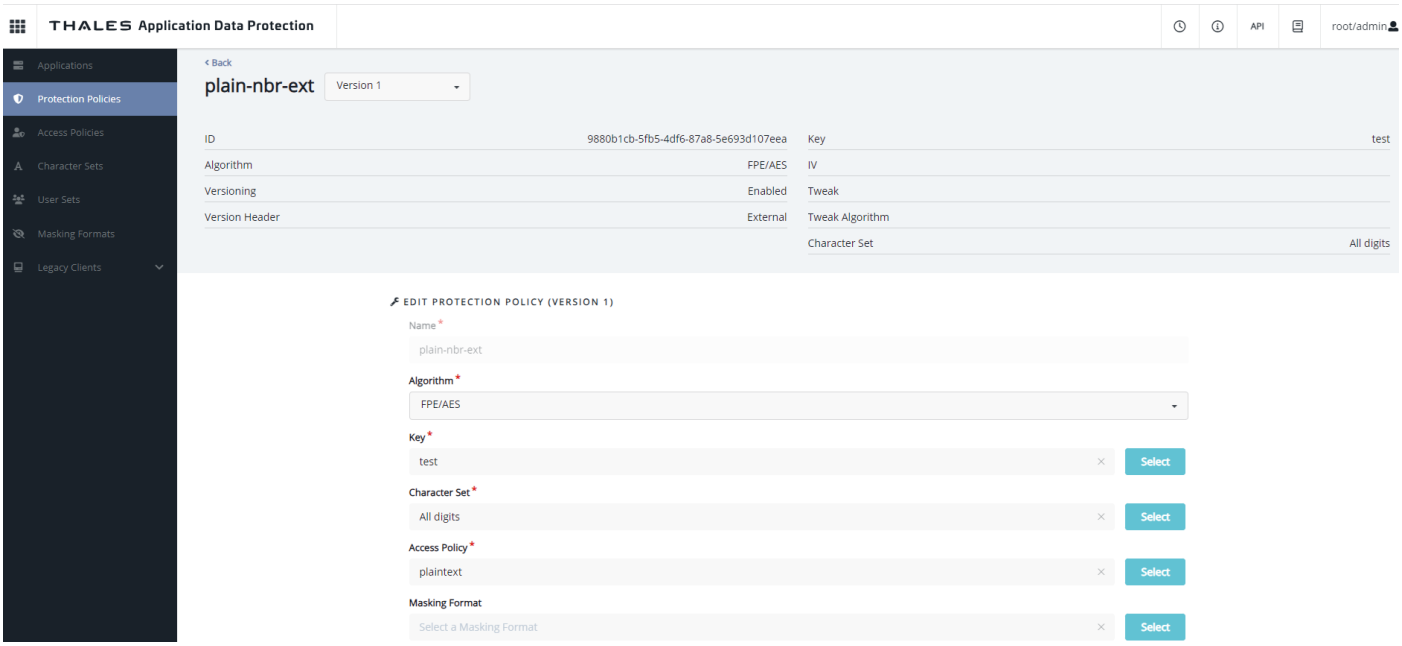
- CipherTrust Manager has already been installed and setup.
- CRDP container is already setup and configured.
- The focus on this document will be the setup required in CM to enable the protect/reveal API's to make calls to the CRDP container.

This tutorial used the standalone implementation of CRDP container which is documented in the Appendix. For more information about the methods of CRDP container deployment see this link: https://thalesdocs.com/ctp/con/crdp/latest/admin/crdp-deploy_alternative/index.html

