

The OSD Security Protocol

Michael Factor* David Nagle† Dalit Naor‡ Erik Riedel§ Julian Satran¶

December 2005

Abstract

The ANSI T10 Object-based Storage Devices (OSD) Standard is a new standard. It evolves the storage interface from fixed size blocks to variable size objects and includes an integrated security protocol that protects storage. This paper presents the requirements, the design tradeoffs, and the final security protocol as defined in the standard. The resulting protocol is based on a secure capability-based model, enabling fine-grained access control that protects both entire storage device and individual objects from unauthorized access. The protocol defines three methods of security based on the applications' requirements. Furthermore, the protocol's key management algorithm allows keys to be changed quickly, without disrupting normal operations. Finally, the protocol is currently being enhanced for version 2.0 of the ANSI T10 OSD standard; future extensions will include privacy and access-control on sections of storage objects.

Keywords: Networked Storage, Object Storage, OSD, Capability-based protocol.

1 Introduction

Object-based Storage Devices (OSD) is an emerging storage paradigm that replaces storage's traditional fixed-size block abstraction with variable-size objects that virtualizes the underlying physical storage. An individual object's data is presented as a linear sequence of bytes and is accessed by specifying the object identifier (OID) and an (offset, length) tuple. All mapping from (OID, offset, length) to the physical storage locations is hidden under the object interface

and managed by the storage device itself.

To ensure secure access to storage, every command must be accompanied by a cryptographically secure capability that identifies a specific object and the list of operations that may be performed against the specified object. Capabilities not only provide the per-device security that is lacking in today's block-based storage, but they also facilitate fine-grained secure access to individual objects. This enables storage-device sharing among diverse applications with unique security requirements. This paper presents the requirements, motivation and a description of the object store security protocol as contained in the T10 OSD (Object-based Storage Devices) standard [T10].¹

Object storage was first proposed in the Network-Attached Storage Devices (NASD) project at CMU as [GNA⁺96, GNA⁺97] and is the basis for two commercial products [Pan, Lus]. The security of object stores was treated in [Gob99], [ACF⁺02] and remains an active area of research (for surveys, see [MGR03, FMN⁺05]). The OSD T10 standard [T10], ratified in September 2004, is currently being field-tested in engineering prototypes and interoperability demonstrations [ISE05, NRR⁺05].

The T10 OSD Security working group set the following goals for the OSD Security protocol.

1. The protocol must prevent against attacks on individual objects, including:
 - Intentional and inadvertent non-authorized access
 - Illegal use of credentials beyond their original scope and lifespan
 - Forging or stealing credentials
 - Using malformed credentials
2. Enable protection against network attacks:
 - Man-in-the-middle attacks on the network

*IBM Haifa Research Laboratory, Haifa, Israel.
factor@il.ibm.com

†Panasas, Pittsburgh, Pennsylvania. dnagle@panasas.com

‡IBM Haifa Research Laboratory, Haifa, Israel.
dalit@il.ibm.com

§Seagate Research, Pittsburgh, Pennsylvania.
Erik.Riedel@seagate.com

¶IBM Haifa Research Laboratory, Haifa, Israel.
Julian.Satran@il.ibm.com

¹Throughout the paper the terms *object store* and *OSD* are used interchangeably.

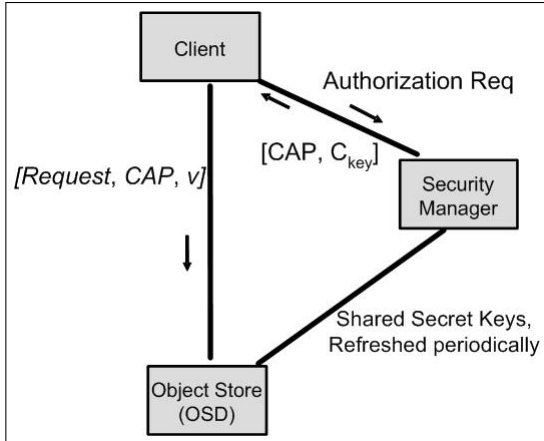


Figure 1: Object Store Security Architecture

- Network errors and malicious message modifications
 - Message replay attacks
3. Defend against other attacks:
 - Prevention of denial of service attacks specific to the protocol
 - Prevention of timing attacks specific to the protocol
 - Protection against ill-behaved clients, in particular with incorrect clocks
 4. Must allow low cost implementation of the critical path for commodity devices.
 5. Should enable efficient implementation on existing network transports by leveraging a secure network infrastructure when one is available to protect against network attacks.

To address these goals and building upon prior work [Gob99], we defined a credential-based access control system composed of three active entities: the object store, a security manager, and a client (see Figure 1). As a capability-based access control system, all requests to the object store must be accompanied by a capability, which encodes a set of rights the holder has on an object, and be cryptographically secured.

