Decentralized Authentication Mechanisms for Object-based Storage Devices [*]

Vishal Kher
Computer Science and Engineering
University of Minnesota
Minneapolis, MN 55455
vkher@cs.umn.edu

Yongdae Kim
Computer Science and Engineering
University of Minnesota
Minneapolis, MN 55455
kyd@cs.umn.edu

Abstract

Network-attached object-based storage separates data-path from control-path and enables direct interaction between clients and the storage devices. Clients interact with the file manager only to acquire the meta-data information and some cryptographic primitives, for example, access keys. Most of the current schemes rely on a centralized file manager to support these activities.

This paper presents security mechanisms for decentralized authentication for object-based storage. The schemes are novel in several ways. First of all, they reduce the load on the file manager and free the system from central point of failure and denial of service attacks. We exploit Role-based Access Control (RBAC) to provide scalability and design authentication schemes that efficiently utilize RBAC. In most of the cases, the client needs to acquire only one access key from the file manager, which can be used by the client to further derive role-keys for the roles that he/she is permitted to play within an organization. Further, the number of cryptographic keys required for the purpose of authentication in these schemes is less as compared to the existing schemes. Finally, we also present two simple schemes that enable the clients to access objects stored on any device on the network using a single identity key.

1. Introduction

Recent advances in network-attached storage have enabled direct interaction between clients and devices. The devices are attached to the client-network, which enables the client to directly access data from these devices, even from remote locations. Clients interact with the file manager only to acquire the meta-data information and some cryptographic primitives, for example, access keys. The client can now directly send a request (for example, read/write) to the device along with the cryptographic primitives acquired from the file manager, which will be used by the device for the purpose of authentication and access control. The file manager is completely bypassed during the data transfer phase, which improves the performance and scalability of the system.

Attaching storage devices to the IP network renders the entire storage system vulnerable to passive attacks such as eavesdropping, traffic analysis and various active attacks such as masquerading, data modification, replaying, and denial of service attacks. The storage devices will have to be active and intelligent enough to authenticate users, restrict access to the objects and their methods, and secure themselves and the objects from potential attacks.

In the existing schemes, a client either acquires a capablitiy key [10, 1, 18] for each object or an identity key [18] from the file manager. Use of identity keys makes revocation difficult whereas, in the prior case, client needs to acquire a large number of keys. The client has to frequently contact the file manager to acquire a key for each object that he wants to access. This imposes a lot of overhead on the file manager, which also presents a single point of failure and an attractive attack target. If the file manager is down or is subject to denial of service (DoS) attack, the entire system can come to a halt. Typically, the file manager is replicated to prevent the failure of one file manager from affecting the entire system. However, replicating the file manager that also acts as an authentication server, simply increases the number of attack points. Further, it also necessitates complex tasks of keeping the security information and policies consistent. For example, the access control lists

* This research is supported in part by Intelligent Storage Consortium at Digital Technology Center (DISC), University of Minnesota.

or keys stored on the replicated file managers should be consistent. Therefore, replicating the file manager is difficult and requires complex management task.

This paper presents security mechanisms for decentralized authentication for object-based storage. Most of the current security mechanisms are credential based. That is, in order to access an object, a client should acquire a credential from the file manager. This can overload the file manager. The main motivation behind the credential-based schemes is that the access control list (ACL) can be maintained by the centralized file manager, which makes modification to the ACLs easier and reduces the complexity on the device. In our schemes, we use *Role-based Access Control (RBAC)* [7]. In RBAC, access decision are based on the "roles" a user[1] plays within an organization. Access permissions are assigned to roles and any user that is a member of a particular role is permitted to perform operations assigned to that role. Changes to role permissions (i.e., to the role-based access control list) are *infrequent* as compared to changes to role memberships. We exploit role-based access control in our schemes and present authentication mechanisms that can be employed in conjunction with RBAC. We attempt to minimize the load on the file manager and reduce the impact of failure of the file manager on the system. If the file manager fails, existing clients can still access objects from OSDs or atleast authenticate themselves to the device. Our authentication mechanisms are based on identity keys. In most of the cases, the client needs to acquire only one identity key from the file manager, which can be used by the client to further derive role-keys for the roles that it is permitted to play within an organization. Further, the number of cryptographic keys required for the purpose of authentication in the presented schemes is less as compared to the current schemes. Finally, in order to enable a client to directly interact with any device in the network by using a single identity key, we present two simple schemes. The first scheme is based on symmetric keys whereas, the second scheme is based on Diffie-Hellman protocol [4].

