Capability based Secure Access Control to Networked Storage Devices

Michael Factor Dalit Naor Eran Rom Julian Satran
Sivan Tal
IBM Haifa Laboratory, Haifa, Israel
{*factor,dalit,eranr,Julian_Satran,sivant*}*@il.ibm.com*

Abstract

Today, access control security for storage area networks (zoning and masking) is implemented by mechanisms that are inherently insecure, and are tied to the physical network components. However, what we want to secure is at a higher logical level independent of the transport network; raising security to a logical level simplifies management, provides a more natural fit to a virtualized infrastructure, and enables a finer grained access control. In this paper, we describe the problems with existing access control security solutions, and present our approach which leverages the OSD (Object-based Storage Device) security model to provide a logical, cryptographically secured, in-band access control for today's existing devices. We then show how this model can easily be integrated into existing systems and demonstrate that this in-band security mechanism has negligible performance impact while simplifying management, providing a clean match to compute virtualization and enabling fine grained access control.

Keywords

storage, security, access control, virtualization, networked storage, capability-based security protocol.

1. Introduction

While access control is common in higher level protocols for accessing remote systems, services and application data (e.g. [21, 28, 25, 14], etc), lower layers protocols are less likely to have integrated access control mechanisms. In particular, the protocols used today for accessing network attached storage devices do not have an integrated strong access control mechanisms. Instead, access is enforced at the network or service delivery level. In this paper we discuss access control in storage area networks (SANs).

Today, access control for SAN attached storage is realized using transport level abstractions and not storage level abstractions. More concretely, access control in current SANs is achieved using Port Zoning and/or LUN Masking. At the heart of both schemes (and their many flavors) are access rules of the form: A requests coming from a node connected to port *a*, can get responses from a node connected to port *b*[1].

The fundamental drawback with this approach to SAN security is it involves entities, the ports, which have nothing to do with the desire to control which executing images can access which persistent storage. From this basic problem stem several concrete issues. For instance, since access is tied to ports, changing the physical connection of a node requires updating the SAN security configuration. This is particularly problematic in a world of compute virtualization, where virtual machines co-exist inside the same physical machine sharing physical resources, and migrate between physical machines. Even without virtualization, this mixing of levels of abstraction is a recipe for management confusion: to specify which (logical) host can access which storage volume, one needs to map from a virtual machine to a physical host, from a host to a port and a port to another port and to a volume. A third issue of trying to provide storage access control at the level of ports is that the access control mechanisms are transport-dependent; thus different mechanisms have to be developed, deployed and maintained for IP SAN, FC SAN, SAS SAN and so on.

For Fibre Channel storage networks, two emerging standards - FC-SP [2] which provides port authentication and N_Port ID virtualization (NPIV) [1] which provides means to dynamically create virtual FC ports - can be combined to address some of the above weaknesses. However, adopting them involves hardware changes, and the access policy is still managed in the port level, meaning they still suffer many of the weaknesses we outlined above.

To address these we weaknesses we propose a new security model enforcing access control to storage in SANs. Our model is based on the Object Store Device (OSD) se-

[1]The above description is a simplified generalization of Port Zoning and LUN Masking. Section 2.2 details more about these access control solutions.

24th IEEE Conference on Mass Storage Systems and Technologies (MSST 2007)
0-7695-3025-7/07 $25.00 © 2007 IEEE

curity model [16, 7], as developed in the OSD technical working group in SNIA [4] building upon research done in CMU [17] and IBM [10]. The OSD model is well understood, has been reviewed and implemented. We apply this model to SCSI logical units[2] (and extents within them) in general. The model (Figure 1) provides a mechanism for enforcing dynamic access policies by requiring that storage I/O commands, initiated by some application client, provide a cryptographically hardened credential. This credential is obtained from a security/policy manager which ensures that only authorized clients are given appropriate credentials for a given storage device. The storage device grants or denies access based on this credential. A secret key shared between the security manager and the storage device is used by the storage device to validate the authenticity of the credential shown by the client.

The cryptographically hardened credential is coupled with the client's I/O command using an encapsulation paradigm, where a secure I/O command encapsulates the original I/O command together with the credentials. Using encapsulation we avoid the need for massive changes in the host or storage. Our approach is called *Capability based Command Security* (CbCS).

CbCS addresses the weaknesses we described above. CbCS provides an access control mechanism that is highly amenable to use in a virtualized environment since it secures the logical entities at the appropriate level of abstraction. It provides fine-grained access control which works at the command level, rather than the connection level. It is independent of transport since it is an end-to-end protocol at the SCSI level. Finally, it simplifies management by providing a single point where storage access control needs to be managed. The CbCS approach is a proposed new SCSI extension standard, currently in review at the T10 technical committee of INCITS [5].

Our main contributions in this paper are:

- We define a logical, in-band access control protocol for storage in SANs, be it a disk a CD or a tape.
- We outline the architecture for realizing an I/O path implementation of the model. In particular, we use command encapsulation, which enables a smooth integration with the SCSI protocol.
- We demonstrate that this protocol can be realized with minimal impact on performance.

Table 1 compares our approach with existing solutions. Equivalent security level can be achieved by using Object Storage Devices with an emulation layer in the host system that maps block logical units to OSD objects. The reasons we have chosen to augment the existing protocol with security features instead of emulating disks on top of OSD devices, are avoiding the complications involved in the protocol conversion, and retaining the use of existing storage system that have a much higher level of maturity than OSD systems.

[2]Logical Unit (LU) is the SCSI terminology for the basic entity serving as a target for SCSI commands. LU is sometimes referred to as LUN - the logical unit number.

