Security Considerations When Designing a Distributed File System Using Object Storage Devices

Benjamin C. Reed, Mark A. Smith, Dejan Diklic
Department of Computer Science
IBM Almaden Research Center

Abstract

We present the design goals that led us to developing a distributed object-based secure file system, Brave. Brave uses mutually authenticated object storage devices, SCARED, to store file system data. Rather than require a new authentication infrastructure, we show how we use a simple authentication protocol that is bridged into existing security infrastructures, even if there is more than one authentication protocol or domain present.

We position our work in the context of some of the current work going on in distributed secure file systems and present our implementation of our file system. We also present some security weaknesses that are shared with other distributed file systems that may not be apparent when designing these systems.

1 Introduction

The ubiquity of Internet access has opened up new possibilities of information sharing. It also brings along with it new challenges in the form of scale and security. We have designed a distributed file system to meet some of these challenges. Specifically we designed a distributed file system with the following design points:

- Untrusted storage devices.
- Independent security domains.
- No central manager or client communication.
- Strong mutual authentication.

Since this is a distributed file system, we must store the data on the network to allow other clients to access it. However, we only trust the network storage devices to store the file system data and meta-data. We want to be able to use client side encryption to encrypt the data and meta-data stored on the device. This means that a device may know that a set of blocks correspond to a file or directory, but it does not know how to interpret the contents of those blocks. The device also does not know how files and directories are related to each other.

The storage devices that store the file system data and meta-data may not all be in the same security domain. We do not want to require cooperation from the different security domains. For example, a file in a subdirectory might be located on a storage device at a different company. While the users of the file system may have a relationship that resulted in the file being stored there, the administrator of the storage device that stores the subdirectory need not have a relationship with the administrator of the device that stores the file. We also want to be able support the scenario in which different authentication schemes are used in the two companies.

Related to the previous requirement, we also do not want to require any kind of central manager for the file system. This may take the form of a central authentication server or a central meta-data server. Basically, each storage device can be managed independently of another device and clients need only do operations and authenticate themselves with devices that hold the data involved in the operation. Further, by not requiring clients to communicate with each other we allow the presence of firewalls and obviate the need for group communication protocols.

Because we are not operating in a single trusted domain, we must be able to do mutual authentication of client and server. Because there are no file servers and clients directly access the network storage devices, each device must make sure that data access is only done by authorized clients. As we do this we want to be able to keep the devices simple and manageable while still adapting to the authentication protocols used in the environment in which the device is located.

These design points lead us to a object based network storage device, SCARED, that can manage the storage of file system data and meta-data without the need of centralized file servers or meta-data servers. On top of SCARED devices we developed a client file system, Brave, that maps a distributed file system onto SCARED devices.

Brave clients trust SCARED devices to store their data and enforce access controls, but they do not trust the devices to manage the file system data or meta-data. There is no communication or trust between Brave clients nor between SCARED devices. When a Brave client communicates with a SCARED device it proves it is authorized to do the requested operation and the SCARED device authenticates all responses.

2 Related Work

In the beginning distributed file systems were simply designed to make remote file systems as easy to use as local file systems. With few exceptions the majority of these file systems were more concerned with making the data available than securing it. NFS [4], the de facto UNIX network file system, and CIFS [8], the de facto personal computer network file system, are the distributed file systems in common use today and are not appropriate for use on untrusted networks due to their weak security features.

Now that the Internet is in general use and vast quantities of data are shared over it, the problem of malicious agents on the Internet has become more important. The susceptibility of the Internet to malicious agents has been known for a long time [2], but the increased connectivity of the Internet has increased the exposure to such agents.

2.1 Secure File Servers

The first secure distributed file system in common use was AFS [10]. This file system was later followed by DFS [11]. AFS servers store sub trees of the file system and use Kerberos [15] to provide authenticated access to the trees. Each AFS server manages the meta-data and has full access to the file data. DFS added the option to do link level encryption to prevent eavesdroppers from discovering the contents of the file system.

NFS V4 [17] has also greatly increased the security of NFS. Like DFS it also provides the ability to encrypt traffic between client and server and it does strong authentication. An NFS server manages a complete file system and thus has access to all the data and meta-data.

2.2 Encrypting File Filters

The Cryptographic File System (CFS) [3] was developed to enable file system clients to keep file data hidden from the file server. CFS trusts the server to store the file data, but it is encrypted by the client before it is sent to the server and decrypted by the client when it is received from the server. CFS also encrypts file names in directories. To make CFS portable it is implemented as a file system filter that manipulates data before sending it to the real file system. The filtering is done using a user-level NFS server that proxies another file system.

