Storage Networking Industry Association
Object-Based Storage Devices

SNIA OSD TWG document cross posted to T10

Object Store Security Document T10/03-279r0

Revision: 8
Last Revised: 8/12/2003

Abstract

This document presents the requirements, motivation and a proposal for the security protocol for object store. This protocol is based upon the original Network Attached Storage Device (NASD) work [8] as well as other work on secure object stores 9.

Related Documents

- The OSD White Paper offers an introduction to OSD and its applications.
- The OSD Requirements Document discusses requirements of the OSD applications discussed in the white paper.
- The T10 SCSI Draft Standard for OSD implements the OSD framework for the SCSI architecture model.

Table of Contents

0 Revision History

0.1 Revision 1

Authors: Dalit Naor, Michael Factor, Julian Satran (IBM), Don Beaver, Erik Riedel (Seagate), and David Nagle (Panasas).

0.2 Revision 2

Authors: Dalit Naor and Michael Factor

0.3 Revision 3

Authors: Erik Riedel

Changes: Added description of Levels 2 and 3. Added sequence diagrams and detailed message arguments. Changed “client” to “host” throughout for consistency. Use “OSD” instead of “object store”. Reorganized items in the introduction. Added more white space for better readability.

0.4 Revision 4

Author: Dalit Naor.

Changes: Incorporated presentation changes and open issues from comments submitted by May 6 from David Nagle, Erik Riedel, Dalit Naor, Michael Factor.

0.5 Revision 5

Author: Michael Factor

Changes: Incorporate changes to open issues as discussed in the SNIA Symposium in Boston. Main changes include, added section on protocol between OSD and Security Manager, added description of error codes, cleaned up presentation of sections 2 and 3

0.6 Revision 6

Author: Michael Factor

Changes: Fixed formatting. Cleanup of Chapter 1 (provided an introduction, deleted objectives section as being repetitive with requirements, deleted some discussion of levels 4-7 of security, editorial corrections). Moved key management to Chapter 8 and added Chapter 8. Added a separate chapter on Nonces (Chapter 4) in place of the section on nonces which was in the discussion of level 2 security.

0.7 Revision 7

Authors: Michael Factor, Dalit Naor

Minor changes from prior revision: Integrate editorial comments, in particular cleanup usage of capability and credential. Add section 2.5 to describe credentials for creating objects without specifying an object ID.

0.8 Revision 6

Author: Michael Factor

Additional minor changes: clarify object version number in credential (and renamed to object version tag), (re)add creation time to credential, clarify which keys protect credentials for commands that are not scoped to a partition. Also remove descriptions of how do we know the appropriate security level; this will not be addressed in the first version of the standard

1 Introduction

Object storage is a new storage paradigm (in particular for network accessible storage) in which the abstraction of an array of blocks is replaced with an abstraction of a collection of objects. In object storage, a client accesses data by specifying the identity of an object along with an offset in the object, and the implementation of the storage is responsible for mapping the offset to the actual location on the physical storage. From a security perspective, the main change between object storage and today's block storage paradigm is that every command is accompanied by a cryptographically secure capability. Thus, object storage provides the means of having secure, fine-grained access to storage.

This document presents the requirements, motivation and a description of the object store security protocol. The goals of this document are multifold. First, it is intended to specify the behavior of the high-level protocol in sufficient detail to allow a direct mapping to a standard specification in a particular transport (*e.g.,* in SCSI). It is also intended to explain the protocol in a way that it can be shared with security experts, outside of the OSD community, to allow an independent review of its correctness. Finally, it is intended as a general background material to explain OSD security.

One major goal for OSD security is to work well both on top of a secure network infrastructure and in environments where there is no such infrastructure. This requirement has led us to define multiple of levels of security which reflect the assumptions on the underlying infrastructure and the protection required.

This document is organized as follows. This chapter describes the basic security model, and the requirements we imposed upon ourselves. The next chapter describes the structure of the capabilities/credentials and the basic message flow; this structure and flow is common for all levels of security. Chapter 3 describes the details of the security level which ensures integrity of the security mechanism; this level is ideally suited for use on top of a secure network infrastructure, but it also can be used in environments where there is no concern of network-type attacks. Chapters 5 and 6 describe two different levels of security intended for use on insecure networks; they differ in whether or not they secure the data. Chapters 7 and 8 describe security aspects that are not on the main data path.

1.1.1 Basic Security Model

The object store security model is a credential-based access control system composed of three active entities: the object store, a security manager, and a client/host. Each entity plays a different role.

As a credential-based access control system, all requests to the object store must be accompanied with a valid capability that allows the host to perform the requested operation. A *credential* is a cryptographically secured capability and a *capability* is a set of rights the holder has on an object (or set of objects).

The role of the security manager is to generate credentials for authorized hosts at the request of the host. The protocol between the host and the security manager is not defined as part of the

OSD protocol; however, the structure of the credential returned from the security manager to the host is defined. In addition, the protocol between the OSD and the security manager is specified.

The role of the OSD is to validate a capability presented by a host:

1. The requested operation is permitted by the capability based on *a)* the type of operation (*e.g.*, read, write) and *b)* a logical match of the specified attributes
2. The capability has not been tampered with, *i.e.*, it was generated by the security manager and was rightfully obtained by the host that presents it (either directly or via delegation).

The object store can validate that a host rightfully obtained a capability since a credential contains both the capability and a secret part (CAP_Key – see section 2), which the host uses to sign its messages to the object store. Without this secret part, which should be transferred on an encrypted channel from the security manager to the host, the host cannot generate validly signed messages. Note this protocol does allow delegation of a credential if a host transfers both the secret part of the credential as well as the public capability arguments.

The role of the host is to follow the protocol. While the host is not trusted to follow the protocol, the protocol is structured in such a way that it is in the host's self-interest to follow the protocol. In other words, if the host does not follow the protocol, it will not receive service from the OSD. The figure below shows this basic flow.

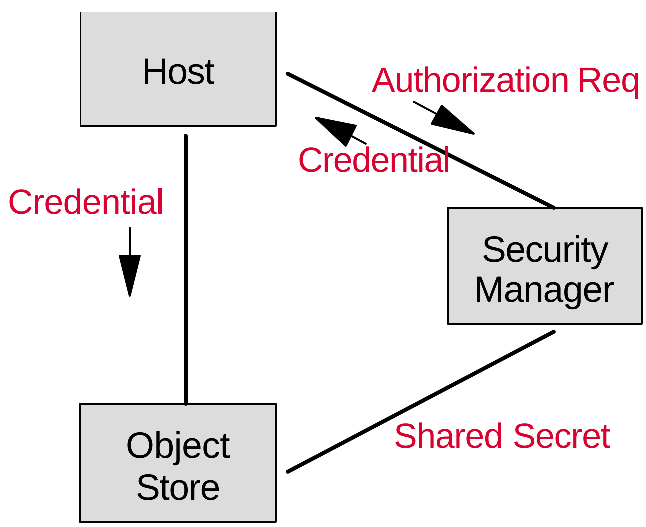


Figure 1. Basic System Structure

We specify seven levels of security, of which only the first three are within the scope of the current proposal:

Level 1 – Integrity of capability
Level 2 – Integrity of arguments
Level 3 – Integrity of data in transit
Level 4 – Privacy of arguments
Level 5 – Privacy of data in transit

Level 6 – Integrity of data at rest
Level 7 – Privacy of data at rest

Levels 2-4 correspond to the security levels defined in the original NASD work [8]. Level 1 is best suited for the case where the network between the OSD and the host is secured; it can be used as another layer on top of the network security 9.

With Level 1, only *access security* is handled within the OSD specification, and *network security* is handled by an external, network-specific means (*e.g.,* IPSec or FCS).

In order to implement Level 3 efficiently, the authentication hashes for user data must be carried by the underlying transports. The structure and interpretation of these hashes will be specified in this document, but an efficient mapping to a particular network transport layer (*e.g.,* FC or TCP/IP) is left to external specifications.[1]

1.1.2 Trust Assumptions

Trust assumptions describe how each element of the system trusts the other elements of the system. The OSD is a trusted component. This means that once a host 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 host can authenticate that it is talking to the intended OSD, *i.e.,* the one for which the security manager has granted it credentials, either via the use of an externally provided authenticated channel or as part of each command using mechanisms defined in this protocol.

The security manager is also a trusted component. After it is authenticated,[2] it is trusted to:

1. safely store long-lived keys
2. compute access controls correctly according to the policy it implements[3]
3. follow the protocol
4. not be controlled by an adversary

The trust assumption on the host is that a user trusts their own operating system to protect them from malicious clients on the same machine, *e.g.,* protect its CAP_Key. We do not trust the host to correctly follow the protocol; however, the host will not receive service if it does not follow the protocol.

1.1.3 Security Flow and Channel Requirements

As mentioned above, when a host wishes to access an object (or set of objects), it makes a request to the security manager for a credential allowing the intended operation.[4] In this

[1] The difficulty is in the ordering of the hashes with respect to the data while in transit and during verification at the device. This is discussed further in Section 6.1.
[2] Authentication of the Security Manager by the host is out-of-scope of this protocol.
[3] The definition of this policy is outside of the scope of this proposal.

request, the host must specify the OSD and partition (see section 2.3.2) on which it wishes to perform the operation; the identity of the object(s) it wishes to access; and the operation(s) it wishes to perform. The security manager upon receiving this request may need to authenticate the host making the request.[5] After authenticating the host, the security manager applies its policy to determine whether the client is authorized to perform the requested operation(s) on the indicated object(s). If not, the security manager will fail the request for the credential. Otherwise, the security manager will generate a credential including the requested capability; this credential is cryptographically secured by a secret shared between the security manager and the OSD. The credential is then sent from the security manager to the host over a channel which is encrypted and authenticated. Other than specifying the structure of the credential returned from the security manager to the client, the protocol between the client and security manager is not defined by the OSD protocol.

The host must present a capability on each operation it executes against the OSD. When the OSD receives the capability, it verifies that it has not been modified, using the secret it shares with the security manager.[6] If the credential has not been modified (and is properly held by the requesting client), the OSD will permit the operation based upon the rights encoded in the capability.

When using Level 1 protection, we assume an existing network infrastructure that provides secure channels (*e.g.,* IPsec) between the OSD and the host. More precisely, if we are running over a secure channel, we require both parties of the communication to know that they are communicating with the parties that originally established the channel (an authenticated but anonymous channel). We do not imply a requirement for privacy, *i.e.,* we assume it is still possible for a malicious party to eavesdrop on the channel.

We also assume there is a channel 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[7] and other administrative security operations (see chapter 7). We believe that the performance of this channel is not an issue.

The channel between the security manager and host 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 it is expected this channel be encrypted.

The channel between the host and the OSD has the most stringent bandwidth 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.

---

[4] The host may request a broader set of rights than what is required for the operation it currently wishes to perform.

[5] It is conceivable that an authentication is not required, *e.g.*, an object with world-wide read permission.

[6] When caching of credentials is possible, some verification steps can be omitted.

[7] The protocol does not specify this period, but we believe tens of minutes or longer would be reasonable.

1.1.4 Layered Approach to Protocol Definition

We take a layered approach to defining the protocol for object store security. This allows an implementation to provide only the desired level(s) of (internal) security and to surface the various layers in a consistent manner.[8] We also want to ensure a consistent message exchange between the elements of the system, regardless of what level of security is supported.