1.1. Object-based Storage Devices (OSD)

Object-based storage devices present an object level abstraction to the clients. These objects can be viewed as virtual containers that have an internal mapping to their corresponding file blocks. The definition of an object depends on the system or application that uses that object. For example, an object can be viewed as a simple file, a database, a database record or a multimedia object. One object may also contain more than one component objects.

1 "User" and "Client" refer to the end user of the system and can be used interchangeably.

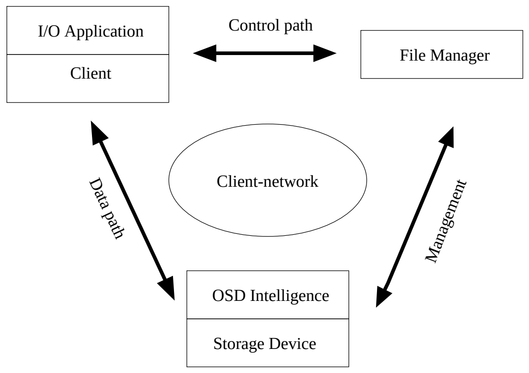

Figure 1. OSD Architecture.

Figure 1 represents the object-based storage architecture. Clients can directly interact with the devices without any intervention of the file manager (indicated by the data path). The OSD intelligence layer is the layer that provides an object level abstraction to the client, performs object specific activities (such as migration, replication), and more importantly performs security related operations such as authentication and access control. In order to grant a client's request for a particular object, a device needs to know whether the client is legitimate and whether he has the required access rights.

1.2. Role-based Access Control (RBAC)

In most of the organizations (such as, commercial organizations, health care etc.), end users do not "own" the information for which they are allowed access. The operations that a user performs are based on the user's role in the organization. In RBAC [7], access decisions are based on the roles that individual users play as part of an organization. Users take on assigned roles (such as doctor, nurse, teller, manager). Access rights are grouped by role name and the use of resources is restricted to individuals authorized to assume the associated role. A role is a set of transactions that a user (or users) can perform within the context of an organization; for example, a University system administrator performs global system management, where as a local system administrator performs departmental management. Associated with each role is a set of transactions performed by that role. Thus, role can be considered as a collection of users and *collection of permissions*. An object in OSD can be viewed as complex objects such as, database records, database etc. These objects can have complex operations or methods. RBAC supports access control at the granularity of such complex methods and is good for object-oriented technology.

RBAC follows the principal of least privilege, where a user is allowed to gain information depending on his job function. The NIST study [8] indicates that permission of roles do not change as frequently as the membership of the role. Under RBAC, roles can have overlapping responsibilities and privileges, which can be represented by forming a hierarchy of roles. RBAC is easy to manage and a natural way of access control in an enterprise and hospital environment. Role definitions, permission assigned to roles can vary from one organization to another. A thorough analysis of job functions and policy decisions are required in order to setup a RBAC framework. Once the basic RBAC framework is setup, the main administrative task is granting and revoking a user's role. Recently, a standard for RBAC was proposed by NIST [6].

1.3. Types of Access Keys

In order to access an object, the client first sends a request to acquire an access key to the file manager. The file manger authenticates the client and returns an access key to the client. The access key is derived from a shared secret between the file manager and the device.

