Securing Wide-area Storage in WheelFS

by

Xavid Pretzer

B.S., Massachusetts Institute of Technology (2008)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2009

© Massachusetts Institute of Technology 2009. All rights reserved.

Author ....
Department of Electrical Engineering and Computer Science
May 22, 2009

Certified by ....
Jeremy Stribling
PhD Candidate
Thesis Supervisor

Certified by ....
M. Frans Kaashoek
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Professor of Electrical Engineering
Chairman, Department Committee on Graduate Theses

Securing Wide-area Storage in WheelFS

by

Xavid Pretzer

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 2009, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

WheelFS is a secure wide-area distributed file system that gives applications fine-grained control over the various trade-offs inherent in wide-area storage. Because of the security risks of running a wide-area application on the public Internet, WheelFS aims to allow for secure operation in an untrusted network with untrusted clients, while still enforcing access control against these clients and allowing them to safely share data with each other. By using SSH connections and RSA public keys, it gives users a familiar and secure authentication interface and maintains efficient secure communication in an untrusted network. It also uses server-supplied SHA-256 checksums to allow secure client-to-client data sharing between mutually-distrustful clients. Experiments on PlanetLab, a wide-area testbed, shows that wide-area file transfer and cooperative fetch operations take 1.5–2 times as long with these security measures compared to the unsecured prototype.

Thesis Supervisor: Jeremy Stribling
Title: PhD Candidate

Thesis Supervisor: M. Frans Kaashoek
Title: Professor of Computer Science and Engineering

Acknowledgments

I am grateful to my friends and colleagues for their assistance throughout the journey to the completion of my thesis.

I would particularly like to thank Jeremy Stribling for his explanations of the inner workings of WheelFS, his extensive support and feedback, and his regular assistance with thorny distributed debugging issues.

Much thanks is also due to my advisor, Frans Kaashoek, for support and feedback throughout the process and for assistance in particular with the design phase of this thesis. Robert Morris also made significant contributions to this thesis's design.

I would like to thank everyone who has contributed to WheelFS for providing me with such a good foundation for me to base my research, and for putting up with the times when I inadvertently broke the build.

Thanks to Jeffrey Hutzelman and Greg Price for some suggestions on algorithmic flexibility to allow migration away from broken cryptographic algorithms.

Thanks to everyone who has worked to create and maintain the PlanetLab distributed testbed, without which I would not have been able to obtain as meaningful results.

Finally, thank you to everyone in PDOS who gave me feedback.

Contents

1 Introduction **11**

1.1 Motivation . . . 11

1.2 Problem Statement . . . 12

1.3 Thesis Contributions . . . 13

1.4 Outline . . . 14

2 WheelFS Overview **15**

2.1 Nodes . . . 15

2.1.1 Storage Nodes . . . 15

2.1.2 Configuration Service . . . 16

2.1.3 Clients . . . 16

2.2 Cues . . . 17

2.2.1 Inter-client Cooperation . . . 17

3 Security Design **19**

3.1 Threat Model . . . 19

3.2 Trust Model . . . 20

3.3 SSH Connections . . . 21

3.4 RSA Authentication . . . 22

3.4.1 Server Identities . . . 22

3.4.2 User Identities . . . 23

3.4.3 Alternative . . . 23

3.5 File Permissions . . . 24

3.5.1 POSIX Interface 25
3.6 Inter-client Communication 27
3.7 Deployment Example 28
3.8 Attack Analysis 29
3.8.1 Malicious Network 30
3.8.2 Malicious Client 30
3.8.3 Lying Cooperative Client 31
3.8.4 Rogue Server 31
3.8.5 Compromised Server 31

4 Implementation **33**
4.1 Overview 33
4.2 Limitations 34

5 Results **37**
5.1 File Transfer 37
5.2 Cooperative File Distribution with **.Hotspot** 40

6 Related Work **43**

7 Conclusion **47**
7.1 Future Work 47
7.1.1 Protocol Evolvability 48
7.1.2 Performance Optimizations 48
7.1.3 Enhanced ACL Support 49
7.1.4 Enhanced PlanetLab Integration 50
7.1.5 Configuration Key Distribution 50
7.1.6 Signed Checksums 50

List of Figures

2-1 WheelFS Structure 16

3-1 Trust Model 20

3-2 Accessing WheelFS permissions via the POSIX interface 26

5-1 File Transfer Times 38

5-2 Cooperative Transfer Times 41

Chapter 1

Introduction

1.1 Motivation

Wide-area applications are increasingly popular in the modern Internet, as running a distributed application that spans multiple distant data centers has distinct advantages. They can be more resilient to site-specific outages such as natural disasters and they can optimize performance such that the closest site to a given user handles that user's requests. Being able to scale up an application by adding a new data center also makes it easier to add capacity without disrupting existing deployments. However, applications running in wide-area networks are faced with many trade-offs, such as having to choose between strong data consistency and quick responses in the face of failures. Furthermore, moving applications out of trusted local-area networks and onto the public Internet dramatically increases their vulnerability to attacks, requiring careful attention to security.

WheelFS [18] is a new distributed file system designed as a storage system for wide-area applications. It simplifies the building of wide-area applications by providing an easy-to-use storage layer that can be conveniently tailored to the requirements of a given application. The original design for WheelFS was based on unencrypted TCP channels for all communication and performed no authentication or access control for clients or servers.

Unfortunately, there are many possible attacks against such a system. An adver-

sary can impersonate a server to gain access to arbitrary data in the system. An adversary can use a client to read or modify data belonging to other users. An adversary can even read or modify data in transit between clients and servers without joining the system at all. Finally, an adversary can act normally to servers while still feeding modified data to other clients via inter-client cooperative sharing. These attacks could prevent WheelFS from being deployed in an untrusted network environment like PlanetLab [4]. Thus, a strong security system that integrates well with WheelFS while maintaining its simple interface would allow for real-world deployments of WheelFS and thus increase its utility.