An object store implementation defines the levels of security it supports. By enriching the information included in a message a higher level of security can be internally provided, as opposed to leveraging an external network security mechanism.

In taking this approach, we want to provide flexibility in choosing how to secure the transport, either internal or external, while allowing an installation to pay only for the level of security needed. This should enable a simplified solution in certain glass-house environments (where no network attacks are expected). It also should enable leveraging existing infrastructure for network security and privacy while avoiding the cost of duplicate mechanisms. At the same time, we must define a mechanism, which an object store can optionally implement to provide network security as part of protocol for use where no secure transport exists.

1.2 Levels of Security

We consider several different levels of security that an OSD could provide. In the first version of the protocol we only directly provide the first three of these security levels. Privacy can be provided through external mechanisms, *e.g.*, running the protocol on an encrypted channel. The levels are incremental and support all the protections of the level below them.

The particular level of security to be used for accessing a set of objects will be defined using a mechanism not specified by the initial version of the standard.

1.2.1 No Security

In the no-security level, the same message structure will be used. However, when an object store is running with no security, the host must place zeros in the message related fields and the object store must not examine these fields.

This is not considered a security level and its support is optional.

1.2.2 Level 1 – Integrity of Capability (Access Control Security)

Access Control Security is the common component to all the levels.

Integrity of capabilities by itself is most useful when the channel between the object store and client is externally secured. In this case, *e.g.,* where we have an authenticated IPSec channel, we still need a mechanism that prevents a host from forging or otherwise modifying a credential and/or replaying a credential over a different authenticated channel. In addition, we need to verify that the host rightfully possesses the credential it is presenting. Without a secure network,

[8] An implementation need not be layered

using only integrity of capability leaves an installation susceptible to certain network attacks, *e.g.,* man-in-the-middle, replay, *etc*.

Support for this security level is optional. However, its functionality must be supported in conjunction with all other security levels.

1.2.3 Level 2 – 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 (hashes) for both commands and data would be too expensive.

With integrity of arguments, malicious hosts cannot replay command parameters, even when running on unsecured networks, but they can use network attacks on the data portion of the messages exchanged between the client and the OSD.

Integrity of arguments prevents a malicious host from accessing a portion of object which was not accessed by some client with a valid credential for the object, or changing a read operation into a write, but it does not prevent a malicious host from modifying the data read from or written to the object.

Support for this security level is optional. If supported, it must be supported in conjunction with the functionality of integrity of capability.

1.2.4 Level 3 – Integrity of Data (Access Control and Internal End-To-End Security)

We assume that integrity of the data includes integrity of the command, *i.e.,* there is no point in protecting the data if the command parameters describing to which object the datum belongs is not also protected. This level of security provides security similar to integrity of capability when the channel between the object store and the client is authenticated. The exact comparison between the two depends on the level of network security that is provided by the external security mechanism. The difference is that this level of the security internally secures the network as an integral part of the object store protocol, thereby defining an end-to-end solution at the storage layer as opposed to building upon pre-existing mechanisms for secure network channels.

Support for this security level is optional. If supported, it must be supported in conjunction with the functionality of the two prior levels.

1.2.5 Privacy

Providing privacy, *i.e.,* encryption, to the command and data, either in flight or at rest is beyond the scope of the current proposal. This includes Levels 4 and 5. Note, there is nothing in this proposal that precludes building upon external mechanisms for encryption.

1.2.6 Summary of Security Levels

All of the security levels are summarized in the table below. The table shows each level on its own as well as each layer when combined with a network security mechanism (such as IPSec) providing integrity and (separately) encryption as well as integrity.

		w/o a secure network	w/ a secure network (integrity)	W/ a secure network (encryption)
None	No Security	No security	Network-level integrity	Network-level privacy
Level 1	Access Security	End-to-end verification of credentials	+ Protection from network attacks	+ Protection from network snooping
Level 2	+ Command Integrity	Protection from mistakes	+ Protection from network attacks (some duplicated work)	+ Protection from network snooping
Level 3	+ Data Integrity	End-to-end verification of requests	Duplicated work	+ Protection from network snooping
Level 4	+ Command Privacy	Protection from traffic analysis on commands	Duplicated work	+ Protection from snooping of data
Level 5	+ Data Privacy	End-to-end protection from snooping of data	Duplicated work	Duplicated work
Level 6	+ Data Integrity at Rest	Protection from modification of data on physical attack	Duplicated work	Duplicated work
Level 7	+ Data Privacy at Rest	Protection from leaking of data on physical attack	Duplicated work	Duplicated work

1.3 Requirements Summary

We have defined a set of requirements for the OSD security model; these requirements attempt to address a range of target platforms for implementing OSD.

On the one hand we believe it is important to enable efficient implementations of the object storage interface in storage controllers; such storage controllers are relatively resource rich, and it is reasonable to envision them containing support for standard network security, *e.g.*, hardware support for IPSec. We wish to be able to use an existing network security infrastructure (when practical) to take advantage of the development and design effort, as well as the administrative and support tools developed for such an infrastructure, *i.e.*, we do not want to (needlessly) reinvent the wheel.

On the other hand there is a requirement to enable efficient implementation in low-end storage devices. These devices are resource poor and the developers of these devices do not want to add additional hardware without a clear justification. These devices will not always support standard network security and in such environments it is necessary to provide end-to-end security against attacks without depending on an external mechanism to secure the network.

We have defined the following set of requirements that must be met by the OSD security model. We distinguish in defining these requirements between access control security (security which is directly tied to the semantics of object storage) and network security (security which is related primarily to network protocols and could be handled separately from the semantics of OSD). The requirements we define are:

- Must prevent against attacks on individual objects. Such attacks include both intentional and inadvertent access to an object in a way not authorized by the security manager. In particular, we must address malicious hosts forging or modifying a credential, a host stealing a credential from the channel between the object store and client,[9] *etc*.
- Must enable protection against attacks on the network such as man-in-the-middle (*e.g.,* a computer posing as an object store), replay, *etc*.
- Must provide a stand-alone solution that works in the event there is no existing network security infrastructure or for whatever reasons the implementer desires not to use an externally secured network.
- Must provide a solution that can use an existing standard network security infrastructure.
- Must not duplicate the cost of security, where it can be avoided. *e.g.,* if the host is running over a secure network with Level 1, it should not incur a higher overhead than a host running over a non-secure network with Level 3.
- Must allow low cost implementation of the critical path.
- Must be simple. In particular, we should use the same structures and same message flow across all the protocol levels.
- Should allow efficient implementation on existing network transports.

1.4 Limitations in the Proposed Version of Object Store Protocol

The version of the protocol defined in the following sections of this document is a first step towards OSD security. As such, it has the following limitations:

- It does not internally support privacy on the channel between the object store and the client
- It does not support privacy for the data at rest
- It requires a communication channel between the object store and the security manager. This channel must be capable of carrying authenticated and encrypted messages.
- Ability to define capabilities that apply to multiple objects where the object to which a capability applies is defined by a predicate on the object's attributes. Note this is not the same as commands which apply to multiple objects.
- Ability to define a capability which applies to only a portion of an object or to only certain object attributes.
- It does not provide a means of determining from the object store what security level should be used.

[9] As stated above, the assumption is that the channel between the host and the OSD is not encrypted, and thus it is possible for a malicious host to eavesdrop on this channel.

2 Structure of Credentials and Basic Message Flow

2.1 Introduction

To enforce legitimate use of capabilities, the client receives from the security manager (over a secure channel) both the *capability (CAP_Args) and* some associated secret information, a *capability key (CAP_Key)*. Together the capability and capability key are the credential. The client sends a capability to the object store as part of each request. The client uses the capability key to compute a *validation tag*, which it appends to each request. The structure of this validation tag depends upon whether an existing network security infrastructure is being used, or whether the network security is provided internally by the protocol. Among other semantics depending upon the security level, the validation tag ensures the capability has not been modified. Using the protocol appropriate for the security level, the object store validates the validation tag and checks whether the operation requested by the command is indeed permissible. Note that the object store does not need to authenticate the client or to have a notion of "client identity".

2.2 Cryptographic Building Blocks

The cryptographic primitive that is used throughout this protocol is a keyed message authentication code. The protocol uses an HMAC-SHA1 [9][10] whose output is 160 bits long. When applicable, the final output of 160 bits is truncated into 96 bits. The HMAC-SHA1 key is 160 bits long. The means by which the HMAC-SHA1 key is generated is not specified by the protocol. Later versions of this protocol may allow an object store to specify alternate cryptographic primitives (see section 8.8).

2.3 Key Management Overview

The credential is based on a secret key that is shared between the object store and the security manager. For each object store s_j, $K_{secret\ key\_j}$ is an authentication key shared between s_j and the security manager. For clarity, when concentrating on a specific object store we omit the index j where no ambiguity arises. In particular, $K_{secret\ key}$ is a 160-bits long SHA1 key. More accurately, there is a hierarchy of keys shared between the object store and the security manager.

This protocol exchanges a secret key between the object store and the security manager:

1. The security manager sends a secret key to the object store along with the key's version number.
2. The object store updates its key, removes any cached credentials established with the previous key, and acknowledges receipt of key.

In chapter 8, we elaborate on the hierarchy of keys and the protocol for exchanging keys.

In a later version of the protocol we may define a mechanism for piggybacking the exchanges of keys over the client-object store channel without requiring a separate channel for the communication between the object store and security manager. As we describe below, since a channel between the object store and security manager is needed for other reasons, we take advantage of this channel for the key exchange.

2.3.1 Maintaining two valid keys $K_{secret\ key}$ simultaneously

A key refresh event between the object store and the security manager invalidates all credentials at once. This results in heavy communication traffic between all clients and the security manager; moreover, all new credentials must be explicitly validated (via MAC calculation) before being cached. This phenomenon may cause undesired performance degradation after the key refresh. To mediate this effect, we allow an object store to declare the last two (or more generally n) refreshed versions of $K_{secret\ key}$ as valid, instead of just the latest one. As a result, the process of validating a credential requires a *key_version* field in the credential to enable the object store to know which key to use in validating the credential.

The number of key versions used is configured between the OSD and the security manager. The OSD implementation can specify the maximum number of key versions it supports; one is a legal value. The maximum number of key versions supported by the protocol is 16.

2.3.2 Partitions

An object store is divided into multiple partitions, each of which carries its own keys for security purposes. Instead of having a separate secret key for each object store s_j, there is a distinct secret key for each two-tuple of object store s_j and partition p_k.

All commands other than key exchange commands (see chapter 8) come with credentials which are protected by the key associated with a specific partition. For most commands, e.g., those that operate on a specific object, the partition used is 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, we use the keys associated with partition zero. Since the commands that operate at the level of the object store and not at the level of individual objects are by their nature very powerful, we want to limit the use of the keys associated with credentials used to execute these commands. We thus define that partition zero should not contain user objects; in addition, to solve the problem of bootstrapping, an object store must always contain a partition zero (e.g., to allow formatting the object store). We note that a realization of the object store standard can define an identity between the root object and partition zero.

