SwiftStack

Data-at-Rest Encryption

At-rest encryption in SwiftStack encrypts all object data, object etags (checksums), and any user metadata values set on objects. The feature is enabled by a cluster operator and is completely transparent to the end-user. Internally-encrypted data is never returned to clients via the API.

All security features start with defining the threat model (i.e. the circumstances you're trying to protect against). For the at-rest encryption feature in SwiftStack, the threat model is rather straightforward: 1) protect object data and user metadata from being exposed if a data drive leaves the cluster 2) inter-cluster data in flight is encrypted.

There are two common cases where data drives might leave a cluster. The first is by accident: an inventory error could misplace a drive taken out of a cluster and put it into a different server. If the drive isn't erased, the data could be exposed.

The second is when hard drives fail, which is a common reality. The drive vendor's "return merchandise authorization" (RMA) process could result in that user data being exposed to unauthorized parties. The at-rest encryption feature in Swift is designed to protect against this sort of data disclosure. When using encryption in SwiftStack, the cluster operator can confidently RMA drives.

HIGHLIGHTS

- Encryption of object data and user metadata on disk
- AES-256 in counter mode
- Encryption option at a cluster level
- Encryption handled by the proxy role
- Inter-cluster data in flight is encrypted

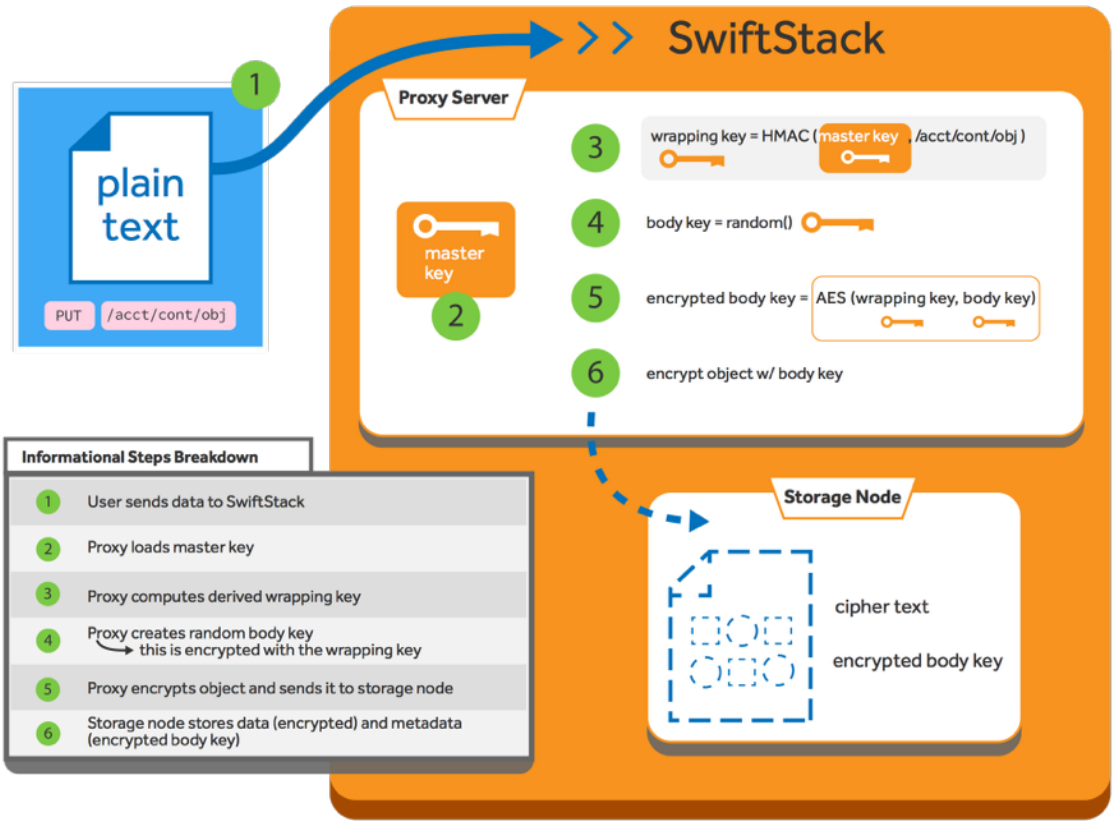
USES

- Protects against data exposure if a drive physically leaves the cluster
- Allows drives to go through an RMA process without data exposure

Additionally, by encrypting data in the proxy tier, all inter-cluster communication is encrypted. While these are typically configured as private networks, this reduces the exposure if this network traffic is compromised. With the combination of client-facing TLS, this provides a mechanism for data to be encrypted at any point the data is on a network.

