SURVEY ON SWIFT OBJECT STORAGE FOR UNSTRUCTURED DATA

Prof. Gururaj Kulkarni ¹, Prof. Pavan Mitragotri ² and Prof. Swarooprani H.Manoor ³
¹,²,³ Department of Master of Computer Applications, KLS’ Gogte Institute of Technology

Abstract- Cloud Storage Systems are increasingly noticed now-a-days as they are promising elastic capability and high reliability at low cost. In these services, the files are stored in an authenticated cloud storage service center. The most important feature in storage is adjusted dynamically, and there won’t be any worry about space being inadequate or wasted. This paper presents comprehensive study of various authentication mechanisms of swift object storage. Also various shortcomings of authentication mechanism are discussed. Unlike file systems, object storage is unstructured. which means files are stored alongside each other in a storage pool rather than in a hierarchy. In addition, the metadata that characterizes object storage is far more detailed than in a file system. Servers use unique identifiers to find objects, allowing them to be retrieved without knowledge of their physical location.

Keywords- Open Stack, Swift, REST, Cloud Storage, Authentication.

1. INTRODUCTION

Now-a-days demands on storage systems are increasing. Users are providing and consuming more data; for example: social media, gaming, online videos, SaaS applications all contributing to vast need for easily accessible storage systems. There are different types of data that have different access patterns and therefore can be best stored on different types of storage systems. Data storage is categorized into three types namely block storage, file storage, and object storage.

1.1. Block Storage:

This type of storage stores structured data, which is represented as equal-size blocks without putting any interpretation on the bits. This kind of storage is useful when the application needs to tightly control the structure of the data. A common use for block storage is databases, which can use a raw block device to efficiently read and write structured data. Additionally, file systems are used to abstract a block device, which then does everything from running operating systems to storing files.

1.2. File Storage:

This is what we are most used to seeing as desktop users. File storage takes a hard drive and exposes a file system on it for storing unstructured data. File system is one where you open and close documents on your computer. Although file storage provides a useful abstraction on top of a storage device, there are challenges as the system scales. File storage needs strong consistency, which creates constraints as the system grows and is put under high demand. In addition, file systems often require other features such as file locking that create a barrier for working well with large amounts of data.

1.3. Object Storage:

This is familiar to those who regularly access the internet or use mobile devices. This type of Storage does not give access to raw blocks of data nor does it give access to file based type. Rather it gives access to entire object or block of information by an API particular to that framework. Objects are accessible via URLs using HTTP protocols, similar to how websites are accessible in web browsers. Object storage abstracts these locations as URLs so that the storage system can grow and scale independently from
the underlying storage mechanisms. This makes object storage ideal for systems that need to grow and scale for capacity, concurrency, or both. The important advantage of object storage is, its ability to distribute requests for objects across a large number of storage servers. This provides reliable, scalable capacity for huge measure of information at a lower expense. [1]

Swift allows for a wide spectrum of uses, including supporting web/mobile applications, backups, and active archiving. Layers of additional services let users access the storage system via its native HTTP interface, or use command-line tools, file system gateways or easy-to-use applications to store and sync data with their desktops, tablets, and mobile devices. Swift is an object storage system, which means it trades immediate consistency for eventual consistency. This allows Swift to achieve high availability, redundancy, throughput, and capacity. With a focus on availability over consistency, Swift has no transaction or locking delays.

Large numbers of simultaneous reads are fast, as are simultaneous writes. This means that Swift is capable of scaling to an extremely large number of concurrent connections and extremely large sets of data. Swift is open source multi tenant, highly scalable and durable object storage system intended to store extensive measure of unstructured data at lower expense. Swift is used by businesses of all sizes, service providers, and research organizations worldwide. It is typically used to store unstructured data such as documents, web content, backups, images, and virtual machine snapshots. Originally developed as the engine behind Rack Space Cloud Files in 2010, it was open-sourced as part of the OpenStack project. Swift is not a traditional file system or a raw block device. Instead, it lets us store, retrieve, and delete objects along with their related metadata in container using a Restful HTTP API.

Swift is basically designed to scale linearly based on how much data needs to be stored and how many users need to be served. This implies Swift can scale from a couple of hubs with a bunch of drives to a large number of machines, even hundreds or petabytes of storage. As the number of requests increments because of system growth in usage, performance does not drop down, to some extent in light of the fact that swift is intended to be scalable with no single purpose of disappointment. To scale up, the system grows where needed by adding storage nodes to increase storage capacity.