Before first access to an object, the client requests a *capability* from the security manager for the desired operation. Assuming the client is authorized to access the object, the security manager returns an appropriate *credential* which includes: 1) the *capability*

that defines the access rights and, 2) the *capability-key* that is based on a secret shared between the security manager and the OSD. On every command, the client sends this *capability* to the object store as part of its request and cryptographically secures the request using the *capability-key*. The object store validates the request, ensuring that the capability has not been tampered with, was rightfully obtained by the client, and that the requested operation is permissible by the capability.

To deliver a complete security system for storage, the standard defines the security protocol between: 1) the OSD and the client and 2) the OSD and the security manager (primarily for key management). Communications between the client and security manager is considered part of the higher-level software system (e.g., NFS, AFS, Oracle), and is therefore outside of the scope of the standard. Furthermore, the protocol does not specify the way the security manager authenticates or authorizes the client.

Storage performance demands that the OSD security protocol support efficient implementations in both secure network environments (e.g., IPsec) and insecure environments. In secure network environments the OSD protocol only needs to provide access control security, *i.e.*, security which is directly tied to the semantics of the object store. Running over insecure networks, the object store protocol itself provides network security as well as access control. To support both environments efficiently, we have defined multiple methods of the protocol in which network security mechanisms are layered on top of access control mechanisms. The decision of which security method(s) to employ is left to the implementation and is chosen from the following methods:

- **NOSEC – No Security.** The same message structure as other methods is used, however, the object store does not verify the authenticity of the capability prior to performing an operation.
- **CAPKEY – Integrity of Capability (Access Control Security).** Access Control Security is the common component to all the methods. CAPKEY by itself is most useful when the channel between the object store and client is externally secured, *e.g.*, an authenticated IPsec channel. CAPKEY provides a mechanism to prevent a client from forging or otherwise modifying a credential or replaying a credential over a different authenticated channel. In addition, it verifies that the client rightfully obtained the credential it is presenting. Without a secure network, using only integrity of capability (the

CAPKEY method) leaves an installation susceptible to a variety of network attacks. It is based on the protocol presented and analyzed in [ACF⁺02].

- **CMDRSP— Integrity of Command and Arguments.** Integrity of command and arguments is most useful when the channel between the object store and the client is not externally secured and where providing integrity for bulk data transfers would be too expensive. CMDRSP adds integrity of arguments to CAPKEY that would otherwise be provided by the underlying secure channel. It prevents a malicious client from being able to replay or modify command parameters, even when running on unsecured networks. It does not prevent network attacks on the data portion of the messages exchanged between the client and the object store *e.g.*, modifying the data in transit.
- **ALLDATA – Integrity of Data (Access Control and Internal End-To-End Security).** This method includes the functionality of CAPKEY and CDMRSP. It provides security similar to CAPKEY when run over an authenticated channel. The exact comparison between the two depends on the level of network security provided. This method internally secures the network as an integral part of the object store protocol, thereby defining an end-to-end solution at the storage layer. It is ideal for an insecure network but provides redundant function when run over a secure network infrastructure.

The security methods are summarized in Table 1. The table shows each method on its own as well as when combined with a network security mechanism (such as IPsec) which provides integrity, data origin authentication, and replay protection. All of these methods use the same basic flow² and the same credential structure.

The rest of this paper is structured as follows. The next section provides background, including a brief general description of object stores as defined by the OSD protocol, the general flow for security and the trust model. Section 3 describes the rationale behind the credential and capability structures. In section 4 we outline the basic mechanisms driving the various security methods. Section 5 discusses some design issues concerning the security manager, and section 6 provides a description of the hierarchy of keys. We

²NOSEC does not require the presence of a security manager.

conclude in section 7 with future enhancements for the next versions of the OSD protocol.

2 Background

2.1 OSD

A T10 OSD [T10] presents a 3-level object hierarchy. At the top of the hierarchy is the *root object*, which represents a single logical unit (LUN in SCSI terminology). Beneath the root object are one or more (logical) partition objects used to manage space and scoping security. At the bottom of the hierarchy are user objects, which hold the actual data and attributes. In addition to user objects, partitions also store collection objects, which are lists of user objects created and maintained by the higher-level software (*e.g.*, file system). The root object is also known as partition zero. By definition, partition zero should not contain user objects; for bootstrapping, an object store must always contain a partition zero.

Objects contain both data, which is byte addressable, and an indexed attribute namespace which is divided into 2^{32} pages of 2^{32} attributes per page. Some attributes are defined by the standard and interpreted and/or maintained by the OSD (*e.g.*, capacity used). Other, application-created attributes, are uninterpreted byte strings stored by the OSD within the associated object's attribute space.

The protocol defines a set of operations which a compliant object store should support. These include creating and deleting partitions and objects, reading/writing data and attributes from objects, and managing the object store. For details on the supported operations, see [T10].