Other file systems have expanded on this idea. CryptFS [19] implements the filter in the kernel using a stackable file system [20] to achieve better performance. The Transparent Cryptographic Filesystem (TCFS) [5] also uses a user-level NFS server but adds integrity checks to file data and also adds the ability to share encrypted files.

2.3 File Servers with Client-Accessible Storage

To increase file system performance Network Attached Secure Disks (NASD) [7] is used to allow clients direct access to file data stored in object storage devices. NASD uses capabilities to provide clients with secrets used to access the object storage device and prove to the devices the ability to request an operation on an object. The communication between client and NASD is both encrypted and authenticated. Although each NASD device does not know how the object it holds fits into the file system, the file server still has full access to the file system meta-data and the ability to access all of the file system data.

While it seems that an object storage device is necessary for enforcing access controls, it turns out that the file server can use slightly more sophisticated capabilities to convey to a block storage device the block ranges that a client can access. A novel distributed file system [1] has been implemented that uses these capabilities and also uses extents for allocating blocks. The use of extents means that in most cases capabilities for just a few block ranges can grant access to all the blocks the client needs to access the file. In this file system the file server knows even more about the file system than NASD since it knows the mapping of files to blocks on the storage devices.

2.4 Highly distributed file systems

The Secure File System (SFS) [14] is a highly distributed file system that combines independent file servers into a single name space by generating a unique identifier for the file server using its location and public key that can be accessed from the root of the file system. Each file server then maintains a file system that clients can access. A secure channel with the file server is established using the public key of the file server. Client access is authenticated using a couple of different client authentication schemes. As with other file servers, an SFS file server has full access to the portion of the file system data and meta-data it manages.

OceanStore [13] is another highly distributed file system. It uses a large number of untrusted storage devices to store redundant copies of encrypted files and directories in persistent objects. Objects are identified by globally unique identifiers (GUID) that are generated in a similar fashion as the unique identifiers in SFS. Each identifier is a hash of the owner's public key and a name. Objects can point to other objects to enable directories. All objects are en-

crypted by the clients. By replicating the objects among servers, clients can even avoid malicious servers deleting their data. The extensive use of replication and public keys make revocation of access and deletion of data difficult to achieve, but it does provide a nice model for a completely decentralized distributed file system.

3 Brave File System Design

For Brave we wanted to be able to have many of the decentralized benefits of OceanStore without having to rely on public keys and large numbers of storage servers. We wanted to be able to allow even more of the file system meta-data to be encrypted than the encrypting file filters. Specifically, file servers serving as the shadows of the filters still know the basic layout of the file system. We also wanted Brave to have the feel of classical distributed file systems: ACLs can be used to immediately revoke access, servers storing file system data can be identified and managed, and data consistency guarantees can be made that are similar to classical distributed file systems.

To achieve these goals we store file and directory data in objects on object based storage devices. Like OceanStore, these objects are used directly to reconstruct the file system without the need of a file server. Unlike OceanStore we trust storage devices to store the data they hold, although we do not trust them to keep it confidential. This allows us to avoid requiring large scale replication. We also want to enable users to control which storage devices the data resides on to fit their requirements. For example, some data may need to be stored on a highly reliable storage appliance that is backed up daily while other data may be stored on an unreliable device in an office. Other factors such as network locality or network bandwidth may make other servers desirable for specific data.

We were able to achieve these desires by basing Brave on SCARED object disks that are used to store the file system data and meta-data. SCARED is only a storage device and does not interpret the data it stores. Thus, SCARED never knows the relationship between objects in a file system. The Brave file system runs on clients and reconstructs the file system using the data stored in the SCARED devices.

3.1 SCARED Object Disks

Our goal for the SCARED object model was to make a device that is able to efficiently and securely store the data for a distributed serverless file system. It is more than a network block device since it manages objects and therefore allows more structured access to the storage. Our object model is also more rich than previous models used by NASD and OceanStore because SCARED also fully supports the storage of directory data. Both NASD and OceanStore clients operate on streams of bytes. SCARED provides support for managing directory entries to ease the synchronization requirements of the Brave clients. However, SCARED does not achieve the level of a distributed file server since the SCARED storage device does not have any sense of how the objects it stores are related within the file system hierarchy.

3.2 Why not a Block Server?

We decided that a block interface to storage devices was an insufficient interface for building a decentralized file system. Part of this reasoning was based on the convenience of using objects to both allow the storage to manage block allocation and the use of handles for doing atomic operations. Since clients do not communicate with each other, having the storage device handle block allocations of files and directories allows the synchronization necessary to avoid allocation conflicts from happening locally at the device.

Using objects also allows other object level operations such as deletions and cache management to be done efficiently. Atomic operations on directory operations are also facilitated using objects.