2.4 Capability Argument and Capability Key

Define:

- *Type* of the credential (4 bits), which must currently all, be zero. This is intended to allow future extension to different types of credentials.
- *MAC Function* is a four bit field indicating the cryptographic primitive used to construct the credential. In the initial version of the protocol, the value of this field must be zero and the HMAC SHA-1 must be used (see section 2.2)

- *Partion ID* is the identity of the partition for which this capability is being generated. Note we do not include the object store ID in the capability under the assumption that it is passed on all commands as part of the addressing.
- *Capability Nonce* to be an *l*-bits nonce (*l*=128) chosen uniquely by the security manager for each credential. The *nonce* may be a counter. We do not specify the means of generating this nonce, leaving the mechanism up to the implementer of the security manager. The role of this nonce is twofold: *1)* to ensure that every credential generated by the security manager is unique; this prevents a host from masquerading as an OSD to another host, which would be possible if both hosts received the same exact credentials and *2)* to serve as an audit field for allowing management applications to track the client which received a capability.

  This nonce has the following structure

  - *Audit tag* is a 32 bit value which the security manager uses in an implementation defined way to associate a credential with the client to which it granted the credential. The correctness of the system will not be dependent upon the value the security manager places in this *audit tag*. However, the overall performance and usability of the system can be improved if this field is used as a audit tag. This field can be used for purposes of auditing and report generation. It can also be used by an object store to better manage nonces in level 2 and level 3 of the protocol.
  - *Random bits* a 96 bit value which must be unique across all credentials with the same audit tag and values for the other fields.
- *Rights string* specifies the rights and object(s) to which they apply. At this point we propose the following structure for the rights string:[10]
  - *Type* – the implementation of the rights string; this is four bits with the following values
    - 0 – a specific object and set of operations is specified
    - 1-15 – reserved
  - *Operations* – a bitmap with one bit per OSD command; this bitmap should contain additional reserved bits for potential extension, without requiring a change in the size of credentials.
  - If the type == 0, then the following additional field is defined
    - *Object* – the local ID of the object to which this command applies.[11]
- *Object Version Tag* – a *k*-bits value (*k*=32) that is maintained as an attribute for each object. It is used to invalidate credentials, which have been issued earlier for the same object. If the security manager wishes to invalidate all credentials it had previously generated for an object, it modifies the value of the attribute associated with the object (see section 7.1); the new value should never have been previously used in a credential for this

---

[10] The size of the rights string is the sum of the sizes of its component fields with any necessary padding.

[11] The space required to encode the local ID will be used for pattern matching on attributes for future types of credentials to be defined.

object ID.   To allow resumed access to the object, the security manager should use this new value in future credentials it generates for this object .

- *Creation Time* – the time the object was created provided as an attribute by the object store. If object IDs are reused, then two creates for an object in a given partition which use the same ID must have different values for the create time. Note, it is clearly acceptable for this value to be unique for every object created in an object store. The size and resolution of this value will be as defined for the creation time attribute of objects.
- *Key_version* – a four bit index indicating the key version of $K_{\text{secret } key.}$ The key version is set at every key refresh between the object store and the security manager. See also section 2.3.1 and chapter 8.
- *Expiry Time* – a 48 bit field giving the time the credential expires in milliseconds since January 1, 1970. The security manager should generate this time. By using an expiry time we allow the security manager to give different lifetimes to different credentials. We assume a weakly synchronized clock between the security manager and the object store. No assumptions are made on the client's clocks. The OSD should not accept a capability with an expiry time in the past.

The credential *C* that the security manager issues for a client is comprised of two components, a "public token" *CAP_Args* and a "secret extra information" *CAP_Key*.

CAP_Args º [rights string, Key_version, Nonce, Object Version Tag, creation time, expiry time, partition ID, object store ID]
CAP_Key º $MAC\_K_{secret\ key}$ *(CAP_Args)*

CAP_Key is the 160-bits long output of the HMAC-SHA1 computation on CAP_Args along with the implicit parameters of the object store ID and partition ID. Note that CAP_Key cannot be truncated (to 96 bits) as it is used later in the protocol as a key to another MAC computation. It is the host's responsibility to keep CAP_Key secret; if CAP_Key is compromised, than it is possible for an adversary to issue requests using the capability if it determines CAP_Args, which are passed on the wire between the OSD and host in the clear.

We note that not all of the fields in the CAP_Args need to be passed explicitly on the wire.   In particular, since the object store knows the creation time and desired version tag for each object, it is not necessary to pass these values. Instead, given the object ID, the object store can determine which object version tag and creation time to use in calculating the CAP_Key. If the host had a credential created using different values for these fields, a MAC calculation would fail and the command would be rejected.

In a similar vein, the partition ID and object store ID do not need to be passed as part of the capability for each command. This is because these fields are part of addressability and will need to be passed as part of the basic command (even if running with no security). In other words, there is no need to pass partition ID and object store ID twice.

The precise treatment of the object version tag, creation time, partition ID and object store ID will be defined by each realization of a concrete object store standard, however, we recommend that they not be passed as part of a capability on each command.

Since the credential includes information which is stored as attributes for the objects (namely the creation time and version tag), we may have a problem of bootstrapping, in particularly 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.,* all object IDs assigned by an external cataloging entity, the use of the creation time may require additional message exchanges and provide no benefit.

To address this, if a credential generated by the security manager uses zero for the version tag/creation time, then when calculating the CAP_Args the object store should not take into account the actual value of the respective attribute associated with the object but rather will use zero (of the appropriate number of bits). When used with the version tag, this essentially creates a credential which cannot be invalidated (other than by a key exchange which invalidates all credentials for the partition generated with the same working key). Note that if a realization of this work as a concrete standard does not pass the complete values of version tag or creation time with each command (see above), it must pass an indication of whether or not these fields should be treated as zero.

To delegate a credential *C* to another host, a host must transfer both the CAP_Args and CAP_Key. While beyond the scope of this protocol, to ensure security, such delegation should be done over an encrypted channel.

2.5 Anonymous Object Creation

To support creating an object where the OSD provides the object ID, the security manager should generate a capability in which the object ID embedded in the rights string is zero and the only right specified in the operations bitmap is object creation. The OSD must not allow such a capability to be used more than once. To minimize the memory requirements the OSD must dedicate to ensuring that such capabilities are used at most once, it is strongly recommended that the security manager construct such capabilities with expiry times very close to the current time.

2.6 Message Flow

Prior to sending an object store command to a target, the client must request the credential from the security manager and in return the security manager sends back both the public part of the credential, *CAP_Args*, as well as the private part, *CAP_Key*. *CAP_Key* should be sent to the client over an authenticated and encrypted, channel to maintain its secrecy. To establish this channel (and also to let the security manager identify the client), the client and the security manager should authenticate each other in a preliminary step. The implementation of this channel and its protocol are not part of the object store protocol.

When the client executes the actual object store command, the object store should validate:

1. The integrity of the public credential *CAP_Args*
2. That the public credential *CAP_Args* is used by a client that legitimately received it.
3. The integrity of the command itself (command and data), as required by the security level.

For that, the client sends, along with the command, the public credential *CAP_Args* along with a MAC-based validation tag, which is computed using *CAP_Key*. Since *CAP_Key* can be computed from *CAP_Args* and the secret shared between the security manager and the object store, the validation tag is also computable by the object store.

The structure of the validation tag and its usage depends on the security level being used. Section 3 describes the validation tag if we assume an external mechanism for the integrity of data and command, namely an authenticated channel such as an IPSec authenticated channel. In Sections 5and 6 no such external mechanism is assumed and therefore the validation tag as well as its validation at the object store is more elaborate.

2.7 Credential Invalidation

As described above, the object store protocol provides two means for invalidating a credential. By the use of object version tag in each credential, we can invalidate all of the outstanding credentials for an object. By a key exchange between the security manager and the object store we can invalidate all credentials a security manager had generated for a particular object store partition. Note, by explicit decision, we have decided not to support an efficient means of externally invalidating all of the credentials given to a particular host by the security manager (but see section 4.4).

2.8 Security Related Error Status

The following error responses related to security can be returned by the OSD to the host. Some of these responses are limited to specific security levels as indicated:

- *NOT_SUPPORTED_CREDENTIAL_TYPE* – the type of the credential is not supported by the object store.
- *CAPABILITY_MISMATCH* – the requested operation is not allowed by the rights string
- *INVALID_MAC* – the message authentication code (MAC) included in the request is not consistent with the credential included in the message; in other words, the *CAP_Key* calculated based upon the credential cannot be used to compute the same MAC as the one included in the message. In the event that the MAC is invalid either this error or INVALID_KEY must be returned (regardless of other errors detected).
- *INVALID_VERSION* – the request includes a credential with a object version tag which is no longer being accepted
- *INVALID_KEY* – the key as indicated by key_version in the credential is no longer valid; the host must retrieve a new credential from the security manager prior to retrying the

operation. An OSD implementation does not need to be able to distinguish this situation from the situation reported by *INVALID_MAC;* in which case it should report *INVALID_MAC*.

- *EXPIRED_CREDENTIAL* – based upon the expiry time, the credential has expired.
- *INVALID_NONCE* – the nonce does not contain a valid timestamp; a recommended time stamp will be returned with this code. This may only be returned for level 2 or level 3.
- *NONCE_NOT_UNIQUE* – a message with this per request nonce has previously been seen by this object store. This may only be returned for level 2 or level 3.
- *CAPABILITY_BLOCKED* **–** The capability was blocked, *e.g.*, based on the capability audit tag. Note that the reason for the "blockade" is not given. This may only be returned for level 2 or level 3.

In addition to these error responses which are specific to security, the following additional errors, which we are not specifically related to security, can be returned:

- *INSUFFICIENT_RESOURCES* – a temporary condition exists which does not allow processing this request due to a lack of resources
- *INVALID_MESSAGE_STRUCTURE* – the structure of the message is not syntactically valid.

3 Level 1 – Integrity of Capabilities

This security level is useful in two scenarios: *1)* where no network attacks are expected to take place (such as a 'glass house' scenario) and *2)* where an authenticated channel between the host and the OSD is assumed. The mechanism for establishing this channel is beyond the scope of the OSD protocol.

3.1 Level 1 Security with Authenticated Channel

For Level 1 security with an authenticated channel, the channel provides integrity of messages as well as an anti-replay mechanism (for the given channel). The OSD-specific protocol prevents copying messages from one channel to another by tying the message to the channel via a *validation tag*; this tag is computed as $MAC_{CAP\_key}(ChannelID)$, where *ChannelID* identifies the communication channel. Given that the OSD knows the channel on which a request was received, the OSD can validate that the $MAC_{CAP\_key}(ChannelID)$ included in a message is for the *ChannelID* associated with the channel on which the message was received. The same validation tag can be used with all requests based upon a given credential.

We do not need this validation tag on OSD responses since *1)* the authenticated channel ensures the host any responses it receives are received from the intended OSD and *2)* we trust the OSDs to not copy messages between channels (see section 1.1.2).

The *ChannelID* is a name for the channel that is unique to this channel between the client and the object store and is known to both ends. The size of the *ChannelID* is transport dependent. The lifetime of the *ChannelID* is no greater than the lifetime of the channel;[14] the lifetime of the *ChannelID* is independent of the lifetime of the key $K_{secret\ key.}$ See section 3.3 for a more precise definition of the assumptions on the channel and *ChannelID.*

Most likely, a value that can be used as a *ChannelID* already exists and was created at the time the channel was established. Otherwise, it requires another message exchange between the client and target, where:

1. Client requests 'open security window' with the object store.
2. Object store responds with a randomly chosen *m*-bit channel name *ChannelID*.[15]

At this point we do not architect such a flow to explicitly have the object store provide the *ChannelID* but rather we assume the channel provides the *ChannelID*.