1.2 Problem Statement

This security design targets a proposed deployment of WheelFS as a service for PlanetLab, a wide-area research network of machines distributed throughout the globe [4]. PlanetLab nodes communicate over the Internet, with no additional security, so an adversary could eavesdrop on communication between nodes to learn private data, or edit messages in transit to modify data. In addition, an adversary could try to hijack WheelFS by connecting malicious nodes to it. Finally, PlanetLab has many independent users who may not all follow the same security precautions, so a PlanetLab user whose account has been compromised could try to read data they lack access to, modify data they lack permission to write, or mislead other clients as to the state of a file being cooperatively fetched. While PlanetLab is a single environment, these characteristics are shared to varying degrees with many other wide-area environments.

The characteristics of PlanetLab lead to several design goals. Due to PlanetLab's use by many researchers from various organizations, WheelFS should support secure use by many mutually-distrustful applications, which requires the system to support access controls based on different user identities. PlanetLab already uses RSA key pairs to authenticate users who connect over SSH; it would be ideal to be able to use these same keys to authenticate users to WheelFS as well. The system should

also be protected against the attacks enabled by communicating on a public network. Due to the large and regularly-changing membership of PlanetLab, the design should be able to handle changes in node membership conveniently and easily. To maintain the advantages of WheelFS's easy-to-use interface, it is also important to make secure operation as simple and unobtrusive as possible for application developers. Finally, WheelFS should keep the overhead of using security to be low to maintain the unsecured prototype's level of efficiency.

Many previous distributed file systems have addressed security; however, these systems have various issues, particularly in credential management and scalability, that prevent them from addressing all of WheelFS's needs. For further discussion of existing security systems, see Chapter 6.

1.3 Thesis Contributions

This thesis describes a security implementation for WheelFS that provides encrypted communication, secure authentication, and verifiable inter-client sharing while minimizing the complexity of a secure deployment. It assumes trusted, non-Byzantine servers, but allows clients or the network to be malicious. It uses SSH [20,21] to provide secure encrypted channels for communication and RSA [13] public keys for authentication. Since these are already in wide use at WheelFS's target deployment environment of PlanetLab, there is little administrative overhead for this system, and it provides secure encrypted communication and authentication for WheelFS. Server-enforced access control lists provide flexible access control and allow isolation of different users and applications. This thesis also adds checksumming (using the SHA-256 hash function [16]) to allow client-to-client communication to be verified and thus to prevent a malicious client from misleading another client with modified data.

With these measures, WheelFS can now be run securely over the public Internet without divulging private data or opening itself up to attacks while remaining easy to manage and practical for realistic distributed applications. This makes it possible

to deploy WheelFS on PlanetLab and many other similar wide-area environments.

1.4 Outline

The remainder of this thesis is organized as follows. Chapter 2 discusses the aspects of WheelFS that are relevant to the security system. Chapter 3 discusses the proposed security design for WheelFS. Chapter 4 details the current implementation of this design, and Chapter 5 presents benchmarking results for this implementation. Chapter 6 discusses previous work in distributed storage security. Finally, Chapter 7 concludes and presents some possibilities for future enhancements of this security design.

Chapter 2

WheelFS Overview

WheelFS is a distributed file system targeted at wide-area applications that allows for fine-grained configuration to handle the trade-offs necessary in wide-area storage [18]. A brief overview of WheelFS's structure is necessary to understand how security fits into the broader picture.

2.1 Nodes

As illustrated in Figure 2-1, WheelFS consists of two types of server nodes, configuration and storage, in addition to client nodes. While all three nodes can run on a given machine, each node can be run independently, and there will often be clients who do not run either of the server nodes. The configuration service tracks the membership of the system and determines how files are spread among the storage nodes. Storage nodes handle the long-term storage of data and handle client requests to read or modify the data. Clients provide an interface to WheelFS for applications and do local caching to improve performance.

2.1.1 Storage Nodes

Storage nodes provide for the persistent storage of WheelFS data and provide that data to clients.

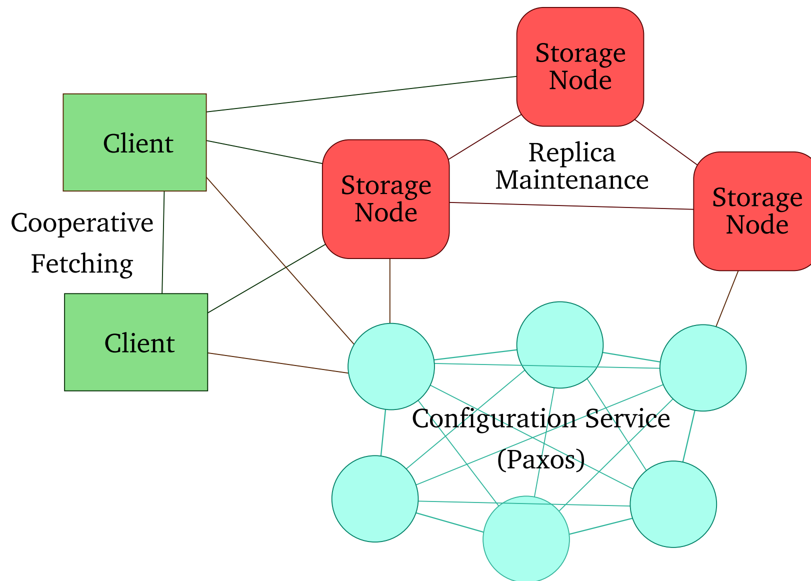

Figure 2-1: WheelFS Structure
WheelFS is made up of configuration nodes, storage nodes, and client nodes.

Each file and directory is managed by a specific storage node called its 'primary'. A file's primary is responsible for choosing backup storage nodes for the file and keeping their replicas of the object up-to-date.