The main motivation for choosing an object model for the storage devices is security. We want to be able to restrict client access at the granularity of files and directories using access control lists (ACLs) as well as capabilities. If we used a block abstraction to store files and directories, the storage device would lack the necessary mapping of blocks to file system objects to restrict access using ACLs. Our use of ACLs also gives us an advantage over NASD because it allows the storage device to map identities to capabilities instead of requiring a meta-data server.

Using a file abstraction to access network storage; as do NFS, AFS, and CIFS; allows for ACLs, but it also exposes all of the file system meta-data to the storage device. We wanted to be able to eventually encrypt as much of the file system meta-data as possible at the client, and using an object abstraction allows more of that information to be kept confidential. Specifically, the file system hierarchy can be kept confidential from the storage device and other untrusted clients.

3.3 Communication Channel

SCARED clients communicate with devices over TCP. We assume that attackers can observe, modify, remove, and insert packets into a TCP connection without detection by the networking layer. We also assume that the networking layer does not provide us any type of authentication information.

In order to enable authenticated communication we use Message Authentication Codes (MAC), specifically a hash based MAC (HMAC) [12]. The specifics of how the secrets are generated will be explained later, but now we must point out that these secrets are bound to capabilities that tell

the device what kinds of things the holders of the secrets can do.

3.4 Object Types

SCARED objects are identified using a 128-bit number called the object id. This object id is assigned by the storage device to the object when it is created as a result of a request by a SCARED client. Clients use this id to access this object until it is deleted as a result of a request by a SCARED client.

The storage device generates object ids in such a way as to ensure that the same number is never generated twice. While we do not make use of this property currently in Brave, later we can use it to generate capabilities that enable operations for specific objects. If an id is reused for a previous object, the capabilities that corresponded to the old object would be able to be used for the new one.

Attached to each object is an info block and ACL. The info block is a variable sized block of information, on the order of 1K. It is updated atomically by clients, and is used by the clients for file system meta-data. The data in the ACL is used by the storage device to restrict access. It can also be updated atomically by the clients. The ACL is made of a list of users and permission bits.

The storage device also maintains create, modify, and access timestamps for objects. To avoid requiring clocks on the storage device these timestamps are updated using times passed by the clients when requests are made.

In addition to timestamps, version numbers are maintained for the info block, ACL, object data, and the object itself. The version number is changed every time the relevant item is updated. This allows for detection and recovery of simultaneous updates of the object and for cache validation. All update operations take a version number that will abort the operation if the version number of the request does not match the actual version number.

When an object is created, the client specifies the creation of a file object or a directory object. Each have specialized operations and structures that are intended to make the storage and management of files and directories easier.

File objects are presented as logically contiguous byte streams. They are read and written by providing an offset relative to the beginning of the stream, and a length. They can also be truncated or expanded. File objects are identical to those used in NASD except for the addition of the info block and the ACL to the object.

Directory objects are presented as arrays of directory entries. Each entry contains an entry identifier (etag), lookup identifier (ltag), a version number, and entry data. The etag and ltag are 128-bit numbers that identify the entry. The etag is assigned by the storage device when the entry is created and cannot be changed.

The storage device will not generate the same etag twice in the same directory object. Not only does this avoid race conditions between deletions, queries, and updates, but it also is used to enable capabilities to restrict access to specific directory entries.

The ltag is generated by the client and can be changed with the restriction that no two entries in the same directory object will have the same ltag. The version number is changed every time the entry is updated. The entry data is provided by the clients and can be changed at any time.

Most directory operations identify the entry on which to operate by providing the etag. The only operation that uses the ltag is the lookup operation. Since lookup is the most common directory operation, we want to be able to find an entry based on the filename. For this reason, clients map filenames to ltags and provide them to the server for lookup operations.

The ltag is calculated by applying a collision resistant hash function to the filename. While this masks the filename from the storage device, applying the function to guesses of the filename can unmask the name quickly. To truly hide the name a keyed hash function must be used.

3.5 Caching

We chose to use a caching mechanism similar to AFS for SCARED. Whenever a client reads an object, the SCARED storage device marks that client as interested in changes to the object. Then if the object changes or the device is no longer able to maintain the list of clients interested in the object, it sends an invalidation message with the object id of the object that changed. Once the invalidation message is sent, the client's interest in the object will get unmarked unless the client again reads the object.

Since clients access SCARED devices using TCP, the client's interest in an object will get unmarked if the TCP connection drops. A client can revalidate its cache if needed by checking to see if the version number has changed.

We chose to use this caching mechanism since it provides stronger caching semantics than time based algorithms such as those used by NFS, but does not require leases or locking such as those used in DFS. The need to avoid locking is particularly important since we want to use SCARED in very wide area networks and we do not want rogue or malfunctioning machines locking out valid machines from accessing objects. In wide area networks it is possible for a malfunctioning machine is in an unrelated administrative domain to cause denials of service due to lock contention.