Below is the protocol flow of messages along with a table that explains the messages and their corresponding arguments. Note that the *OpenWindow* message is *not* needed for every *ReqCap* message. Furthermore, it may not be needed at all if a *ChannelID* is already exchanged.

[14] Note it would be permissible to change the *ChannelID* for an existing channel; this would invalidate cached credentials.

[15] Analogously, a 'close security window' clears knowledge of the session at the object store.

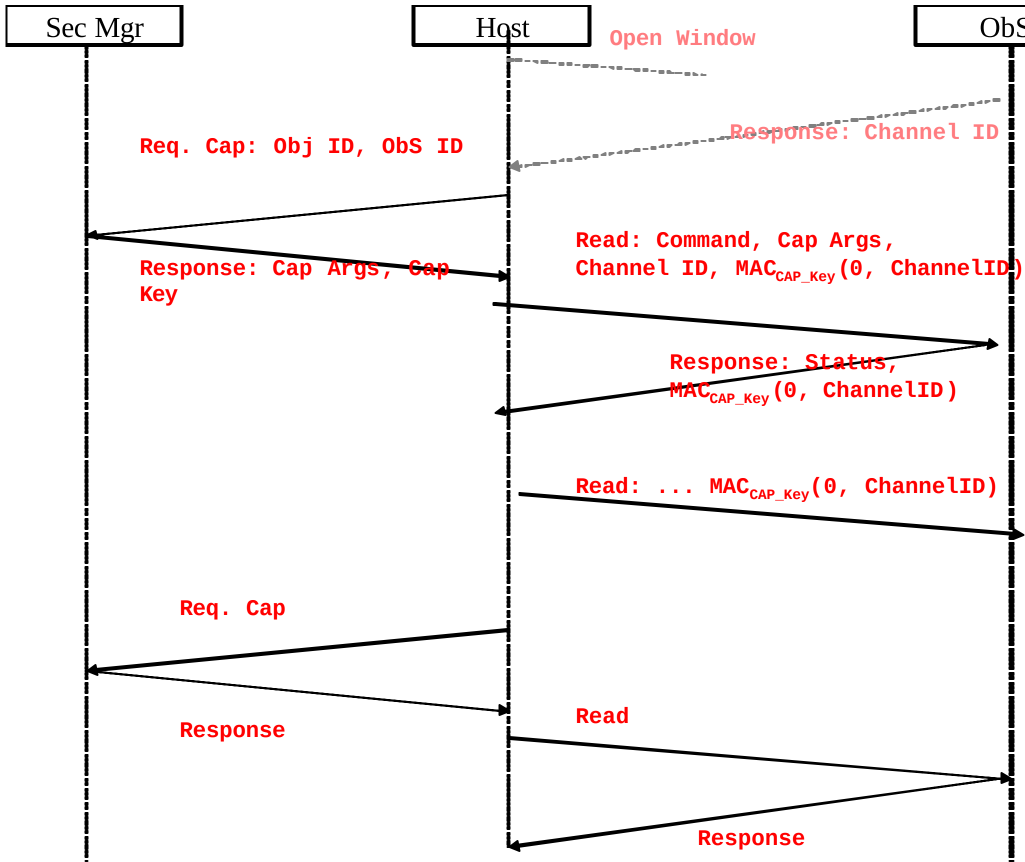


Figure 2. Flow of Messages for Level 1 Security. The establishment of the authenticated channel is not shown. The *OpenWindow* exchange will not be required for most channels.

Message	Argument	Explanation
ReqCap	ObjID	Object identifier
	Partition ID	Partition ID
	ObSID	Object Store identifier
ReqReturn	Version	
	Rights } **CAP-Args**	
	Expiration	
	Partition ID	
	Creation	
	Capability Nonce	
	CAP_Key	MAC_{secret_key}(CAP_Arguments)
OpenWindow		
WindowReturn	ChannelID	
ReadData	ObjID	
	Partition ID	Partition ID
	ObSID	Object Store identifier
	CAP-Args	

	Offset	
	Length	
	Nonce	Ignored (all zeros)
	ReqMac	MAC_{CAP_Key}(ChannelID)
ReadReturn	Status	
	RetMac[16]	MAC_{CAP_Key}(ChannelID)

3.2 Level 1- security without network security

As noted above, this level of security is also useful when no network attacks are expected to take place. In the event no secure network infrastructure is used, level 1 security protects the integrity of the capability. The protocol is identical to the one described above. However, since the *ChannelID* is not in practice tied to a channel and there is no true means for tying a message to the channel. An OSD implementation cannot, however, ignore the value of the validation tag if level 1 is being used without a secure network infrastructure since the validation tag is also used to validate the capability has not been modified. In this case, zero should be used as the value of the *ChannelID*.

3.3 Assumptions on Network Infrastructure for End-to-End Security

We place the following requirements on the channel and *ChannelID* if we want to ensure and end-to-end security solution using level 1 of OSD security:

- Within the lifetime of a key, $K_{secret\ key,}$ all channels established with a given object store from any host must receive unique channel IDs.
- There must be a means for the host and OSD to associate a received message with the *ChannelID* for the channel on which the message was received.
- Assuming that the channel provides the value of the *ChannelID,* this value must be non-forgeable.
- The channel must be authenticated (although it may be anonymous) in the sense that it must ensure both parties can be guaranteed all messages in a session come from the same party.

The channel must ensure message integrity, *i.e.,* non-modification of message contents by the network.

3.4 Client-Object Store Message and Flow

1. Client sends a command to the object store, along with the public token *CAP_Args* (defined above) and a 96-bits long validation tag $V = MAC_{CAP\_key}$ *(ChannelID). V* is computed using HMAC-SHA1 on the *ChannelID,* truncated to 96 bits.

[16] As discussed above, this MAC is not necessary – is it only used for symmetry with level 2 and level 3 security

2. Verification at object store:

   1. The validation tag *V* equals MAC_{CAP_key} *(ChannelID)*, where *CAP_Key* is obtained as $MAC_{K\_secret\ key}$ *(CAP_Args)*.
   2. The rights string in the *CAP_Args* allows the requested operation
   3. The *key_version* is current.
   4. The capability's *Version Tag* is either zero or equal to the version tag attribute of the object. [17]
   5. The capability's *Creation Time* is either zero or equal to the creation time attribute of the object.

   If any of the checks fails, the request is denied. If checks (a) and (c) pass, the object store may cache the token *CAP-Args* associated with channel *ChannelID*.

An object store implementation may cache the validation calculations. In particular, if *CAP_Args* has ever been presented to the object store on this channel within the lifetime of current *ChannelID*, the request may be granted without re-validation (*i.e., without redoing step 1)*. The authenticated channel assures that another client is not replaying *CAP_Args* on this channel, rather it is currently presented by the same entity that presented it in the past, and hence a re-validation is not necessary.

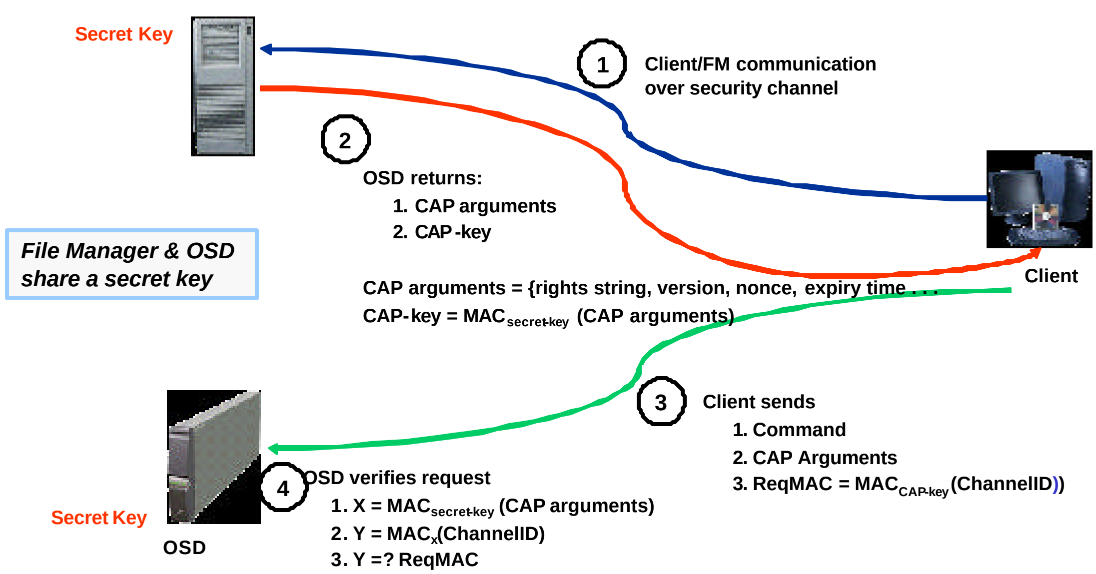


Figure 3. Flow for Level 1 Security

[17] This check can be implicit in checking the validation tag as the object version tag is part of the CAP_Args and if the version tag is incorrect, the object store will not be using the same CAP_Key as the host. This comment applies as well to the creation time. It also applies to the other security levels. It is not repeated.

3.5 Performance Considerations

For level 1 security, we have the following performance considerations:

- A client does not need to request a new credential on every command; rather the client can reuse the *CAP_Args* and *CAP_Key* on multiple commands for the same object(s).
- The client does not need to recalculate the ReqMAC on each command; rather, this needs to be calculated only once per credential.
- The object store does not need to recalculate X and Y 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.

4 Per Request Nonces for Level 2 and Level 3

Level 2 and Level 3 of the security protocol use Nonces included in each request to prevent replay. The requirements for correctness of a nonce-based approach to preventing replay are as follows:

- The object store must not accept the same nonce more than once.
- The object store must not accept a nonce that was rejected in the past.[18]

In addition, it is acceptable for an implementation to reject valid requests with unseen nonces if necessary to ensure that the two basic requirements are met.

There are three main means of generating nonces:

- Random
- Session based
- Time based

We believe it is fairly easy to argue that the time-based protocol has better performance and space requirements than either a session-based or random generation protocol if all entities in the system are well-behaved. On the other hand, the time-based protocol can have extremely large memory requirements or require frequent changes of the secret keys if enough clients in the system are not well-behaved. In addition, assumptions on strong clock synchronization between the clients and object store are problematic both from a practical and security perspective.

The approach to nonces we define is a time-based approach modified to have only weak dependencies upon client clocks and augments to minimize the impact of poorly behaved entities.

4.1 Background

We define a system running level 2 or level 3 security as *well-behaved* if at any given time t, the total number of *far-in-the-future messages* (messages with a nonce whose time is greater than $t + d$) which have been sent to an object store from all clients, is less than k, for implementation-defined values d and k. In other words, a system is well-behaved if the number of far-in-the-future messages, which an object store has received, is bounded. Similarly a client running level 2 or level 3 security is defined as *well-behaved* if it does not send any nonces for a time greater than $t + d$ where t is the current time of the target object store. An *ill-behaved client* and *ill-behaved system* have the obvious definitions. Malicious intent is not required for a client to be *ill-behaved*. Also note that if there is malicious intent, the maliciousness is not necessarily directly from the ill-behaved client; for instance, a malicious time-server can causes clients to be ill-behaved.