2.1.2 Configuration Service

WheelFS configuration nodes use the Paxos protocol [11] to coordinate when configuration and storage nodes enter and leave the system. Both storage nodes and clients use the configuration service to obtain the mapping from files and directories to storage nodes; they cache this information locally and refresh it periodically.

2.1.3 Clients

The WheelFS client uses the FUSE userspace file system library [7] to provide a standard POSIX file system interface to applications. Using FUSE as opposed to an in-kernel file system does cost some in terms of efficiency, but it makes testing and portability easier.

Clients maintain a local cache of recently-accessed data for efficiency. Storage

nodes give leases to clients, promising to notify clients when files change within a certain period of time; thus, a client can assume its cached data is still current if it has not received a notification.

2.2 Cues

The novel feature of WheelFS is its 'semantic cues', special path name components which allow parameters to be configured on a per-file or directory basis. For example, reading `/wfs/`**`.EventualConsistency`**`/foo.txt` is a request to read `/wfs/foo.txt` with relaxed consistency requirements. For the most part, cues are orthogonal to WheelFS's security information, and thus, with one exception (see section 2.2.1), will not be discussed here.

2.2.1 Inter-client Cooperation

When opening a file using the **.Hotspot** cue, the client will fetch from the file's primary a list of clients that have blocks of that file cached, and will attempt to fetch blocks from other clients' caches when possible instead of directly from the primary. This allows many clients to read a large file simultaneously without overloading the file's primary. The primary selects clients to return based on their proximity to the requesting client using Vivaldi network coordinates [6], so this also has the advantage of reducing network latency.

Chapter 3

Security Design

This thesis first defines the security model for WheelFS and the security goals this leads to, and then discusses a security design that achieves these goals.

3.1 Threat Model

WheelFS is concerned with a variety of attacks that are possible due to its communication taking place over the public Internet. In particular, it assumes an adversary can read any data that passes over the network or intercept traffic and modify it. An adversary can also create their own instance of any WheelFS node and try to join it with a WheelFS deployment.

In addition, WheelFS is concerned with malicious clients, mainly due to the possibility of security holes in applications running on WheelFS or compromises of identities belonging to individual WheelFS users. Thus, WheelFS is concerned about an adversary with a valid client identity attempting to read or modify files in ways that violate his identity's permissions. WheelFS is also concerned about clients misleading other clients about file contents during inter-client cooperative fetching.

The design in this thesis is not concerned with compromises of storage nodes or configuration nodes. In WheelFS's target deployment, these will be managed by a single group that can enforce strong security practices.

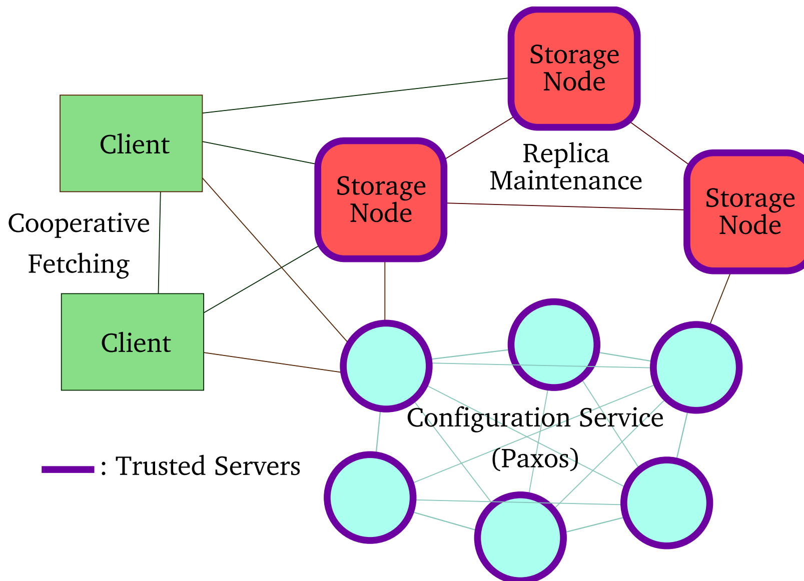

Figure 3-1: Trust Model
Configuration and storage nodes are trusted, but clients are not.

3.2 Trust Model

The trust model for WheelFS is one where servers, both storage nodes and configuration nodes, are trusted, but clients and the network itself may be malicious. This model is illustrated in Figure 3-1. While this does limit WheelFS's use in certain situations, it allows for an efficient and easy-to-use security setup for WheelFS's target case where all servers, while physically separated, are still run by the same organization. If individual applications seek additional end-to-end security properties, they can encrypt or sign data at the application level.

WheelFS does not attempt to handle Byzantine behavior of an authenticated server, whether due to a server compromise or a malfunction.

WheelFS manages its untrusted clients by requiring each client operation to be associated with an authenticated user, and then assigning each user permissions using access control lists that limit what operations are permitted. There are several advantages of this model over one where any authenticated client is allowed to perform any valid operation, such as NFSv3 [15] and Pangaea [14]. This setup allows multiple applications to run on the same WheelFS instance without exposing one ap-

plication to risks caused by security holes in another. Similarly, even within a single application, ACLs can be used to prevent an application from, say, overwriting its own code in WheelFS, limiting the potential impact of an attack. It also allows the maintainers of various applications to be given direct WheelFS access even if they are not completely trusted, and makes it easier to recover if a maintainer's machine is compromised or stolen.

WheelFS uses the POSIX/FUSE interface to manage ACLs, rather than introducing a separate API.

3.3 SSH Connections

To prevent adversaries in the network from being able to observe confidential data or modify data in transit, all RPCs in a secure WheelFS instance are sent through SSH channels. SSH provides well-tested and reliable encrypted channels without a large amount of overhead [23]. Due to SSH's widespread use, tools for generating and examining SSH keys are widely available.