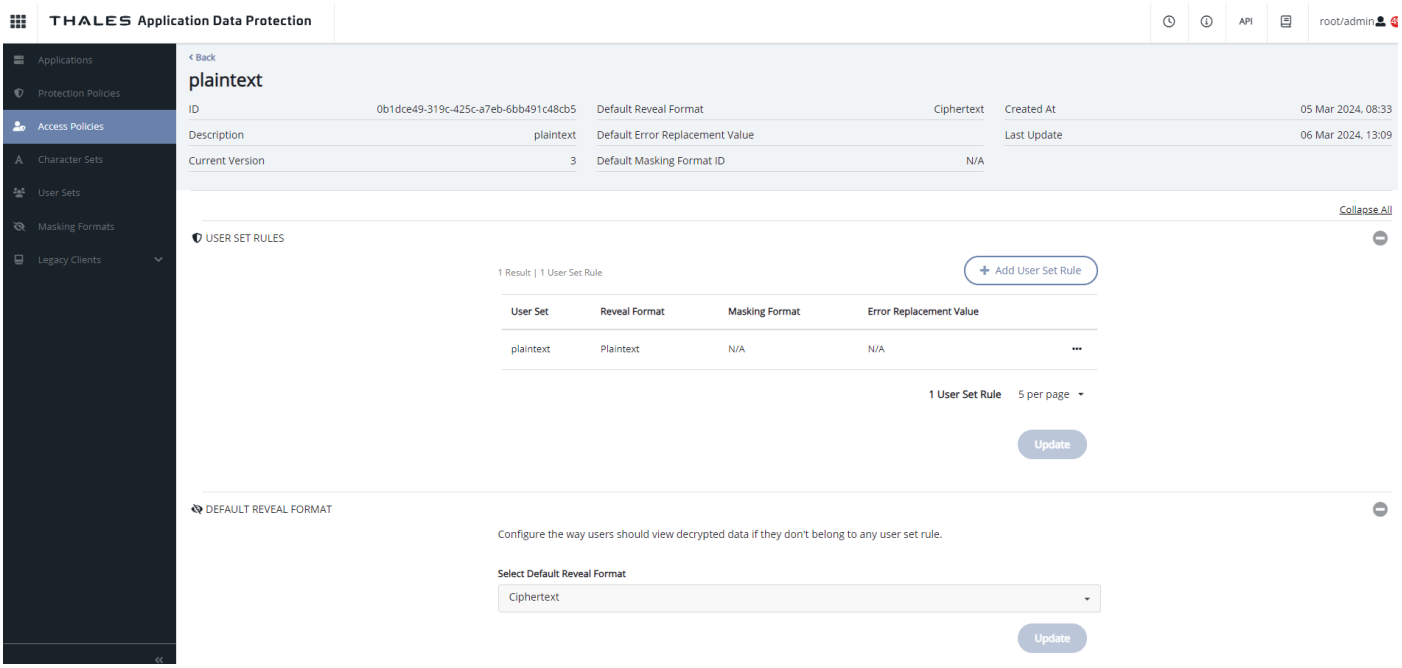
Here is a link the Thales documentation for CRDP.
https://thalesdocs.com/ctp/con/crdp/latest/index.html
There is also an online demo at: thales.navattic.com/thalesprotectreveal

Protect/Reveal CM Setup Tutorial

Protect/reveal are the API's that are implemented by CRDP. The API's depend on policies in CM which are the building blocks of what composes an application found in the Application Data Protection tile. The first policy is called a protection policy which defines how to **protect** a specific piece of data. The protection policy includes entities such as algorithm, key, IV, access policy name, and character set.



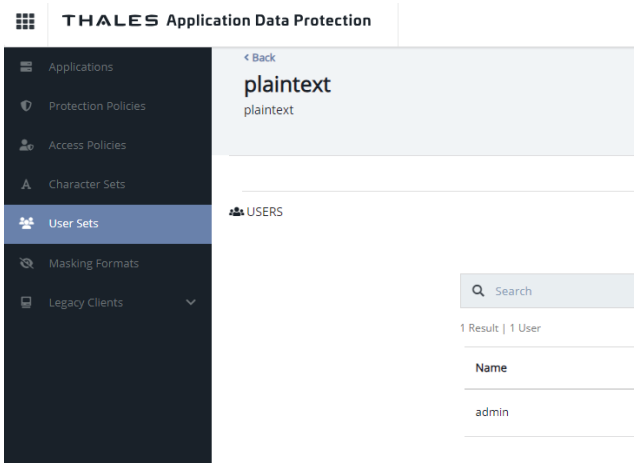
As you can see from above, a protection policy defines a set of rules that govern the cryptographic operations. If you notice above the protection policy, make reference to something called an access policy. Access policies are set of rules that define how the decrypted data will be **revealed** to the application users. Access policies contain set of rules that govern how the decrypted data will be revealed for any specific user. Each access policy has a default reveal format for any "user" that is not part of any user set.