With growth in mobile technology, need for object storage is rapidly increasing. As more and more data is being stored in object storage format, studying security aspects of object storage is need of hour. In this paper, we are presenting in depth study of SWIFT Object Storage authentication mechanism and we will also discuss in short implementing new authentication mechanism for SWIFT storage. This paper is organized in following sections. In section 2, we are discussing SWIFT Object Storage its architecture and authentication mechanisms, Section 3 presents future enhancement and in section 4 we conclude paper.

II. SWIFT OBJECT STORAGE

Swift is a highly available, distributed, eventually consistent object/blob store. Organizations can utilize swift to store loads of information productively, securely, and inexpensively. [2] Object storage is only accessible with a specialized client application. The main reason why Swift serves so well for highly-available, unstructured application data is that its design, just like Amazon S3 (Block Storage), incorporates eventual consistency. In swift objects are protected by storing multiple copies of data so that if one node fails, data can be easily retrieved from another node. Under certain circumstances if multiple nodes fail data will be available to user. Swift’s design for eventual consistency means that there is a guarantee that the system will eventually become consistent and have the most up-to-date version of data for all copies of the data but still provide availability to data should hardware fails. This type of design will make it optimal when performance and scalability are considered, particularly for massive, highly distributed infrastructures with unstructured data serving global sites. [3]

Swift architecture can be viewed as logical organization and physical organization of objects.

2.1. SWIFT Architecture:
2.1.1. Logical organization of objects in Swift:

A tenant is assigned an account. A tenant could be any entity: a person, a department, a company, and so on. The account holds containers. Each container holds objects, as shown in the following Fig. 1. A tenant can create additional users to access an account. Users can keep adding containers and objects within a container without having to worry about any physical hardware boundaries, unlike traditional file or block storage. Containers within an account need to have a unique name, but two containers in separate accounts can have the same name. Containers are flat and objects are not stored hierarchically. [4]

![Fig. 1 Logical organization of objects in Swift](image)

2.2.2. Physical data location hierarchy:

Swift completely abstracts logical organization of data from the physical organization. At a physical level, Swift classifies the physical location into a hierarchy, as shown in the Fig. 2

Region: Swift stores data in regions that are geographically separated and thus suffer from a high-latency link. A user may use only one region, for example, if the cluster utilizes only one datacenter.

Zone: Within regions, there are zones. Zones are a set of storage nodes that share different availability characteristics. Availability may be defined as different physical buildings, power sources, or network connections. This means that a zone could be a single storage server, a rack, or a complete datacenter depending on your requirements.

Storage servers: A zone consists of a set of storage servers ranging from just one to several racks.

Disk: Disk drives are part of a storage server. These could be inside the server or connected via a JBOD.

![Fig. 2 Physical data location hierarchy](image)

Swift allows users to store unstructured data objects with a canonical name containing three parts: account, container, and object as depicted in Fig. 3.

Account: Account storage will contain the metadata about the account itself and also it contains the list of containers in that account. /Account/Container

Container storage will contain the metadata about the container itself furthermore it contains the list of objects in that container. /Account/Container/Object.
Object storage will contain the metadata about the object and data object itself.

2.2.3. Authentication System in Swift:

Swift does not require a specific default authentication service. Instead, it allows administrators to plug one or more authentication services into its framework. This flexibility lets you choose authentication middleware components that meet your needs, and enable them in the proxy server configuration of your Swift cluster. Swift comes with two types of auth middleware, TempAuth and Keystone Auth. TempAuth is not suitable for production use because it deliberately compromises security for the sake of convenience, but it might be acceptable for testing Swift. For production environments where your cluster needs to integrate with other OpenStack products such as Nova and Glance, the Keystone Auth system is better. There are also many popular third-party auth middleware components from which to choose. Some have been developed by the Swift community, while others are offered as proprietary auth middleware components as part of a vendor’s Swift products.

2.2.4. Authentication works as follows:

- A user presents credentials to the auth system. This is done by executing an HTTP REST API call.
- The auth system provides the user with an AUTH token.
- The AUTH token is not unique for every request, but expires after certain duration.
- Every request made to Swift has to be accompanied by the AUTH token.
- Swift validates the token with the Auth system and caches the result. The result is flushed upon expiration.
- The Auth system generally has the concept of administrator accounts and non-admin accounts. Administrator requests are obviously passed through.
- Non-admin requests are checked against container level Access Control Lists (ACL). These lists allow the administrator to set read and write ACLs for each non-admin user.
- Therefore, for non-admin users, the ACL is checked before the proxy server proceeds with the request. The following Fig. 4 illustrates the steps involved when Swift interacts with the Auth system.
Fig. 4 Swift and its interaction with the authentication system.