3.6 Brave File System

Since we are building a file system on top of SCARED devices, there is a natural mapping of file system structures to SCARED structures: each directory is stored in a directory object and each file is stored in a file object. In our initial implementation we maintain this one-to-one mapping. In

the future, files and directories may be striped across data and directory objects to improve performance and scaling. Mirroring may also be used to improve performance and reliability. Even with striping and mirroring, the basic concepts presented here remain unchanged.

To introduce Brave, we must first introduce the mount point or root of the file system. The root of the Brave file system is a directory. In Brave there is nothing special about the root; any directory object can be used as the root of the file system. This directory object contains an entry for each file or directory in the root of the file system. Brave clients will get information stored in each entry to discover the top level of the file system hierarchy, and then recursively traverse directory objects to find desired files or directories.

Figure 1 shows the data structures that are stored in the directory entry. Only the first three fields in the entry are used by the SCARED device. The rest of the fields are stored in the entry data by the SCARED device and left uninterpreted. The lookup tag (ltag) is generated by hashing the file name at the client. The current implementation of Brave uses SHA [18] to hash the file name to a ltag. By storing the hash of the file name in the lookup tag, the SCARED disk is able to return the desired entry on a lookup without sending all the directory entries to the client. This saves network bandwidth, as well as optimizing one of the most common directory operations. By using a hash of the file name instead of the file name itself, the storage device is able to search on a fixed size number and in the future can preserve the secrecy of the file name if constructs such as HMACs [12] are used.

Because the lookup tag is the hash of the file name and not the file name itself, the file name of the directory entry needs to be stored in the directory entry. A SCARED device does not use the file name since lookups are done with an ltag, so the file name is stored in the entry data, which is the uninterpreted portion of the entry.

The main purpose of a directory entry is to provide a mapping between a name and a file or directory. For this reason, a pointer to the location of the file or directory follows the file name. The pointer comes in two forms: a symbolic link or an object identifier and host name (hard link). A hard link is composed of the host name of the SCARED device that stores the object containing the file or directory and the object identifier (OID) of that object. A hard link preserves referential integrity. This means that an object referenced by a hard link will not be deleted until the link is deleted.

Unlike a hard link, a symbolic link does not retain referential integrity. Instead, the symbolic link is a string that is passed back to the operating system to be resolved. The string need not reference a file that is part of the Brave file system or even a file that exists.

Referential integrity is preserved by using the info block that is part of every SCARED object. The info block is stored uninterpreted by the SCARED device. The Brave clients store the entry tag, the OID, and the host name of the directory entry that references the object. Using the entry data and the info block, clients have pointers from the entry to the object and vice-versa. The atomic updates of info blocks also enable us to perform atomic operations across devices.

Figure 2 shows an example of a directory containing a file and directory. The file is stored on the same SCARED device as the directory, but the directory is stored on a different device. It is important to note that the SCARED device stores the info block, the entry data, and the file data, but does not interpret the contents. The figure shows both the forward links from the entries to the objects they reference, and the backward links from the objects to the entries that reference them.

4 Security Infrastructure

When designing the security interface to SCARED devices we wanted to make the administration of the devices as simple as possible and be able to fit into existing security infrastructures.

4.1 Key Management

Access to a storage device is controlled using a single secret, K_d which is shared by the SCARED device and its owner. When the owner wishes to generate a key for another user to allow access to the device, it generates a capability. Although SCARED supports a variety of capabilities, currently we only use identities. This capability, $Kdata_u$, will be encoded as a concatenation of three values:

$$Kdata_u = \{\text{“Identity”}, ids, expiration\}$$

where “Identity” is a constant string indicating an identity capability. ids is the set of ids, including group ids, used by the user. $expiration$ is the time limit, defined by the device, of the validity of the capability.

The owner then uses a pseudo-random function prf) to generation a new secret,

$$K_u = prf_{K_d}(Kdata_u)$$

and gives K_u and $Kdata_u$ to the user. Note that K_u must be kept secret, but $Kdata_u$ is not considered secret. We call K_u the access key since it is used by the SCARED device to validate access.

To authenticate responses from a SCARED device a similar key is generated called a response key, K_r. The data used to generate the key, $Kdata_r$, does not have any capabilities encoded in it since it is only used to authenticate responses and does not have any access privileges.

When an access key is generated, the ids in the capability must be verified to ensure that user is not given the ability

<table>
<tr><td rowspan="2">Entry tag</td><td rowspan="2">Lookup tag</td><td rowspan="2">Version</td><td rowspan="2">Filename</td><td rowspan="2">Location type</td><td>OID</td><td>Hostname</td></tr>
<tr><td colspan="2">Symlink</td></tr>
</table>

Figure 1: Brave directory entry layout.