How it works

SwiftStack has a two-tier architecture: there's a proxy server that handles most of the API and coordinates requests to the storage nodes, and there's the storage nodes which actually persist the data. SwiftStack's encryption is implemented completely in the proxy server. If you're deploying separate physical proxy and storage servers, this design allows for all of the crypto knowledge to stay on the proxy server and not even be on the servers where the data is persisted at all.



Each object stored in SwiftStack is encrypted with it's own unique, randomly-chosen key. Internally, this is called the "body key". This randomly chosen body key is itself encrypted with the object's derived key. This is a technique called "key wrapping" where one key is encrypted with another key. The derived key is the HMAC (keyed-hash message authentication code) of the cluster's master key and the full path to the object. The cluster's master key is available to the proxy server. It's very important that this master key is protected from untrusted parties.

Since key wrapping is used, it means that any eventual support for rotating encryption keys will not require re-encrypting the entire object contents. A SwiftStack cluster will only have to re-encrypt the body key. Although key rotation is not supported yet, the foundation for it exists in this initial version.

SwiftStack uses AES in counter mode because it has the property that byte offsets in the plain text and the cipher text are the same. This allows range requests to encrypted data to be easily supported. One of the goals of at-rest encryption in SwiftStack is that it be completely transparent to the end user. All existing API functions should work in exactly the same way for both encrypted data and non-encrypted data.

Enabling Encryption in SwiftStack

Enable for New Clusters

To ensure all potentially sensitive user data is protected the Encryption feature is enabled Cluster wide. You should enable encryption before you deploy on new hardware to ensure that all user data committed to disk is protected.

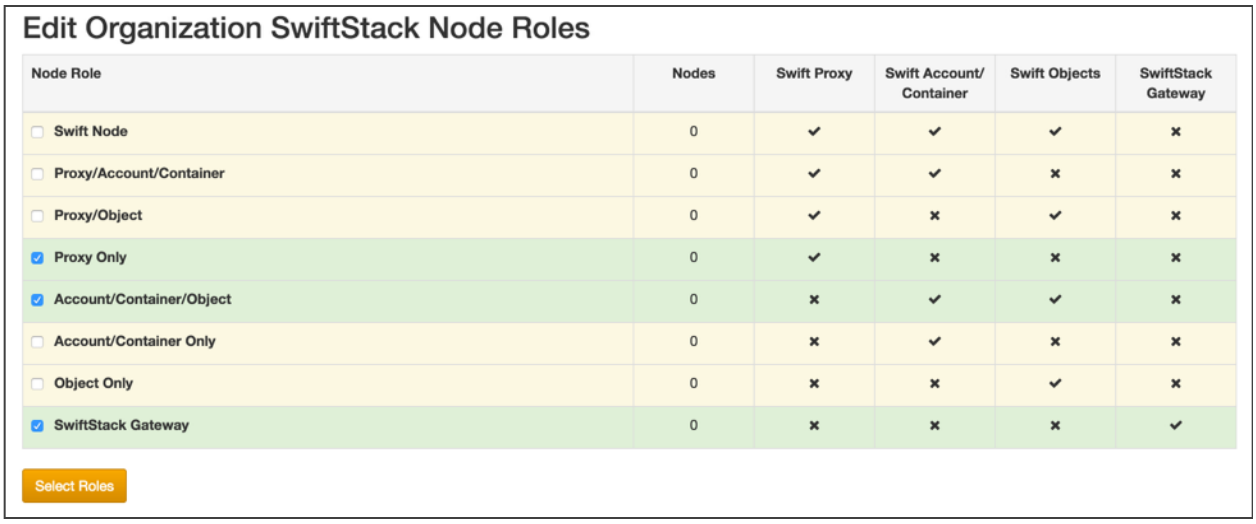
Note: Currently, you may only enable the Encryption feature before the initial deployment of the cluster. If you are interested in using encryption on an existing SwiftStack cluster, please contact support.

Configure Roles

To ensure all potentially sensitive user data is protected the Encryption feature is enabled Cluster wide. You should enable encryption before you deploy on new hardware to ensure that all user data committed to disk is protected.

Optimal threat isolation requires you to segregate your access tier from your storage tier. Encrypted data is never stored in the access tier. No unencrypted key material is ever written into the storage tier.

Navigate to the Node Roles under the **Organization** tab.



Enable the *Proxy Only* Role.