SWIFT authentication is performed before sending user request to proxy server. Requester information is validated in way defined by authentication middleware. After successful validation of user, validated information is added to WSGI environment. Format and details of information needs to be added to WSGI server depends on middleware authorization module needs. To avoid conflict with other authentication servers used by SWIFT cluster, it is advised to prefix their tokens and swift storage accounts with configurable reseller prefix, like “AUTH_” which is default in case of TempAuth.

Authorization is performed through callbacks by the Swift Proxy server to the WSGI environment’s swift. Authorize value, if one is set. The swift Authorize value should simply be a function that takes a Request as an argument and returns none if access is granted or returns a callable (environ, start response) if access is denied. This callable is a standard WSGI callable. Generally, you should return 403 Forbidden for requests by an authenticated user and 401 Unauthorized for an unauthenticated request. For example, here’s an authorize function that only allows GETs (in this case you’d probably return 405 Method Not Allowed, but ignore that for the moment).

2.2.5. Following are the swift authentication methods:

1) TempAuth:

TempAuth is the other middleware auth system shipped with Swift. It is intended for testing rather than production, but it is a complete auth system. TempAuth stores usernames, passwords, and group membership in its middleware configuration section in the proxy-server.conf file, which is as insecure as it sounds. In addition to the security concern, this also means that whenever a user is added or a password is changed, the proxy server process must be restarted. Although TempAuth is not suitable for production deployments, depending on your security considerations it might be sufficient for test clusters. Additionally, because it is a reference implementation of a fully functional (if poorly persisted) auth middleware, it has served as a template for the development of several other auth middleware packages. TempAuth accepts usernames in the format account name:username, so that the same Swift account may be accessed by many users. For example, an account with a storage URL https://www.swift.com/v1/AUTH_acc_name might be accessed by users acc_name:user1 and
acc_name:user2. These two users would have different passwords, different group memberships, and so forth. Once it authenticates, TempAuth grants authorization access based on user type and groups. TempAuth has the concept of admin (account-level access) and non-admin (container-level access) users within an account. [5]

TempAuth stores user name and password in plaintext format at /etc/swift/proxy_server.conf file including admin. This file has access to all users who has access to proxy server node. Admin can access any of account details.

2) SWAuth:

SWAuth is an auth middleware component for Swift that uses Swift itself to store account data. It has the advantages of not requiring the potentially complex deployment of another system (such as LDAP), and of having the authentication information stored in Swift’s highly available infrastructure. It has the disadvantage of requiring a full Swift retrieval operation for each unscathed auth operation, and there are certainly systems that are better optimized than Swift for the task of authenticating users.

SWAuth stores only admin user name and password in plaintext format at /etc/swift/proxy_server.conf file. This file has access to all users who has access to proxy server node. Admin can access any of account details. [5]

3) Keystone:

Swift is able to authenticate against keystone. Keystone is Open Stack’s standard identity service, which makes it easy to integrate with other OpenStack services. [6] To use the keystone middleware, we need to first configure the auth token middleware. Authentication, token validation, retrieval of real user verification data is done by auth token middleware. The keystone middleware performs authentication and maps keystone users to swifts ACL. The OpenStack Identity Service, known as Keystone, provides services for authenticating and managing user accounts and role information for our OpenStack cloud environment. It is a crucial service that underpins the authentication and verification between all of our OpenStack services and is the first service that needs to be installed within an OpenStack environment. Authentication with OpenStack Identity Service sends back an authorization token that is passed between the services, once validated. This token is subsequently used as your authentication and verification that you can proceed to use for that service, such as OpenStack Storage and Compute. As such, configuration of the OpenStack Identity Service must be done first and consists of creating appropriate roles for users and services, tenants, the user accounts, and the service API endpoints that make up our cloud infrastructure. [3]

OpenStack Identity enables:

- Configuration of centralized policies across users and systems
- Creation of users, tenants and define permissions for swift storage through the use of role-based access control (RBAC) features
- Integration with existing directories, allowing for single-source of identity authentication.
- As a user get a rundown of the administrations that you can get to and make API request or log into the web dashboard to make assets claimed by your account.
- Keystone stores user name and password in database. User will provide authentication details by setting environment variable OS_AUTH_URL, OS_USERNAME, OS_PASSWORD.