2.2 Security Flow

A client wishing to access an object requests a *capability* from the security manager and must specify the (OSD name, partition ID, object ID) tuple to fully identify the object. The security manager upon receiving this request may need to authenticate the client making the request; such authentication is beyond the scope of the OSD protocol. The security manager determines whether the client is authorized to perform the requested operation on the specified object. If the operation is permitted, the security manager generates a credential including the requested capability; this credential is cryptographically secured by a secret shared between the security

	Description	Without a secure channel	With a secure channel
NOSEC	No security	No security	Network-level integrity
CAPKEY	Access control	End-to-end verification of credentials	+ Protection from network attacks
CMDRSP	Access Control, Command Integrity	Verification of command (not data)	+ Protection from network attacks (some duplicated work)
ALLDATA	Access Control, Command Integrity, Data Integrity	End-to-end verification of request	Duplicated work

Table 1: The OSD Protocol Security Methods. Each method is presented on its own and combined with a secure channel that ensures integrity, data origin and authentication.

manager and the OSD as defined in section 6.³

The credential is then sent from the security manager to the client. While the security specifications of this channel are outside of the scope of the standard, it is expected that the channel will provide privacy (encryption). This is needed for protecting the capability key which is used by the client to authenticate that it legitimately obtained a capability.

The client must present a capability on each OSD command. Before processing the command, the OSD verifies: 1) that the capability has not been modified in any way, and 2) that the capability permits the requested operation against the specified object. If the tests pass, the OSD will permit the operation based upon the rights encoded in the capability.

A client may request a credential that permits multiple types of operation (*e.g.*, Read + Write + Delete). This allows the client to aggressively cache and reuse credentials, minimizing the number of messages to the security manager.

2.3 Channel Requirements

When using the CAPKEY method, we assume a network infrastructure that provides secure channels. More accurately, the channel needs be *authenticated but anonymous*, namely both parties need to know that they are communicating with the parties that originally established the channel. Our protocol assumes it is possible for a malicious party to eavesdrop on the channel.⁴ We also assume there is a channel

³The OSD standard makes a distinction between a *policy* manager, which determines authorization, and a *security* manager, which generates the appropriate credential.

⁴Privacy is beyond the scope of the first version of the T10 protocol, but may be obtained from the network infrastructure if needed in a particular environment.

for communication between the OSD and the security manager.

Looking at the bandwidth and latency requirements of the various channels, the channel between the security manager and the object store has the least stringent requirements. This channel is used only for a periodic key exchange and other administrative security operations, with limited performance concerns. The channel between the security manager and client has medium network requirements, since it is used for a message exchange for each unique credential required by the client. In some configurations this could become a performance issue, since this channel must be encrypted to ensure secure operation. The channel between the client and the OSD has the most stringent bandwidth and latency requirements as every request to the OSD flows on this channel. Because of the heavy traffic on this channel, it is not reasonable to assume that by default this channel is encrypted.

2.4 Trust Assumptions

The OSD is a trusted component. This means that once a client authenticates that it is talking to a specific OSD, it trusts the OSD to: (1) provide integrity for the data while stored (2) follow the protocol (3) not be controlled by an adversary. The client can authenticate the intended OSD either via the use of an externally provided authenticated channel (for example if the CAPKEY method is used) or as part of each command using mechanisms defined in this protocol (via the CDMRSP and ALLDATA methods).

The security manager is also a trusted component. After it is authenticated, it is trusted to: (1) safely store long-lived keys (2) compute access controls cor-

rectly according to the policy it implements (3) follow the protocol (4) not be controlled by an adversary.

Finally, the trust assumption on the client is that users trust their own operating system to protect them from malicious users on the same machine who may access data in local memory. We do not trust the client to correctly follow the protocol; however, the client will not receive service if it does not follow the protocol.

3 Credentials and Capabilities

3.1 Overview

The OSD protocol is a capability-based protocol which cryptographically enforces the integrity of the credential and its legitimate use by the client.

Recall the two-steps required by the client to access an OSD object:

- Prior to sending an OSD command to an object store target, the client requests a credential with certain permissions from the security manager (over a secure channel) and in return the security manager sends back a pair $[CAP, C_{key}]$. Together, the capability CAP and capability key C_{key} form the **credential**. C_{key} is secret whereas CAP is public.⁵
- The client sends to the OSD, together with the request, the capability CAP along with a validation tag v , namely $[Request, CAP, v]$. v is computed by the client using C_{key} . The structure and usage of the validation tag depends on the security method being used, as described in section 4.

We will show below that C_{key} can be computed from CAP by the object store, hence the validation tag is also computable by the object store. Based upon the security method, the object store validates the validation tag and checks whether the operation requested by the command is indeed permissible by the capability CAP . Note that the object store does not need to authenticate the client or to have a notion of client identity.

Throughout the protocol there is a need to use a pseudo-random function as a cryptographic primitive. The T10 standard chose HMAC-SHA1 [BCK96] whose output is 160 bits long, as the only required function. In this paper we use the notation \mathcal{F}^{pr} to denote this function.

⁵ C_{key} should be sent to the client over an authenticated and encrypted channel to maintain its secrecy.

