Secure-storage for cloud-storage system

## Introduction

Nowadays, cloud storage systems are becoming more and more popular. But how to achieve security as well as convenience on these systems remains a question. We will present a possible solution to this problem as following and test it on OpenStack Swift. Swift is a cloud-storage system on OpenStack which uses object stores. We deploy database server in server layer to generate and store the encryption keys. Besides, we alter client and server scripts of Swift, so it they send and receive necessary encryption information and choose encryption ways flexibly.

## Scope

Swift has an original encryption system which encrypts the objects on server-end and then sends it to the backend storage node. The system uses AES\_CTR with one key for one user only. And the key is written into a local configuration file in plaintext. It has many significant drawbacks. First, if the key is lost or altered due to some reason, then all of the objects encrypted by previous key will be unable to be decrypted. What’s more, if the key is leaked, (the configuration file is not encrypted itself),all the encrypted objects are no longer secure. In addition, this one-key for one-user mechanism is very rigid. As long as the system is enabled, every object to be uploaded after will all be encrypted with the same key as well. Users might want to decide whether to encrypt an object and want to divide objects into groups with different encryption level.

## Contributions

**Flexible level chosen by client**. In our encryption system design, we let the client choose whether to encrypt a file each time before uploading and which level it should be encrypted on. There are three levels it can choose from: account, container and object. The account level is the lowest security level. Each account has only one key for this level. If it got hacked or leaked, all objects using encryption level on account will be unsecure. And object level is the highest, each object has its own unique key.

**Key stored in another database server instead of storage server.** If an attacker got all files on storage-server, he still can’t get the files which are encrypted. The keys are stored in another database server. For an attacker who wants to get all files, he has to get both storage server and database server. In original plan, storage server is a highly vulnerable target. In our plan, we transfer some risks from it to another server, so we can focus on how to secure the database server. Storage node will have less burden and more efficiency.

## Challenges

Our design will work best under two assumptions. The client trusts the cloud-storage system provider and the transport channels are secure. We do the encryption on server-end. So, when server get the file, it’s still in plaintext. Chances that server might see these objects and attackers may get the files while they being transmitting in channels. We can also encrypt files before sending to servers, but it will cause problems as for deduplication. A possible solution is to encrypt the file and adds a md5 tag of the plaintext to it. But if doing that, client will have to do more work rather than just put an object which sacrifices efficiency on client-end.

This system will work for best the companies whose users are willing to take more convenience and hope to let the provider do security work. And it will provide access to solve the problem of deduplication on server-end.

The main challenges are how to deploy this database layer. Because Swift’s deployment uses a monolithic configuration setup, it’s difficult to identify the original script of a specific function. And because the servers run continuously, it’s hard to debug them. Besides, it’s still in debate which end should this encryption system should be deployed on. We could do a little adaptation work to let the client to do the encryption work as well. But mainly focus on convenience for clients and provide access for deduplication, we decide to deploy it on the server end. It could still corporate with specific client’s need using the methods I presented above.

Besides, Swift transports and reads files in bytes stream, but writes it in chunks. So it will cause problems for big-sized file. The blocks when read may differ the blocks when write. So to solve that, we divide the file into same size chunk both while uploading and downloading.

## Design

While uploading a file:

1.A client sends a request tagged with the encryption level it chooses as well as the project to a proxy server node.

2.The proxy gets the request and review the encryption info in headers. If the info exist, proxy will create an IV for later encryption and tagged it to the headers. Proxy finish process work and pass request to the storage node.

3.The storage node get the request and review the encryption info. If it exists, then review the IV. If IV not exists, create one. (This could happen if client directly contacts with storage node.)

The storage sends a query request to the database node regarding the token of the object.

4.Database query the result. If the key exists already, it will be sent back to storage node. If not, the database will send a response back to storage server waiting for it to assign a new key.

5. Storage server use the key(either from database or it creates one by itself) to do encryption work and write the object into device. Encryption info will be written into object’s metadata.

While downloading a file:

1.Storage server gets the request of downloading an file. It reads the body of the file in buffer. Then it will send a query request to database.

2.Database query the result and send the key back.

3.Storage server uses the key to decrypt it and sends it to the client.