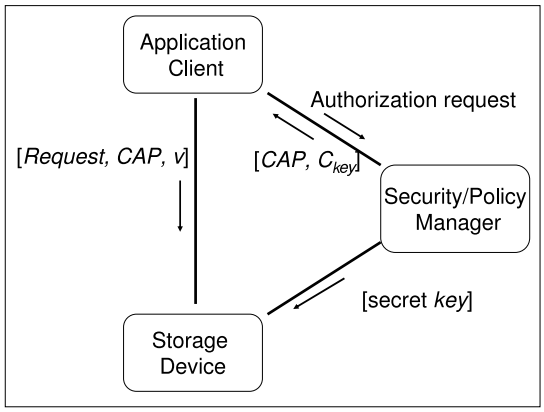

Figure 1. The security model architecture.

The rest of the paper is organized as follows: Section 2 gives background on virtualization and current access control solutions in networked storage. Section 3 elaborates on the model, the aspects of integrating it into SCSI, and its implementation architecture. Section 4 deals with our prototype implementation and its performance. Section 5 looks at related work and section 6 concludes with a discussion and further work.

2. Background

2.1. Virtualization

Compute virtualization is the ability to run multiple instances of an operating system (i.e., virtual machines) on a single physical system. The virtual machines run on a software layer called a VMM (virtual machine monitor) or hypervisor. Roughly speaking, the VMM exposes a virtual hardware environment to the virtual machines. There are several approaches to virtualization (e.g. [20, 27]) and many implementations (e.g. [11, 9, 6]) One feature of virtualization servers is the ability to migrate virtual machines across physical servers. Migration is a useful tool for administrators of data centers and clusters: It facilitates fault management, load balancing, and non-disruptive hardware maintenance [12]. Security mechanisms must be adapted to virtualized environments. Existing mechanisms which are based on physical properties become problematic as physical resources are being virtualized. Access control is one such instance, as outlined below.

Table 1. CbCS vs. current access control approaches in the SAN

	Traditional Zoning/Masking	Zoning/Masking with NPIV/FC-SP	CbCS
Prevents identity spoofing	No	Yes	Yes
Supports differentiated access per LU extent	No	No	Yes
Supports differentiated access per command	No	No	Yes
Supports physical adapter/port sharing	No	Yes	Yes
Transport layer independent	No	No	Yes
Single point of management	No	No	Yes

2.2. Historical SAN security

The access control to storage volumes in FC SAN is done today using port zoning and LUN masking. In this section we overview these methods.

2.2.1. Port zoning. Port zoning is an access control functionality supported by all the fabric switch vendors today. There are several flavors of port zoning and some variations between different vendor implementations. A storage area network (SAN) fabric consists of one or more switches having fabric ports (F_Ports) which are connected to node ports (N_Ports) located in server hosts and in storage systems. A given N_Port communicates with the fabric through its F_Port connection. A zone is a set of FC ports (either N_Ports or F_Ports, or both) that are allowed to pass commands and data between one another. This can be viewed as a virtual fabric residing inside the physical fabric. Zones may overlap, that is a port can be a member of more than one zone. Zoning methods can be categorized along two dimensions: the way enforcement is done and the port type on which it is based (F_Port or N_Port).

Enforcement Methods: There are two methods of enforcing zoning by the fabric switches:

1. Enforcing zones through discovery: When connecting an N_Port to a switch F_port there is an establishment phase in which the connected N_Port is informed with the set of N_Ports addresses it can communicate with. This, however, does not prevent a given N_Port from sending data to some known N_Port address which does not appear in the set.

2. Enforcing zones on frame by frame basis: With this method, each frame is examined for its source and destination ports, and if the two don't belong to a common zone - the frame is dropped.

Zoning According to Port Type: Zoning can be done by grouping either F_Ports or N_Ports in zones.

1. F_Port based zoning: Every two F_Ports in the switch are either allowed or disallowed to exchange data. This method has the merit that it cannot be attacked by host port ID spoofing. However, it is hard to manage, as any change in cabling requires change in zoning configuration. Moreover, control over which host port can connect to each F_Port has to be implemented by some means. For that purposes, switch vendor provide a port binding feature that binds N_Ports to F_Ports. Unfortunately, port binding suffers from port ID spoofing similarly to N_Port based zoning.

2. N_Port based zoning: Each N_Port connected to the SAN can be grouped with other N_Ports defining an N_Port zone. The advantage of this zoning type is that the hosts and storage systems are explicitly defined as zone members. This is easier to manage, and also allows moving systems and N_Ports around the fabric without affecting the zoning configuration. However, this method is vulnerable to spoofing the identities of N_Ports.

Port Zoning Limitations: Port zoning suffers from the following limitations:

1. Port Zoning is vulnerable to identity spoofing, unless it is complemented with port authentication.

2. Both F_Port and N_Port based zoning are not amenable to virtualization, as they are based on ports which are naturally shared between virtual servers and/or dynamically allocated to virtual servers.

2.2.2. LUN masking. Zoning alone does not provide sufficient segregation of storage access in all SAN environments: One of the main features of networked storage is the ability to have shared storage systems. A shared storage system may contain many LUs serving multiple host servers. Every LU within the storage system may be accessed through any of the system's N_Ports. Clearly, port zoning does not provide sufficient access control because of the need to segregate host access between the different LUs residing in the *same* storage system, using the *same*

N_Ports. To enable segregation of host access on LU basis, shared storage system vendors support various management functions which are referred to as LUN masking.

One way of masking LUNs is by defining which LUs are accessible through each storage system N_Port. This is sometimes called port binding, but should not be confused with switch port binding mentioned above. A management interface enables binding LUs to specific storage system ports. This can be used in conjunction with port zoning to limit the LUs visible to hosts. A host can only access LUs that are bound to storage target ports that have a common zone with the host port(s).

Although enabling the segregation between LUs, LUN masking suffers from the same basic vulnerability of N_Port based zoning as it works on host port basis, thus leaving the same management and security vulnerabilities.