Access policies specify:

- **User Set**: Contains the list of users who needs access to data.
- **Reveal Format**: Determines how the decrypted data will be revealed to specific application users. Following reveal formats are available:
    - **Error Replacement Value**: Client returns the error_replacement value, like a fixed response or null, to the application user.
    - **Ciphertext**: Client returns the ciphertext to the application user.
    - **Masked Value**: Client dynamically masks the data, as an example only revealing the true last four digits and returns the masked value to the application user.
    - **Plaintext**: Client decrypts data and returns the plaintext to the application user.

Different users will reveal data differently. These users are part of user sets. Here is an example of a user set with one user in it.



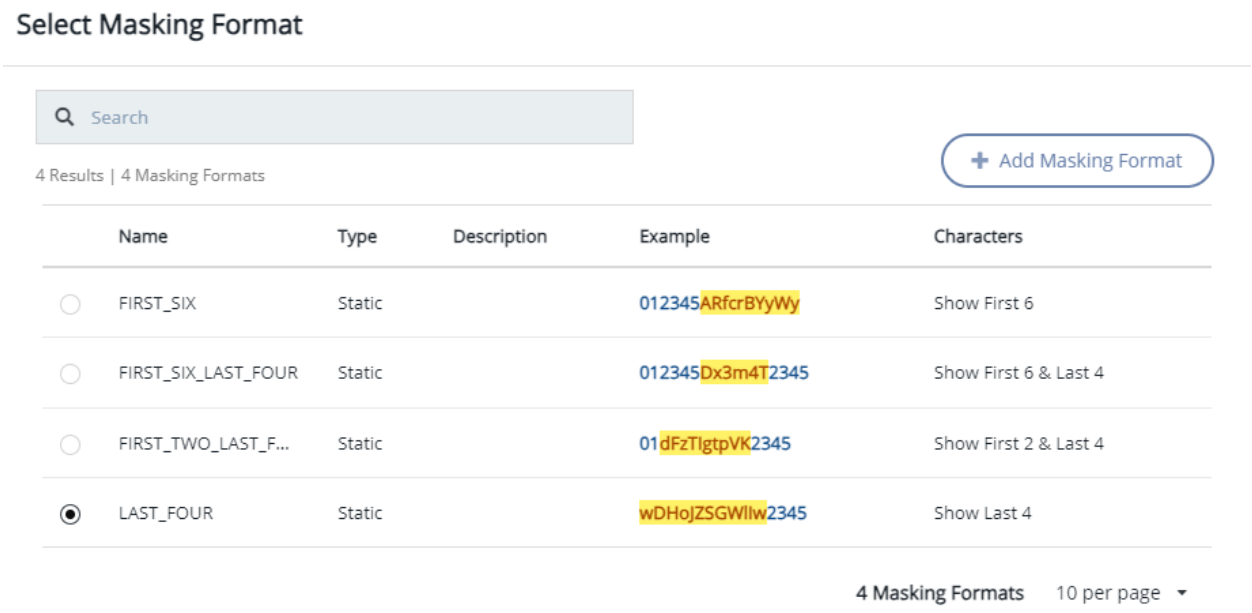
The users in a userset are **NOT** users in CM who require CM credentials. It is assumed that the users in a user set have already been authenticated by an external identity provider by the customers application. The user in a userset will reflect application users that are used in a Reveal request. This is needed so the necessary permissions can be applied and reveal the correct results (clear text, cipher text or masked data).

Masking

There are two types of masks when making a protection profile. One is controlled in the UI for creating of a protection policy and the other is dynamic depending on who the user is so it is based on an access policy.

Option 1. Protection Policy Mask (Static).