C. ACL’s:

Swift supports access control at two levels: the account level and the container level. The responsibility for interpreting and enforcing access control rules falls to the authentication and authorization system, rather than to core Swift. However, the following access control patterns are defined in core Swift, and auth systems are generally expected to follow them.
Account-level access control grants privileges over an entire account. For example, account-level read-only access allows designated users to read anything (except privileged security-related metadata) in the account. Account-level read-write access adds the ability to create and overwrite objects, containers, and (unprivileged) metadata in the account. Account-level admin access, also known as swift owner access, represents full privileges equivalent to ownership of the account.

Container-level access control grants privileges within a specific container. Because this type of access operates at the container level, a user might have certain permissions in some containers in another user’s account, but different permissions (or no permissions) in other containers. The access levels for containers are simply “read” and “write.” Users might have container read access and be able to download any object within the container, or have container write access and be able to upload objects to the container, or have both read and write access to the container. Container-level access control can also be used to grant permissions to unauthenticated users to read and/or write objects within the container.

III. IMPLEMENTATION

3.1. Configuring proxy server and authentication server:

In order to integrate authentication service with swift object storage we need to configure .conf file and proxy-server.conf file.

Configuring proxy-server.conf file: The proxy server accepts user request and transfers it to appropriate location on swift (account, container and object) and directs the requests successfully. This file acts as a mediator between auth system and swift. The path to configure this file is “/etc/swift/proxy-server.conf”.

Restart proxy server: After editing the proxy-server.conf file, the proxy server should be restarted in order to apply the changes that are done.

Restart authentication server: After editing the auth.conf file, the auth server should be restarted in order to apply the changes done.

B. Algorithm

Account creation:

| Code Block |
|---|
| Do acc = keystone account-create --os-endpoint=endpoint --os-token=token --name=accountname IF acc is not equal to null THEN Do display “account created” return -1 ELSE Do display “account creation failed” return 0 |

Account deletion:

| Code Block |
|---|
| IF account _exist THEN Do delete = keystone account-delete --os-endpoint=endpoint --os-token=token --name=accountname Do display “account deleted” IF delete not equal to null THEN return error ELSE return success ELSE Do display “account does not exist” |
Container creation:

```
Do token= curl -X POST -i https://storage-url/v2.0/Tokens -H “Content-type: application/json” -d
        '{"auth":{"passwordCredentials":{"username":"user","password":"pass"},"accountName":"account"}}

Do spit token to extract account-id and auth-token
Do container=curl -X PUT -i storage-url /account-id/container-name -H ‘X-Auth-Token:auth-token’ Do display “container”
```

Container deletion:

```
Do token= curl -X POST -i https://storage-url/v2.0/Tokens -H “Content-type: application/json” -d
        '{"auth":{"passwordCredentials":{"username":"user","password":"pass"},"accountName":"account"}}

Do spit token to extract account-id and auth-token
Do delete=curl -X DELETE -i storage-url/account-id/container-name -H ‘X-Auth-Token: auth-token’ Do display “delete”
```

Uploading object:

```
Do token= curl -X POST -i https://storage-url/v2.0/Tokens -H “Content-type: application/json” -d
        '{"auth":{"passwordCredentials":{"username":"user","password":"pass"},"accountName":"account"}}

Do spit token to extract account-id and auth-token
Do upload=curl -X PUT -i storage-url/container-name -H "X-Auth-Token: token" -T file.txt storage-url/container/file.txt
Do display “upload”
```

Downloading object:

```
Do token= curl -X POST -i https://storage-url/v2.0/Tokens -H “Content-type: application/json” -d
        '{"auth":{"passwordCredentials":{"username":"user","password":"pass"},"accountName":"account"}}

Do spit token to extract account-id and auth-token
Do download=curl -X GET -i storage-url/container/file.txt -H “X-Auth-Token: token” storage-url/container/file.txt
Do display “download”
```

Listing objects:

```
Do token= curl -X POST -i storage-url/v2.0/Tokens -H “Content-type: application/json” -d
        '{"auth":{"passwordCredentials":{"username":"user","password":"pass"},"accountName":"account"}}

Do spit token to extract account-id and auth-token
Do list = curl -i storage-url/container/object -X GET -H “X-Auth-Token: token”
Do display “list”
```

Deleting objects:

```
Do token= curl -X POST -i storage-url/v2.0/Tokens -H “Content-type: application/json” -d
        '{"auth":{"passwordCredentials":{"username":"user","password":"pass"},"accountName":"account"}}
```
C. Curl Commands

- Account create (Creates a new account)
  
  ```
  Account create --name account -name (--description account -description) (--enabled true/false) Arguments
  
  Account name- New account name (must be unique).
  Account -description- Description of Account. Default is none. enabled- Initial tenant enabled status.
  Default is true.
  ```