In the worst case, with the time-based nonces, an object store implementation must ensure that it has sufficient memory[19] to remember the nonce from each message it could receive in the

[18] This holds regardless of the reason the message was rejected

period between working key exchanges. This is to prevent messages from being replayed: the implementation must also ensure that any message which has ever been rejected will never become valid in the future. In other words, given an object store which can handle *n* messages a second and a key exchange every *e* seconds, the object store needs to be able to remember *ne* nonces.[20] This, admittedly unlikely worst case, would occur if every message received was for the end of the period in which the key was valid. Note an alternative would be to allocate a fixed amount of memory, much less than for *ne* nonces and if this memory fills up, for the object store to force a key exchange. This leaves open a denial of service (DOS) attack in which all existing capabilities are invalidated. The goal of our modifications to a pure-time-based nonce protocol is to reduce the easy of this DOS attack

One way to mitigate the amount of memory required to handle ill-behaved systems[21] is to design the messages in such a way that the object store would be able to reduce its memory requirements by organizing the nonces into groups. If the far-in-the-future messages are limited to a subset of the groups of nonces, the implementation can decide to reject nonces belonging to the problematic groups, while continuing to accept other nonces. Clearly, the efficiency of such an approach depends on the accuracy of the grouping. We leverage the audit tag field of the nonce[22] in the *CAP_Args* for this purpose; see section 2.4.

4.2 Requirements

In addition to the general requirements listed above, we place the following requirements on the protocol:

- The changes to allow better behavior in ill-behaved systems should incur no additional cost in the case of a well-behaved system.
- An implementation that chooses so must be able to bound the amount of memory required for correct behavior independent of the frequency in which the key is exchanged between the object store and security manager (*i.e.,* safety with bounded memory), while sill ensuring liveness for well-behaved clients in many scenarios where there are ill-behaved clients.
- We must allow freedom to the object store implementer to trade-off between implementation complexity and overall system behavior in the event that clients are not well-behaved.

4.3 Structure of the Per Command Nonce

When working with the time-based nonces, on each request, the host generates a nonce by combining a 48-bit time representing the number of milliseconds since January 1, 1970. The nonce also includes a 48-bit random number.

---

[19] Clearly various compression techniques could be used; for example see [11].

[20] This is the number of nonces that must be remembered; the memory that is required is implementation dependent and may need to take into account compression techniques.

[21] Although there are still scenarios in which correct behavior entails either remembering all nonces or forcing a key exchange.

[22] Not to be confused with the per command nonce described in this section

4.4 Use of Nonce for Anti Replay

Define the *current interval* for an object store whose clock currently is at time t as the period of time beginning with $t\text{-}d_1$ and ending with $t+d_2$, where d_1 and d_2 are values determined by the object store implementation. The current interval defines the time-based nonces the object store expects to receive from well-behaved clients. The object store can accept any valid request received in this time range. To prevent replay, the object store must have sufficient resources to remember all nonces seen in this range. Messages received with nonces less than $t\text{-}d_1$ do not need to be remembered.[23] Define as *far-in-the-future* a nonce for a time greater than $t+d_2$. By definition, such nonces will only be sent by ill-behaved clients.

When the object store receives a request in level 2 or level 3 with a nonce in the current interval, the object store must remember the nonce in a *current interval nonce list*.[24] While the only requirement from the protocol is that anti-replay be provided, the space allocated to the current interval nonce list should be sufficient[25] to hold the number of nonces that can be received by the object store during the time of the current interval, *i.e.,* a function of the size of a nonce (12 bytes) times the number of messages the object store can receive in time $d_1 + d_2$.

Note, the object store must remember the nonce even if the message fails verification of the MAC. This is required to prevent the following, replay-like attack. Assume an adversary hijacks a request to the object store, modifies the command portion of the request and forwards the request to the object store. The object store will send an *INVALID_MAC* error response to the client. The client may then decide to regenerate the request with a new nonce and MAC. Assuming this request is executed, the adversary can now replay the original request. We should point out that a client should be suspicious of an *INVALID_MAC* response which does not itself contain a valid MAC.

If the nonce in a request is for a time that is older than the current interval, the object store rejects the request without further processing with an *INVALID_NONCE* error message. The *INVALID_NONCE* response includes the current time of the object store, allowing the client to try again with a nonce that will fall in the current interval. The object store does not need to remember the nonce. A well-behaved client will (logically) reset its clock to be that of the object store for future messages it sends.

Finally, if the nonce in a request is a *far-in-the-future* nonce, the object store must remember the nonce in the *far-in-the-future nonce list*.[26] The object store implementation may reject the command with an *INVALID_NONCE* status or it may decide to process the request as described for messages received with a nonce in the current interval, as long as the nonce uniqueness is guaranteed.[27] If an *INVALID_NONCE* response is returned, as above, it will

[23] Note we assume that if the clock of an object store is set backwards, a key exchange with the security manager will also take place.
[24] Note, the reference to a current internal nonce list is for explanatory purposes only; an implementation may choose any mechanism to remember previously seen nonce as long as the basic requirements are met.
[25] After any compression techniques
[26] Note, the reference to a far-in-the-future nonce list is for explanatory purposes only; an implementation may choose any mechanism to remember previously seen nonce as long as the basic requirements are met.
[27] But this does not enable the client to be informed that it should update its clock

include the current time of the object store and a well-behaved client will logically reset its clock to be that of the object store.

We define the size of the *far-in-the-future nonce list* to be large enough to hold some number, *k*, of nonces,[28] where *k* is implementation dependent, not specified by the protocol, and may vary at different times for a given implementation. Clearly, a nonce can be removed from the far-in-the-future nonce list when the nonce represents a time prior to the start of the current interval. If an implementation ensures the basic requirements, a nonce can be removed from the far-in-the-future nonce list at other times.

To verify that a nonce has not previously been seen, the object store must look in both the current and far-in-the-future nonce lists.[29]

If the object store receives more than *k* far in the future nonces, *i.e.*, the object store has run out of resources to remember far-in-the-future nonce, the object store implementation has several options, as long as it guarantees the basic requirements of not accepting the same nonce more than once and not accepting a nonce that was previously rejected.

One option, the "big hammer" option, is for the object store to refuse to accept any more messages using the same working key which was used for the capabilities in the messages with the far in the future nonces. In this case, the object store may return an indication of *INVALID_KEY* when it receives requests with this working key. It is implementation dependent as to how the security manager is notified that the working key needs updating. Options include (but are neither limited to, nor required to include) having the security manager poll the object store and having the client pass on an indication to the security manager.

The drawback of the "big hammer" option is that it invalidates all capabilities whose corresponding credential was created with the given working key. In other words, all clients which have capabilities for the given object store partition created with the same version of the working key are impacted.

To mitigate the likelihood an implementation needs to resort to the big hammer, the implementation can organize the far-in-the-future nonce list based upon the architected audit tag that the security manager places in the credential.[30] One option an implementation can choose is to partition this nonce list based upon the audit tag. For instance, if the object store receives more than *c* far-in-the-future nonces with a given audit tag created by the same working key, the object store can refuse to receive additional requests with the given audit tag until the oldest request in the far-in-the-future nonce list for this audit tag is older than the start of the current interval. If the object store is refusing to receive requests with a given audit tag or capability, it should return *CAPABILITY_BLOCKED*. For this to work, the object store must always remember the *c* newest far-in-the-future nonces received with a given audit tag. In this case, the

[28] Again, this may be after compression

[29] The description of separate current and far-in-the-future nonces lists is for explanatory reasons only; an implementation that ensures the basic requirements need not have separate lists.

[30] The implementation may arrange the *far in the future set* in any manner, e.g., it according to the nonce hash value. However using audit tags is a reasonable choice as they identify the “source” of the attack.

object store only needs to "drop the hammer" if more than *k/c* clients are not well behaved. Other implementations are clearly possibly as long as they meet the base requirements.

We require that *c* be a value that is visible to a client. Clients may send a batch of requests without waiting for a response. In this case, a client needs to be able to determine how many outstanding requests it can send to an object store without risking having the object store decide it is ill-behaved and thus refusing to accept requests from it.

4.5 Host Protocol

To prevent replay of responses, hosts must maintain nonce lists in the same way the object store supports nonce lists

4.6 Use of Time

The only requirement for the time used to determine nonce timestamps is that it be monotonically increasing, although weakly synchronized clocks between the OSD and hosts will avoid additional messages. This time must never go backwards without a key exchange. In order to catch the time up to an external “real time”, the OSD may choose to accelerate or decelerate the passage of time until it has caught up or the “real time” has caught up. Any OSD 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.

4.7 Additional Attributes on Partition Object

To allow implementing a complete solution, an object store implementing level 2 or level 3 security, must define the following attributes on a partition object:

- *NUM_REQS_BEFORE_BAD* – the minimum number of requests which are far-in-the-future which a client may send, prior to the object store determining that the client is ill-behaved.[31] This guarantee will only hold if there are not too many clients sending NUM_REQS_BEFORE_BAD at the same time. Note that one and zero are legal values.
- *WORKING_KEY_FROZEN(i)* – an array of $n=16$ Boolean attributes, where the *i*'th attribute is true if an object store needs to "drop the hammer" and refuse any credentials created with the *i*'th version of the working key (as indicated in the key_version) field of the credential. An OSD sets bit *i* when it, of its own initiative, invalidates working key *i* and an OSD unsets bit *i* when it receives and accepts a key management command that defines a new value for working key *i*.
- *OLDEST_VALID_NONCE* – the minimum number of milliseconds older than the object store's current time a nonce that is received will be considered valid; this attribute maps to the value d_1 defined above. Zero is a legal value implying the absence of information.

[31] This is the "maximum" number of requests that a client trying to be well-behaved can issue without receiving a response from any, and be confident that the OSD will not invalidate the associated working key in the case that its nonces are in fact far-in-the-future relative to the OSD clock.

- *NEWEST_VALID_NONCE* – the minimum number of milliseconds newer than the object store's current time a nonce that is received will be considered valid; note an object store implementation may decide to treat as valid nonces that are even newer than this. This attribute maps to the value d_2 defined above. Zero is a legal value implying the absence of information.

5 Level 2 – Integrity of Arguments

This security level does not make any assumption about the security of the underlying network and internally provides end-to-end protection for the arguments at the level of the OSD protocol.

The Host makes a request for a capability to the Manager and the manager returns the credential composed of *CAP_Args* as well as the *CAP_Key*.

The Host then presents the command, including the *CAP_Args* and the *Cmd_Args*, along with the *ReqMac* to the OSD. The *ReqMac* is a MAC using the *CAP_Key* of the *Cmd_Args* and a *nonce* constructed as described in the prior chapter. This ensures that the *Cmd_Args* are not modified in transit. The *Nonce* ensures that the command is not being replayed from some point in the past.