The main overhead of using SSH channels is when setting up a channel for the first time and authenticating the sender and receiver to each other. Thus, as with the original TCP channel implementation, the channel is kept open after an RPC completes and reused for later RPCs with the same sender and receiver. Only a large fixed number of connections are kept open at a time; less-recently-used connections will be closed if too many new connections are opened.

There are several reasons to use SSH channels instead of other secure channel abstractions, such as TLS [19]. WheelFS's target environment, PlanetLab, already uses SSH keys for authentication, allowing users to reuse these same keys to connect to WheelFS. In addition, SSH keys allow for decentralized creation of credentials, making it easier to delegate the ability to create new users and reducing the administrative overhead.

Another potential advantage to using SSH is the ability to use agent forwarding, allowing an administrator to prove her identity on a less-trusted machine while stor-

ing her private key only on a different, more trusted machine. While the current implementation does not support agent forwarding, this feature would be a useful addition in the future.

There are some disadvantages to using SSH connections. Encrypting all communication increases the CPU usage of WheelFS nodes and increases the latency of RPC handling. We analyze the overhead of using SSH connections in Chapter 5.

3.4 RSA Authentication

There are two types of identities in WheelFS: server identities and user identities. Both types of identities correspond to standard RSA public key/private key pairs [13]. However, because of the requirements of WheelFS, the public keys for the two types of identity are distributed and trusted differently.

3.4.1 Server Identities

Each configuration and storage node identifies itself using a server private key stored locally on the node.

Server public keys are distributed in the configuration service. A node joining the system needs to have received out-of-band the public keys for one or more reachable configuration nodes. When it retrieves the membership of the configuration service from one of those nodes, it also retrieves all other server public keys. These keys will be cached on local disk, so if the node is restarted and its initial configuration nodes are no longer available, it can contact other nodes whose keys it has acquired. The client periodically refreshes its cache by requesting the list of public keys again from the configuration service.

Configuration nodes store all server public keys in files on local disk. When a new server joins the system, its public key must be added on each configuration node; similarly, a public key can be revoked by removing it on each configuration node.

A common alternative to giving each server a unique identity and private key is to give all servers the same private key. While this simplifies adding new servers to

the system, it makes recovery from a compromised server difficult. While WheelFS's model assumes that servers are not compromised, in real-world deployments this is difficult to ensure. Thus, it still makes sense to give administrators some means to manually recover from a compromised server, and giving each server or at least each site a unique identity makes it possible to revoke compromised keys and recover from a compromised server without needing to migrate to a new instance of WheelFS.

3.4.2 User Identities

Each client identifies itself using a private key from a user identity. This key is stored on the local disk of the client. Each client action identifies a username it is acting as, and a client can only act as a user if it can cryptographically prove to the remote node that it has the corresponding private key.

Unlike with server keys, the public keys for users can be stored directly in world-readable files in WheelFS, since files in WheelFS can be securely read knowing only the server keys. This makes adding new users and managing user keys much easier, and allows user management permissions to be delegated using WheelFS ACLs. (Write permission to these key files and to their directory would be restricted to administrators by default, but, for example, a deployment could give users the permission to maintain their own public keys.) If a user's private key gets compromised, an administrator can simply remove the corresponding public key from the appropriate file in WheelFS, limiting the impact of the compromise.

3.4.3 Alternative

An alternative to the SSH public key model would be a certificate-based public key infrastructure, as used by TLS [19]. In such a system, there would be an ultimately trusted certificate authority that would be trusted to sign certificates associating public keys with users and also to delegate this authority. A given node would only need to start with the public key for a small number of authorities, and it would then be able to add new public keys by verifying the signature of the authority on

a certificate presented by the authenticating party. WheelFS files could be used to store 'revocation lists', lists of keys that have been compromised and should no longer be trusted.

This system would also work, and it has some advantages and disadvantages relative to the SSH-based system. It would only require a single certificate to be distributed out-of-band to be able to authenticate all configuration nodes, without requiring the configuration service to distribute identity information as the SSH-based system does. However, with TLS the certificate authority becomes a single point of compromise for the system, and thus needs to be managed carefully to maintain the system's integrity. The centralized certificate model also makes it more difficult to safely delegate user-creation ability; for example, with the standard TLS certificate model it would be impossible to give someone the ability to create new user identities without also being able to create new server identities.

The main reason this design uses SSH is because WheelFS's target environment, PlanetLab, lacks an existing TLS infrastructure and already uses SSH keys. For environments with an existing public key infrastructure, the TLS encryption and authentication method would be a logical choice.

3.5 File Permissions

WheelFS uses a standard access control list model for file permissions. Each user can be assigned admin, write, read, and advisory execute permissions for individual files or directory. A user needs admin permission to change the permissions on a file. Any user with a server key implicitly has all permissions on all files.

The motivation for using this model, as opposed to a simpler single-owner model, is for flexibility. In large-scale distributed file systems there is often demand for complex ACLs [25]. This would be less necessary for a system that targeted single-application deployments or one that permitted no sharing between applications. However, for a shared large-scale deployment of WheelFS on PlanetLab, this flexibility should prove useful.

In some situations, it may be useful to be able to assign permissions to groups of users instead of individual users; this could easily be implemented by storing group membership as a list of users in a file in WheelFS. This system would also allow group membership management to be delegated to non-administrator users using standard ACLs. The permissions given to a special group named "anyuser" apply to all users.

3.5.1 POSIX Interface

The standard POSIX permissions model allows only for a single owner and group for each file, and only assigns permissions for these two and a pseudo-group consisting of all other users. While this may be adequate for local files, as discussed above WheelFS has a more flexible access control list model. However, providing useful information about these access control lists via the standard POSIX mechanism is nontrivial. Sticking to the familiar interface provided by FUSE is one of WheelFS's strengths, but this makes it difficult to display different permission semantics cleanly. WheelFS uses a two-part solution.