- **Capability Keys**: A capability key indicates the capabilities (access rights) of the client over a particular object. For example, a capability key can be generated as $capKey = MAC_K(accessrights, O, expiry)$ where K is the shared secret between the file manager and the device that stores object O and MAC is a secure message authentication code such as HMAC [2]. $expiry$ indicates the duration for which this key is considered to be valid. The client can then present this capability key to the device to authenticate himself. In order to authenticate the client, the device generates the $capKey$ (since it knows K) and verifies whether client's $capKey$ is authentic. If so, it grants the requested operation if the particular operation is listed in the $accessrights$ argument. An advantage of this scheme is that the device is unaware of the client's identity and his access rights. Usually the $capKey$ is valid for a short interval of time. A downside of this scheme is that the client has to acquire a key for each object he wants to access, which incurs a lot of overhead on the file manager.
- **Identity Keys**: An identity key allows the device to verify the identity of a particular client. For example, an identity key can be generated as $idKey = MAC_K(identity, expiry)$, where K is the shared secret between the file manager and the device. In this case, the device has to store an access control list (ACL) along with each object. The device first verifies the identity of the client by verifying the $idKey$ and then verifies whether the client has the requested rights listed in the ACL of the object. If these tests succeed, the device grants the request.
- **Role Keys**: Existing schemes are either based on capability key or identity key. We introduce a third type of key, role key, which is used in our system for authentication purpose. A role key allows the device to verify the role of the beholder. For example, a role key can be generated as $roleKey = MAC_K(role, expiry)$ where K is the shared secret between file manager and the device. Role key can also be derived by using the identity key of the client. For example, the identity key of the client can be generated as: $idKey = MAC_K(id, H(roles), expiry)$, where $H(roles)$ indicates a hash of concatenation of all the roles a particular client can play. The client can then generate a role key using this identity key. The client then presents this role key to the device. To verify the role key of the client, the device first generates the identity key of that client. Using this identity key the device can then generate a role key to verify whether the client is allowed to play that role. In this case, role key corroborates the identity as well as the role of the client.

1.4. Organization

The remainder of this paper is organized as follows. Section 2 describes related work. Section 3 describes the assumptions and goals of our system. Section 4 gives an overview of our protocol. The details of these protocols are presented in section 5. Section 6 explains possible revocation mechanisms. Section 7 presents two optimizations that further help to reduce the total number of keys in our system. Section 8 presents a preliminary comparison with other capability based and identity based systems. Finally, section 9 draws conclusions and outlines future work.

2. Related Work

The main goal of distributed file systems is to allow users to access their data from remote locations in the same way they would access local files, by making the network transparent. The focus of initial research of these systems was on availability, performance, and other distributed systems issues rather than security. However, due to the popularity of the Internet and due to the enormous amount of electronic data transferred on the network and stored on these file servers, securing this data and the entire system from abuse has become an important requirement.

AFS [11, 20] was one of the first file systems to take security into consideration. It enables co-operating hosts (clients and severs) to efficiently share file system resources across both local and wide-area networks. Authentication in AFS is done using Kerberos [16]. NFS has enhanced its security recently in NFS V4 [17], which provides authentication, integrity and privacy of the data on the network. CFS [3] was one of the first file system that pushed the file encryption services into the file system. This user level virtual file system performs the file encryption and key management functions. TCFS [14] and cryptFS [5] further extended CFS by making file encryption transparent to the user. Smart-Card-based secure file system [13, 12] supports end-end data encryption as well as decentralized access control.

NASD [9, 10] was one of the first systems that enabled clients to directly access the objects stored on the storage devices by separating the data path from the control path; thus, improving the performance of the system. The clients interact with the file manager to obtain meta-data information and cryptographic capabilities that apart from other things include the access rights of the client for that object, access control version, and capability key for the client. After acquiring this information, the clients can directly store/retrieve the objects from the storage devices. Devices can authenticate the clients on the basis of the capabilities passed to the clients by the file manager. These capabilities are based on a common key shared between the file manager and the storage device that stores the requested object. The client obtains a capability for each object; therefore, the file manager has to be online and presents an attractive attack target and a central point of failure. If the file manager is down the entire system comes to a halt. Further, the capability is bound to the device and the object; therefore, if the object is replicated (or migrated) to a different device or is a compound object with component objects stored on different devices, clients will have to acquire one or more fresh capability keys. The access control version of each object is stored along with the object. By changing the access control version in the capability of the client, the file manager can instantaneously revoke the client. However, this revokes the capability key of all the clients accessing that object. NASD devices are unaware of the file system structure and simply store the objects and enforce the policy decisions made by the file manager. The capability keys need to be transmitted through secure channel (possibly using SSL or IPSec). Opening new connection using these techniques enables another type of denial-of-service attack, since the file manager has to be involved with public key operations. SNAD [15] extends NASD to provide end-end data encryption.

SCARED [19] extends NASD to provide mutual authentication between clients and storage devices. It supports authentication based on capability key (as in NASD) as well as identity keys. In the latter case the storage device also stores the access control list (ACL) along with the object. These keys can be long lived as compared to NASD, which makes revocation difficult. In the case of identity keys, each client needs to acquire keys equal to the number of devices; therefore, the total number of keys in the system is equal to the number of clients times the number of devices as compared to the number of objects times the number of clients in the case of NASD.