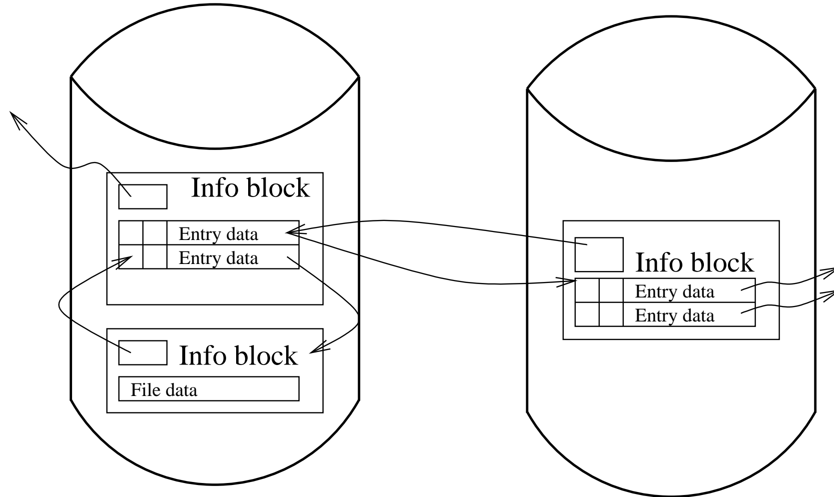

Figure 2: An example directory structure stored in a meta-data object.

to access files for identities other than her own. Thus the owner must authenticate the user. Since response keys do not enable any kind of device access, the owner need not authenticate the user, instead, the user must ensure that the owner that is generating the response key is the real owner of the device. We originally designed the system to use request keys and response keys to allow clients to share access keys, but still authenticate responses without exposing themselves to attacks from other clients. As will be pointed out later when talking about exposures that can arise using shared local caches, response keys also enable a kernel to validate responses without having to act on behalf of any user.

Once the user has K_u and $Kdata_u$, when $Kdata_u$ is sent to the SCARED device with the secret K_d, the device can calculate K_u which is then used as a shared secret between the user and device. Apart from establishing a shared secret the derivation using $Kdata_u$ binds the information encoded in $Kdata_u$ to K_u. Thus the SCARED device can make access decisions based on the identities encoded in $Kdata_u$.

While the exact protocol is explained in previous work [16], suffice it to say that the request and response keys are used to establish a fully authenticated channel between client and server. As noted later, the response key can also be used to enable link-level encryption.

4.2 Manual Key Distribution

The simplest form of key distribution is when an owner, or administrator acting on behalf of the owner, generates a key for a user and gives it to the user. In this case the user and owner mutually "authenticate" when making the request. Once the user has the $Kdata_u$ and corresponding K_u along with $Kdata_r$ and its K_r, it can pass the pair to the brave file system and access the storage device.

4.3 Key Servers

While manual key distribution may be a simple and convenient mode of operation when very few people and devices are involved, the problem becomes much more complicated when either of these numbers becomes large. (It can be noted that there is a case where a large number of equally trusted devices can share a key which would make this scenario reasonable with large numbers of devices and few users.) To handle larger number of users and devices, and to fit into existing security infrastructures, we use a trusted third party, called a key server, that holds the keys on behalf of the owner and authenticates users using these infrastructures.

Figure 3 shows two storage devices whose keys are generated by different key servers. Each device shares their secret with the key server that will generate their keys for clients that will access the device. When a client needs to access the storage device *d1*, the client will ask it to identify its key server. In figure 3 the key server for *d1* is *s1*.

Figure 3: An example of two storage devices with two different key servers.

The client will then authenticate using an existing protocol, such as LDAP [9] or SSL [6], to get keys to access *d1* and then start making requests to *d1*.

One thing to note is that we cannot rely on *d1* revealing the correct key server. If an attacker is masquerading as *d1*, he could lie about the key server and direct the client to a key server under his control. For this reason when the client requests keys from the key server, it must verify that the key server is authorized to issue keys for the storage device.

The two other things we require of existing infrastructures are the ability to establish a private channel and the ability to map users to groups. After authenticating a user, the key server must be able to pass K_u and K_r to the user in a confidential way. Further, when generating K_u the key server must be able to enumerate the identities used by the user.

Each SCARED device in the Brave file system may be using a different key server to distribute keys and none of these key servers need be in the same security domain or even use the same authentication scheme as any other. Since a client does not know a priori which key server is used by a given storage device, we encode this knowledge in a device descriptor block that is stored on the storage device and is publicly accessible. This way when a client first accesses a device it can discover the key server used by the device and obtain the keys necessary to make requests.