When a user lists the permissions of a file directly, WheelFS displays as much of the ACL as it can under the POSIX model's limitations. It will choose the first user alphabetically who corresponds to a user on the client machine[1] and has admin on a file as the owner and display her corresponding permissions as the owner permissions. For the 'group' permissions, it will choose the first user alphabetically that is a group on the client machine, is not the owner, and has more permissions than the "anyuser" WheelFS group.[2] It will list the "anyuser" group's permissions as the other permissions. If no appropriate users or groups are found, a default user or group with ID 0 is used, and it is shown with no permissions.

In addition to this default display, a user can view or modify permissions for a specific user with the **.Ident** cue. For example, listing the permissions for `.Ident=bob/foo` will display bob's permissions, regardless of the permissions normally shown for `foo`

[1] e.g, has an `/etc/passwd` entry

[2] Linux systems generally have a group of the same name as every user, so this effectively gives us the ability to display two users' permissions.

```
$ cd /wfs/
$ ls -l file
-rw-r-----  1 dennis bob      42 May  8 15:38 file
$ ls -l .Ident=alice/file
-rw-rw----  1 root   daemon   42 May  8 15:38 file
$ ls -l .Ident=bob/file
----r-----  1 nobody bob      42 May  8 15:38 file
$ chmod u+x .Ident=bob/file
$ ls -l file
-rw-r-x---  1 dennis bob      42 May  8 15:38 file
$ chown root .Ident=bob/file
$ ls -l file
-r-xr-----  1 bob    chuck    42 May  8 15:38 file
$ ls -l .Ident=bob/file
-r-xr-x---  1 bob    bob      42 May  8 15:38 file
```

WheelFS ACLs:

alice	arw
bob	r
chuck	r
dennis	arw
anyuser	

⇒

alice	arw
bob	rx
chuck	r
dennis	arw
anyuser	

⇒

alice	arw
bob	arx
chuck	r
dennis	arw
anyuser	

Figure 3-2: Accessing WheelFS permissions via the POSIX interface
Initially, `alice` and `dennis` have read, write, and admin permissions, whereas `bob` and `chuck` have only read. There are local users called `bob` and `dennis` and local groups called `bob` and `chuck`. The anyuser group has no permissions. How the ACL is changed in this example is shown below the transcript.

and whether or not there is a local user named bob. If bob has the admin permission for the file, **.Ident**=bob/foo will show his permissions as both owner and group; otherwise, it will show them as just the group permissions. (The username and group name shown are the specified user if available on the local system, or otherwise the default user or group. The user 'nobody' is used as the owner if the specified user does not have admin.) Using the 'chmod' command to change the owner or group permissions via that path will affect only bob's permissions. Using the 'chown' command to a non-nobody local user sets the admin bit for the user specified in the cue, while chowning to 'nobody' removes the admin bit for this user. See Figure 3-2 for an example of an administrator interacting with an ACL..

This interface was designed to make permission management possible with the POSIX/FUSE API, rather than by introducing another API. However, because the permissions model WheelFS uses is different than the one used by POSIX and the set of possible usernames is different, this leads to a complex interface that may be confusing to those not familiar with it. In effect, this ends up being a standard API from a technical perspective but a custom API from a user perspective, which is undesirable from a usability perspective. Section 7.1.3 discusses some potential improvements to this interface.

3.6 Inter-client Communication

The system described so far is sufficient to authenticate all nodes to each other, and since storage and configuration nodes are trusted, this is enough to ensure proper enforcement of access control and that clients receive valid data in most cases. However, this does not account for the situation where, using the **.Hotspot** cue, a client fetches data directly from another client's cache. Since clients may be malicious, without additional measures a client with read permission on a file could mislead another client using the **.Hotspot** cue by sending incorrect data.

To avoid this problem, WheelFS generates a checksum for each block[3] of each file cached by a client. The checksum used is a SHA-256 checksum [16], which means it is computationally infeasible for a malicious client to discover an alternative block that would have the same checksum. When a client wants to fetch this file using **.Hotspot**, it first asks a trusted storage node to identify the clients that have blocks from that file cached and the checksums for those blocks. Then, the client contacts the returned clients directly, fetching blocks from them without going through the storage node. The fetching client checks each block thus fetched against the block's checksum, and discards blocks when the checksums fail to match. Since the checksums were received from a trusted source, when the checksums match the data must be correct. This validation is inspired by the BitTorrent peer-to-peer file sharing protocol [5].

[3]WheelFS uses 64 kilobyte blocks internally.

Currently, a client does not penalize another client when blocks mismatch. A more robust system would be to track how often nodes send mismatched blocks, and prefer clients that have better success rates; this reduces the ability for malicious clients to introduce additional latency for file read operations by deliberately serving incorrect blocks. However, the current system is sufficient for correct operation.

A second problem is that a malicious client without read permission on a given file could try to fetch a block of that file from another client's cache. This is prevented by having each client enforce permissions locally when sending blocks to other clients. Since clients mutually authenticate using their user keys, this is straightforward. When a client receives a request for a block from a file it has cached, it checks the requesting client's username against the cached ACL for the file, and replies with an error if the requester does not have the appropriate permissions. Obviously, this does not prevent one malicious client from sharing blocks it has access to with a second malicious client that does not have such access, but as preventing that is clearly impossible, this measure is sufficient.

3.7 Deployment Example

To better understand the practical implications of this design, let us walk through an example of an administrator setting up a new WheelFS deployment.

First, he obtains servers to serve as the storage nodes. He generally would use these servers or a subset of them additionally as configuration nodes. On each server, he uses the `ssh-keygen` program to generate a new RSA key pair. He stores the private key from the key pair in a local file that will be accessible to the WheelFS server process. He copies the public key for each server into a list of authorized host keys, which he copies onto each configuration node. He then starts the configuration service on each configuration node (generally as a system daemon). After waiting a short amount of time for the configuration service to stabilize, he can start the storage service on each storage node. This requires providing the storage node with at least one public key for a configuration node. By talking to that node, the storage

node obtains the full list of public server keys, which enables them to securely contact other storage nodes.