Another extension to the NASD scheme is presented in [1]. This protocol separates mechanisms used for transport security from those used for access control. It requires that the underlying transport layer is using IPSec. The protocol is then built over this secure communication layer. As in NASD, in order to access an object, the client has to acquire capability key from the file manager. The protocol is session based; client establishes a connection with the storage device, and the device delivers a channel ID to the client. This channel ID along with access credentials is used to setup an authenticated channel between the client and the device. This prevents information from one channel to be replayed onto another channel. Caching the verified credential for each session and using the cached copies for future authentication can increase the speed of the verification process. This scheme faces the same problems as in NASD mentioned above.

3. System Assumptions and Goals

In this section we describe our system assumptions, goals, and what kind of security threats will be faced by our system. The schemes presented in this paper are developed with these assumptions in mind.

3.1. Assumptions

File Manager

File manager is a trusted entity. It is responsible to securely setup the system, necessary security parameters, and store the keys or other security parameters securely. It knows the legitimate users of the system and has a secure way to authenticate these users. It also has a secure way to communicate with the clients. Should a file manager be compromised, we term this as a total break, meaning the entire system is compromised.

Clients
Clients or end users are not trusted. Clients are capable of performing all kinds of active and passive attacks. Even a legitimate (rogue) client or an insider can try to break security schemes. For example, he can attempt to impersonate other legitimate clients, attempt to collaborate with other clients to break the security schemes, or attempt to perform illegitimate access.

Communication Links
The communication links are completely insecure. We do not assume any kind of underlying secure protocols such as IPSec. Since the communication links are insecure, an adversary or even a rogue insider can perform active/ passive attacks such as eavesdropping, masquerading, inserting and modifying data, etc.

Devices
Devices are trusted to grant access to legitimate users of the system and play their role appropriately and securely.

3.2. Goals

1. Remove central point of failure and distribute security functions appropriately between the file manager and the devices.
2. Reduce total number of access keys required in the system.
3. Minimize the number of interactions between the client and the file manager. The client need not contact the file manager to access each object at least for security purposes. This will reduce the overhead of the file manager. The file manager can remain offline for most of the time.
4. Provide client to device mutual authentication.
5. The protocols should be secure in the face of various network attacks.
6. Minimize the performance overhead imposed by the cryptographic operations.
7. Improve scalability of the storage system to support frequent replication and migration of objects.

4. System Overview

We follow the basic OSD model and build our system on this model. We use RBAC in our system. We assume that a RBAC framework already exists. The policies on which this framework is built is outside the scope of this paper. We also assume that decisions regarding various roles and role permissions are already made. These decisions can vary from one organization to another.

In our approach, the file manager performs user to role association. It maintains a database of all possible roles within an organization and their corresponding members. By maintaining this database at the centralized entity, frequent changes to the database can be easily handled. A device stores role-based access control list along with each object. The reader should recall that changes to role-based access control list are infrequent as compared to changes to UNIX style ACLs.

The file manager shares a secret key K with each device. In order, to access an object, a client first sends a message to the file manager requesting an identity key. The file manager authenticates the client, and acquires all the roles that particular client can play within the organization. This information is fetched from the database maintained at the file manager. The file manager then generates an identity key for the client as follows:

$$idKey_c = MAC_K(I_c, H(roleList))$$

$roleList$ is concatenation of all the roles the client can play within the organization and I_c is the identity of the client. This identity key is then transferred securely to the client along with the $roleList$. This ensures that no malicious entity can acquire the client's identity key. Further, the identity key is derived from the secret shared between the file manager and the device; therefore, only that device and the file manager can regenerate and verify the client's identity key. After receiving this identity key the client can directly access objects stored on the device. The client first generates a role key depending on the current role played by the client. By allowing the client to generate his own role key, the client does not have to contact the file manager in order to acquire a key for each role. The role key is derived from $idKey_c$ as follows:

$$roleKey_c = MAC_{idKey}(currentRole)$$

In order to access an object stored on the device, the client sends a message to the device, which among other things includes I_c, $roleList$, $currentRole$. Along with the message, the client also sends a $MAC_{roleKey_c}$ on this message using the role key corresponding to $currentRole$. The device can then generate $idKey_c$ in a similar way as the file manager. Using the $currentRole$

