Accessing Encrypted Documents using Secure Indices and Openid Connect in Cloud Storage

Chethana H R
MTech Student, Computer Science & Engg
Dr Ambedkar Institute of Technology
Bangalore, India

Suresha D
Assistant Professor, Computer Science & Engg
Dr Ambedkar Institute of Technology
Bangalore, India

Abstract— **Cloud computing is a compilation of existing techniques and technologies, packaged within a new infrastructure paradigm that offers improved scalability, elasticity, business agility, faster startup time, reduced management costs, and just-in-time availability of resources. Cloud storage is one of the fastest growing areas under cloud services. Along with many advantages it also imposes serious security issues. Unless dealt with security issues properly, we cannot utilize the best of cloud storage. In this paper, we propose the use of OpenId Connect which has OAuth underpinned along with secure indices which is used for efficient search. OpenId Connect provides an extra layer of authentication along with OAuth functionalities.**

Keywords— cloud storage, Dropbox, encryption, indices, opened connect

I. INTRODUCTION

Cloud storage is one of the most rapidly growing cloud services. Today, there are many cloud-storage services, e.g., Google Drive, Amazon Simple Storage Service (S3), Dropbox etc. With such a service, we can retrieve or revise the latest versions of our files over the internet. Some of us may use it as a networked storage, with which we can synchronize our files across different computers at school, work, or home. For example, S3 is part of the Amazon Web Services, which Amazon claims to provide highly scalable, reliable, secure, and inexpensive infrastructures to web application developers and operators. Amazon stores data across several large-scale data centers. Amazon provides strong physical and logical security to these data centers. Despite such great security measures as advertised by Amazon, cloud-storage services still face serious security challenges[1]. There is an increasing tendency that these cloud storage servers suffer targeted attacks because they hold users' private information, some of which might be quite valuable.

As a result, recently several cloud-storage service providers started to provide encryption protection to user files in the cloud. For example, Dropbox not only stores user files in encrypted form but also employs strict access control to limit read permission to authorized users [4]. More specifically, Dropbox claims to encrypt user files using AES-256, the encryption standard used by many banks to secure customer data. However, encryption of user files is performed after the files are uploaded, and it is Dropbox, as opposed to the user, who manages the encryption keys. In this case, one might question the security, as dishonest insiders and intruders who have compromised Dropbox's system may still be able to obtain the files in plaintext without their owners' permission. Even if users are willing to trust service providers like Dropbox, encryption does impose significant limits on data use. For example, it is difficult, if not impossible, to perform keyword search in encrypted files, and as a result, users can no longer easily ask for files or documents containing a set of specific keywords.

In "Design and Implementation of Multi-User Secure Indices for encrypted cloud storage"[5], author has investigated on the problem of *multi-user* secure indices for encrypted cloud storage. In such a system, files are encrypted in the cloud, and yet users with proper authorization can obtain those files containing some user provided keywords. An interesting real-world application of such a system is the web service of a job bank, in which the storage server stores its resume database. In this scenario, users fill in their resumes online before they can get job matches from the job bank. The companies (queriers) who want to hire some of these potential employees can specify some general conditions not pertaining to sensitive information of the applicants. If a company is interested in some specific candidates, it sends requests to retrieve the private information, and the candidates can decide whether the company can view their resume or not. Author has concentrated on architecture for achieving multi-user secure indices for encrypted cloud storage. Second, in addition to secure indexing, it also provides efficient search among a large number of user documents.

In this paper, we have proposed an extension to the above mentioned work. Author in [5], has made use of OAuth Protocol. Here we are replacing OAuth with OpenId Connect. OpenId connect is built on the top of OAuth and hence it's more robust. OAuth is the protocol which is used only for the authorization and open id connect is very similar to OAuth but it combines the feature of OAuth also. OpenId connect helps to extract basic profile information of querier, which can be sent to Owner(uploader) for verification, before granting the access to the querier.

II. RELATED WORK

In [5], author has provided architecture for achieving multi-user secure indices for encrypted cloud storage. In addition to secure indexing, author has also provided design for efficient search among a large number of user documents. The architecture consists of three kinds of servers, namely, proxy server, index server, and storage server, each of which manages a unique aspect of the system. According to the author, an attacker will need to compromise all three servers at

IJERT

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

the same time can he or she obtain the stored files in plaintext form.