4.3.1 Password Based Key Server

A simple example of key server is a password based key server. This key server has access to a database of passwords for a set of user ids and a database of groups that correspond to a user id. In our implementation we maintain the databases at the local machine, other databases such as Sun's Network Information System (NIS) and LDAP [9] could also be used.

When a client contacts the server, it sends its user id and proves that it knows the password and which point the key server generates the key pairs with the list of identities of the user stored in the key data and sends them to the client.

In order to send the keys to the client the server must have a confidential channel. Further, the server must be able to prove to the client that it is indeed able to serve keys for the storage device. Because the information about which key server corresponds to a storage device was obtained without any secrets to authenticate the response, the client cannot be sure that the keys it is receiving is for the real device or a fake. We get around both of these problems by using SSL [6] with server authentication. The certificate received from the server lists the name of the storage device in its distinguished name. This gives us a secure channel for the client to negotiate and receive keys and also proves to the client that it is communicating with the correct key server for the device.

Note that when the client receives the certificate from the key server at the beginning of the SSL exchange, it must decide whether or not to accept the signer of the certificate. Clients can have a variety of policies for certificate acceptance, even a list of valid certificates if necessary. There is no requirement to have a single trusted signer.

4.3.2 Certificate Based Key Server

We also designed a key server that authenticates clients based on their X.509 certificate. This implementation is similar to the above except that the clients identities are stored in the certificate rather than a database. SSL is used to establish a channel to the key server and the key server simply validates the client's certificate and then puts the identities from the certificate into the key data that is sent to the client.

5 Implementation Experience

When we implemented our design, there were some things that turned out not to be as practical as envisioned. The use of a single file system cache in the kernel also posed some problems that needed to be addressed to plug security holes. Finally some issues related to encryption turned out to be harder than envisioned while others proved easier.

5.1 ACLs Versus Credentials

When we originally designed the Brave file system we were completely capability based. In each directory entry there were capabilities and encryption keys used to access the object referenced by the entry. This object was encrypted with keys of the users and groups that could access that object. Thus it was easy to give and revoke access to files and directories. Although a revoked user or group could copy off the necessary keys before the revocation occurred,

it was deemed acceptable since they could have also copied the content before revocation. The bigger problem turned out to be revoking group and user encryption keys. That entailed doing a large scale re-encryption of every occurrence of the key in the file system. The other thing to note is that a client must use a unique key for each object in the network. In the case of objects with users or groups of differing access privileges there may be more than one key for each object.

Because the revocation of keys was such a problem, we broadened our concept of capabilities to include the capability to act on behalf of a user and groups. We called keys with this new capability identity keys. This let us introduce ACLs into the object module and greatly reduce the number of keys generated by the user. In general, to access any object on a device the user needs only one key to identify themself. This also means that the number of keys managed by the storage device depends only on the number of users verses the number of users multiplied by the number of objects accessed.

5.2 Local page cache

Local caching of the data and meta-data from storage devices is necessary to achieve good performance from a distributed file system. This caching also has the potential to leak data to other clients. When we implemented Brave as a client file system, we only cached a data from a remote object once. This may result in a situation where a user reads a file that has be read into the cache by another user. The kernel checks the identity encoded in the key data to ensure that the user can actually access the file, but it has no way of knowing if the key data is actually valid. Before identities in key data can be used to access cached data, they must have been validated by the storage device. So in the scenario just described, the kernel would need to do a simple operation on the device using the key given to it by the second user to ensure that the key data can be trusted if the key has not been used before.

There is also the dual attack in which the attacker provides bogus response keys to the kernel to trick it into putting data from a bogus server into the cache. When another user accesses the data with a real response key, the key will never get used since the data is already in the cache. For this reason, the kernel should get its own response keys and only use request keys obtained from users. While our derivation scheme is designed to permit this and the key servers outlined here support this mode of operation, there may be security infrastructures that do not support the ability for an unauthenticated client to authenticate a response key obtained from a key server.

There is also a problem resulting from the asynchronous nature of cache invalidations. If an attacker prevents the receipt of a cache invalidation from the storage device, the attack will only be detected after a request is generated by the client. This means that a client that has achieved a good working set in its cache could be prevented from ever receiving cache invalidations by an attacker. To limit the length of time such an attack will succeed we will do periodic requests to the storage device that are only for heart beat purposes.