parameter the device can generate role key and verify the authenticity of the message sent by the client. The device can also verify whether the client is eligible to play that role by checking the $roleList$. Finally, the device verifies whether the role-based access control list of the requested object permits the requested operation for $currentRole$. If all tests succeed, the device grants the client's request and sends an appropriate response to the client. Including $H(roleList)$ while deriving $idKey_c$ ensures that the client cannot play any role other than those specified in $roleList$. Attaching MAC with every request ensures authenticity of the origin as well as integrity of the message. To prevent replay attacks, we also include freshness information along with each request and response. Note that, if the client is granted new roles different from those mentioned in the $roleList$, the client will have to acquire a new identity key. However, the frequency of acquiring new identity keys will be less than that of capability keys.

5. Protocol Details

Notations used for the protocol description are as follows:

C	Client
D	Device
FM	File manager
I_c	Unique identity string of C
$idKey_c$	C's identity key assigned by the fi le manager
$roleKey_c$	C's role key corresponding to its current role
K	Secret key shared between FM and D
MAC_{k_i}	Keyed-MAC function using key k_i
H	Collision resistant hash function

Identity Key Generation
A client can acquire his identity key from the file manager on the first access to an object or an identity key can be transferred to the client when his account is created. The protocol messages are shown below.

$C \rightarrow FM$: identity key request (1)
$FM \rightarrow C$:
$idKey_c = MAC_K(M, H(roleList)), M, roleList$ (2)
Where $M = I_c, expiry$

On receipt of message (1), file manager authenticates the client and verifies that the client is a legitimate user of the system. The authentication mechanism can be one of the standard mechanisms and is out of scope of this paper. Using key K, the file manager creates an identity key for the client as shown in (2) above, which is transferred securely to client C. The field $expiry$ indicates the expiry of the identity key. $roleList$ indicates all the roles that the client is eligible to play, for example, programmer, manager, designer, etc. During creation of the identity key, the file manager includes a hash of all the roles that the client can play. If a client has multiple roles, the file manager can compute hash on concatenation of all the roles.

Role Key Generation
Before sending a request, the client generates a role key as $roleKey_c = MAC_{idKey_c}(currentRole)$. $currentRole$ indicates the current role that the client is using to access an object. For example, a client can have two roles A, and B. The $roleList$ in step (2) above is $A||B$. However, a client can only play certain roles at time (such decision are part of policy decisions of the client's company). If a client is allowed to play only one role at a time, then $currentRole$ is either A or B.

Freshness Guarantee
Since the communication links are insecure, an attacker or rogue insider can replay previous messages sent to and from legitimate clients and attempt to impersonate legitimate clients. In order to prevent replay attacks its important to guarantee the freshness of each message sent to and from the client. The freshness guarantee in our protocol is similar to that used in SCARED [18].
Protocol messages are shown below:

$C \rightarrow D$:
$M_1 =$
$\{NonceRequest, r, M\}, MAC_{roleKey_c}(M_1)$ (3)
Where $M = I_c, roleList, currentRole, expiry$

$D \rightarrow C$:
$M_1 = \{NonceReply, r, s\}, MAC_{roleKey_c}(M_1)$ (4)

On receipt of request (3), the device verifies whether $currentRole$ is listed in $roleList$. Next, the storage device can generate $idKey_c$ by using M and performing MAC_K(M). The device then generates $roleKey_c$ in a similar way used by the client. Using this $roleKey_c$ the device can verify the authenticity of the message. If the test succeeds, the device can conclude that the client is authentic and the request was generated by the client. The device then sends back a response as shown in step (4). Similarly, on receipt of message (4), the client can verify whether the message was generated by the the device by verifying the MAC. The value r in the response is used by the client to match the response with the original request. Communication can be session based or timer based. Fields r and s indicate these freshness parameters and are used to guarantee freshness of future communications. If the communication between the client and the device is session oriented, then the client and device can exchange the initial session counter using the above two messages. Once the initial counter is established, the counter can be incremented for successive messages for that session. If the communication is timestamp based, it is assumed that the clocks of the device and the client are synchronized; therefore, no communication is required. If their clocks are not synchronized they can exchange messages (3) and (4) as shown above and synchronize their clocks periodically. Clients can save one round of communication by merging freshness protocol along with the request protocol.