There are 4 system components:

1. Client : There are 2 types of clients: Uploader and Querier. Uploader is one who uploads the files and he is the owner of the file that he uploads. Querier is one who requests for the file.

2. Proxy Server : when files are uploaded, the proxy server parses the files and hashes important keywords. The hashed keywords are then sent to the index servers for building secure indices. Each uploaded file is given an identifier and should be encrypted before transmitted to the storage server. The encryption could be done either by the uploader or the proxy server with a randomly generated key. If the files are encrypted by the proxy server, it can either send back the encryption key to the uploader or encrypt it using the uploader's master key derived from his or her username and password before storing the encrypted key on the proxy server. The proxy server is implemented in a way that both the encryption key and the uploader's master key would be deleted right after the uploading procedure. Therefore, even if the proxy server is compromised, this mechanism limits the damage to those users who are online.

For the query procedure, the proxy server hashes the keywords and sends the hashed values to the index servers to do the search. The search results are then used to retrieve the files containing keywords from the storage server. Before the querier can actually read the searched files, the proxy server asks the file owners (uploaders) for access grants.

3. Index server: The index server builds the indices that allow efficient keyword search. The indices are based on hashed values so the index servers do not have information about the original keywords. The separation of the keys and the indices as well as the long hashed keywords make it a lot more difficult to gain any information from solely compromising the index server. Furthermore, divided hashed keywords are sent to different index servers separately in order to minimize leakage of distribution information.

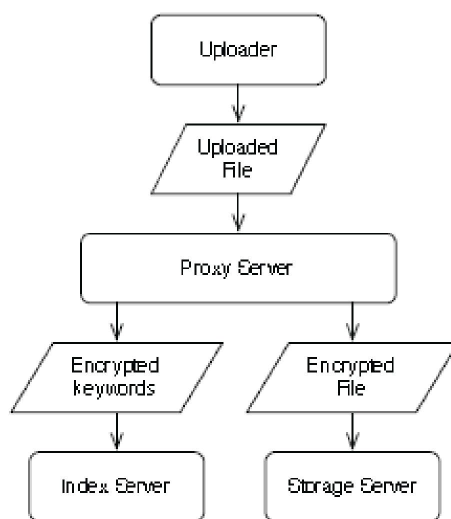

Fig 1 : Input and Output of the system components in the workflows of uploading File

4. Storage server: The storage server provides backing store that actually store the encrypted files. In addition, it also records metadata such as file names and last modification timestamps.

Fig. 1 & 2 illustrates the inputs to and outputs from the three types of servers during the procedures of uploading files and querying..

OAuth open standard for authorizing has been used to provide secure search over encrypted cloud storage. OAuth provides a method for clients to access server resources on behalf of a resource owner. It also provides a process for end users to authorize third-party access to their server resources without sharing their credentials (typically usernames and passwords) using user-agent redirections.

There are four roles in the open authorization standard.

- A resource owner grants client access to protected resources.
- A client wants to access the protected resources.
- An authorization server responds to the client requests by granting an access token if the client is authorized.
- A resource server gets an access token from the client and sends the corresponding protected resources to the client.

As shown in Fig. 3, it is straightforward to fit the scenario into OAuth 2.0, namely, the uploader acting as resource owner, querier as client, proxy server as authorization server, index server as proxy server's back-end, and storage server as resource server. Via the procedure of OAuth, the querier can access the uploaded files on behalf of the uploaders as long as he or she gets the authorization grants from them.

There are two major tasks executed by the proxy server, one for the uploaders, the other for the queriers. Steps followed by the proxy server for the uploaders are:

1) After receiving an uploaded file, the proxy server generates a random key for encrypting the file.
2) Generate a unique ID for the file.
3) Find specific keys of attributes from the look-up table.
4) Parse the whole file and divide it by the fields word by word, and hash the keywords by the corresponding keys of attributes.