5.3 Passing keys to the kernel

As noted earlier, we use key servers to bridge the simple capability based security protocol with existing security infrastructures for managing groups and users. The kernel needs keys and corresponding key data in order to make requests to the storage devices, and only the user has access to the secrets to communicate with the key server. To facilitate the transfer of these secrets on Linux, a special file is created in the /proc file system. The user then runs a user-level application that will continually read that file for key requests from the kernel. Whenever the kernel needs a key it will find an open read request on that special file that is being issued by an application with the same user identifier as the process making the file system request. The kernel will then cause the read to return with the name of the device the kernel needs to access, and its key server, if known.

The user-level application will then contact the key server, authenticate the user to the key server and ensure the key server is authorized to generate keys for the storage device, and it will pass the key and key data received from the server to the kernel by writing to the special file. If needed, the application may need to prompt the user with authentication information or secrets to access private keys.

It is also possible that the user previously received a key and key data for the storage device using manual key distribution and stored them in the local file system. In this case the application simply retrieves the keys locally.

5.4 Encryption Revocation

Brave is architected such that the file system data and meta-data can be encrypted at the client. Client machines that have a shared secret can simply encrypt the meta-data and data as it is sent to and from the storage device. This model works and is simple and even practical in the case of a business that outsources its storage management and is able to put a copy of the same key in all the clients.

The use of a shared secret is too simple for general use. In general we would like to encrypt data and meta-data on a per file or directory basis using encryption keys known only to users and groups, not clients. For these cases we have a file system filter that encrypts file data using user managed keys much along the lines of other cryptographic file systems. In this case Brave simply provides an authenticated repository. Because SCARED devices have less knowledge of the structure of the file system, it is possible to hide even the relationship of objects in a file system from devices and eavesdroppers on the network.

5.5 Link Encryption

As we discuss in the next section, client encryption is not sufficient to protect some of the confidentiality of the distributed file system. Because parameters of requests are sent in the clear, an eavesdropper can figure out what parts of objects are being accessed and how. In many cases this information leakage can be used to attack the system.

Because the response key is a shared secret between the storage device and the client, we can use it to generate a session key to do link level encryption. This results in a fully authenticated confidential channel between client and server. Although it may be tempting to skip encryption of the client encrypted data to be stored on the device, if it is not re-encrypted using the session key, an attacker can detect duplicated byte patterns in the data and glean information about the data being accessed. While it may seem intractable, using the burstiness of data access makes this a very reasonable attack.

6 Security Weaknesses

Even with the use of encryption information is still leaked. Obviously, a storage device knows a lot about the meta-data of an object, but there are even more sources of information available to a storage device. Traffic analysis yields relationships between objects on a disk, so if related files and directories are on the same device, the device can deduce relationships. For example, when a READ operation on a directories followed by GETINFO operations on a set of objects, the device can deduce that an "ls -l" has occurred and therefore figure out which objects belong to the given directory.

Although it is tempting to believe that by using client encryption link level encryption is not needed, the lack of link level encryption means that an eavesdropper on the network can do even more traffic analysis than the storage device since it can correlate traffic from a client to different storage devices. It can also correlate network events, such as remote system logins, to storage requests to deduce the location of specific files, such as $/etc/passwd$. Since encrypted content is only changed by the clients, an eavesdropper can figure out when a file changed and which part has been updated.

Note that even block based storage devices are susceptible to this kind of analysis. Because blocks that belong to a given file or directory are usually allocated contiguously for performance reasons, and read sequentially, traffic analysis allows a storage device or eavesdropper (if the link is not encrypted) to figure out the blocks that correspond to a file system object.

Another exposure that is not immediately evident is that by allowing direct access to the storage devices by clients, a client who does not have access to a directory may still be able to access its child. Normally if a client cannot access $/a/b$, it will not be able to access $/a/b/c$ even if the permission of the file c allows the client access. When using object disks, the client can access c if the ACL of c allows the client access and the client has c's object id, which is not considered secret. Note that this problem occurs with most distributed file servers at the administrative boundaries. In AFS clients can directly access a volume on the system if the ACL permits, even if it cannot access the parent. In OceanStore, knowledge of a GUID allows access to that object if the ACL permits, even if access to the parent is not permitted. SFS has the same scenario when using unique identifiers. It should be noted that our original design that used encrypted capabilities in directory entries to access the object referenced by those entries did not suffer from this exposure. Only by having access to the parent could a client access an object.

While it is relatively easy to encrypt content as it is generated and changed, it is very difficult to change encryption keys. Re-encrypting an entire file system is not practical. Even if each object has its own encryption key and only that key is re-encrypted, walking the entire file system is not very tractable. Although it is convenient to say that theoretically only new content needs to be encrypted with new keys since old content is already compromised, in practice that will rarely be the case and compromised encryption keys need to be changed to prevent further compromise of old content.