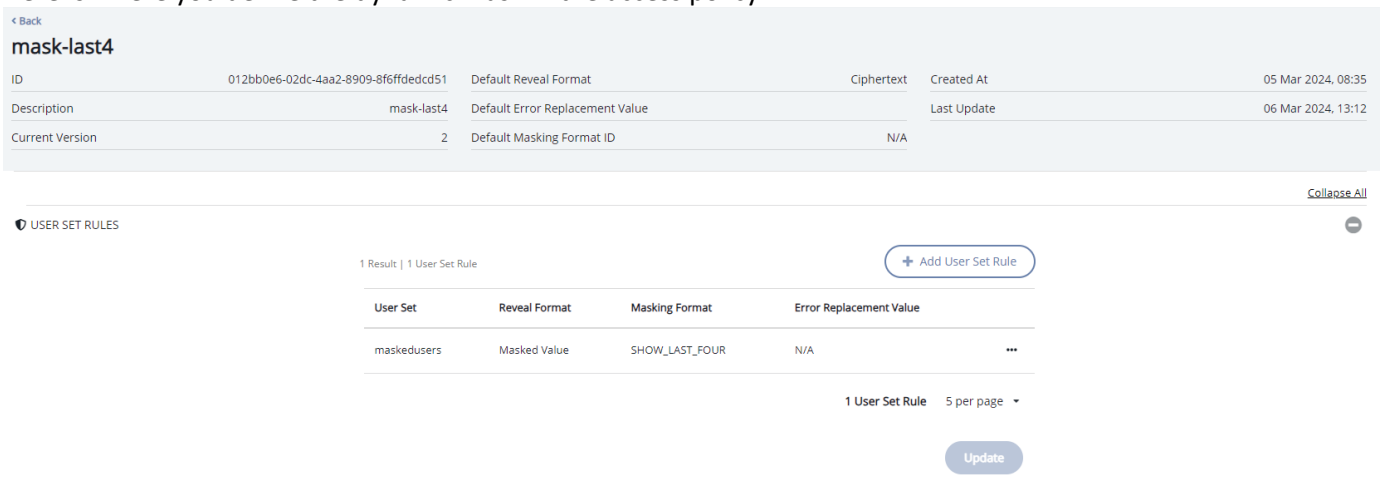
Here are the options for masking when creating a protection policy:



This type of mask determines what parts of the sensitive data will **NOT** be protected. As an example if someone were to run a query against the database and the LAST_FOUR masking policy was set then the query would show the original last 4 digits since it was not protected.

Option 2. Access policy mask (dynamic)

Here is where you define the dynamic mask in the access policy.



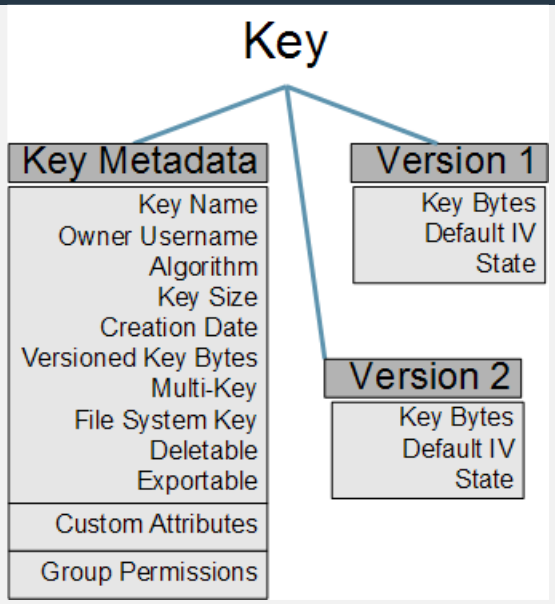
You can see that any user in the userset called maskeduses will only reveal the last 4 digits of the original sensitive data. As an example if someone were to run a query against the database the query would show **ciphertext** for all the characters of the original sensitive data.

Key Versioning.