2.3. New enhancements

The root problem that causes port zoning and LUN masking to be unusable for virtualized environments is that there is no one-to-one mapping from an N_Port to a server, when there are multiple virtual servers using the same adapter port. The N_Port is physical, and may be shared among multiple logical (virtual) servers. A new approach that addresses this issue virtualizes the N_Port, and enables assignment of virtual N_Ports to virtual machines. Industry-standard N_Port ID Virtualization (NPIV [1]) is the capability for a single N_Port to use multiple port identities.

Combining this approach with port authentication [2] addresses some of the issues raised above. However, these solutions are specific to the Fibre Channel transport layer, thus other types of transport layers still require their own solutions. Moreover, since this solution is at the fabric 'level' it still suffers the weaknesses of "all or nothing" access control and the need to map logical entities to ports.

3. The CbCS model and architecture

3.1. The protocol

The CbCS protocol is a capability-based SCSI-extension protocol which cryptographically enforces the integrity of the capability and its legitimate use by the client. The protocol is based on the OSD security model [10, 7, 16, 17], mapping the objects to logical units of any device type, with appropriate adjustments. Many of the protocol details are similar to the OSD security protocol. In this section we give an overview of the access control protocol, and describe the major changes from the OSD protocol.

3.1.1. Basic flow. Like OSD, CbCS involves three active entities: an application client, a storage device, and a security manager (See Figure 1). As a capability-based access control system, requests to the storage device must be accompanied by a cryptographically secured capability[3] , which encodes a set of rights the holder has on a logical unit. The capability is obtained from the security manager by authenticating to it, and specifying which logical unit is to be accessed.

In return to the capability request, the security manager sends back a pair $[CAP; C_{key}]$ called a credential.

CAP is the capability which describes the rights that the authenticating client has for accessing the requested logical unit (e.g. read only permissions). The capability contains additional properties detailed in 3.1.3, enabling features such as fine-grained revocation of capabilities and performance optimizations. C_{key} (Capability Key) is a cryptographic hash taken over the capability by the security manager using a secret key. C_{key} must be securely kept by the client. The secret key used for the computation of C_{key} is a symmetric key shared with the storage device (see Figure 1).

In order to prevent network replay attacks, where an adversary might replay a legitimate I/O request, the client needs to tie the credential obtained from the security manager to the channel over which the I/O command is issued. This is done by an additional cryptographic hash taken over a unique identifier associated with the channel, using C_{key} as the secret key. That integrity check value is called validation tag and is included in each command. The next section elaborates more on the validation tag and channel identifier.

Once the validation tag is calculated the client can issue an I/O command together with the necessary capability and validation tag. This is done using command encapsulation: The client encapsulates the I/O command together with the capability and validation tag within a standard wrapping SCSI command.

Once the storage device gets the wrapping SCSI command, it extracts the capability part from it. The storage device calculates C_{key} from the capability using the secret key shared with the security manager. It then calculates the expected validation tag. If the expected validation tag matches that from the SCSI command the capability is considered authentic and the encapsulated SCSI command can be serviced given that it is permitted by the capability.

To enable flexibility in the upper software layers, federation of client identities and client authentication methods are not defined by the protocol.

[3] Since the protocol is defined at the command level, not necessarily every I/O request must have this capability. For example, the protocol may be applied only to commands that read or write user data from a storage device.

3.1.2. The security features of the protocol. As shown in [16], the above protocol ensures that the capability presented by the application client is not modified, forged or replayed. To ensure that the credential is not replayed the protocol uses some channel unique identifier *assumed* to be featured by the underlying channel. The protocol assumes that communication sent over the channel can be trusted. Thus, the validation tag which is a cryptographic hash of the channel id using C_{key} attests both the integrity of the capability and the reliability of the transport channel on which it was received. A replay attacker sending the validation tag over another channel would fail as his message is received on a different channel.

The OSD security protocol includes higher levels of security, such as securing the integrity of commands, responses and data (and not only integrity of capability as described above). As our protocol uses the same model, all these security methods can be deployed under our model. However, we concentrate on integrity of capability only. We believe that end-to-end security, providing access control security along with integrity and confidentiality of all the data in-flight, should be achieved by combining this protocol with lower-level protocol at the transport layer to provide secure communication channel between the initiator and the target. When the initiator and the target use a secure communication channel, the additional security methods do not provide additional security. Moreover, they create redundant functionality that only adds complexity and could potentially harm performance. See "Security Methods" in [16] for more information.

3.1.3. Capability arguments. The capability CAP is described as

$$CAP = \{Permissions, SecurityInfo, ExpireryTime, Audit, LUDescriptor, PolicyAccessTag\}$$

1. The *Permissions* field encodes the set of allowed functions on the logical unit. We elaborate more on that field in the next section.

2. The *SecurityInfo* field identifies the shared key used for the capability-key calculation and the cryptographic hash function (e.g. HMAC-SHA-256.)

3. The *ExpiryTime* field specifies the expiration time of the capability. The capability can be reused as long as it is not expired. Thus, the application client does not have to obtain a credential for each I/O.

4. The *Audit* field contains a value that the security manager may use to associate the capability and credential with a specific application or client (thus achieving confinement of credentials as described in [18]). However, the protocol does not mandate a specific algorithm to use, and use of this field is optional and considered to be vendor-specific.

5. The *LUDescriptor* field uniquely identifies the logical unit to which this capability applies. We elaborate more on that field in the next section.

6. The *PolicyAccessTag* field is a settable LU attribute, and it provides a mechanism to invalidate all outstanding credentials for a given logical unit: A valid capability must match the policy access tag of the logical unit; hence, we can invalidate credentials for a logical unit by modifying the value of its policy access tag.

3.2. Integration with SCSI

Integrating the protocol with the SCSI standard has the following aspects:

1. Adding the credential (capability and validation tag) to the SCSI command. This is done using a newly defined command called Encapsulating SCSI Command (ESC). This command allows encapsulating another SCSI command with additional parameters. The capability and validation tag are the encapsulating parameters in our case.