Finally, there is a user interface problem of reflecting the security domains involved and the level of security when accessing a file. Consider the case $/a/b$. Since a is directly referenced by the directory that has been mounted by a client, the client will have a good idea about the authentication methods used to access a. It is possible that a is on a storage device that uses weak authentication, or possibly a is on a storage device that does not do any authentication at all. Thus, even though b may be on a highly secured device, since a is not well authenticated, information returned by a for b may be subverted to point to a bogus b on a bogus device. This highlights that a path is only as trusted as its weakest link. The file system layer of most file systems and most user interfaces do not reflect different levels of trust in a file or directories path.

7 Conclusions

The Brave file system has strong security features with semantics and behavior that resemble other classical distributed file systems.

Our use of SCARED object storage devices allows us to store file system data and meta-data on SCARED devices without having to expose knowledge of this data to the devices. In particular, all file and directory data can be encrypted as well as the relationships between files and data. This allows us to encrypt even more of the file system meta-data than encrypting file filters.

Because each device can be administered separately, a Brave file system can span different administrative domains. By using key servers, even different security infrastructures can be used in the same file system.

By using the SCARED object model not only are distributed file system operations enabled, but the ability to control access to individual objects at the device is enabled. This allows us to avoid having file servers to manage file and directory access.

Finally, we enable strong mutual authentication between clients and devices using a simple keying scheme that requires a storage device to manage a single secret.

The design of Brave enables many of the features of more experimental file systems while maintaining semantics needed for use as normal distributed file system.

References

[1] Marcos K. Aguilera, Minwen Ji, Mark Lillibridge, John MacCormick, Erwin Oertli, Dave Andersen, Michael Burrows, Timothy Mann, and Chandramohan Thekkath. Block-level security for network-attached disks. In *USENIX Conference on File and Storage Technologies (FAST)*, March 2003 (to appear).

[2] S. M. Bellovin. Security problems in the TCP/IP protocol suite. *Computer Communication Review*, 19(2):32–48, April 1989.

[3] Matt Blaze. A cryptographic file system for UNIX. In *First ACM Conference on Communication and Computing Security*, pages 9–16, November 1993.

[4] B. Callaghan, B. Pawlowski, and P. Staubach. NFS version 3 protocol specification. RFC 1813, June 1995.

[5] Giuseppe Cattaneo, Luigi Catuogno, Aniello Del Sorbo, and Pino Persiano. The design and implementation of a transparent cryptographic filesystem for unix. In *USENIX Annual Technical Conference 2001*. USENIX, June 2001.

[6] T Dierks and C Allen. The TLS protocol version 1.0. RFC 2246, January 1999.

[7] Garth A. Gibson, David F. Nagle, Khalil Amiri, Fay W. Chang, Eugene M. Feinberg, Howard Gobioff, Chen Lee, Berend Ozceri, Erik Riedel, David Rochberg, and Jim Zelenka. File server scaling with network-attached secure disks. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (Sigmetrics)*, pages 272–284, June 1997.

[8] The Open Group. *Protocols for X/Open PC Internetworking: SMB, Version 2*. September 1992.

[9] J. Hodges and R. Morgan. Lightweight directory access protocol (v3): Technical specification. RFC 3377, September 2002.

[10] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. In *ACM Transactions on Computer Systems*, volume 6.1, pages 51–81, February 1988.

[11] M. L. Kazar, B. W. Leverett, O. T. Anderson, V. Apostolides, B. A. Bottos, S. Chutani, C. F. Everhart, W. A. Mason, S.-T. Tu, and E. R. Zayas. Decorum file system architectural overview. In *Proceedings of the Usenix Summer 1990 Technical Conference*, pages 151–164. USENIX, June 1990.

[12] H. Krawczk, M. Bellare, and R. Canetti. HMAC: Keyed-hashing for message authentication. RFC 2104, February 1997.

[13] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *ASPLOS*, December 1999.

[14] D. Mazieres, M. Kaminsky, M. Kaashoek, and E. Witchel. Separating key management from file system security. In *SOSP*, December 1999.

[15] C. Neumann and T. Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications Magazine*, September 1994.

[16] Benjamin C. Reed, Edward G. Chron, Randal C. Burns, and Darrell D.E. Long. Authenticating network-attached storage. *IEEE Micro*, 20(1):49–57, January/February 2000.

[17] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. NFS version 4 protocol. RFC 3010, December 2000.

[18] U.S. Government. Proposed federal information processing standard for secure hash standard. Federal Register, January 1992.

[19] Erez Zadok, Ion Badulescu, and Alex Shender. Cryptfs: A stackable vnode level encryption file system. Technical Report CUCS-021-98, Columbia University, June 1998.

[20] Erez Zadok and Jason Nieh. Fist: A language for stackable file systems. In *Proceedings of the USENIX Annual Technical Conference*. USENIX, June 2000.