Although encryption provides a high level of data security, it is possible that given enough time and resources, a skilled attacker could compromise an encryption key. The best way to limit the effect of this attack is to rotate the keys used to encrypt your data. Key rotation should be included as a regular part of your security maintenance plan to help ***reduce the risk*** of a breach.

Key Versioning Overview

CipherTrust Manager supports key versions. A versioned key maintains the same key metadata (key name, owner, algorithm, key size, etc), but has a unique set of bytes for each version. Therefore, each version is different enough for encryption purposes, but similar enough to allow for easy management. Each key version has its own key bytes, default IV, state, and creation date.



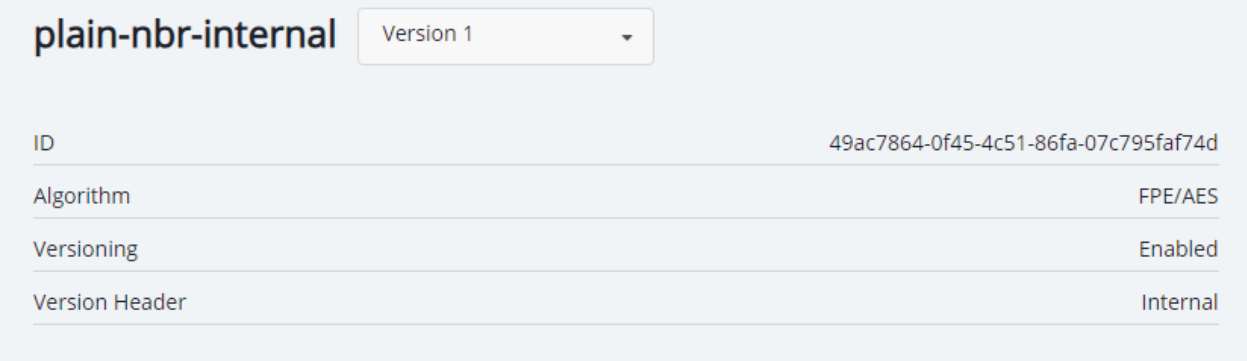
Although most key managers implement a version key capability, which does simplify the management of key rotations, not all systems and vendors support it. Keeping track of non-current keys should not be a maintenance burden to the operations team and procedures should be in place to retire non-current keys at a reasonable timeframe.

Other important things to consider on the topic of key rotation are the different types of key rotation that are available, what the security auditors require, frequency of key rotation and how it might impact current systems when the key rotation occurs as well as maintaining data integrity to existing systems. **For Protect/Reveal the key versioning is 7 bytes. The first byte is reserved for the schema of the protection policy, the next three bytes represents the protection policy number and the last 3 bytes represents the key version.** Here are some samples:

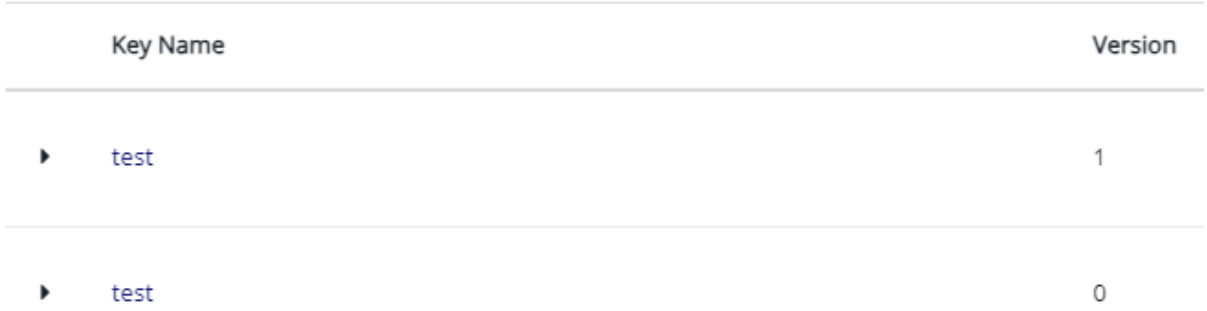
1001001 is Protection Policy ver1 using Key (named within) at version 1
1001002 is Protection Policy ver1 using Key (named the same within but new ver) at version 2
1002002 is Protection Policy ver2 using Key (named the same within but new ver)
1003001 is Protection Policy ver3 using Key (different key name within) at version 1

Protect/Reveal and key versions.