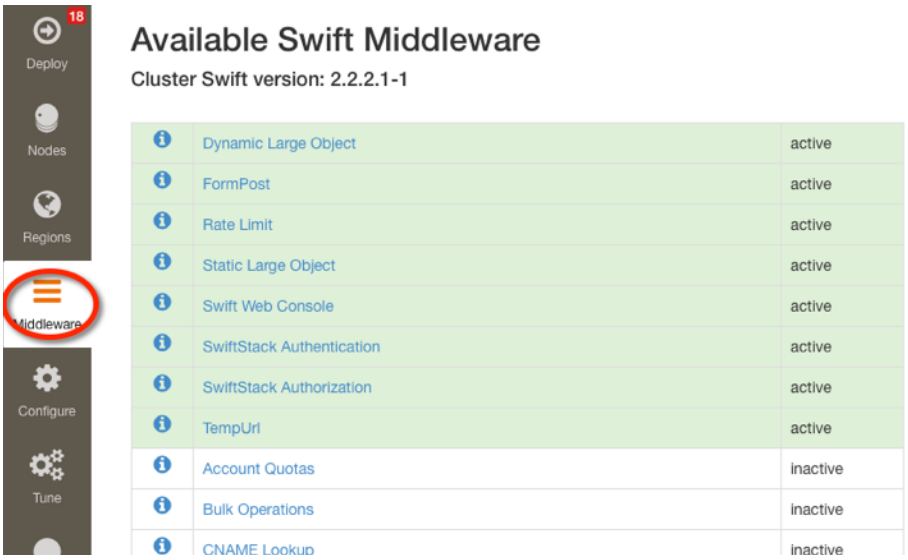
You must also enable one or more of the storage only roles, *Account/Container/Object, Account/Container Only,* or *Object Only.*

Avoid using roles that combine the *Proxy* service with *Account/Container* or *Object* services by disabling the *Swift Node* role.

Enable Encryption

All the steps to Configuring Your Cluster and Provisioning Your Node are preformed like normal, but before Deploying Changes to the Cluster you should stop to enable Encryption!

Encryption is enabled via Middleware.



1. Enable the Encryption Middleware

Be sure the **Enabled** box is checked.

`disable_encryption`

> Only used for testing/benchmarking. Setting this to True will temporarily disable encryption of new data. Previously encrypted data stored in the cluster will still be decrypted. Any data written while the cluster is configured with disable_encryption will be stored in cleartext.

2. Enable the Keymaster Middleware

Be sure the **Enabled** box is checked.

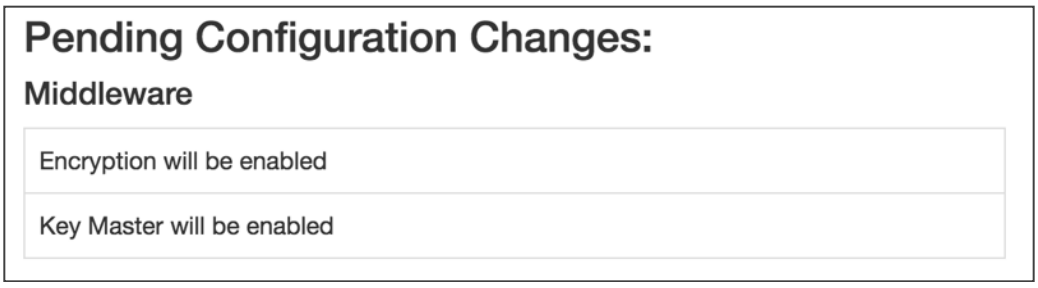
`encryption_root_secret`

> The `encryption_root_secret` option holds the master secret key used for encryption. The security of all encrypted data critically depends on this key and it should therefore be set to a high-entropy value. For example, a suitable encryption_root_secret may be obtained by base-64 encoding a 32 byte value generated by a cryptographically secure random number generator.
>
> The `encryption_root_secret` value is necessary to recover any encrypted data from the storage system, and therefore, it can never be changed and will not be displayed via the SwiftStack controller.

Deploy the Cluster

You may now proceed Deploying Changes to the Cluster.

You should confirm that both the Encryption Middleware and Keymaster Middleware will be enabled before you deploy.



Encryption Details

Note: Topics discussed in this section refer to low-level functionality and configuration settings that are automated by SwiftStack and are not intended to be interacted with by end users. SwiftStack is powered by OpenStack Swift at the core and using these options are required by community users only.

The goal of this section is to give you a complete understanding of how encryption works and is implemented in SwiftStack.

Encryption of data at rest is implemented by middleware that may be included in the proxy server WSGI pipeline. The feature is internal to a Swift cluster and not exposed through the API. Clients are unaware that data is encrypted by this feature internally to the Swift service; internally encrypted data should never be returned to clients via the Swift API.