Request/Response Protocol
The request/response protocol is shown below:

$C \rightarrow D$: $M, MAC_{roleKey_c}(M)$ (5)
where $M =$
$\{Operation, Oid, I_c, expiry, r, s, currentRole,$
$roleList\}$

$D \rightarrow C$: $M =$
$\{Response, r, s\}, MAC_{roleKey_c}(M)$ (6)

In order to authenticate the client, the device checks whether $currentRole$ is included in $roleList$. It then calculates $H(roleList)$ ($roleList$ can be obtained from message M). It can then generate $IdKey_c = MAC_K(I_c, expiry, H(roleList))$, and $roleKey_c = MAC_{idKey_c}(currentRole)$. The device can now verify whether MAC generated by the device matches with the MAC sent in the message. By using r and s in the message the device can verify the freshness of the message. If the verification is successful, the device checks whether the role that the client wants to play has the required access rights by checking the role-based access control list of the requested object. If all the tests succeed, access is granted to the client. The client can verify the response from the device in a similar fashion by verifying the MAC of the response.

6. Revocation

Revocation can be achieved in following ways:

1. Each identity key has an associated key expiry information. The device needs to check whether an identity key is expired on every request. The expiration time might be specified as a timer or in terms of number of accesses a client can perform.
2. The system can maintain a revocation list, which can be stored on fast LDAP servers. Each device can periodically download (or check) the revocation list (for example, once a day) or the file manager can push the revocation list to OSDs. The old entries in the list can then be discarded, which will keep the revocation list small. The revocation list can maintain information regarding revoked clients, client's individual roles, and a particular role from the organization.
3. Although changes to the role-based access control list are infrequent, the changes can still occur. In order to speed the look-up process, each device can maintain a list per role that contains pointers to all the objects accessed by that role. In order to find the location of an object accessed by a particular role, the list can be indexed by the identity of the object. However, there are two disadvantages of this scheme. First, the size of the list maintained for each role can be large. Second, this list should be updated every time an object is added or deleted by a role.
4. In order to revoke a client's individual role, the file manager can assign an expiry for each role that was included in the $roleList$ used while deriving identity key for a client. The expiry field can be a part of the $roleList$. The device can verify whether the role-key has expired by checking the expiry field corresponding to that role. Note that in this case, the client can still use his identity key to generate role key for other roles.

7. Optimizations

Identity keys are derived from a secret key shared between the file manager and the device. Therefore, the identity key of a client is bound to each device. In order to access objects from multiple devices, client has to acquire multiple identity keys. Therefore, the client still has to contact the file manager to acquire all the identity keys. To reduce the number of client to file manager interactions, these identity keys can be batched and transferred to the client at once. We can further reduce the number of identity keys that a client needs to acquire by using two simple approaches, namely, by grouping the clients or devices, and using Diffie-Hellman protocol.

7.1. Grouping

Devices can be grouped together and the file manager can assign one key to each group. All devices within one group share the group key amongst themselves and with the file manager. The identity key of clients is derived from this group key. Since, the devices share a common secret, client's can directly interact with any device within this group by just using one identity key. Thus, the number of identity keys per client depends on the number of groups. If there are N groups of devices, then a client will have to acquire at the most N identity keys to interact with each device in these groups.

Another approach is to group clients. File manager creates N groups and assigns each legitimate client to one of the groups. The client to group assignment can be random or can be based on other factors such as, the client's role. A group can be a subdivision within an organization, for example, storage division. The identity keys for all members of one group are derived by the file manager using the group key of that group. That is, instead of using the secret key shared with the device, file manager will use the group key of a client's group to generate an identity key for the client. If file manger wants to give access to the members of a particular group to a particular device, the file manager can simply give the group key of that group to the device. Since, this group key is used to generate the identity key of the clients belonging to that group, the device can verify the identity keys as well of roles keys of all the members of that group. By storing or deleting a particular group's key on the device, file manager can enable or disable group's access to the device. Thus, the file manager can impose another level of access control. Members of a group can interact with a device if that device has the group key of that group.