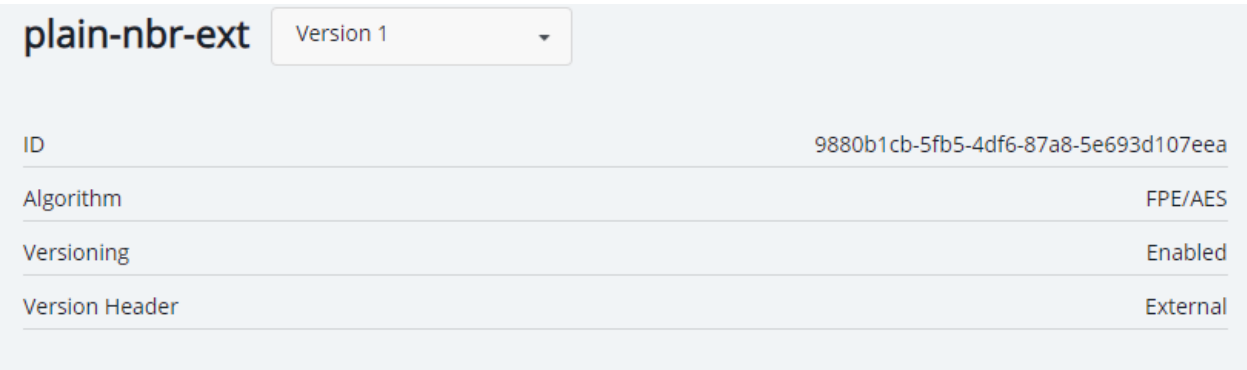
As noted in the link above there are a couple of ways to handle the key version when working with FPE. The first is to store the 3 byte header with the data. This is called internal mode. The other option is to store the 3 byte header in another location. This is called external mode. Protect/reveal make the implementation of this very simple using the UI. The screenshot below shows the version will be internal which means it will be appended to the front of the ciphertext.



For example, if we wanted to encrypt this **data**: 0123456789012345
The ciphertext would be: **Encrypted Data**: 10010014735860402263035
As you can see there a 7 bytes of meta data that are returned with the ciphertext. The fist 4 bytes reflect the policy version used and the next 3 bytes are the key version used. When decrypted the reveal api knows the header location (the 3 right bytes in blue) and will use it to determine what version of the key to use. In this case we have two version of the key and Version 0 will be used to decrypt the data:



When using the internal mode, you can see that it will adjust the size of the original data to accommodate the key version and policy version. From a coding perspective it makes it very easy to implement because there are no additional code changes required to deal with the key version, but organizations need to determine if changing the datatype of the sensitive field can be done without too much disruption to the application and database schemas. The screenshot below shows the version will be external which means must be stored in another column in the database or dataset.



To protect using CRDP:

```
curl -X POST -H "Content-Type: application/json" -d '{"protection_policy_name": "plain-nbr-ext","data": "0123456789012345"}' localhost:8090/v1/protect
```

Output:

```
{"protected_data":"4735860402263035","external_version":"1001001"}
```