- The following data are encrypted while at rest in Swift:
- Object content i.e. the content of an object PUT request's body
- The entity tag (ETag) of objects that have non-zero content
- All custom user object metadata values i.e. metadata sent using X-Object-Meta- prefixed headers with PUT or POST requests

Any data or metadata not included in the list above are not encrypted, including:

- Account, container and object names
- Account and container custom user metadata values
- All custom user metadata names
- Object Content-Type values
- Object size
- System metadata

Encryption scheme

Plaintext data is encrypted to ciphertext using the AES cipher with 256-bit keys implemented by the python cryptography package. The cipher is used in counter (CTR) mode so that any byte or range of bytes in the ciphertext may be decrypted independently of any other bytes in the ciphertext. This enables very simple handling of ranged GETs.

In general an item of unencrypted data, **`plaintext`**, is transformed to an item of encrypted data, **`ciphertext`**:

```
ciphertext = E(plaintext, k, iv)
```

where E is the encryption function, k is an encryption key and iv is a unique initialization vector (IV) chosen for each encryption context. For example, the object body is one encryption context with a randomly chosen IV. The IV is stored as metadata of the encrypted item so that it is available for decryption:

```
plaintext = D(ciphertext, k, iv)
```

where D is the decryption function.

The implementation of CTR mode follows NIST SP800-38A, and the full IV passed to the encryption or decryption function serves as the initial counter block.

In general any encrypted item has accompanying crypto-metadata that describes the IV and the cipher algorithm used for the encryption:

```
crypto_metadata = {"iv": <16 byte value>,
                   "cipher": "AES_CTR_256"}
```

This crypto-metadata is stored either with the ciphertext (for user metadata and etags) or as a separate header (for object bodies).

Key management

A keymaster middleware is responsible for providing the keys required for each encryption and decryption operation. Two keys are required when handling object requests: a container key that is uniquely associated with the container path and an object key that is uniquely associated with the object path. These keys are made available to the encryption middleware via a callback function that the keymaster installs in the WSGI request environ.

The current keymaster implementation derives container and object keys from the encryption_root_secret in a deterministic way by constructing a SHA256 HMAC using the encryption_root_secret as a key and the container or object path as a message, for example:

```
object_key = HMAC(encryption_root_secret, "/a/c/o")
```

Other strategies for providing object and container keys may be employed by future implementations of alternative keymaster middleware.

During each object PUT, a random key is generated to encrypt the object body. This random key is then encrypted using the object key provided by the keymaster. This makes it safe to store the encrypted random key alongside the encrypted object data and metadata.

This process of *key wrapping* enables more efficient re-keying events when the object key may need to be replaced and consequently any data encrypted using that key must be re-encrypted. Key wrapping minimizes the amount of data encrypted using those keys to just other randomly chosen keys which can be re-wrapped efficiently without needing to re-encrypt the larger amounts of data that were encrypted using the random keys.

Note: Currently, you may only enable the Encryption feature before the initial deployment of the cluster. If you are interested in using encryption on an existing SwiftStack cluster, please contact support.

Encryption middleware

The encryption middleware is composed of an *encrypter* component and a *decrypter* component.

Encrypter operation

— Custom user metadata —

The encrypter encrypts each item of custom user metadata using the object key provided by the keymaster and an IV that is randomly chosen for that metadata item. The encrypted values are stored as *Object Transient-Sysmeta* with associated crypto-metadata appended to the encrypted value. For example:

```
X-Object-Meta-Private1: value1
X-Object-Meta-Private2: value2
```

are transformed to:

```
X-Object-Transient-Sysmeta-Crypto-Meta-Private1:
  E(value1, object_key, header_iv_1); swift_meta={"iv": header_iv_1,
                                                  "cipher": "AES_CTR_256"}
X-Object-Transient-Sysmeta-Crypto-Meta-Private2:
  E(value2, object_key, header_iv_2); swift_meta={"iv": header_iv_2,
                                                  "cipher": "AES_CTR_256"}
```

The unencrypted custom user metadata headers are removed.

— Object body —

Encryption of an object body is performed using a randomly chosen body key and a randomly chosen IV:

```
body_ciphertext = E(body_plaintext, body_key, body_iv)
```

The body_key is wrapped using the object key provided by the keymaster and a randomly chosen IV:

```
wrapped_body_key = E(body_key, object_key, body_key_iv)
```

The encrypter stores the associated crypto-metadata in a system metadata header:

```
X-Object-Sysmeta-Crypto-Body-Meta:
    {"iv": body_iv,
     "cipher": "AES_CTR_256",
     "body_key": {"key": wrapped_body_key,
                  "iv": body_key_iv}}
```

Note that in this case there is an extra item of crypto-metadata which stores the wrapped body key and its IV.