Next, the administrator needs to set up the initial WheelFS directory structure and permissions. He starts the WheelFS client on one of the servers. Using the server's server private key, he is granted full access to WheelFS as user 'host'.[4] This allows him to create files and directories and assign access control lists. In particular, he creates a `/wfs/users/` directory and in it creates a file named after each user he wishes to grant access to the system. These files contain the corresponding user's public keys. To create groups, he creates a `/wfs/groups/` directory with a file for each group containing the list of users in that group. He probably also creates a `/wfs/home/` directory that contains a directory for each user, ACLed such that they have full rights on it, that users can use for personal storage. Depending on the purpose of the deployment, he may also create folders for specific distributed applications (with permissions given to a user representing the application and to the application admistrators), a folder for binary programs, or whatever structure makes sense.

A user can start a WheelFS client and connect to the deployment as soon as her public key is present in `/wfs/users/`. (Her private key needs to be on the client machine in a location readable by the client.) At this point, the system is fully functional.

3.8 Attack Analysis

To better illustrate how this security system works, we present here several possible attacks against WheelFS and how WheelFS handles them. For this analysis, we assume that the cryptography used by SSH, RSA, and SHA-256 is secure.

[4]Alternately, he could do this on a separate trusted machine with another key added to the server key list.

3.8.1 Malicious Network

One common attack is for an adversary to be able to eavesdrop on or control some intermediate routers on the network. However, because communication is encrypted, being able to eavesdrop on the network does not give any information other than some idea of the activity level of the WheelFS instance and the nodes that are participating. While this information could be useful to an adversary in some situations, WheelFS does not consider it private information, so this design does not attempt to obscure it.

Clearly, given sufficient control of the network, an adversary can prevent nodes from communicating with each other, causing a denial of service. This is unavoidable. The adversary could also do a more targeted attack; for example, prevent a certain client from contacting a certain storage node that is the primary for data it needs. Since the storage node can still communicate to the configuration service to renew its lock on the data, WheelFS will keep the same node as that data's primary, and that client will never be able to access it. WheelFS does not attempt to deal with this attack either; it would be noticeable were it to happen, and it does not compromise any of the data in the system.

A more serious attack would be for the adversary to intercept communication between two nodes and alter it in some way, or alternately redirect traffic to a node to a malicious node under his control. However, because the connections are encrypted and authenticated for both parties, without the private keys involved the adversary cannot perform these man-in-the-middle attacks.

3.8.2 Malicious Client

A malicious client is allowed by WheelFS's trust model, and a client could try to perform illegal operations. However, because access control lists are enforced both by the client and by the storage nodes, such attempts would fail.

A client could also attempt to exhaust server resources by performing a large number of valid operations. There is no current mechanism for preventing this other

than having an administrator revoke the offending user's public key. For environments with large numbers of users, it might make sense to add some rate-limiting or fairness-enforcing mechanism.

3.8.3 Lying Cooperative Client

A client could try to send incorrect data when it receives a **.Hotspot** request from another client, but the other client's checksum verification will reduce this attack to a form of denial of service.

3.8.4 Rogue Server

An adversary could try to connect a storage or configuration node under her control to the system. Without access to a valid server private key, the server would not be able to convince the configuration service to let it join, and nodes would refuse its RPCs. Thus, this attack would be ineffective.

3.8.5 Compromised Server

A more dangerous case is if the adversary manages to compromise a server machine and gain access to its private key. While this violates WheelFS's trust model, in the real world no system has perfect security, and there are many ways outside of WheelFS that a node could be compromised.

If either type of server node is compromised, the adversary can read arbitrary files in WheelFS and make arbitrary modifications by using the server's private key to run an authenticated superuser client. WheelFS, as it is currently set up, provides little to recover from these unauthorized modifications. WheelFS's **.Versioned** cue will keep a persistent version history for a file, enabling the manual recovery of pre-compromise versions for files whose primary was not on the compromised node. The versioning system makes no guarantee on the persistence of previous versions when the primary of a file is lost or compromised.

When the compromise is discovered, it is desirable to remove the compromised node from the system. The compromised node's public key can be removed from the key list on each uncompromised configuration node, rendering the compromised node unable to communicate with the configuration nodes, and eventually with all other nodes once the other nodes receive the updated public key list from the configuration service. When a node notices that the public key for an open channel has been revoked, it will close that channel, and thus end communication with the compromised node.

If the compromised node is a configuration node, the situation is more complicated. Nodes who join the system using that node could be directed to a separate WheelFS deployment under the adversary's control, which not only gives the adversary complete control of the new node's view of the file system state but also makes recovery difficult; each misdirected client would need to be restarted and directed to an uncompromised configuration node.

If the compromised node was the configuration service's current master, the situation is even more severe. The master can propagate public keys for rogue storage nodes under the adversary's control and assign them locks on all files, effectively directing both already-connected and newly-joined clients to a separate WheelFS instance. If the master is taken offline by an administrator, however, the system will eventually return to its proper membership as the rogue storage nodes' locks expire and further communication from them is rejected from the new, uncompromised master.

The security of storage nodes and particularly configuration nodes is essential to a secure WheelFS deployment.

Chapter 4

Implementation

The current WheelFS prototype implements the key parts of the security design, though some features have yet to be implemented.

4.1 Overview

Implementing the security subsystem involved contributing 2,800 of WheelFS's 26,000 lines of C++ code. These additions make heavy use of the OpenSSL library [24] for SHA-256 checksums and libssh [3] for secure authentication and communication. In addition, 450 lines of modifications to libssh's C code were necessary, due to its incomplete implementation of server support for RSA authentication and some threading issues.