Key Management The security manager and the object store share a set of symmetric secret keys, arranged hierarchically and updated periodically as described in section 6. A credential is based on one of the keys from this hierarchy, chosen according to the command being issued. When a key is updated, all credentials based on that key are no longer valid.

Given a command, which key in the hierarchy should be used? An object store is divided into multiple partitions, each of which carries its own keys. A partition is essentially a separate security domain. All commands other than key exchange commands come with credentials which are protected by the *working key associated with the partition containing the object being operated upon*. For those commands which operate at the level of an entire object store, *e.g.*, the commands for formatting the object store or creating/removing partitions, the working key associated with partition zero is used.

3.2 Capability Arguments and Capability Key

The standard defines the credential as a pair $[CAP, C_{key}]$. The capability CAP is the public part of the credential and contains a set of fields that specify what command functions the command may request, on which storage construct (*e.g.* object or the entire device), and other information used for validation purposes. The exact structure is specified in [T10, Section 4.9.2] and below is a high-level description of these fields as well as the rationale for including them in the capability. C_{key} is the secret part of the credential and is defined as

$$C_{key} \equiv \mathcal{F}_K^{pr}(CAP)$$

Here \mathcal{F}^{pr} is the pseudo-random function and K is a secret key from the key hierarchy that is shared between the security manager and the object store.

The capability CAP is described as

$$CAP \equiv \left[\begin{array}{l} \text{SecurityInfo, ExpiryTime, Audit,} \\ \text{Discriminator, ObjCreationTime,} \\ \text{ObjType, ObjDescriptor,} \\ \text{Permissions} \end{array} \right]$$

- The **SecurityInfo** field identifies the shared key to be used, the cryptographic algorithm, and the security method.
- The OSD capabilities are time-based; **ExpiryTime** specified the expiration time. Section 5 elaborates on this field.

- The `Audit` field is reserved as a vendor specific value that the security manager may use to associate the capability and credential with a specific application or client. Use of this field is optional and it may be used by the anti-replay protection described in section 4.2 to help group client requests.
- `ObjCreationTime`, `ObjType` and `ObjDescriptor` together uniquely identify the storage construct to which this capability applies and its version. `ObjType` can be one of four: `root`, `partition`, `collection` or `user`. The `ObjDescriptor` contains the object identifier and its policy access tag (see below). Since objects with the same identifier may be created at different times, the `ObjCreationTime` uniquely identifies the object over time.
- The `Discriminator` field is a nonce which ensures the uniqueness of the capability. This is needed to avoid scenarios in which the same capability CAP (and therefore C_{key}) is given by the security manager to different clients requesting the same rights to the same (set of) object(s).⁶
- The `Permissions` field encodes the set of allowed operations on the storage construct; permissions include: `read`, `write`, `get/set attributes`, `policy/security` (*i.e.*, changing keys), `object management` and others.

3.3 Special Considerations

Credential Revocation: Rapid revocation of access rights is a very important operation for distributed systems. For example, if a file's ownership is modified, the system should immediately stop all file access until the security manager re-evaluates each client's access rights. To build a revocation mechanism above the OSD would require complex software, such as a distributed callback module; such an approach is known to be inefficient and difficult to ensure correctness in the face of network partitions. The OSD protocol recognized this basic function and enabled immediate revocation via two mechanisms for invalidating a credential

The first mechanism, key exchange, is a coarse-grained approach that exchanges the key between the security manager and the object store. By changing the shared key, *all previous credentials a security*

manager had generated for a particular object store partition are now invalid.

The second mechanism is fine-grained and invalidates all outstanding credentials *for a given object* by utilizing the *object policy access tag*. This tag is a settable object attribute (typically by the security manager only, since it requires a special permission to set). A valid credential must match the policy access tag of the object; hence, we can invalidate all outstanding credentials for an object by modifying the value of its policy access tag.

Bootstrapping Since a credential includes information which is stored as attributes for the objects (namely the creation time and policy access tag), we may have a problem of bootstrapping, in particular if the security manager does not have this information in its memory. How does the security manager generate a credential to read these attributes if it does not know these attributes? In addition, in certain usage scenarios, *e.g.*, if object IDs are assigned by an external cataloging entity, the use of the creation time may require additional message exchanges and provide no benefit.

To address this, the security manager has the option to generate a capability with a wild card (zero) for the policy access tag or the creation time. In this case, when calculating the capability arguments, the object store should not take into account the actual value of the respective attribute associated with the object. When used with the policy access tag this essentially creates a credential which cannot be invalidated during its life-span other than by a key exchange.

Delegation: It is possible for a client to *delegate* a credential to another client, by transferring both CAP and C_{key} . While beyond the scope of this protocol, to ensure security, such delegation should be done in a private manner (*e.g.*, over an encrypted channel). If desired, such delegation may be prevented by use of the `Audit` field. As shown in [HKN05], an appropriate use of the `Audit` tag leads to "security confinement" thereby preventing leakage of information to unauthorized users.