- Account delete (Deletes an existing account)
  
  ```
  Account delete < Account > Arguments
  
  Account - Name or ID of account to delete.
  ```

- Container create (The command used to create user container in account)
  
  ```
  The command used to create user container in account is as follows: curl -X PUT -i storage_url/container -H “X-Auth-Token: token”
  ```

- Container delete (The command used to delete user container from account)
  
  ```
  The command used to delete user container from account is as follows: curl -X DELETE -i storage_url/container -H “X-Auth-Token: token”
  ```

- List containers (The command used for listing containers)
  
  ```
  The command used for listing containers is as follows: curl -X GET -i storage_url /v1/tenant-id -H ‘X-Auth-Token: token’
  ```

- Upload object (Command used to upload objects in user container)
  
  ```
  Command used to upload objects in user container is as follows:
  curl -X PUT -i -H “X-Auth-Token: token” -T object-name storage_url/cont/obj-name
  ```

- Download object (Command used to download objects in user container)
  
  ```
  Command used to download objects in user container is as follows:
  curl -X GET -i storage_url/container/object-name
  ```

- List object (Command used to list objects from user container)
  
  ```
  Command used to list objects from user container is as follows:
  curl -X GET -i storage_url/v1/account-id/container -H ‘X-Auth-Token: token’
  ```

IV. FUTURE ENHANCEMENT

In this paper we have discussed about authentication server and middleware namely TempAuth, Keystone and SWAuth for swift storage. As future work, we need to study integration of LDAP and Active Directory with swift storage. Based on study, we may need to implement our own authentication server and middleware.

V. CONCLUSION

As discussed above, both TempAuth and SWAuth stores username and password in plain text format on proxy server node. Even communication of authentication is done using plain text format. So both TempAuth and SWAuth are not much secured. Keystone stores user information in database. But communication of authentication information is done in plain text format. So keystone is a better option as compared to TempAuth and SWAuth but not complete secured option. After studying LDAP and Active Directory integration with swift storage and based on our findings we may need to implement our own authentication server and middleware.

REFERENCES

[1] L. G.-M. L. Q. D. Q. L. L. Guo Yu-Feng, "A Novel Intelligent Object Storage Controller for," in The 1st International Conference on Information Science and Engineering (ICISE2009), 2009.
[2] Neelam Patil, Shubhada Walekar, Trupti Mane, Prof. A.S.Shitole, "Profiling and Benchmarking of Openstack..."
Swift,” International Journal of Advanced Research in Computer and Communication Engineering, vol. Vol. 3, no. Issue 1, pp. 5237-5240, 2014.

[3] N. N. Bhanu Prakash Gopularam, “Data Confidentiality in Public Cloud: A Method for Inclusion of ID-PKC Schemes in OpenStack Cloud,” International Journal of Computer Applications (0975 – 8887), vol. Volume 93 – No.8, pp. 40-45, 2014.

[4] OpenStack Foundation, “Swift Architectural Overview,” OpenStack Foundation, [Online]. Available: http://docs.openstack.org/developer/swift/overview_architecture.html. [Accessed 23 June 2015].

[5] S. M. a. P. S. Hala Albaroodi, "CRITICAL REVIEW OF OPENSTACK," Journal of Computer Science 10, pp. 23-33, 2014.

[6] N. Jayaprakash, D. Dhanasekaran, "Preserving Data Security in Cloud Storage," N. Jayaprakash and D. Dhanasekaran, pp. 795-799, 2015.

[7] Sridevi Bonthu,Y S S R Murthy,M. Srilakshmi, "Building an Object Cloud Storage Service System using OpenStack Swift," International Journal of Computer Applications (0975 – 8887), vol. Volume 102 – No.10, pp. 39-42, 2014.

[8] Omar SEFFAOU, Mohammed AISSAOUL, Mohsine ELEULDJ, "OpenStack: Toward an Open-Source Solution for," International Journal of Computer Applications (0975 - 8887), Vols. Volume 55 - No. 03, pp. 38-42, 2012.

[9] OpenStack Foundation, "Welcome to Swift’s documentation!,” OpenStack Foundation, [Online]. Available: http://docs.openstack.org/developer/swift/. [Accessed 27 June 2015].

[10] R. C. H. Hrishikesh Dewan, "A Survey of Cloud Storage Facilities,” in 2011 IEEE World Congress on Services, Bangalore, 2011.

[11] OpenStack Foundation, "Middleware," OpenStack Foundation, [Online]. Available: http://docs.openstack.org/developer/swift/middleware.html. [Accessed 27 June 2015].