WheelFS's use of SSH connections is effectively a subset of the SSH protocol [20, 21,22,23]. It only uses a single channel per connection and uses the "issue banner" that is typically used to send a message to the user before login to instead inform the RPC caller of the identity the RPC target is identifying itself as; this allows both parties in inter-client authentication to validate each other's identity.

The WheelFS channel classes use an abstract interface; in this way, the original TCP channels are preserved for testing or use on virtual private networks, and new methods for securing channels or for authentication can easily be added in the future.

4.2 Limitations

There are several aspects of the security design that are not yet implemented. These aspects, however, are all administrative convenience features, not core protocol components. Thus, while they would be important for production use of WheelFS, their absence is unlikely to affect the validity of this thesis's results.

Key Distribution Key distribution for both server and user public keys is currently out-of-band: the keys are stored in files on the local disk of each node, both servers and clients. This successfully allows all nodes to verify that they are talking to a valid server before sending confidential data or trusting provided information. However, this makes changing the set of valid server or user private keys difficult, since each node must be modified, making adding new users or servers to the system and responding to compromised keys difficult. Propagating server keys via the configuration service and storing user keys directly in WheelFS will reduce the administrative burden of key management.

Groups Group lists and group permissions (other than for the 'anyuser' group) are not yet implemented.

Multi-User Clients Currently, each client only manages a single RSA private key, and thus can only act as a single user. In the future, it would be more efficient to allow a client with access to several users' keys on a single machine to provide access to WheelFS for any of them. This would require a modification to the WheelFS SSH protocol to allow a client to authenticate as more than one user for a given channel.

Agent Forwarding Support The current implementation does not support SSH agent forwarding. Allowing for SSH agent forwarding would make it more convenient for administrators to access WheelFS from a variety of machines without needing to copy their private key, and would also promote better security by reducing the number of machines with nontransient copies of users' private keys. This would

require implementing the SSH agent forwarding protocol in libssh.

Chapter 5

Results

To evaluate the performance cost of this security system, we ran several experiments on the PlanetLab wide-area test bed comparing the unsecured WheelFS to the secured WheelFS on several basic workloads. In particular, we sought to determine the performance impact this security system has on different types of WheelFS file transfer, both plain node-to-node transfers and cooperative **.Hotspot** fetches.

Due to the fact that PlanetLab nodes have many other applications running on them at all times, with widely varying CPU and bandwidth usages, obtaining consistent results on PlanetLab is challenging. For this analysis, we look mainly at the median results from multiple trials to minimize the effects of unusually busy or unusually free nodes, though we also discuss the level of variability in the collected data.

5.1 File Transfer

To measure the raw overhead of SSH channels on file transfer speed, we conducted a simple experiment on PlanetLab, measuring the time to read at MIT a file randomly-generated at UCSB for various file sizes. For comparison, we also measured the time to transfer equivalent files via unsecured TCP connections, and via SCP, a file transfer program based on SSH. The results of this experiment can be seen in Figure 5-1.

Due to the inconsistent CPU and bandwidth load on the PlanetLab environment,

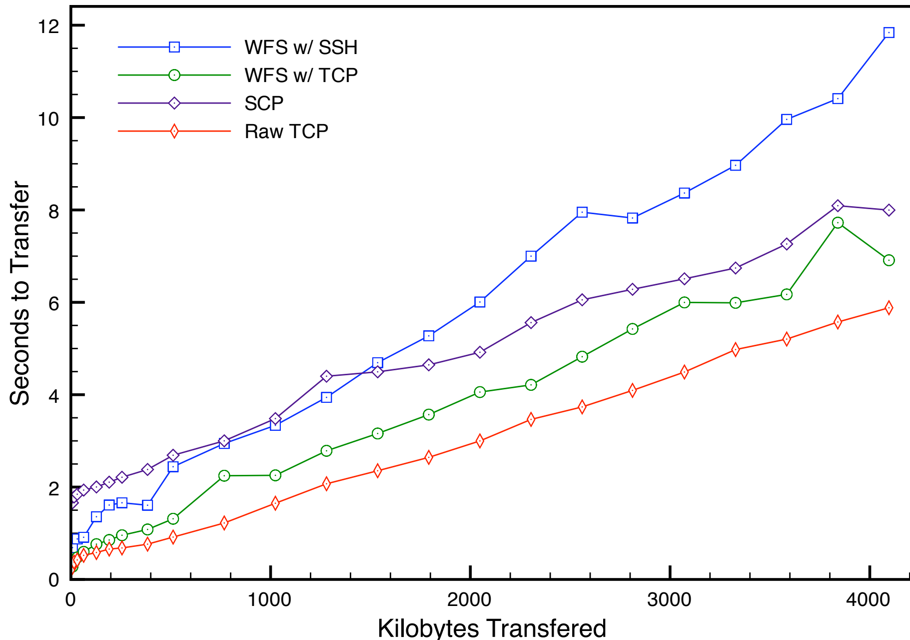

Figure 5-1: File Transfer Times
Time to transfer a random file by file size for plain TCP, SCP transfers using SSH, WheelFS with TCP channels, and WheelFS with SSH channels.

the running times of individual experiments varied widely, with many experiments close to the graphed medians but a few much slower outliers. (In a few cases, there were enough spikes to pull the median up to a small peak, such as for WheelFS with TCP transferring 3840 kB.) The standard deviations for each data point varied widely with no consistent pattern; for example, the standard deviation for SSH-based WheelFS transferring 192 kB was 25.5 seconds, whereas for 384 kB it was 0.7 seconds. This variation seems to be caused mainly by occasional huge outliers which are likely to be due to transient load spikes at one or more nodes or transient connectivity problems between nodes. These outliers are particularly severe when using SSH connections; SSH connections are more sensitive to CPU load due to SSH's CPU-intensive encryption operations. Testing with a WheelFS deployment with multiple WheelFS nodes on a single PlanetLab machine suggests that CPU load spikes are a major cause of these outliers. While WheelFS would likely exhibit more even performance if given more consistently-used hardware, these occasional large delays

would be a concern for any widespread deployment on PlanetLab.