2. Identifying the SCSI LU to which access is granted. The exact format of SCSI LU identification is derived from the SCSI standards that require a logical unit to have a globally unique designator (see [3]). It is possible to address a LU in finer granularity by adding extent within the LU, using SCSI logical block addresses (see [3]). Thus, separate capabilities can be generated for different LU extents.

3. Formulate the permissions. One way to formulate the permissions is to identify which access rights are relevant to SCSI commands (e.g. read/write), and define for each SCSI command the access rights it requires. Section 4.1 details the way we handled permission in our prototype.

4. Provide within the SCSI protocol means to: Query and set LU security attributes, such as the policy access tag, and perform key exchange between the security manager and storage device. Additional standard commands are added for those purposes in the proposal under review in the T10 committee, and they are not detailed in this paper.

5. Provide a key management framework. The key management framework defined for OSD is closely related to the object store structure. Adapting the protocol to

logical units implies the need for a modified framework. However, the principles behind the framework are the same as in the OSD protocol (see [16] section 5).

3.3. Implementation architecture

3.3.1. The basic architecture. In this section we detail the architectural considerations and changes involved in realizing the model. Figure 2 depicts a typical architecture with a Linux server and a networked storage system.

The application client runs in the operating system having a layered I/O stack. In Linux this stack involves three layers: the host bus adapter (HBA), which connects the host to the SAN, the SCSI driver and the block device driver.

The model's storage system is a SCSI device connected to the SAN using a network adapter on top of which there is a SCSI driver. In addition there is a policy/security manager which communicates with security clients residing in the host and the storage system.

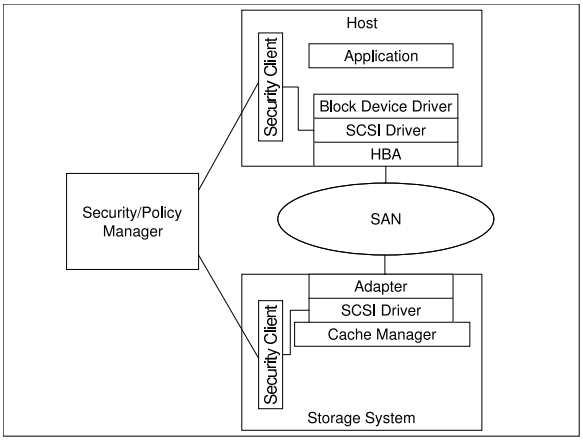

Figure 2. System architecture.

In order to realize the model, a change is required in the host component that constructs the I/O commands to include credentials, namely, the SCSI driver. A similar change is required on the controller.

The general purpose of the security client on the host is to authenticate the entity requesting access against the security manger, retrieve the credential and calculate the validation tag. The host side security client should provide an interface for fetching the credential based on the logical unit accessed and the ID of the accessing entity. The purpose of the security client in the storage side is to exchange the secret keys with the security/policy manager and to validate the authenticity of the capability using the validation tag received as part of the I/O command.

3.3.2. Architecture in virtualized environments. In virtualized environments we identify the virtual machine as the logical entity granted access to storage. Different virtualization systems require different implementations. [20] classifies virtualization along two dimensions: The interface they provide to virtual machines and the underlying platform they are built upon. We are interested in the I/O interface given to the virtual machines and the hardware interface used by the hypervisor or VMM to do the actual I/O.

The IBM POWER^{TM} hypervisor [9] has a way to provide the generic SCSI system within the virtual machine a direct mapping of virtual LUNs to real LUNs. In this case the virtual machine acts as the host in the basic architecture: The SCSI driver which needs to be changed is the virtual machine driver, and the security client runs in the virtual machine.

On the other hand VMMs such as Denali [29] and Xen [11] provide generic devices to the virtual machine, eliminating the SCSI stack from the virtual machine. This requires a different solution. Figure 3 depicts the I/O stack in Xen. In Xen virtual machines are called user domains, and there is a special domain called domain0 which is the only domain having direct access to physical devices. The user domain operating system is provided with a virtual block device (VBD) having a front-end driver in the user domain operating system, and a back-end driver in domain0, which manages all VBDs.

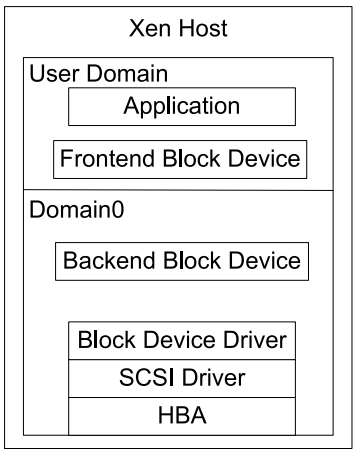

Figure 3. The Xen block I/O architecture.

4. Our prototype

4.1. Implementation details

We have built a prototype system to demonstrate the feasibility of the good path data flow in a virtualized environment.

The application client is a Xen user domain. The storage device is a storage system, which exposes logical units to the Xen server which runs user domains. The security manager is realized by code running on the client system, which constructs the capability structure and computes the capability key. Since we are interested in the I/O path we did not implement the security manager as a separate server which requires authentication. Our prototype is realized by the following components:

Access control permissions: We have classified a subset of the SCSI commands as either read, write or control commands. The permissions granted to user domains to access a LU consist of three bits controlling Read/Write/Control access. The actual policies and the secret keys were implemented as files located in the client and storage systems. Our prototype does not include client authentication. Credential requests are emulated by directly invoking functions that read data from files and return the appropriate credentials.

The Xen virtual block device: An architectural issue arising when coming to realize the model is associating the requesting user domain with the I/O command constructed in the I/O stack in domain0. This association is done using Xen user domain id. This id is unique and persistent across Xen systems, making it suitable for migration of a virtual machine between systems without changing the access control policy. To do the actual association we have added the user domain's unique id to the I/O request structure constructed in the back-end driver of the virtual block device. This block I/O structure is part of the data handed finally to the SCSI subsystem in domain0, which constructs the actual SCSI command, and needs to set the proper credential within the command.