4 The Security Methods

Recall that the OSD protocol defines three types of security methods which, depending on the underlying network's security assumptions, guard against a wide

⁶This addresses a potential security hole in the CMDRSP and ALLDATA security methods. Without this, a client could masquerade as an object store for another client, if both clients get the same authorization for a given object.

range of security attacks.⁷ This section elaborates on these mechanisms. We first show in Section 4.1 the basic mechanism for protecting credential integrity as used in the CAPKEY security method. In the absence of a secure channel, integrity of the command and an anti-replay mechanism is also needed; the 'per request nonce' mechanism that is used in CMDRSP and ALLDATA security methods is addressed in section 4.2.

4.1 Integrity of Credential

It is necessary to bind a given credential to a particular secure (IPSec-like) channel between the client and object store. Binding ensures that an eavesdropper cannot replay a credential over a different secure channel.

Given a credential and a channel, the OSD protocol ties the message to the channel via a **validation tag**. This tag is computed by the client as

$$\mathcal{F}_{C_{key}}^{pr}(ChannelID)$$

where *ChannelID* identifies the communication channel and is unique to this combination of client, OSD, and the particular link on which they communicate.⁸ C_{key} is the capability key associated with the command.

The *ChannelID* is chosen by the OSD; the OSD protocol provides a mechanism for a client to request the value of *ChannelID*. Given that the OSD knows the channel on which a request was received and its *ChannelID*, the OSD can verify that the validation tag attached to the request equals $\mathcal{F}_{C_{key}}^{pr}(ChannelID)$. Note that the same validation tag can be used with all requests based upon a given credential.

This mechanism is used by the CAPKEY security method. Since the capability key C_{key} with which the validation tag $\mathcal{F}_{C_{key}}^{pr}(ChannelID)$ is computed depends on the credential *CAP*, it ensures that the request is authenticated by a client who legally obtained the credential, whether directly from the security manager or indirectly via delegation.

4.2 Per Request Nonces

The CMDRSP and ALLDATA methods do not rely on an underlying security transport mechanism;

hence the OSD protocol must provide its own mechanisms to protect message integrity and prevent message replay. Message integrity is naturally provided by attaching a message integrity value to each protocol message computed over the command and its arguments, and possibly over the data (for ALLDATA); this corresponds to the validation tag of the CAPKEY method. To prevent message replays, the protocol uses *per-request nonces*.

Correctness of any nonce-based approach requires that the object store: 1) not accept the same nonce more than once and 2) not accept a nonce that was rejected in the past (see 4.2.3). It is acceptable to reject valid requests with previously unseen nonces if necessary. Note that since replay attacks are possible only throughout the lifetime of a particular key, keeping *nonce-history* is required only between two consecutive key exchanges.

Typically, there are three means of generating per-request nonces: random, session-based, and time-based. Random nonces require too much space (one per request seen, including invalid requests, between key exchanges) to be implemented correctly. A session-based approach requires per-client session setup and penalizes the common case to accommodate the uncommon. If all entities in the system are well-behaved, a time-based protocol has the best performance and lowest space requirements. However, the time-based protocol can also have large memory requirements if enough clients in the system are ill-behaved (e.g. have incorrect clocks). Requiring strong clock synchronization between the clients and object store is problematic from both a practical and a security perspective.

The approach we define is time-based modified to have only weak dependence on correct client clocks and augmented to minimize the impact of ill-behaved entities (clients or time servers). Below is an overall description of this approach, including motivation and rationale; details can be found in [T10, Section 4.10.7].

On each request, the client generates a nonce by combining a 48-bit time representing the number of milliseconds since January 1, 1970 with a 48-bit random number. The object store target maintains a time window and a list of nonces. If the window is kept sufficiently large, this scheme approximates a random approach. As elaborated below, if malicious or ill-behaved clients can be correctly identified with per-client audit tags, then the approach is essentially session-based with overhead only in the presence of malicious clients.

⁷We do not elaborate on the NOSEC method since by definition it does not provide security.

⁸The standard requires that *ChannelID* be a random value, but in theory it could also be a nonce.

4.2.1 Goals

In designing the nonce protocol, we wanted to ensure that an implementation can 1) limit the cost that well-behaved systems need to pay in order to accommodate ill-behaved systems 2) bound the amount of memory required to store nonces between key exchanges; 3) minimize the number of key exchanges required solely to reset the nonce history; 4) trade complexity and space against system performance in the face of ill-behaved clients.

4.2.2 Details

The details of nonce implementation are left to individual system designers, but the externally visible behavior is defined by the standard.

The protocol creates a window around the current time as seen by the object store. Request nonces in the window can be serviced as valid. Nonces before the start of this window may be rejected and need not be remembered. Nonces beyond the window are known as *far-in-the-future nonces*. Such nonces must be remembered until the next key exchange in order to prevent an attack where a client is tricked into sending an OSD request with future nonces that are saved by an adversary until they become valid. Nonces for requests with invalid working keys can always be discarded.

Once nonce storage has been exhausted, the object store must either stop servicing requests or force a new exchange of working keys. As the time window moves forward, nonce storage can be reclaimed and further requests serviced. In a random nonce approach, there would be no way to reclaim storage until a change of working keys.