The OSD then verifies that:

1. the *Nonce* is fresh, *i.e.,* it has not been seen before
2. the *Cmd_Args* are compatible with the *CAP_Args* (i.e., the rights string permits the operation)
3. the Version Tag and Creation Time are valid
4. *CapY* matches *ReqMac* as sent by the host

Where *CapY* is calculated using

CapX = a MAC computed using the *secret_key* on the *CAP_Args* (this is the *CAP_Key)*
CapY = a MAC using CapX on the Cmd_Args and the Nonce

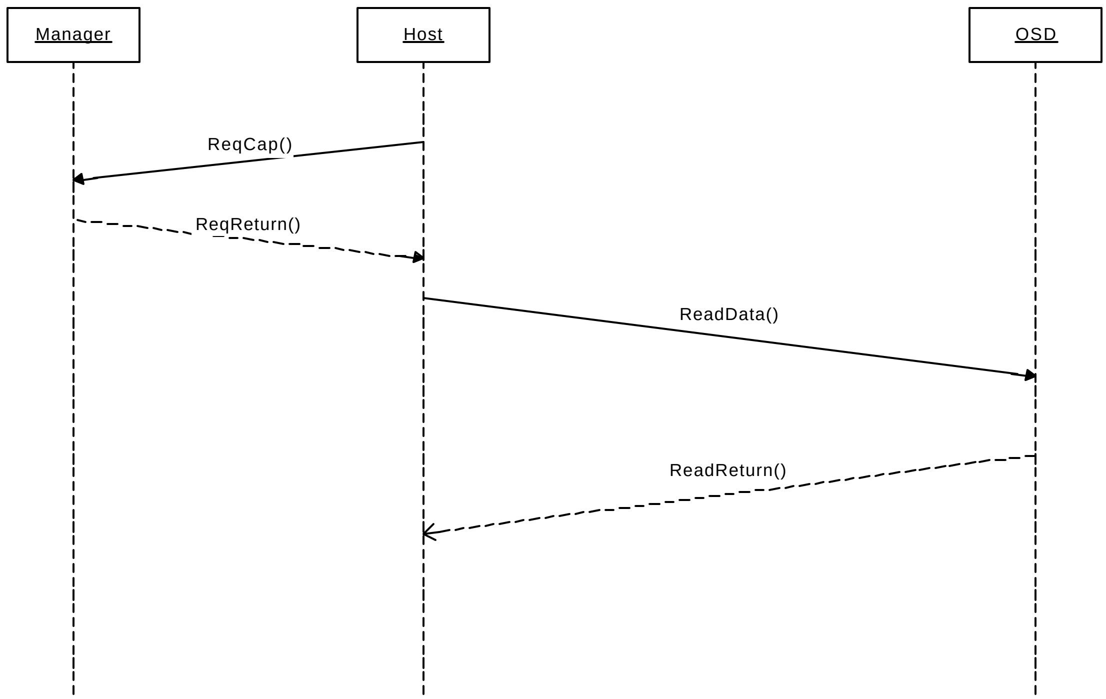
If any of these conditions cannot be verified, the request is rejected and no further command processing is performed other than processing related to the nonce as described above. Nonce related failures are handled as described in the prior section. Other failures are reported with a *Status* as described in Section 2.8.

In all cases:

1. A *RetMac* is computed using *CapX* on the *Status* and the *Nonce* (from the original request) to allow the host to verify the response

Note we can safely apply this MAC to all messages, including with a status of *INVALID_MAC* without becoming susceptible to a black box attack due to the properties of HMAC we are using.[32] See section 2.2.

[32] The Message Authentication Code (MAC) has the Computation - Resistance property [1], namely, given text-MAC pairs (x_i, h_k(x_i)), it is computationally infeasible to compute any other text-MAC pair (x_j, h_k(x_j)) for any new input x_j [3] .



Message	**Arguments**	**Explanation**
ReqCap	ObjID	Object identifier
	Partition ID	Partition identifier
	ObSID	Object Store identifier
ReqReturn	Version	
	Rights **CAP-Args**	
	Expiration	
	Partition ID	
	Creation	
	Capability Nonce	Capability nonce
	CAP_Key	MAC_{secret_key}(CAP_Arguments)
ReadData	ObjID **CmdArguments**	
	Partition	
	Offset	
	Length	
	Nonce	Per command nonce
	CapArgumentsCAP_Args	
	ReqMac	$MAC_{CapKeyCAP\_Key}$(ObjID, Offset, Length, Nonce)
ReadReturn	Status	return code from the request, success or failure
	RetMac	$MAC_{CapKeyCAP\_Key}$(Status, Nonce)

5.1 Performance Considerations

For level 2 security, we have the following performance considerations:

- A client does not need to request a new credential on every command; rather the client can reuse the *CAP_Args* and *CAP_Key* on multiple commands for the same object(s).
- The object store does not need to recalculate *CapX* on each exchange with a client.

6 Level 3 – Integrity of Arguments and Data

This security level does not make any assumption about the security of the underlying network and provides end-to-end protection at the level of the OSD protocol. In addition to the protection of Level 2, this level also includes integrity checking of the data portion of the command.

The Host makes a request for a capability to the Security Manager and the host returns a credential, namely the *CAP_Args* as well as the *CAP_Key*. As in level 2, the Host then presents the command, including the *CAP_Args* and the *Cmd_Args*, along with the *ReqMac* to the OSD. The *ReqMac* is a MAC using the *CAP_Key* of the *Cmd_Args* and the *Nonce*.

In addition, the *DataMac* is a MAC using the *CAP_Key* of the *Data* and the *Nonce*. On a WRITE command, the *DataMac* is computed by the Host, on a READ it is calculated by the OSD.

The OSD then verifies that:

1. the *Nonce* is fresh, it has not been seen before
2. the *Cmd_Args* are compatible with the *CAP_Args* (i.e., the rights string permits the operation)
3. the Version Tag and Creation Time are valid
4. *CapY* matches *ReqMac* as sent by the host

Where *CapY* is calculated using

CapX = a MAC computed using the *secret_key* on the *CAP_Args* (this is the *CAP_Key)*
CapY = a MAC using CapX on the Cmd_Args and the Nonce

In addition for Writes the OSD verifies

5. (WRITE) *DataZ* matches *DataMac* as sent by the Host

Where

DataZ = a MAC computed using *CAP_Key* on the *Data* and *Nonce*

If any of these conditions cannot be verified, the request is rejected and no further command processing is performed other than processing related to the nonce as described above. Nonce related failures are handled as described in the section 4. Other failures are reported with a *Status* as described in Section 2.8.

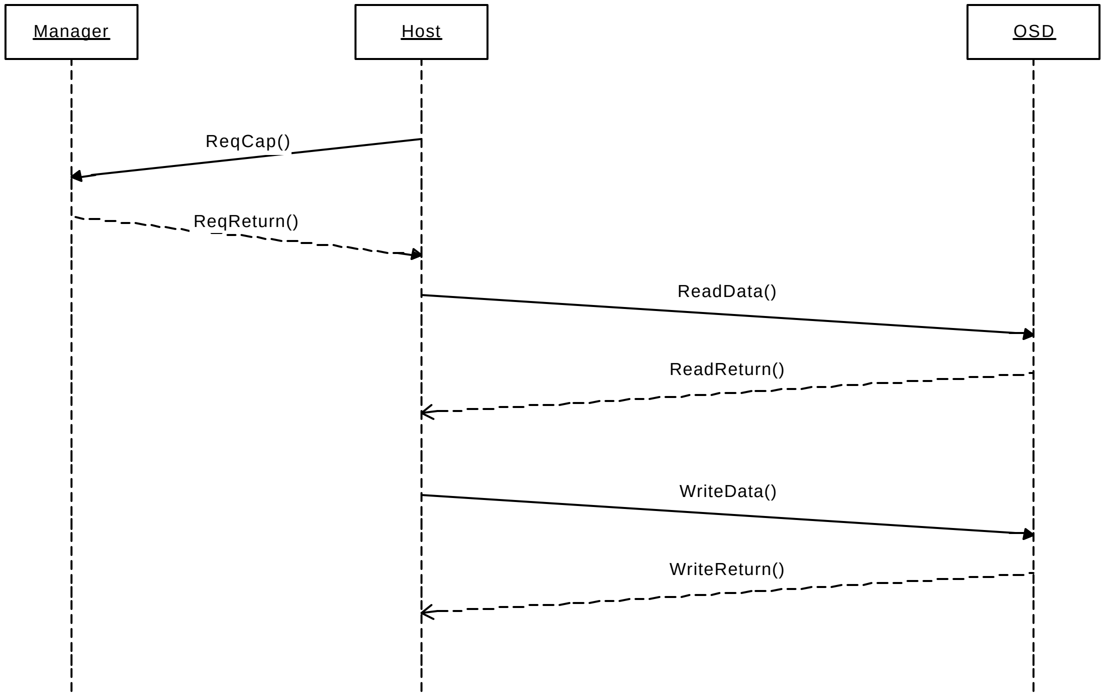
In all cases:

1. a *RetMac* is computed using *CapX* on the *Status* and the *Nonce*

In addition for successful read commands, the OSD returns

2. a *DataMac* is computed using *CAP_Key* on the *Data* and *Nonce*

to allow the host to verify the response.



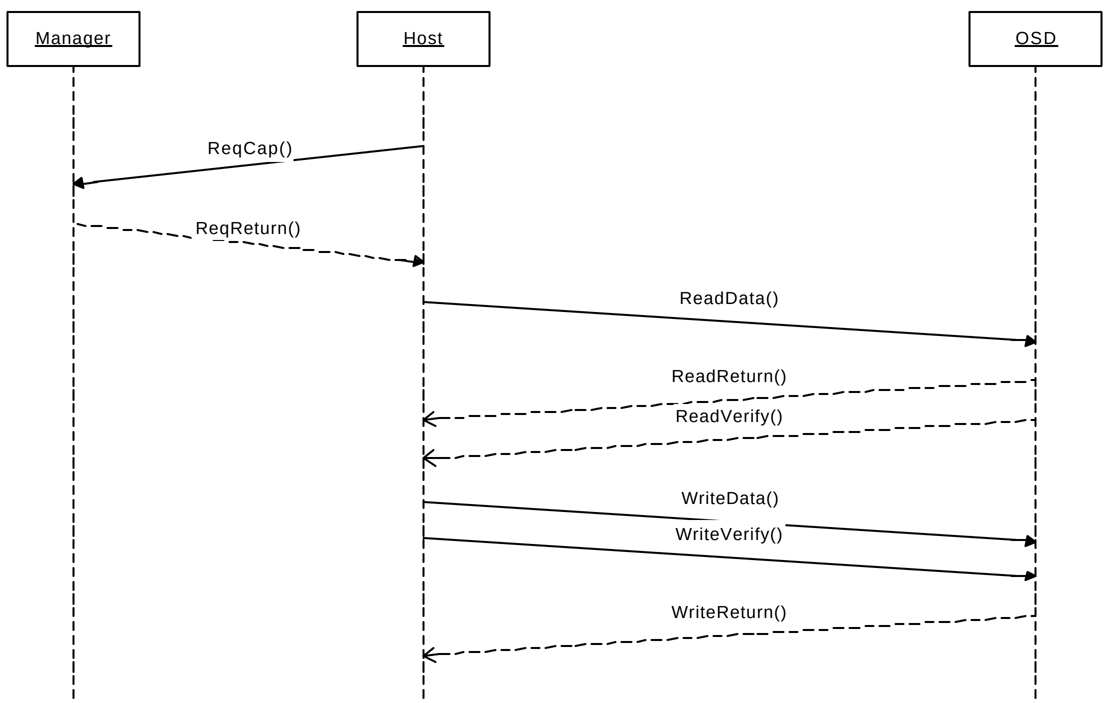
Message	Arguments	Explanation
ReqCap	ObjID	Object identifier
	Partition ID	Partition Id
	ObSID	Object Store identifier
ReqReturn	Version	
	Rights **CAP_Args**	
	Expiration	
	Partition ID	
	Creation	
	Capability Nonce	Capability nonce
	CAP_Key	MAC_{secret_key}(CAP_Args)
ReadData	ObjID **Cmd_Args**	
	Partition	
	ObSID	
	Offset	
	Length	
	Nonce	Time-based nonce
	CAP_Args	
	ReqMac	MAC_{CAP_Key}(Cmd_Args, Nonce)
ReadReturn	Status	return code from the request, success or failure
	DataMac	MAC_{CAP_Key}(Data, Nonce)
	RetMac	MAC_{CAP_Key}(Status, Nonce)
WriteData	Cmd_Args	