The SCSI driver in the host: In the I/O stack every physical device is associated with an instance of a driver according to its type. For example, different types of SCSI devices (disks, tapes, CDs) are associated with instances of SCSI drivers having the appropriate type. For our prototype, we defined a new type of SCSI device for block logical units with CbCS access control[4]. The SCSI driver we associated with this type is a modified SCSI disk driver, changed to:

- Obtain the appropriate capability and validation tag from the security client according to the user domain id issuing the command and the target logical unit.
- Wrap the original SCSI command descriptor block (CDB) inside a larger CDB that contains the capability and the validation tag.

The host side security client: The host side security client is implemented as a kernel module, so that it can be accessed efficiently from the SCSI driver. As mentioned above, the host security client provides an interface for the host SCSI driver to obtain the capability and validation tag. Because our prototype does not include a full implementation of security manager, the host side security client emulates the security/policy manager. The host side security client calculates the capability-key and validation tag once upon first request and fetches them from memory on subsequent requests. We mention that this mode of operation is similar to a real world scenario, where the capability is not retrieved from the security manager on every command, and the validation tag does not change as long as the same channel is used.

The host's HBA: The SCSI standard supports CDBs up to 255 bytes in size [3]. However, current Fibre Channel HBA drivers support only CDBs up to 16 bytes in size, as larger CDBs are not in use. The QLogic Fibre Channel HBA card supports a mode of operation where the Fibre Channel frame is constructed in software (In the usual mode of operation the card would accept the SCSI command and construct the Fibre Channel frame using its firmware). Thus, we needed to modify the open source Linux driver of the QLogic HBA to construct a Fibre Channel frame containing the larger CDB, and use this mode of operation.

The SCSI target driver: The SCSI driver in the storage system was modified to:

- Declare its exported logical units as having the new SCSI device type.
- Strip the credential from the wrapper SCSI command, and forward it to the storage side security client for validation.
- Upon successful validation pass the encapsulated SCSI command to the code that normally processes the commands.

The storage side security client: The security client exposes an interface to the SCSI driver for command validation. To do this it calculates the C_{key} and validation tag based on the appropriate key[5] and the received capability.

[4] This implementation differs from the standard proposal, which introduces a new flag in the logical unit's VPD to denote use of access control security rather than a new device type.

[5] For our prototype, keys were hard coded for each LUN.

Table 2. Number of lines added/modified

Module	Lines of code
Newly introduced code	
LibTomCrypt	580
Security client code	1240
Modified/Added code to existing components	
Xen code	20
Linux kernel SCSI code	140
QLogic HBA	200
Storage system	90
Total:	2270

The security client caches the calculated validation tag[6] in a hash table with the capability serving as a hash table key to save cryptographic computations on future access.

All cryptographic hash calculations are HMAC-SHA1 calculations. We have used the LibtomCrypt [15] open source library to do all calculations in both the host and storage system.

The total amount of code added/modified in our prototype is less than 2300 lines. Table 2 lists the amount of added/modified code by components.

4.2. I/O path performance

Realizing the CbCS model brings up the question of performance impact. The prototype considers the I/O path only and does not take the authentication into account, as it is not on the critical I/O path.

We compare the I/O performance of the described environment with and without the access control mechanism. We do this using synthetic I/O and Benchmarks.

4.2.1. Hardware. The test environment has the following specification:

- Host. IBM blade server HS20-8843, with a single Intel Xeon 2.80GHz CPU[7], 512MB RAM and 2Gbps QLogic Fibre Channel Adapter. We used Xen 3.0 based on Linux kernel 2.6.
- SAN. IBM BladeCenter FC Switch and a Brocade switch.
- Storage system. A storage system, based on two Linux machines, each equipped with two Intel Xeon CPU 3.00GHz CPUs, 8GB RAM and 2X2Gbps Fibre Channel Adapters.

[6] As long as the same channel is used for a given capability the validation tag does not change.

[7] This model has two processors, but we configured the OS to use a single processor for the purpose of measuring command latency accurately.

4.2.2. Synthetic I/O. We first compare a CbCS secured setup to a standard setup using synthetic I/O measured on a per I/O command basis. We have performed the measurements in 3 modes:

1. Kernel. The measurements were taken within the SCSI device driver in Domain0. The time measured is the time elapsing between initializing the SCSI command and the time the SCSI disk driver completes the command.
2. User. The measurements were taken from a user mode process running in Domain0. We measured the elapsed time of a blocking read or write command done directly to the device, bypassing any local file system caching.
3. Virtual machine. Same as in User mode only the process runs in a virtual machine, meaning the I/O is performed against a virtual device mapped to a physical device through Domain0. Again, there is no local file system caching.

Note that the kernel mode and user mode cases are equivalent to a regular Linux system (running on bare metal without VMM) and the results are applicable to such systems as well.

In each mode we have done separate I/Os for read and for write, each with block sizes of 512 bytes, 4kb and 64kb. Each of the above combinations was measured for both cache misses and cache hits in the storage system. Altogether we have tested 12 different types of I/O commands in each mode. Each command type was executed 100000 times.

Computing the average and standard deviation over all 100000 commands in all configurations gave standard deviation that is much larger than the time difference between the CbCS and standard setups. In fact in some configurations CbCS performed better than the standard setup. Taking out 5% of the commands significantly reduced the standard deviation and the averages were comparable.

Looking at 95% of the data, it turns out that there is a constant overhead of approximately 8 microseconds per SCSI command incurred by CbCS in all modes and command types. Figure 4 shows the absolute times of all cache hit measurements, demonstrating the constant additive overhead incurred by CbCS.

Considering the CbCS protocol, this constant overhead is expected. The changes are independent of the actual transfer length requested by each command.