— Entity tag —

While encrypting the object body the encrypter also calculates the ETag (md5 digest) of the plaintext body. This value is encrypted using the object key provided by the keymaster and a randomly chosen IV, and saved as an item of system metadata, with associated crypto-metadata appended to the encrypted value:

```
X-Object-Sysmeta-Crypto-Etag:
  E(md5(plaintext), object_key, etag_iv); swift_meta={"iv": etag_iv,
                                                      "cipher": "AES_CTR_256"}
```

The encrypter also forces an encrypted version of the plaintext ETag to be sent with container updates by adding an update override header to the PUT request. The associated crypto-metadata is appended to the encrypted ETag value of this update override header:

```
X-Object-Sysmeta-Container-Update-Override-Etag:
    E(md5(plaintext), container_key, override_etag_iv);
    meta={"iv": override_etag_iv, "cipher": "AES_CTR_256"}
```

The container key is used for this encryption so that the decrypter is able to decrypt the ETags in container listings when handling a container request, since object keys may not be available in that context.

Since the plaintext ETag value is only known once the encrypter has completed processing the entire object body, the `X-Object-Sysmeta-Crypto-Etag` and `X-Object-Sysmeta-Container-Update-Override-Etag` headers are sent after the encrypted object body using the proxy server's support for request footers.

— Conditional request —

In general, an object server evaluates conditional requests with **`If[-None]-Match`** headers by comparing values listed in an **`If[-None]-Match`** header against the ETag that is stored in the object metadata. This is not possible when the ETag stored in object metadata has been encrypted. The encrypter therefore calculates an HMAC using the object key and the ETag while handling object PUT requests, and stores this under the metadata key **`X-Object-Sysmeta-Crypto-Etag-Mac`**:

```
X-Object-Sysmeta-Crypto-Etag-Mac: HMAC(object_key, md5(plaintext))
```

Like other ETag-related metadata, this is sent after the encrypted object body using the proxy server's support for request footers.

The encrypter similarly calculates an HMAC for each ETag value included in If[-None]-Match headers of conditional GET or HEAD requests, and appends these to the If[-None]-Match header. The encrypter also sets the X-Backend-Etag-Is-At header to point to the previously stored X-Object-Sysmeta-Crypto-Etag-Mac metadata so that the object server evaluates the conditional request by comparing the HMAC values included in the If[-None]-Match with the value stored under X-Object-Sysmeta-Crypto-Etag-Mac. For example, given a conditional request with header:

```
If-Match: match_etag
```

the encrypter would transform the request headers to include:

```
If-Match: match_etag,HMAC(object_key, match_etag)
X-Backend-Etag-Is-At: X-Object-Sysmeta-Crypto-Etag-Mac
```

This enables the object server to perform an encrypted comparison to check whether the ETags match, without leaking the ETag itself or leaking information about the object body.

Decrypter operation

For each GET or HEAD request to an object, the decrypter inspects the response for encrypted items (revealed by crypto-metadata headers), and if any are discovered then it will:

1. Fetch the object and container keys from the keymaster via its callback
2. Decrypt the **`X-Object-Sysmeta-Crypto-Etag`** value
3. Decrypt the **`X-Object-Sysmeta-Container-Update-Override-Etag`** value
4. Decrypt metadata header values using the object key
5. Decrypt the wrapped body key found in **`X-Object-Sysmeta-Crypto-Body-Meta`**
6. Decrypt the body using the body key

For each GET request to a container that would include ETags in its response body, the decrypter will:

1. GET the response body with the container listing
2. Fetch the container key from the keymaster via its callback
3. Decrypt any encrypted ETag entries in the container listing using the container key

Impact on other services and features

Encryption has no impact on *Object Versioning* other than that any previously unencrypted objects will be encrypted as they are copied to or from the versions container. Keymaster and encryption middlewares should be placed after **`versioned_writes`** in the proxy server pipeline.

Encryption has no impact on the *object-auditor* service. Since the ETag header saved with the object at rest is the md5 sum of the encrypted object body then the auditor will verify that encrypted data is valid.

Encryption has no impact on the object-expirer service. **`X-Delete-At`** and **`X-Delete-After`** headers are not encrypted.

Encryption has no impact on the *object-replicator* and *object-reconstructor* services. These services are unaware of the object or EC fragment data being encrypted.

Encryption has no impact on the *container-reconciler* service. The *container-reconciler* uses an internal client to move objects between different policy rings. The destination object has the same URL as the source object and the object is moved without re-encryption.