7.2. Diffie-Hellman Based Authentication (DHA)

DHA uses Diffie-Hellman protocol [4] to share a key between a client and a device. Diffie-Hellman protocol is used to generate the initial long term common secret between these two entities. After the initial setup the long term secret will be used as a symmetric key along with keyed-MAC. The advantage of this scheme is that clients can share a secret directly with a device without any intervention of the file manager. This reduces the load on the file manager and also provides mutual authentication between the device and the client. This protocol does not require any private channel between the client and the file manager.

7.2.1. DHA: DHA is Diffie-Hellman protocol with authentic public keys. These public keys are signed by a certification authority or the file manager. In order to verify the authenticity of a sender's public key, a verifier will have to verify the certificate issued by a certification authority, which could be a separate entity or file manager. If both the sender and the receiver possess an authentic certificate, then a secret key is established with the Diffie-Hellman protocol.

- One time setup: File manager (or any other certification authority) selects an appropriate cyclic group G with generator g. Both G and g are system wide public parameters. It then issues each legitimate entity (client and device) a certificate that binds the entities public key g^a to its identity. Using A's certificate B can verify whether g^a is A's authentic public key. Let $Cert_A$ denote certificate issued by the file manager to entity A. For each client, the certificate will also contain all the roles of that client.

- KeySetup: The following depicts the interaction between the client C and device D to establish a common secret key K_c.

  $C \rightarrow D$: $M = \{KeySetup, g^c, Cert_c, I_c\}$
  $D \rightarrow C$: $M = \{SetupResponse, g^d, Cert_d, I_d\}$

  I_c and I_d denote identity of client and the device respectively. C and D verify the authenticity of each others public key and calculate $(g^d)^c$, $(g^c)^d$ respectively to establish a shared secret K_{cd}=g^{cd}.

- Request Protocol: The client can generate the role key using K_{cd} as explained before. The request protocol is similar to that explained in section 5.

7.2.2. Discussion: DHA allows a legitimate client to directly establish a secret key with a device. The key generation and certificate verification process is expensive as compared to symmetric key based approaches. However, if the public keys and the certificates can be made available to each client (or even to each device) apriori, then key setup phase is not required and each entity can calculate the shared secret offline. Further, after successful verification, devices can cache the shared secret keys K_{cd} along with the client roles and use the cached keys to verify future requests. The device does not have to perform signature verification on each phase, which will speed up the request protocol. Thus, by making the public keys available apriori and caching the secret keys one can reduce the number of messages sent to the device as well as the computational overhead incurred due to key generation and signature verification operations. DHA provides mutual authentication as a client directly shares a key with a device. The client can establish a secret (if not established before) with the any device and access the object without any intervention from the file manager, which improves the scalability of the system. DHA also minimizes the load on the file manger as, the file manager can remain offline after initial system setup. In fact, the file manager does not have to perform any key management activities. Key management can be separated from the file manager by using another entity or a certification authority that performs key management functions.

As each entity requires to store only one secret key a corresponding to its public g^a the number of keys required is equal to the number of clients plus the number of devices.

8. Comparison

A preliminary comparison of our schemes with other capability based systems and identity based system is presented. As compared to capability based systems, our system requires less number of keys. Therefore, client to file manager interaction is reduced and the computational overhead on the file manager is reduced. Our schemes are based on RBAC, which is a natural way of making access decision within an organization. After setup of the RBAC framework, the changes to permissions assigned to roles are infrequent as compared to UNIX style ACLs. Further, by using grouping or DHA we can reduce the total number of keys required in the system. By using DHA we can reduce the number of keys to the number of clients plus the number of devices as compared to other identity-based systems where the number of keys required is equal to the number clients times the number of devices. Finally, after the initial setup the file manager can be offline for security purposes. In fact, file manager does not have to perform any key management operations.

9. Conclusions and Future Work

In our approach, we have attempted to reduce the computation overhead on the file manager. Even if the file manager is down the existing clients of the system can still interact with the devices. RBAC is easy to manage and a natural way of access control for commercial organizations. RBAC is highly flexible and can satisfy various access control requirements. We presented authentication mechanisms that utilize RBAC. In our system, operations on the object can be fairly complex, for example, insert record, deposit money etc. The authentication protocol is robust against networks attacks and provides client to device mutual authentication. The basic protocol coupled with grouping or DHA can further reduce the total number of keys in the system. By using DHA the total number of keys is number of clients plus number of devices. After the initial setup the file manager can be completely offline. The computation overhead imposed by DHA can be minimized by precomputing the shared keys and caching the DHA shared keys on the client and the device.