To better understand the 8 microseconds overhead we have used an additional setup called "large CDB", where we construct a CDB of size necessary to pass the encapsulated command with the credential, but we don't populate it with a credential and perform no validation. This setup

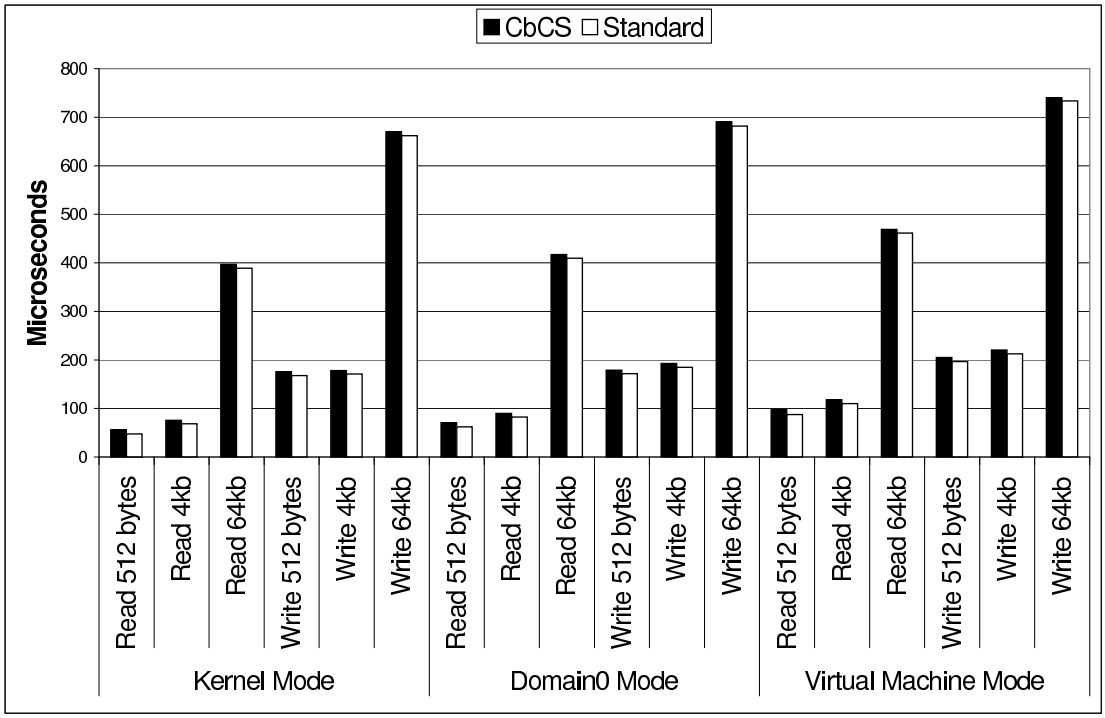

Figure 4. Cache hit I/O absolute times. CbCS shows a constant overhead of about 8 microseconds over the standard setup.

was used to estimate the overhead of using large CDBs in both the host and the storage systems. Looking at 95% of the data (dropping 5% to reduce standard deviation) we see that the constant time difference between the large CDB and standard setups is about 3 microseconds.

We conclude that the host and storage system security code takes 5 microseconds on average. The security code in the host fetches the cached credentials and copies them into the SCSI CDB. On the storage system the security code fetches the cached validation tag and compares it with the received validation tag. Note that in the most common case no cryptographic computations are performed as they are done only once and their results are cached.

To estimate the time saved using caching of the cryptographic computations we have used the synthetic I/O to measure the actual time spent in the security code with and without the caching. It turns out that with the host hardware listed above (using only one CPU) approximately 3.8 microseconds are saved when using caching. On the storage system we did not observe any difference. This can be explained by its significantly stronger hardware.

4.2.3. Benchmarks. To estimate the CbCS overheads at the application level we have tested the CbCS and standard setups using the postmark [19] and Bonnie++ [13] benchmarks. Each benchmark was executed in 4 configurations: CbCS and standard setups in both domain0 and virtual machine modes as above. The benchmarks were executed using the hardware listed above, over an ext3 file system. Each configuration test was executed 50 times. After each execution the file system was un-mounted, the device formatted and re-mounted again.

The **Postmark** benchmark is designed to simulate short lived small files workloads. The benchmark workload first creates files, and then performs read/write transactions on randomly chosen files. Each run was configured to create 30,000 files and to perform 50,000 transactions. Other than that we have used the benchmark defaults. We have used version 1.5 of Postmark. Figure 5 shows for each of the configurations its maximum, average and minimum total execution times. Although the averages show that CbCS seemingly performed better than the standard setup, the maximum and minimum values show that the difference between the CbCS and standard setups are in fact indistinguishable.

Bonnie++ [13] performs sequential I/O on large files ensuring bypass of the filesystem's cache. For the domain0 mode a 1 GB file size was used as the host RAM is 512 MB. For the user domain mode a 512 MB file size was used as the virtual machine RAM was defined to be 256 MB. Other than that we have used the benchmark's default. The Bonnie++ benchmark performs sequential read and write oper-

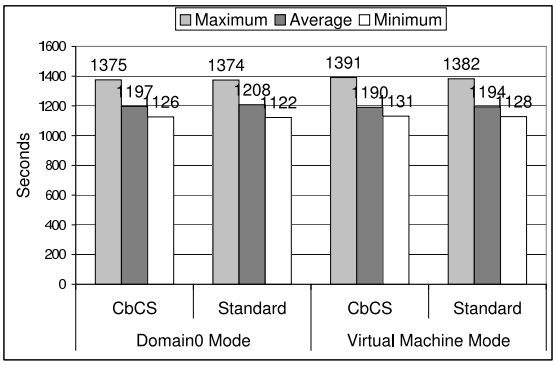

Figure 5. Postmark results.