This behaviour could have potentially exposed the system to denial of service attacks by malicious clients, who will send many *far-in-the-future* nonces to exhaust the available nonce storage causing the object store to stop servicing. Alternatively, it could lead to frequent forced key exchanges, thereby degrading the service for all other well-behaved clients. To mitigate the memory required to handle ill-behaved clients, the object store may organize nonces into groups. If far-in-the-future nonces are limited to a subset of the groups of nonces, the implementation can choose to reject only nonces belonging to the problematic groups and continue to service well-behaved clients. The efficiency of such an approach depends on the accuracy of the grouping and the object storage may leverage the audit tag field of the credential-nonce for this purpose. In such an implementation, an OSD could free all remembered nonces

which were used with capabilities with a given audit tag value. The OSD would then automatically reject all commands received with capabilities containing that audit tag, without even examining the nonces. This behavior would need to continue until the next relevant key exchange.

4.2.3 Additional Considerations

Correcting ill-behaved clients. If the nonce in a request is outside the time window, the object store may reject the request without further processing with an INVALID_NONCE error message. This response includes the current time of the object store, allowing a client to try again with a nonce within the window.

Response to erroneous requests. The object store must remember a nonce even if the message fails verification of the integrity check. An attacker may modify a valid request to fail the integrity check, saving the original request for later. The client receives an INVALID_INTEGRITY_VALUE response and resends the request with a new nonce and integrity value. If the second request succeeds, the attacker can now replay the saved message at some future point.

Use of time. The only requirement for the time used to determine nonce timestamps is that it be monotonically increasing. Weakly synchronized clocks will help to avoid additional messages. In order to catch the time up to an external "real time", the object store may choose to accelerate or decelerate the passage of time. Alternatively, an object storage that is unsure of the time, or concerned about a time-based attack, may choose to expand the size of its nonce lists as it sees fit. This may slow performance, but does not affect security.

4.3 Performance

Below are a number of performance considerations for the different security methods.

CAPKEY: As long as the same channel is used,

- A client does not need to request a new credential on every command; rather, the client can reuse the CAP and C_{key} on multiple commands for the same object(s).
- The client does not need to recalculate the validation tag on each command; rather, this needs to be calculated only once per credential.

- The object store does not need to recalculate C_{key} and $\mathcal{F}_{C_{key}}^{pr}(ChannelID)$ on each exchange with a client. Rather since we assume a secure channel, these values need to only be calculated the first time object store sees a given capability on a given channel.

CMDRSP:

- A client does not need to request a new credential on every command; rather the client can reuse the CAP and C_{key} on multiple commands for the same object.
- The object store does not need to recalculate C_{key} on each exchange with a client; rather this need be calculated only the first time the OSD sees a capability. After that, an implementation can cache this computation.

ALLDATA: Here an additional concern arises for efficient implementation. The efficient computation of the integrity value on the data is straightforward in the case of Read. As data is read from the media, the integrity value is computed and it is sent as part of the status message at the end of the command.

The case of Write is more difficult. If the integrity value on the data is sent in the same message as the command in the command header, then the client must make two passes over the data - one to compute the integrity value and a second to send the data. In order to avoid this, the integrity value is attached to the end of the data buffer and the integrity value field in the command is ignored at verification. Note that this requires the OSD to hold the data until all data has arrived.

5 The Security Manager

The T10 OSD standard carefully defines the protocol between the security manager and the OSD, various security methods, and the structure of the credential used by clients to gain access to storage. However, to ensure flexibility in the upper software layers (*e.g.*, different file systems or applications each with their own security policy), the precise security manager policies are not defined by the standard. Issues such as OSD security method, access to data and/or attributes, and the type of communications channel between client and security manager are determined by the upper-layer software. This allows the upper-layer software to: (1) make decisions based on environment (*e.g.*, secured SAN vs. insecure LAN), (2)

usage models (*e.g.*, sensitive corporate data vs. open web server data), (3) provide diverse security policies based on file system or application demands (*e.g.*, NFS vs. CIFS).

To ensure correctness of the OSD security protocol itself, the interactions between the security manager and the OSD are defined using the OSD capability model. OSDs require the security manager to present a valid capability authorizing the operation. Because security-level commands such as changing keys (the SET_KEY commands) require a high degree of security, the security manager must use the appropriate method of security as specified for the partition with which it is interacting. This prevents the potential security weakness of attempting to use a SET_KEY command with the NOSEC security method.

Finally, the protocol includes a capability expiration time that limits the time a compromised capability can cause and limits the time a client can access an object. Because the capability expiration time within the capability must be interpreted by the OSD, we require some degree of clock synchronization between the OSD and Security manager. The protocol for synchronizing the clocks is not specified as part of the object store protocol, but expects that a standard clock synchronization protocol is implemented in a secure manner.