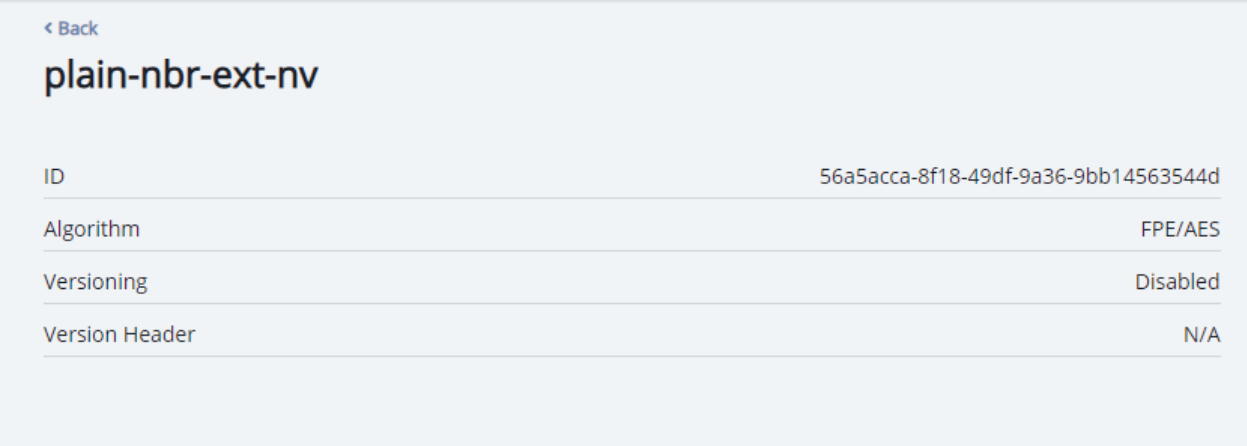
To reveal data using CRDP:

```
curl -X POST -H "Content-Type: application/json" -d '{"protection_policy_name": "plain-nbr-ext","protected_data": "4735860402263035","username": "admin","external_version": "1001001"}' localhost:8090/v1/reveal
```

Output:

```
{"data":"0123456789012345"}
```

The third and final mode of versioning is **NOT** to version. Here is an example of this configuration:



Key versioning is an important part of any security strategy to help ***reduce risk and increase compliance*** but for some use cases it may not be practical which is why the option of no version key is provided.

Key versioning summary
Protect and reveal enable organizations a lot of options to how they handle changes to their application security environment and this capability facilitates enabling crypto agility because versioned keys is part of the profile and not left up to each and every development team to implement.

By providing the version info, for any specific piece of data, we can ensure that the proper cipher, key and parameters defined in that version of the Protection Policy. This is important so that it can be used later when we need to reveal the original data. While providing these examples there were no code changes required other than for the use case where we needed to extract the key version when external mode was used.

Developer friendly API's.

Many encryption implementations tend to be overly complex to implement because the API's are not intuitive with few examples and not very well documented. In addition to that most developers must learn a whole new terminology and technology to properly implement. Usually, it involves a complex

set of APIs that require developers to know quite a bit about crypto… some would say “they need to know too much”. Protect/reveal hides this complexity into two calls PROTECT and REVEAL.

Developers will call PROTECT and provide the plain text data to protect and the name of the Protection Policy to use.

Here is an example of the REST API using CRDP:

```
curl -X POST -H "Content-Type: application/json" \-d '{"protection_policy_name":
"<protection_policy>","data": "<data>"}' \<ip>:<port>/v1/protect
```

Here is an example of the REST API using CRDP:

```
curl -X POST -H "Content-Type: application/json" \-d '{"protection_policy_name":
"<protection_policy>","protected_data": "<data_to_be_revealed>","username": "<user>","external_version":
"<external_version>"}' \<ip>:<port>/v1/reveal
```

For more examples see appendix.

Developer friendly API’s summary

As you can see from above the developer does **not** need to know all the specifics about the key used, initialization vectors, key algorithm, tweaks etc. All the security specifics are now incorporated into the CipherTrust Manager controlled by the security admin who is the person responsible to create protection policies and access policies for the entire enterprise to use. The developers can focus on providing increase application value by improving the functionality of the applications and not have to worry about infrastructure implementations having to do with security.