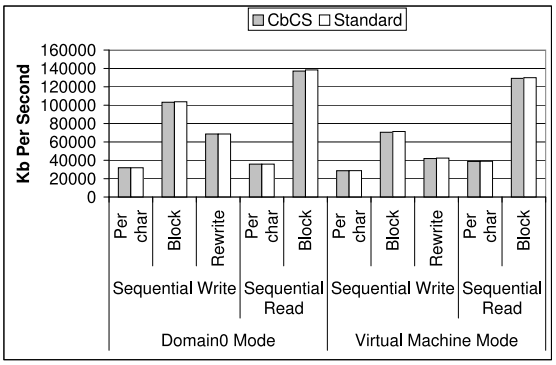

Figure 6. Bonnie++ results

ations. The sequential writes are done per char, block-wise and rewrites. The sequential reads are done per char and block wise. We have used version 1.03 of Bonnie++. Figure 6 shows the resulted kilobytes per second rates. CbCS average rate degradation is 0.57% reaching its maximum of 1.2% in virtual machine mode rewrite test. We mention that for all tests the rate difference between CbCS and standard setups is smaller than the standard deviations. We conclude that Bonnie++ shows a negligible difference between the CbCS and standard setups.

5. Related work

In his PhD thesis [17], Gobioff introduces a new design for access control to network attached storage. Moving the storage device from the server to the network gives rise to various network attacks. Coping with these network attacks must be done in accordance with higher level applications to enforce their access policies over storage resources. [17] suggests a cryptographic capability system which enables application file managers to define policy decisions, which are enforced by the storage, in an asynchronous way, such that the file manager does not have to be available during actual I/O. In this work we adapt Gobioff design concepts, which were part of the Network-Attached Secure Disk (NASD) architecture (later evolved to the object storage devices (OSD)).

Aguilera et. al. [8] apply the same security model as we do to enforce access control in a different setting. Their application of the model is done in a network attached disks (NADs) file systems framework. In a NAD filesystem, a file operation goes through a metadata file server, which provides the client with the block mapping of the file. Using the block mapping, the client can access the network attached storage - which works at the block level - directly. [8] add security management functionality to the metadata file server, such that any client request is passed an authorization check. Having passed the authorization check, the client is provided with the block mapping, accompanied with the required capability to perform the necessary operation on each of the blocks in this mapping. Similar to our approach, [8] also require no changes in the data layout on disk, and can be incorporated relatively easily in existing NAD file systems. [8] introduce a different capability revocation scheme required due to the large amount of capabilities which need to be granted.

Reed et. al. [26] define an authentication framework for network attached storage. Their framework features client and storage authentication, access control enforcement, and integrity and freshness of messages without doing key exchange and encryption. Freshness is guaranteed using either nonce and counters or synchronized timers. Authentication, access control and integrity are achieved by cryptographically hardening request and response messages with the appropriate keys in a similar way done in our model: A key, called disk key, is shared between an administrator/security manager and a storage device. The disk key is used to derive capability keys and identity keys, which in turn are used for access control enforcement, authentication and integrity checks. The capability keys are generated using a cryptographic hash function which hashes some access permissions using the disk key. The identity keys are generated similarly, only the hash is taken over some identity string.

Miller et. al. [24], and Mazieres [23] enforce access control without the use of a centralized server, using encryption of blocks on the disk. Access control is achieved by distributing keys to authorized clients. Revoking access in such filesystems implies re-encrypting the data with a new key, and so is expensive.

Leung and Miller [22] modify the OSD security protocol to fit to peta-byte scale storage systems. In such systems the security server must generate thousands of capabilities per metadata request. The integrity check done on non-cached capabilities in the storage also causes a significant degradation in such large scale systems. They introduce a coarse grained capabilities having expressiveness equal to a collection of many finer grained capabilities, and can

span over many objects, rather than just one. Capability integrity is kept using a public key signature, rather that the capability key which requires a shared secret between the security manager and hundred of thousands of OSDs. [22] also introduce a new and simple capability revocation scheme. Although CbCS has inherently less capabilities, as they are granted for LUs, it might be that some of these ideas can serve in taking CbCS to large scale environments.

6. Conclusions and further directions

The CbCS protocol provides enhanced security to storage devices without changing the whole storage access interface paradigm. The prospects of CbCS adoption depends on many factors. Technical issues that need to be resolved and advances to be made on one hand, and market demand that needs to be built up on the other hand. This section addresses those considerations in brief.

A modification is required in the SCSI system to embed the existing I/O commands in new commands containing credentials (when necessary). Storage systems should provide support for validating credentials and for the new security management commands. A management function should also be developed. The accumulated impact of all these changes on the existing systems, multiplied by the number of different systems, is significant despite the fact that the storage layout and access patterns remain the same. Our prototype system leads us to an optimistic view of the feasibility of applying these changes with a reasonable development effort.

There are still numerous development and deployment issues to resolve on the way to commercial productization. End-to-end solutions have to be developed, and in particular virtualization creates a variety of scenarios that need to be addressed. As the access of a virtual machine to its storage is mediated by a hypervisor and/or by an I/O partition with different storage abstraction schemes, a trust model should be established for each environment, and definition of roles of each component in obtaining and using credentials of a VM to its storage. Another issue that should be addressed is how to securely boot from an external device. How can a credential be obtained before accessing the boot image? For virtualization environment, the issue of secure VM provisioning arises. When an entity requests permission to access an image for provisioning a VM, how can we trust its innocence for accessing the storage only for provisioning?

Standardization of the protocol is a key step towards wide deployment, and a standard proposal is now under review at the T10 technical committee of INCITS. Another requirement for making the deployment of the protocol practical in large commercial SANs is taking into account gradual migration of currently deployed SANs, where host support and storage systems support are likely to be available at different times from different vendors. It is thus required that security attributes should be settable and retrievable at the LU level. Specifically:

- The protocol should allow for a storage system to support both regular and secure LUs at the same time through the same target port.
- The protocol should provide an application client (in a host system) means to determine the security access controls applied to any particular LU.
- The protocol should maintain compatibility for old application clients such that rejecting their access to secure LUs is done in a way that allows for graceful failures.