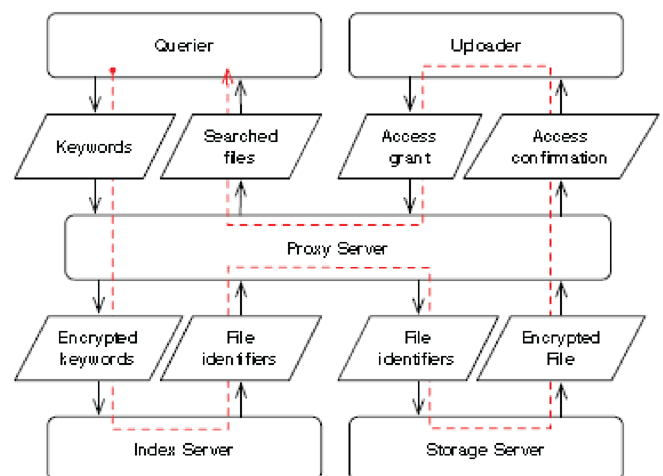

Fig 2 : Input and Output of the system components in the workflows of Querying

(This work is licensed under a Creative Commons Attribution 4.0 International License.)
IJERT

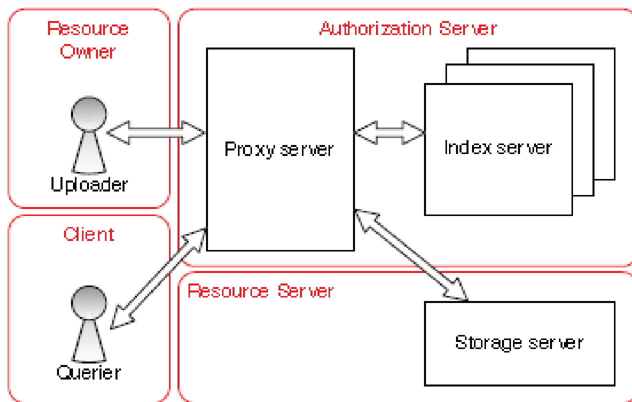

Fig 3 : Roles of various system components in OAuth 2.0

6) Encrypt the original file with the key generated in 1), and record the metadata such as owner's identity, and timestamp, etc.
7) Transmit the encrypted file and its metadata to the storage server.
8) Encrypt the key for encrypting the file with the owner's master key, and store it in the table.
9) Remove all the intermediate information, such as random keys of files and keywords

The steps followed by the proxy server for the queriers are:
1) Find corresponding keys of attributes from the lookup table.
2) Hash the queried keywords with the specific key, and transmit the hashed values to the index server.
3) The index server transmits the search result (file ID) to the proxy server if there are matched files.
4) Send the file ID to the storage server, and ask for the encrypted file and its metadata.
5) Send an e-mail with the querier's information as well as the queried file ID to the file owner.
6) If the file owner allows the querier to access the file, then, the file owner logs in the proxy server to decrypt the file.
7) Encrypt the file with a temporary key, and send a message to the querier with the encrypted file.
8) The querier may read the file when logging in to the proxy server again.

III. ANALYSIS

As an extension to the above work, we can introduce OpenId Connect in place of OAuth. OpenID Connect 1.0 is a simple identity layer on top of the OAuth 2.0 protocol. It allows Clients to verify the identity of the End-User based on the authentication performed by an Authorization Server, as well as to obtain basic profile information about the End-User in an interoperable and REST-like manner.

In general OpenId is all about authentication that is proving that you are who you say you are and OAuth is all about authorization that will allow access to data or functions without requiring to have original authentication once again.

According to the details mentioned in [2], OpenId Connect specifies how identity providers and relying parties can use OAuth 2 0 to communicate identity data to one another. The spec adds identity details and inserts a number of default values without having to fill in the blanks of many OAuth implementation details. This makes OAuth implementation easier comparatively. Also it can be easily portable across different vendors. OpenID Connect Protocol Suite and OAuth underpinnings are as shown in Fig 4.

OpenId Connect obtains profile information through which clients(Uploaders in this case) can verify the identity of the querier based on the authentication performed by an authorization server. This information helps uploaders to grant access to the querier for the requested files. OAuth provides a general method for third party applications to obtain and use limited access to HTTP resources. It does not give any standard methods to provide identity information or any profile information. Without profile information OAuth cannot perform authorization. So, in OAuth, every time, end user need to provide his profile information for getting authorization.

OpenID Connect implements authentication as an extension to the OAuth 2.0 authorization process. Use of this extension is requested by Clients by including the openId scope value in the Authorization Request. Information about the authentication performed is returned in a JSON Web Token (JWT) [JWT] called an ID Token. Using dynamic registration, OpenId Connect obtains profile information needed for openID provider. After receiving all credentials, it performs authentication and returns the result in the form of ID token.