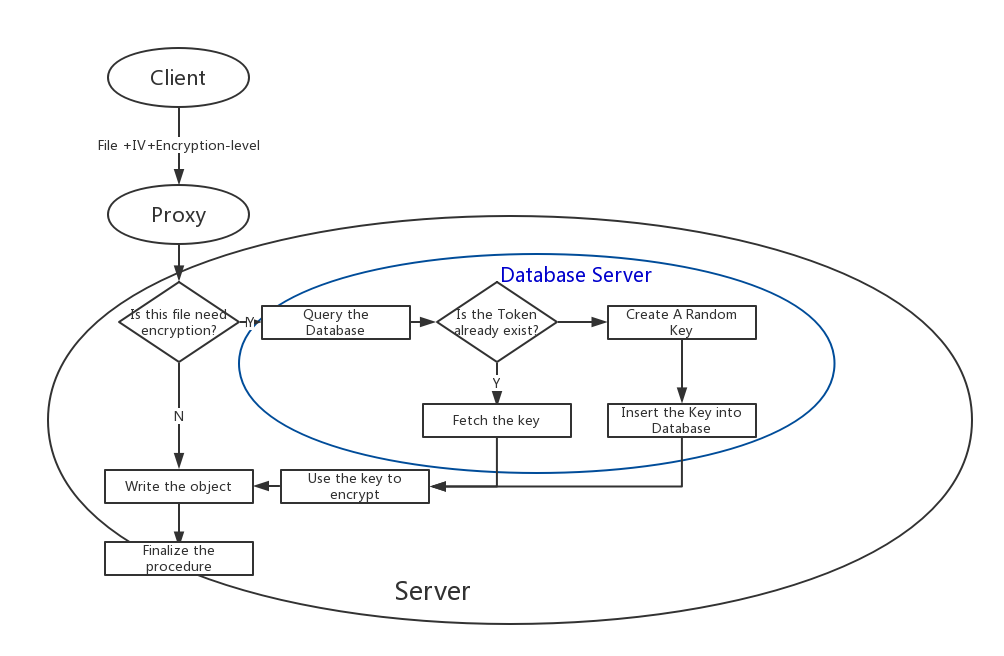


Fig 1: Overview of system’s procedure while uploading

Class key\_generate(object):

#Create this engine while encrypting

def \_\_init\_\_(self,level,token):

result=query\_database(DBconn,level,token)

if result is not null:

key=result

else:

key=random(32)

insert\_database(DBconn,level,token,key)

self.keytext=key

Fig 2: Code of key generate and store in database

Class PUT\_object:

……

If ‘X-Encryption-level’ not in headers:

swift.writer(chunks)

else:

keye=keygenerate(headers[‘X-Encryption-level’],md5(token))

encrykey=keye.keytext

chunks.encrypt(encrykey,iv)

swift.writer(chunks)

……

metadata[‘Encryption-level’]= headers[‘X-Encryption-level’]

metadata[‘Encryption-token’]=token

metadata[‘IV’]=iv

put(metadata)

Class GET\_object:

If ‘Encryption-level’ not in metadata:

Swift.send(body)

Else:

chunks=split\_body(body)

keyd=keyfetch(metadata[‘Encryption-level’],md5(token))

decrykey=keyd.keytext

chunks.decrypt(decrykey,metadata[‘IV’])

body=join\_body(chunks)

Swift.send(chunks)

Fig 3 Code of Storage Server

create database keyencry

create table keytable(group\_type varchar(10),group\_token char(32),keytext char(32))

alter table keytable add primary key(group\_type,group\_token)

Fig4 Code of database

## Deployment

We run our test on cloudlab on a 4-node cluster. They are client, proxy server, storage server and database server respectively.

|  |  |  |
| --- | --- | --- |
| IP | Node type | Service |
| 192.168.0.1 | Storage | account-server, container-server and object-server |
| 192.168.0.5 | Proxy | proxy-server |
| 192.168.0.7 | Database | MYSQL |
| 192.168.0.9 | Client | Python-swiftclient |

Table 1: The deployment of the cluster

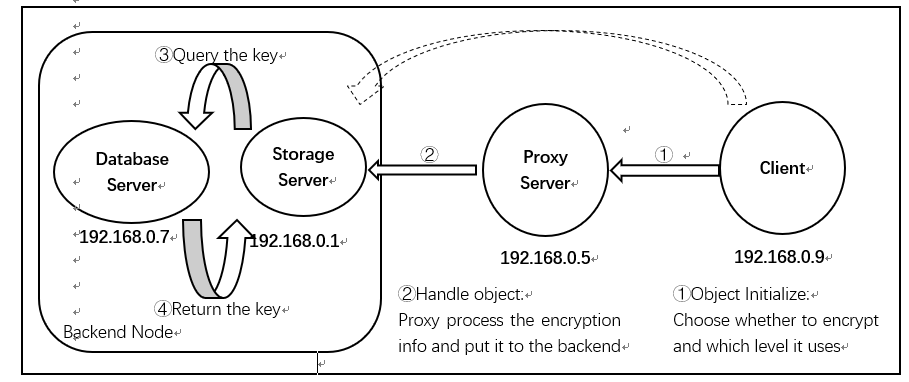


Fig 5 : Overview of the deployment architecture

The test is successful on cloudlab, which means we can let the storage server now will get more relief regarding encryption work. The keys can now be stored in database server and this server won’t do other work so the efficiency will be higher. We can use some measures to secure the database server such as firewall settings and other encryption on database itself.

## Further

1 Deduplication

We can consider more for deduplication. Like to same-level identical files, we only have to store one, to different level we save multiple copies for users asking for high-secure storage.

A possible method for deduplication is that we could encrypt files by session key. And use different keys to encrypt the session key. So duplication of files changes to duplication of keys, which saves a lot space for storage. But I haven’t figure out a way to decrypt the session key quickly, maybe group signature can be a solution.

2 Client encryption system

This system can cooperate with any client-end’s encryption system to provide additional security when transporting and privacy. But to cooperate with deduplication, it needs to carry the etag of file before sending to proxy. Considering using etag of md5(which only has 32 bits for a file), this method is a practical one.

## Conclusion

Our encryption system adds a database layer in cloud-storage system. So we could make it more flexible for server to manage and secure the keys. And for client now, they could choose the encryption level themselves. Besides, this system while provide more support for further deduplication work and client encryption system deployment. It shows this system can be practical for some cloud-storage system provider regarding both efficiency and security.