Appendix

Docker commands for CRDP

Sample command to run CRDP as a standalone container.

```
docker run -e KEY_MANAGER_HOST=192.168.159.134 -e REGISTRATION_TOKEN=yourregtoken -p 8090:8090 -e
SERVER_MODE=no-tls thalesciphertrust/ciphertrust-restful-data-protection
```

Sample command to verify CRDP

```
curl http://localhost:8090/healthz -H 'Content-Type: application/json' -X GET
{"status":"OK"}
```

Sample command to check docker logs

```
docker logs ecd9b2869b99
{"level":"info","time":"Mon, 01 Apr 2024 12:45:26 +0000","msg":"going to initialize shield"}
{"level":"info","time":"Mon, 01 Apr 2024 12:45:26 +0000","msg":"registerClient: Going to register
the client"}
{"level":"info","time":"Mon, 01 Apr 2024 12:45:26 +0000","msg":"registerClient: Register the
client successfully"}
{"level":"error","time":"Mon, 01 Apr 2024 13:12:32 +0000","msg":"Request failed with error:
cannot locate protection policy with name testpp-external-versioned, url:
https://192.168.159.134:443/api/v1/data-protection/protection-policies/testpp-external-versioned,
methodType: GET, statusCode: 404"}
```

Sample docker ps command

```
crdpuser@ubuntu:~$ docker ps
CONTAINER ID   IMAGE                                                  COMMAND
CREATED       STATUS
PORTS
NAMES
ecd9b2869b99   thalesciphertrust/ciphertrust-restful-data-protection   "/crdp"                  2
hours ago   Up 2 hours   0.0.0.0:8090->8090/tcp, :::8090->8090/tcp
hardcore_lumiere
```

CRDP REST API Samples.

Policy internal versioned key: plain-alpha-internal

```
crdpuser@ubuntu:~$ curl -X POST -H "Content-Type: application/json" -d
'{"protection_policy_name": "plain-alpha-internal","data": "abcdabcd"}' localhost:8090/v1/protect
{"protected_data":"1003001q3oaAndi"}
```

Sample output from container logs.

```
{"level":"info","time":"Mon, 01 Apr 2024 13:14:24
+0000","msg":"","app_name":"crdpapp1","audit":true,"client_id":"49e81d27-f362-4fea-911c-
844549a8cefb","endpoint":"/v1/protect","jwt_username":"","key_name":"test","key_version":"1","met
hod":"POST","protection_policy_name":"plain-alpha-
internal","protection_policy_version":"3","source_ip":"172.17.0.1","status":"Success"}
```

Syntax for reveal:

```
curl -X POST -H "Content-Type: application/json" \-d '{"protection_policy_name":
"<protection_policy>","protected_data": "<data_to_be_revealed>","username":
"<user>","external_version": "<external_version>"}' \<ip>:<port>/v1/reveal
```

Reveal for a non-authorized user:

```
crdpuser@ubuntu:~$ curl -X POST -H "Content-Type: application/json" -d
'{"protection_policy_name": "plain-alpha-internal","protected_data": "1003001q3oaAndi"}'
localhost:8090/v1/reveal
{"data":"1003001q3oaAndi"}
```

Reveal for an authorized user:

```
crdpuser@ubuntu:~$ curl -X POST -H "Content-Type: application/json" -d
'{"protection_policy_name": "plain-alpha-internal","protected_data":
"1003001q3oaAndi","username": "admin"}' localhost:8090/v1/reveal
{"data":"abcdabcd"}
```

Policy show last 4: last4-static-plain-nbr-ext

```
crdpuser@ubuntu:~$ curl -X POST -H "Content-Type: application/json" -d
'{"protection_policy_name": "last4-static-plain-nbr-ext","data": "386-76-1952"}'
localhost:8090/v1/protect
{"protected_data":"248-15-1952","external_version":"1001001"}
```

Reveal for an authorized user:

```
crdpuser@ubuntu:~$ curl -X POST -H "Content-Type: application/json" -d
'{"protection_policy_name": "last4-static-plain-nbr-ext","protected_data": "248-15-
1952","username": "admin","external_version": "1001001"}' localhost:8090/v1/reveal
{"data":"386-76-1952"}
```

Reveal for a non-authorized user:

```
crdpuser@ubuntu:~$ curl -X POST -H "Content-Type: application/json" -d
'{"protection_policy_name": "last4-static-plain-nbr-ext","protected_data": "248-15-
1952","username": "bob","external_version": "1001001"}' localhost:8090/v1/reveal
{"data":"248-15-1952"}
```