As mentioned in [3], there are 3 entities in OpenId Connect system.
1. User: Acts like resource owner in OAuth
2. OIDC Server: Acts like authorization server and resource server
3. OIDC client: Acts like Relying party(Client in OAuth)

The 3 main extensions of OpenId Connect to OAuth that are helpful in our scenario are as follows :
1. Authn request: Gathers user information
2. ID token objects: Provides information about authentication events like authentication context and time
3. UserInfo: Supplies identity information about authenticated users

The specifications of OpenId Connect are described in [4]. The OpenID Connect protocol, in abstract, follows the following steps.
1. The RP (Client) sends a request to the OpenID Provider (OP).
2.The OP authenticates the End-User and obtains authorization.
3. The OP responds with an ID Token and usually an Access Token.
4. The RP(Relying Party) can send a request with the Access Token to the UserInfo Endpoint.
5. The UserInfo Endpoint returns Claims about the End-User.

IV. CONCLUSION

Since OpenID Connect provides an extra layer of security and makes OAuth implementation easier it can be implemented to various cloud storage servers which are using OAuth at present.

IJERT

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

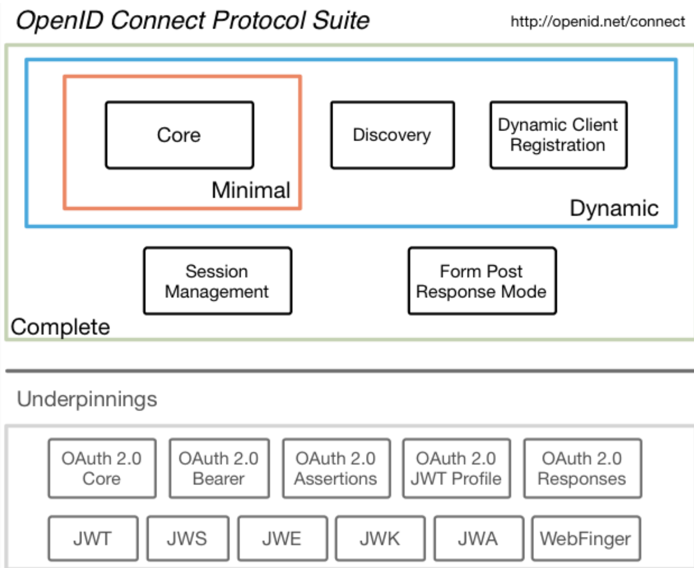

Fig 4 : OpenID Protocol Suite

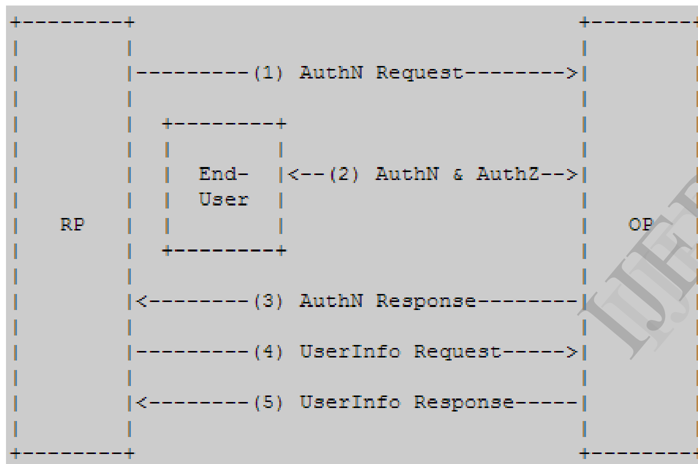

Fig 5 : Steps followed by OpenID Connect

REFERENCES

[1] Amazon, "Amazon S3, cloud computing storage for files, images, videos," http://aws.amazon.com/s3/.
[2] http://windowsitpro.com/identity-management/what-are-oauth-20-and-openid-connect.
[3] http://www.slideshare.net/oliverpfaff/openid-connect-an-emperor-or-just-new-cloths.
[4] Dropbox, "How secure is Dropbox?" https://www.dropbox.com/help/27/en.
[5] Mao-Pang Lin∗, Wei-Chih Hong†, Chih-Hung Chen‡ and Chen-Mou Cheng§ ∗Trend Micro, Taiwan, Design and Implementation of Multi-user Secure Indices for Encrypted Cloud Storage 2013, Eleventh Annual Conference on Privacy, Security and Trust (PST).
[6] http://openid.net/specs/openid-connect-core-1_0.html#CodeFlowSteps.

(This work is licensed under a Creative Commons Attribution 4.0 International License.)