	Nonce	Time-based nonce
	CAP_Args	
	ReqMac	MAC_{CAP_Key}(Cmd_Args, Nonce)
	DataMac	MAC_{CAP_Key}(Data, Nonce)
WriteReturn	Status	
	RetMac	MAC_{CAP_Key}(Status, Nonce)

6.1 Implementation Efficiency

The efficient computation of the *DataMac* is straightforward in the case of READ. As data is read from the media, the MAC 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 *DataMac* is sent in the same message as the command, then the Host must make two passes over the data – one to compute the MAC and a second to send the data. In order to avoid this, there must be an additional message as shown in the following.



ReadVerify	DataMac	MAC_{CAP_Key}(Data, Nonce)
WriteVerify	DataMac	MAC_{CAP_Key}(Data, Nonce)

Implementation of this additional message must be supported by the underlying transport in order to achieve the necessary efficiency.

7 Security Manger – OSD protocol

While the precise behavior and policies applied by the security manager are not defined by this protocol, the interactions between the security manager and the OSD are defined.

The OSD treats commands from the security manager in the same way it processes commands received from a host. In other words, these commands must contain a valid capability authorizing the operation. A security manager must use the appropriate level of security as specified for the partition with which it is interacting.

7.1 Invalidation of capabilities for a Specific Object

The security manager can invalidate all previously issued capabilities for a given object by informing the OSD that it should only accept capabilities for the object with a given object version tag. The parameters that must be provided in this command include:

- *Object* – the identity of the object to which this command applies. This should include the partition ID and local ID
- *Object Version Tag* - the value below which no capability will be accepted for this object.

This function will be realized as a set attribute on the indicated object. In addition to allowing set attributes, the capabilitiy provided for this function must include administrative rights.

7.2 Clocks and Expiry Time

The OSD must reject any capabilities that have expired. Since the time placed in the capability comes from the security manager's clock, for the OSD to be able to properly interpret the expiry time in the capability, we require some degree of synchronization between the clocks of the OSD and Security manager.

The protocol for synchronizing the clocks is not specified as part of the object store protocol. The expectation is that a standard clock synchronization protocol will be used; we also believe it makes sense to allow multiple such protocols to be implemented. The specification of the protocol is beyond the scope of this document.

We do, however, assume that this protocol will be implemented in a secure manner, *i.e.,* we do not want an adversary to be able to change the time for the OSD or Security Manager. Such an action could constitute an attack, which increased the effective lifetime of legitimately issued capabilities. Depending upon the implementation, it could also extent the time during which a secret key is used.

8 Key Management

The credential is based on a secret key that is shared between the object store and the security manager. In order to prevent an adversary from obtaining too many credentials generated with the same key, keys must be refreshed regularly. Thus, a key management scheme is required.

8.1 Requirements

- The security manager should be able to replace the object store keys in a secure manner even if the channel it has with the object store is not secure.
- The security manager (or a higher level authority) should be able to divide the drive into multiple partitions. Each partition should carry its own keys for security purposes. Thus, a credential generated for one partition cannot be valid for another.
- A key refresh should invalidate all the credentials generated by that key.
- The key refresh scheme should not necessarily lead to a surge in the communication caused by clients requesting a new valid credential.
- The security manager has a source for random bits.
- The object store is not required to have a source for generating random bits.
- The drive manufacturer cannot assume to know the identity of the drive purchaser.
- The drive manufacturer should not have control over the drive once it is initialized. *i.e.*, the manufacturer should not be able to know the secret keys that are used to generate credentials.
- A drive crash should not necessarily invalidate valid credentials.
- Provisioning a new drive should not require mechanical actions to configure the security mechanism.

8.2 Key Hierarchy

We suggest using the key hierarchy proposed by Gobioff in [7]. The key hierarchy is comprised of 4 layers as described below:

- **Master key** – held by the disk owner. Used to initialize the drive and to create the drive key. This key does not change unless the drive owner is changed. As the top most key in the hierarchy it should be used as little as possible in order to reduce its exposure, and it would be preferable if this key could be immutable as long as the drive does not change owners.
- **Drive key** – held by the disk owner, used to divide the drive into multiple partitions and to create the partition keys. This key is used very rarely and is changed only if either it is (suspected to be) compromised, or the drive owner changes, or a (rare) key refresh operation is carried in order to increase security.

- **Partition keys –** held by the (partition's) security manager. Used solely to create the working keys. The partition keys are changed infrequently, but in a regular manner to increase security.
- **Working keys –** held by the (partition's) security manager. Used to generate the cap-keys. The working keys are refreshed frequently (e.g., on an hourly or daily basis) in order to limit the number of credentials that are generated by the same key.

8.2.1 Master Key

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, it is desired to limit its use as much as possible. Thus, the only use of the master key is to initialize the drive and to set the drive key. This master key does not change unless the drive owner is changed, *e.g.,* the drive is sold. We denote the Master key by **Km**.

8.2.2 Drive Key

The *drive 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 drive key. Once the drive key is set it can be used to divide the drive into partitions and to set the partitions' keys. The drive key can be changed in case it was compromised, or as part of a scheduled update operation in order to maintain security. We denote the drive key by **Kd**.

8.2.3 Partition Key

An object store can be divided into multiple partitions, formerly known as *security classes*, which carry their own keys for security purposes. From the perspective of the security manager, it will have a distinct secret key for each two-tuple of object store s_j and partition c_k. We denote the key of partition j by $\mathbf{Kp_j}$.

8.2.4 Working Key

The working keys are used to generate the capability keys for a particular partition; hence they should be refreshed very frequently, *e.g.,* on an hourly basis. However, since a key refresh event between the object store and the security manager invalidates all credentials generated by that key at once, a simplistic scheme which keeps only a single working key for each partition would result in an undesired performance degradation as all the clients would be required to communicate with the security manager in order to get new credentials; moreover, all new credentials must be explicitly validated (via MAC calculation) before being cached by the object store. To mitigate the undesired effects of a key refresh, the following optimization, as suggested in [8], can be used: an object store may declare the last two (or more generally *n*) refreshed versions of the *working key* as valid, instead of just the latest one. As a result, the process of validating a capability requires a *key_version* field to be incorporated in the capability indicating which key should be used in the validation process.[33]

[33] For more details on this mechanism, see the object store security document

The number of active key versions used is configured 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. The OSD implementation can specify the maximum number of key versions it supports; one is a legal value. The maximum number of key versions supported by the protocol is 16.

We denote the working key of partition *j* with version *i* by $\mathbf{Kw_{j,i}}$.

8.3 Key Exchange Protocol

We present a protocol for key exchange that applies well-known techniques for key updates[34] and does not use encryption.

The protocol has the following characteristics:

- Except for the topmost key, keys of one level can be replaced only by using a higher-level key. We describe how the master key is set in the Drive Initialization section.
- 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 exchange of a key at a given level invalidates all keys at lower levels (e.g., a new partition key invalidates all working keys).

We propose that the drive use a pseudo random number generator to generate the keys using a random string (a seed) which is sent to it by the drive owner / security manager. Note that the security manager and the drive must use the same generation procedure.

A cryptographic pseudo random number generator may be constructed either from a good MAC function, *e.g.,* SHA1, or a block cipher function, *e.g.,* AES. The specific cryptographic pseudorandom number generator we propose is one that utilizes the cryptographic hash SHA-1, as defined in FIPS 186, Section 3.3. Upon selecting the seed *s*, it basically applies the MAC function to the values *s* and *s+1* using a shared (secret) key.

Again, *TimeNonce* refers to the 12-bytes nonce structure defines in the OSD protocol (a 32-bits timestamp followed by 64 random bits).

We require that at each level, there will be two keys rather then one. The first key is used for message authentication and the second for key generation. For example, instead of having one master key, Km, we have two keys, a keyed MAC key, denoted $\mathbf{K_{m\_A}}$, used for message authentication and a second key for the pseudo random number generator, denoted $\mathbf{K_{m\_G}}$, used for key generation. The same scheme holds for every level. As before, we defer the discussion on how to set the master keys to the *Drive Initialization* section.

[34] See for instance section 12.3.1 of [3], Remark 12.19 (pp. 498-490), states that the confidentiality of the key update is not necessary, and that it may be avoided by employing instead a key derivation from a pseudorandom permutation.

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.

8.3.1 Setting the Drive Key

In order to set the drive key, a **SetKey** message is sent (as described below), protected by the master key. This command will include a *Seed, which* is a random string of length 160 bits computed by the drive owner; LSB (least significant bit) of the seed must be zero.

The new drive authentication key and generation key are computed by applying the generator function on the seed to obtain two distinct pseudo random numbers as follows:

$$K_{d\_G} = G_{Km\_G}(\mathit{Seed\ or\ 0x01})$$
$$K_{d\_A} = G_{Km\_G}(\mathit{Seed})$$

8.3.2 Setting a Partition Key

In order to set the keys of a specific partition, a **SetKey** message is sent (as described below), protected by the drive key. The command will include a seed as defined above as well as a *Partition Number, which* is the number of the partition for which the key is to be set.

The new partition authentication key and generation key are computed by:

$$K_{p,\text{partition number}\_G} = G_{Kd\_G}(\mathit{Seed\ or\ 0x01})$$
$$K_{p,\text{partition number}\_A} = G_{Kd\_G}(\mathit{Seed})$$

Note, setting a partition key invalidates all working keys for the partition and thus all capability keys for the partition.

8.3.3 Setting a Working Key

In order to set the working keys of a specific partition, a **SetKey** is sent (as described below), protected by the partition key, *e.g.*, for partition j, the security manager uses Kp,j. The command will include a seed and partition number as defined above, as well as a *Version Number,* which is the version number of the key to be set.

The new working authentication key and generation key are computed by:

$$K_{w,j,\text{version number}\_G} = G_{K\ p,j\ \_G}(\mathit{Seed\ or\ 0x01})$$
$$K_{w,j,\text{ version number}\_A} = G_{K\ p,j\ \_G}(\mathit{Seed})$$

8.4 Using the standard protocol to Set Keys

Instead of defining a set of specific protocol messages to be used for key management, we can use a single new SetKey command 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 SetKey on the object for the partition and the drive key by invoking SetKey on the object for the object store as a whole.