6 Key Management

The OSD protocol defines its own key management. All credentials are based on a secret key that is shared between the object store and the security manager. To prevent an adversary from obtaining too many credentials generated with the same key, keys must be refreshed regularly. Key management requirements are:

- The security manager should be able to replace individual object store keys in a secure manner without requiring a secure communications channel.
- The OSD should support multiple capability generating keys that provide security isolation for different objects.
- The protocol should also provide a 'heavy-weight' key refresh mechanism preserving forward secrecy.
- A random source to generate keys is required only from the manager, and not from the object store.

- The drive manufacturer: (1) cannot assume to know the identity of the drive purchaser; (2) should not have control over the drive once it is initialized; (3) provisioning a new drive should not require mechanical actions to configure the security mechanism.
- A drive crash should not necessarily invalidate valid credentials.

6.1 Key Hierarchy

The key hierarchy is comprised of 4 layers:

1. **Master key.** Used to initialize the drive and to create the root key.
2. **Root key.** Used to create OSD partitions and their associated partition keys.
3. **Partition key.** Used solely to create the working keys, partition keys are changed infrequently, but in a regular manner to increase security.
4. **Working keys.** Used to generate the capability keys.

The *master key* is the topmost key in the hierarchy. It allows unrestricted access to the drive. Its loss is considered a catastrophic event. Due to the importance of the master key, the protocol limits its use to the infrequent event of setting the root key. This master key does not change unless the drive owner is changed. Master Key refreshes are done via a Diffie-Hellman key exchange protocol [DH76], to guarantee perfect forward secrecy.

The *root key* provides an unrestricted access to the drive, very much like the master key, except that it cannot be used either to initialize the drive or to set another master key or a new root key. Once the root key is set it can be used to set the partitions' keys. The root key can be changed in case it was compromised, or as part of a scheduled update operation in order to maintain security.

Partition keys. An object store is divided into multiple partitions, each having a unique partition key to generate its own working keys for that partition.

The *working keys* are used to generate the capability keys used by clients to access individual objects. Because of their frequent use working keys should be refreshed very frequently, e.g., on an hourly basis. Unfortunately, a key refresh immediately invalidates all credentials generated by that key. This could result in a significant performance degradation as all the clients would be required to communicate with the security manager in order to get new credentials. The

load on the object store device would also increase because all new credentials would have to be explicitly validated before being cached by the object store. To mitigate these problems, the object store may declare - multiple (up to 16) refreshed versions of the working key as valid. This effectively defines multiple working keys that are valid concurrently. Therefore, a key refresh would impact a limited number of capabilities. To support this feature, the protocol required a *key_version* field to be incorporated in the capability indicating which key should be used in the validation process.

The number of active key versions used is agreed between the OSD and the security manager. When setting a new working key, the security manager tags the key with a version number (between 0 and 15); the object store uses this tag to determine which key to use in validating a command.

6.2 Key Exchange Protocol

The Key Exchange protocol is accomplished via two commands, `SET_KEY` and `SET_MASTER_KEY`. Except for the topmost key, the key exchange protocol allows keys of one level can be replaced only by using a higher-level key. The exchange of a key at a given level invalidates all keys at lower levels (e.g., a new partition key invalidates all working keys). The compromise of a key at a given level does not reveal information on keys in higher levels, or on other keys (if multiple key versions exist) at the same level.

The refresh of the topmost key (the Master Key) implements the Diffie-Hellman key exchange protocol [DH76], thus achieving forward secrecy. For all other keys, an old key is used to generate the newly computed key by sending a newly created seed to a pseudo-random-number generator (PGRN) based on \mathcal{F}^{pr} .

We require that at each level, there will be two keys rather than one. The first key is used for computing C_{key} (that is, to generate the request credentials) and the second key for the pseudo random number generator used for key generation. Note that the protocol does not describe how random seeds are generated. It is the responsibility of the security manager to create them as random as possible.

Instead of defining a set of specific protocol messages to be used for key management, we use the `SET_KEY` and `SET_MASTER_KEY` OSD commands along with the basic OSD security mechanisms. We assume that we have objects (or pseudo objects) with known identifiers representing the object store as a whole as well as each partition.

The partition and working keys are set by invoking SET_KEY on the object for the partition and the root key by invoking it on the object for the object store as a whole.

7 Protocol Enhancements for Future Versions

The OSD Version 1 security protocol described in this paper provides a solid mechanism for ensuring secure access to stored objects. Several security enhancements are currently being addressed by the T10 OSD Security Working Group. These include:

- **Privacy.** There is no encryption in the current Version 1 OSD standard. Users must instead rely on encryption at different layers of the protocol stack (i.e., IPsec, application). However, we feel that privacy is a core requirement for future storage. Therefore we are working on leveraging privacy mechanisms defined by other standards for providing encryption for data in transit and at rest (and the associated infrastructure for encryptions key management).
- **Fine-grained capabilities.** The capability's permissions field supports access control on individual objects data/attributes and the commands that can be applied. A finer-granularity, particularly the need to specify access control over individual attributes is important to overall system performance because it allows the higher-level software layers to securely delegate specific attribute access and updates to untrusted clients. However, simple bit-vectors within the SCSI CDB are insufficient. Therefore, we are investigating using lists of attribute IDs created by the security manager and stored within the object store device to support efficient fine-grained access control over attributes.
- **Complex capabilities.** Currently, software systems that wish to manipulate multiple objects must obtain individual capabilities (one per object) or very powerful capabilities. To reduce the security manager load that individual capabilities create, we are investigating the use of multi-object capabilities (i.e., a capability that applies to more than one object). One proposed approach is to allow a single capability to apply to all user-objects contained within a collection-object. Another approach is to encode multiple object IDs within a single capability. This has

the drawback of potentially creating very large capabilities.