We plan to further explore role-based access control models to understand how role-based access control lists can be efficiently designed and implemented. We plan to analyze access patterns to understand how frequently the role-based access control list is changed and frequency of role revocation. A potential optimization to schemes presented in this paper is to store role-based access control list in a single object. This list will include all roles and their associated permissions. This list can be maintained by the file manager. By keeping the list centralized, changes to the list can be done efficiently. Along with each object, we can store the list of roles that can access the object. We can then use a combination of identity key and capability key for authorization purpose. We plan to explore authentication schemes further in this setting and implement these schemes on top of the Lustre file system.

References

[1] Alain Azagury, Ran Canetti, Michael Factor, Shai Halevi, Ealan Henis, Dalit Naor, Noam Rinetzky, Ohad Rodeh, and Julian Satran. A two layered approach for securing an object store network. In *Proceedings of the First IEEE International Security In Storage Workshop*, December 2002.

[2] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash function for message authentication. Lecture Notes in Computer Science, 1996.

[3] Matt Blaze. A cryptographic fi le system for unix. In *Proceedings of the 1st ACM Conference on Communications and Computing Security*, pages 9–16, Fairfax, VA, 1993. ACM Press.

[4] W. Diffi e and M. E. Hellman. New directions in cryptography. *IEEE Trans. Inform. Theory*, IT-22:644–654, November 1976.

[5] I. Badulescu E. Zadok and A. Shender. Cryptfs: A stackable vnode level encryption fi le system. Technical Report CUCS-021-98, Columbia Univ., New York City, NY, 1998.

[6] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST standard for role-based access control. ACM Transactions on Information and System, August 2001.

[7] David F. Ferraiolo and D. Richard Kuhn. Role-based access controls. In *15th NIST-NSA National Computer Security Conference, Baltimore, Maryland*, October 1992.

[8] D.F. Ferraiolo, D.M. Gilbert, and N. Lynch. An examination of federal and commercial access control policy needs. In *NIST-NCSC National Computer Security Conference*, 1993.

[9] Garth Gibson, David Nagle, Khalil Amiri, Fay Chang, Eugene Feinberg, Howard Gobioff, Chen Lee, Berend Ozceri, Erik Riedel, David Rochberg, and Jim Zelenk. File server scaling with network-attached secure disk. In *Proceedings of the ACM International Conference on*

Measurement and Modeling of Computer Systems (SIGMETRICS '97), June 1997.

[10] H. Gobioff. *Security for high performance commodity subsystem*. PhD thesis, CMU, July 1999.

[11] J.H Howard. An overview of the andrew file system. In *Proceedings of the USENIX Winter Technical Conference*, Dallas, TX, February 1998.

[12] J. Hughes and C. Feist. Architecture of the secure file system. In *Eighteenth IEEE Symposium on Mass Storage Systems*, pages 277–290, San Diego, CA, April 2001.

[13] J. Hughes, M. O'Keefe, C. Feist, S. Hawkinson, J. Perrault, and D. Corcoran. A universal access, smart-card-based, secure filesystem. In *Atlanta Linux Showcase*, October 1999.

[14] Ermelindo Mauriello. TCFS: Transparent cryptographic filesystem. *Linux Journal*, 40, August 1997.

[15] E. Miller, D. Long, W. Freeman, and B. Reed. Strong security for distributed file systems. In *Proceedings of the Conference on File and Storage Technologies (FAST 2002)*, pages 1–13, January 2002.

[16] B. Clifford Neumann and Theodore Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications*, 32(9):33–38, September 1994.

[17] Brian Pawlowski, Spencer Shepler, Carl Beame, Brent Callaghan, Michael Eisler, David Noveck, David Robinson, and Robert Thurlow. The NFS version 4 protocol. March 2000.

[18] B. Reed, E. Chron, R. Burns, and D. D. E. Long. Authenticating network attached storage. *IEEE Micro*, 20(1):49–57, January 2000.

[19] Benjamin C. Reed, Mark A. Smith, and Dejan Diklic. Security considerations when designing a distributed file system using object storage devices. In *Proceedings of the First IEEE International Security In Storage Workshop*, December 2002.

[20] M. Satyanarayanan. Integrating security in a large distributed system. In *ACM Transactions on Computer Systems*, volume 7, pages 247–280, 1989.