The parameters of the command are:

- One of the following, DriveKey , PatritionKey, or WorkingKey depending upon the key being set
- an 8-byte string composed of a 1-byte KeyVersion[35] followed by 7 bytes that uniquely identifies the key (a counter will do). In particular, the key identifier indicates the Partition number. This information can be used for auditing and other reporting purposes.
- the information that is needed to infer the next key, i.e., Value is set to be the Seed that is used to generate the two corresponding keys (message authentication key and key generation).[36]

The command is sent using the OSD security protocol as appropriate for the level of security being used by the object store. For messages sent to set the key for the drive, the object representing the drive must be queried to determine the appropriate security level. The CAP-Args right-string must contain an indication that keys can be set. Note that the *CAP_Key* that corresponds to the credential issued on this command is computed using K_{higher_A}. Specifically, CAP_key = MAC_K_{higher} (CAP-Args).

8.5 Drive Initialization

The protocol gives full power over the drive to the possessor of the master key. Thus, using and setting the master key should be done in the most secure environment possible. To allow setting the master key after the drive is obtained from a vendor, we assume that the drive comes from the manufacturer with an initial master key built-in. This master key is also provided *in a secure manner* (*e.g.,* a floppy, a separate email message) to the owner. Before the drive is used for storing the client data, the drive must be initialized. The initialization is done by replacing the initial master key with a new one, generated by the security manager / drive owner. Note that

- The manufacturer cannot access the drive if initialization was done properly since the new Master Key is known only to the owner.
- If the drive has been initialized elsewhere (mistakenly or maliciously) this will be detected by the owner as the initial Master Key that was provided to the owner will no longer work.

The following command will be used to set the master key. The message is authenticated using the previous master key denote by Km_A_old

SetMasterKey msg $M_{Km\_A\_ previous}$ (**SetMasterKey**, msg)
Where msg = Seed, TimeNonce

- Seed is a random string of length 160 bits computed by the drive owner; the LSB (least significant bit) of the seed is zero.

[35] In the range 0-15.

[36] There is an assumption for Level 2 security that the attribute value is part of the command parameters and thus protected by the per command MAC.

The effect of this command is to set the master key as follows:

$$K_{m\_G\_new} = G_{Km\_G\_previous}(\textit{Seed or 0x01})$$
$$K_{m\_A\_new} = G_{Km\_G\_previous}(\textit{Seed})$$

Note, if one is concerned that an entity may listen on the wire as well as steal the master key provided by the object store manufacturer, there is nothing that prevents sending these commands via a direction connection and not over a network.

We point out that this differs from the suggestion in [8] is that the drive comes in an *uninitialized state*, where it has no partitions and no valid keys. Here, before the drive is placed in the general network, the owner initializes it using a secure network, e.g., a cable directly attached from the owner laptop to the drive.

8.6 Storing Long Lived Keys

The drive keys are considered highly secret information. It is important to protect them from being leaked to an adversary. In order to protect the drive the keys should be stored in a tamper resistant[37] nonvolatile manner and maybe even protected by tamper resistant software shield. Note that only the master key must be remembered in a tamper resistant manner. The seeds that were used to create all other keys can be saved in a nonvolatile memory and used to recompute the keys in case of a drive crash.

Note, the object store should not remember the messages sent to set the master key in a manner that could be externally accessible.[38]

8.7 Secure Computation

In order to conform with FIPS 140-1 [5] level 4, storing keys, computing the credential keys and the key exchange protocol should be done in a secure coprocessor.

8.8 Parameterizing Cryptographic Primitives

We would like to provide the flexibility of having an object store support multiple implementations of the cryptographic primitives, i.e., MAC functions. To do this, a root object will support an attribute which provides the cryptographic primitives an object store prefers; this will be provided as an ordered and numbered list of primitives, where number zero is the highest preference. We will allow an object store to support up to sixteen primitives. Note all objects stores must support an HMAC SHA-1.

[37] See *Security Engineering - A guide to building dependable distributed systems*, by Ross Anderson, John Wiley & Sons, Inc. pp.277-304.
[38] The actual requirement for correctness may be slightly weaker than this, but this seems to be sufficient, if not completely necessary.

When the user gets the initial key for the object store, the key will also specify which cryptographic primitives to use with the initial key exchange; the number of this combination will also be specified.

The CAP_Args includes a four bit field indicating the cryptographic primitive used to construct the credential. The security manager will place in this field the number of the cryptographic primitives used in constructing thecredential. The security manager will need to take into account the clients capabilities when it gives a credential to the client. The client will need to use the cryptographic primitive upon which it agreed with the security manager. The intent of this approach was to allow a smooth upgrade of a system, in which some clients may not support a newer cryptographic primitive.

In the first version of the standard we will only support a single MAC function. Later versions of the standard will need to address the security issues that arise in using multiple MAC functions with a single key.

9 References

[1] Azagury, R. Canetti, M. Factor, S. Halevi, E. Henis, D. Naor, N. Rinetzky, O. Rodeh, J. Satran, "A Two Layered Approach for Securing an Object Store Network," *First IEEE International Security In Storage Workshop*, Greenbelt, MD, Dec 2002

[2] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography,* CRC Press 1996. pp. 325.

[3] A. J. Menezes, P. C. Van Oorschot, S. A. Vanstone*, Handbook of Applied Cryptography*, by, CRC Press 1996. Section 12.3.1

[4] AES Advanced encryption standard

[5] FIPS 140-1 security standard

[6] FIPS Publication 186, Section 3.3

[7] H. Gobioff, *et al.*, Security for network attached storage devices, Technical report, CMU-CS-97-185.ps

[8] H. Gobioff, *Security for a High Performance Commodity Storage Subsystem*, PhD thesis, Carnegie Mellon University, 1999.

[9] H. Krawczyk, M. Bellare, R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, http://www.ietf.org/rfc/rfc2104.txt

[10] M. Bellare, R. Canetti, H. Krawczyk, "The HMAC Construction", *Cryptobytes* Vol. 2, No. 1, Spring 1996.

[11] M.K. Aguilera, M. Ji, M. Lillibridge, J. MacCormick, E. Oertli, D. Andersen, M. Burrows, T. Mann, C.A. Thekkath, *"Block-Level Security for Network-Attached Disks," 2nd Usenix Conference on File and Storage Technology*, San Francisco, CA, March 2003.

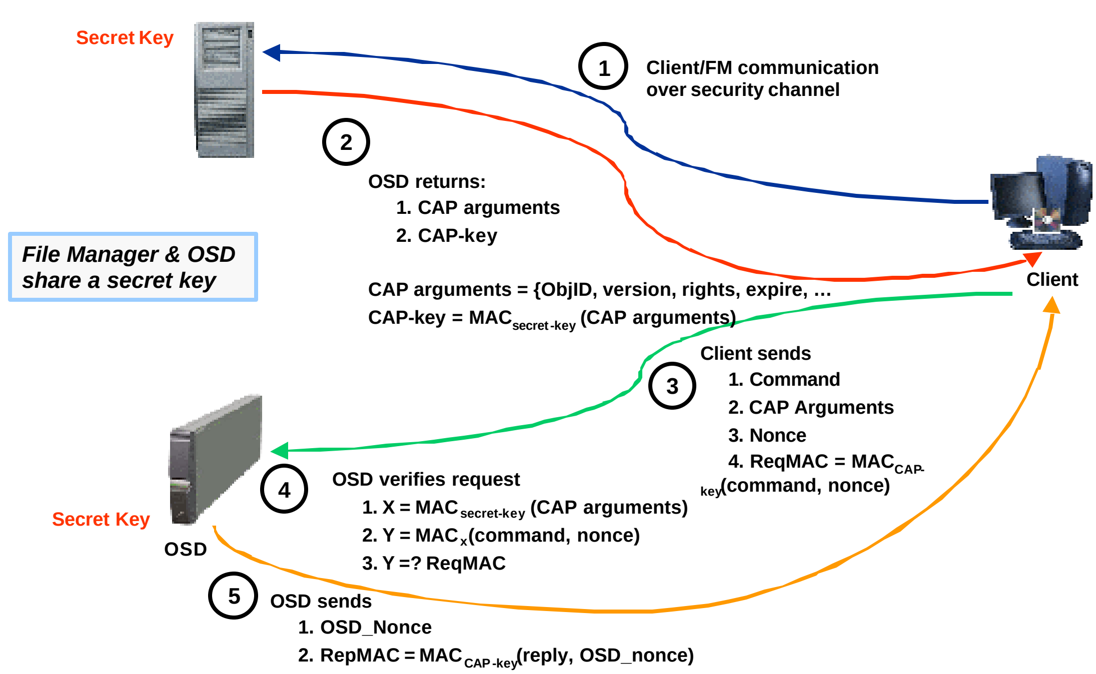
10 Appendix

10.1 Comparison to Original Approach

We now summarize the original NASD protocol and describe the differences between the original protocol and the current proposal.

10.1.1 Original NASD Proposal

As we stated, we want to enable providing only integrity of capabilities (if desired). In the original NASD work 9, which is the starting point for this work, integrity of capabilities is intertwined with network security. To access an object, a host receives a credential composed of a capability and a CAP-key from a Security Manager; the CAP-key is derived from the capability using a secret shared between the object store and the security manager. On each request, the CAP-key is used to authenticate the request: the CAP-key is used as key for a MAC on the nonce and the command/data. The nonce provides anti-replay, *i.e.*, provides a function of network security. The MAC on the command/data provides integrity of command/data. The use of the CAP-key for computing the MAC implicitly provides integrity of capability; if the capability had been modified then the object store would fail in its attempt to validate the MAC of the command/data. The CAP-key is also used by the object store to authenticate its reply to the client.



This approach requires both the host and the object store to calculate a new MAC for each command. However, if we have a secure or trusted network, a direct application of original NASD protocol involves redundant computation. In particular, if we were running on top of an IPSec authenticated channel we would have:

- Two mechanisms for anti-replay
- Two mechanisms for integrity of data

This leads to our challenge: Define integrity of capabilities and integrity of command/data such that integrity of capabilities uses a subset of the cryptographic structure.
In addition to this major challenge, there are some additional minor issues with the original definition of the protocol. These issues led to additional changes from the original NASD protocol in the version of the object store security protocol presented in the following sections.

10.1.2 Ability to Use Either Channel ID or Command Unique Nonce

By replacing the command unique nonce with a channel ID, we are able to extend the original NASD protocol into a protocol that supports running on an externally secured channel without incurring unnecessary overhead. Since the channel ID does not change on each command, it is not necessary to recalculate a MAC that involves this channel ID on each command. However, since the channel ID is tied to the channel and the channel is authenticated, receipt of a MAC based upon this channel ID enables the object store to be certain that the capability it is receiving was legitimately obtained by the client.

10.1.3 Unique Value Added to *CAP_Args*

To avoid scenarios in which the same *CAP_Args* and *CAP_key* is given by the security manager to different clients requesting the same rights to the same (set of) object(s), we add a unique value to each *CAP_Args*. This change closes a potential security hole in the version of the protocol using internal security. Without this change a client could masquerade as an object store for another client, if both clients get the same authorization for a given object.