- **Integrity of data at rest.** Efficient data integrity at rest to ensure that the object's data and meta-data has not been modified.

Acknowledgements

The OSD Security protocol was designed as part of the OSD-v1 standard within the OSD SNIA working group [SNI], and many members of that working group contributed to the security aspect as well. We wish to specifically thank Sami Iren, Don Beaver and Mallikarjun Chadalapaka. Very special thanks also to Ralph Weber, editor of the T10 OSD standard; his diligent editing often forced us to clarify our ideas down to the bit level. David Blake from EMC carefully reviewed the standard and provided us with useful comments and improvements. Shai Halevi, Noam Rinetzky, Ohad Rodeh and Allon Shafrir from IBM contributed to the design of the protocol and in particular Noam pointed out the potential security hole described in section 3.2.

References

- [ACF⁺02] Alain Azagury, Ran Canetti, Michael Factor, Shai Halevi, Ealan Henis, Dalit Naor, Rinetzky Noam, Ohad Rodeh, and Julian Satran. A two layered approach for securing an object store network. In *Proceedings of the First International IEEE Security in Storage Workshop*, pages 10–23, Greenbelt, MD, 11 December 2002.
- [BCK96] M. Bellare, R. Canetti, , and H. Krawczyk. Message authentication using hash functions: The hmac construction. *RSA Laboratories' CryptoBytes*, 2(1), Spring 1996.
- [DH76] W. Diffie and E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(5):644–654, 1976.
- [FMN⁺05] M. Factor, K. Meth, D. Naor, O. Rodeh, and J. Satran. A position paper on object storage. In *IEEE Symposium on 'Global Data Interoperability - Challenges and Technologies*, pages 119–123, Sardinia, Italy, June 2005.
- [GNA⁺96] Garth A. Gibson, David F. Nagle, Khalil Amiri, Fay W. Chang, Eugene Feinberg, Howard Gobioff, Chen

- Lee, Berend Ozceri, Erik Riedel, and David Rochberg. A case for network-attached secure disks. Technical Report CMU-CS-96-142, Carnegie Mellon University, Pittsburgh, PA, 26 September 1996. <http://www.pdl.cmu.edu/PDL-FTP/NASD/TR96-142.pdf>. [T10]
- International Committee for Information Technology Standards (formerly NCITS). *SCSI Object-Based Storage Device Commands (OSD)*. Document Number: ANSI/INCITS 400-2004. Technical editor: Ralph O. Weber. December 2004.
- [GNA⁺97] Garth A. Gibson, David F. Nagle, Khalil Amiri, Fay W. Chang, Howard Gobioff, Erik Riedel, David Rochberg, and Jim Zelenka. Filesystems for network-attached secure disks. Technical Report CMU-CS-97-118, Carnegie Mellon University, Pittsburgh, PA, July 1997. <http://www.pdl.cmu.edu/PDL-FTP/NASD/CMU-CS-97-118.pdf>.
- [Gob99] Howard Gobioff. *Security for a High Performance Commodity Storage Subsystem*. Ph. d. dissertation, CMU-CS-99-160, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, July 1999. http://www.pdl.cmu.edu/PDL-FTP/NASD/hbg_thesis.pdf.
- [HKN05] Shai Halevi, Paul A. Karger, and Dalit Naor. Enforcing confinement in distributed storage and a cryptographic model for access control, 2005. Cryptology ePrint Archive: Report 2005/169.
- [ISE05] IBM, Seagate, and Emulex. Storage networking world conference: An interoperability demonstration of an osd-based file system, Spring 2005.
- [Lus] Lustre. <http://www.lustre.org>.
- [MGR03] Mike Mesnier, Gregory R. Ganger, and Erik Riedel. Object-based storage. *IEEE Communications Magazine*, 41(8):84-90, August 2003.
- [NRR⁺05] Dalit Naor, Petra Reshef, Ohad Rodeh, Allon Shafir, Adam Wolman, and Eitan Yaffe. Benchmarking and testing osd for correctness and compliance. In *Proceedings of the 2005 IBM Verification Conference*, November 2005.
- [Pan] Panasas. <http://www.panasas.com>.
- [SNI] Osd: Object based storage devices technical work group. http://www.snia.org/tech_activities/workgroups/osd/.