Another key factor in making this mechanism a reality in commercial systems is market demand for secure access control to networked, shared, storage systems. This paper describes significant security vulnerabilities of existing SANs. However, these vulnerabilities are not yet acknowledged widely enough to make this a critical concern in the IT world. IT organizations have used the same mechanisms for long years, and became very much used to their limitations, such that those limitations are often perceived as given facts. With lack of clear evidence for successful attacks, the intention is drawn to other areas of IT security. The emergence of compute virtualization could catalyze the need for enhancing SAN security, as it opens a gap between the logical and the physical entities in the network. As the virtualized environments will become more mature and large commercial virtualized systems will become ubiquitous, the security concerns about them will rise and comprehensive security solutions will be pursued more seriously and intensively. Will CbCS become part of many large scale mission critical IT systems? We believe that the answer to that question depends, among other factors, on our ability to standardize and develop effective implementations of the mechanism in a timely manner, such that the mechanism will be ready for the challenge when the demand for improved solutions for storage access control security rises up as one of the top priorities in the IT industry.

References

[1] Fibre channel link services. http://www.t11.org/ftp/t11/pub/fc/ls/06-393v6.pdf.

[2] Fibre channel security protocols. http://www.t11.org/ftp/t11/pub/fc/sp/06-157v3.pdf.

[3] Scsi primary commands. http://www.t10.org/ftp/t10/drafts/spc4/spc4r09.pdf.

[4] Storage networking industry association. http://www.snia.org.

0-7695-3025-7/07 $25.00 © 2007 IEEE

[5] Technical committee t10. http://www.t10.org.

[6] Virtual pc overview. http://www.microsoft.com/windows/virtualpc/evaluation/techoverview.mspx.

[7] Scsi object-based storage device commands. In *T10 Technical Committee of INCITS*, October 4, 2004.

[8] M. K. Aguilera, M. Ji, M. Lillibridge, J. MacCormick, E. Oertli, D. G. Andersen, M. Burrows, T. Mann, and C. Thekkath. Block-Level Security for Network-Attached Disks. In *Proc. 2nd USENIX Conference on File and Storage Technologies*, Mar. 2003.

[9] W. J. Armstrong, R. L. Arndt, D. C. Boutcher, R. G. Kovacs, D. Larson, K. A. Lucke, N. Nayar, and R. C. Swanberg. Advanced virtualization capabilities of power5 systems. *IBM J. Res. Dev.*, 49(4/5):523–532, 2005.

[10] A. Azagury, R. Canetti, M. Factor, S. Halevi, E. Henis, D. Naor, N. Rinetzky, O. Rodeh, and J. Satran. A two layered approach for securing an object store network. In *SISW '02: Proceedings of the First International IEEE Security in Storage Workshop*, page 10, Washington, DC, USA, 2002. IEEE Computer Society.

[11] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauery, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *In Proceedings of the 19th ACM Symposium on Operating Systems Principles*, October 2003.

[12] C. Clark, K. Fraser, S. Hand, J. G. Hanseny, E. July, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *In Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, May 2005.

[13] R. Coker. Bonnie++ benchmark tool. 2004. http://www.coker.com.au/bonnie++/.

[14] M. DeBergalis, P. F. Corbett, S. Kleiman, A. Lent, D. Noveck, T. Talpey, and M. Wittle. The direct access file system. In *FAST*, 2003.

[15] T. S. Denis. Libtomcrypt - modular cryptographic library. http://libtomcrypt.org.

[16] M. Factor, D. Nagle, D. Naor, E. Riedel, and J. Satran. The osd security protocol. In *SISW '05: Proceedings of the Third IEEE International Security in Storage Workshop (SISW'05)*, pages 29–39, Washington, DC, USA, 2005. IEEE Computer Society.

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

[18] S. Halevi, P. A. Karger, and D. Naor. Enforcing confinement in distributed storage and a cryptographic model for access control. Cryptology ePrint Archive, Report 2005/169, 2005. http://eprint.iacr.org/.

[19] J. Katcher. Postmark: A new file system benchmark. In *NetApp Technical Report TR-3022*, 1997.

[20] S. T. King, G. W. Dunlap, and P. M. Chen. Operating system support for virtual machines. In *USENIX Annual Technical Conference, General Track*, pages 71–84, 2003.

[21] J. T. Kohl, B. C. Neuman, and T. Y. Ts'o. The evolution of the Kerberos authentication service. In *Proceedings of the Spring 1991 EurOpen Conference*, 1991.

[22] A. W. Leung and E. L. Miller. Scalable security for large, high performance storage systems. In *StorageSS '06: Proceedings of the second ACM workshop on Storage security and survivability*, pages 29–40, New York, NY, USA, 2006. ACM Press.

[23] D. Mazieres. Don't trust your file server. In *HOTOS '01: Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, page 113, Washington, DC, USA, 2001. IEEE Computer Society.

[24] E. L. Miller, D. D. E. Long, W. E. Freeman, and B. Reed. Strong security for network-attached storage. In *FAST '02: Proceedings of the Conference on File and Storage Technologies*, pages 1–13, Berkeley, CA, USA, 2002. USENIX Association.

[25] B. Pawlowski, S. Shepler, C. Beame, B. Callaghan, M. Eisler, D. Noveck, D. Robinson, and R. Thurlow. The NFS version 4 protocol. *Proceedings of the 2nd international system administration and networking conference (SANE2000)*, page 94, 2000.

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

[27] R. Rose. Survey of system virtualization techniques.

[28] J. G. Steiner, B. C. Neuman, and J. I. Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of the USENIX Winter 1988 Technical Conference*, pages 191–202, Berkeley, CA, 1988. USENIX Association.

[29] A. Whitaker, M. Shaw, and S. Gribble. Denali: Lightweight virtual machines for distributed and networked applications, 2002.