Despite the irregularity of the data, the median point-to-point transfer time is basically linear in all four cases, as can be seen in the graph. WheelFS with SSH channels is consistently slower than WheelFS over TCP by a factor of approximately 1.5, whereas the cost of using SCP over plain TCP is a roughly constant 2-second additional latency.

SSH connections have several potential causes for slowdown. First, there is some additional latency to do authentication when making new connections. Secondly, each packet has a small additional header and thus requires some additional bandwidth [22]. Finally, the cost for encrypting all messages incurs additional CPU usage which can be significant for machines, like those in PlanetLab, which have large amounts of existing CPU load. The startup cost seems largely insignificant for WheelFS, especially since WheelFS's knowledge about the remote node reduces the amount of negotiation necessary to establish a new connection. Tests with a WheelFS deployment on a single PlanetLab node show that WheelFS with SSH channels takes 3 times as long as WheelFS with TCP channels when bandwidth is not a factor, suggesting that the main cost of SSH channels is the continual cost of encryption.

The fact that using SSH with WheelFS has a larger performance penalty than using SCP over raw TCP seems odd at first. However, it turns out to be a matter of bandwidth over latency. SCP requires little response from the remote server; while it takes longer to send any given chunk of data due to encryption, it can pipeline its operation by encrypting the next block while the previous block is being sent. Thus, when bandwidth is the limiting factor using SSH does not cause any slowdown except for the startup cost of authentication. WheelFS, on the other hand, operates via RPCs, so each time data is transfered, it has to encrypt the data, send the request, decrypt the request, encrypt a response, send it, and decrypt the response before execution can continue. Unlike with SCP, SSH encryption operations are on the critical path in WheelFS, so the cost of SSH is a multiplicative factor rather than a constant startup cost. This test uses the **.Wholefile** cue, which causes multiple blocks from a file to be requested in parallel. This reduces the overhead of RPC-based

transfers, but it is still present.

SCP taking longer than the secured WheelFS for small files is somewhat surprising. This is partially because of SCP's need for more elaborate negotiation due to not knowing details about the remote server and its need to consider multiple types of authentication; these contribute to a greater startup cost unrelated to the size of the transmitted file. When the SSH server is given nonstandard configuration to reduce the amount of negotiation needed, this fixed delay is reduced by approximately half a second.

The experiments show that while SSH connections do impose overhead, performance is similar to WheelFS's performance using TCP connections. Increasing the block size used by the client or increasing the level of parallelism in the RPCs used for client operations could be investigated as ways to reduce the overhead of SSH connections overhead and increase overall performance. Section 7.1.2 discusses some major changes to the WheelFS protocol that could lead to improved performance.

5.2 Cooperative File Distribution with .Hotspot

To measure the overhead of SSH channels in another common scenario, cooperative fetching, and also to measure the cost of checksumming, we conducted a second experiment on PlanetLab, measuring the time for 10 nodes divided between 3 remote sites to read a file randomly-generated at MIT with the **.Hotspot** and **.Wholefile** cues. These cues cause blocks from the file to be fetched in a random order and for clients to share blocks to reduce storage node load and reduce transfer time. We noted the average time for each of the ten remote nodes to read the file for a WheelFS deployment using SSH with and without SHA-256 checksumming and one using TCP, again with and without checksumming. The median results for repeated executions of this experiment can be seen in Figure 5-2.

There are noticeable peaks in the data, particularly for the SSH connections when transferring larger amounts of data. These exist for similar reasons to the high variability in the previous experiment. Due to time constraints and the increased amount

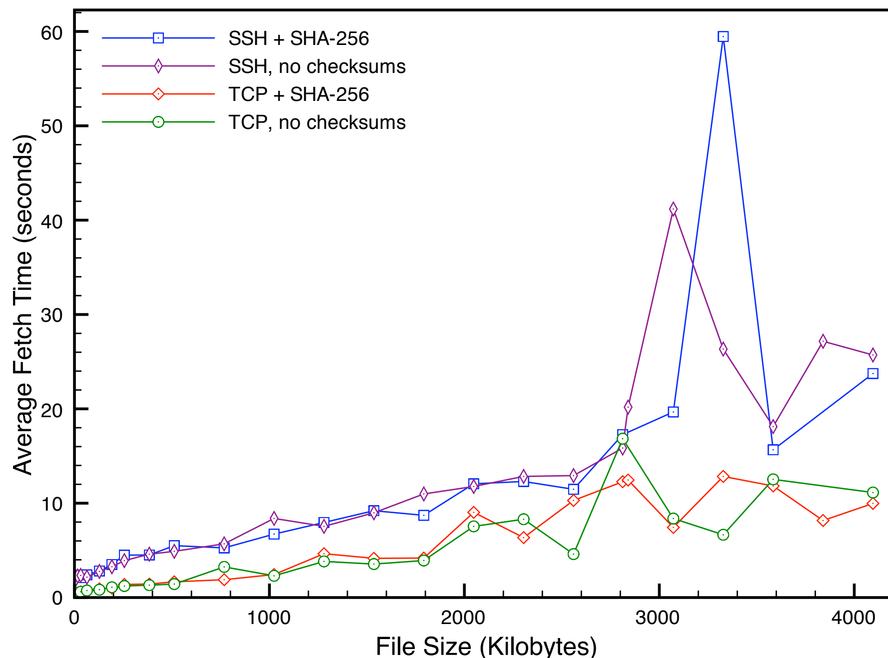

Figure 5-2: Cooperative Transfer Times
Average time for 10 nodes to cooperatively fetch a file by file size for WheelFS with TCP channels and with SSH channels, each with and without SHA-256 checksums.

of time it takes to run an instance of this experiment compared to the previous experiment, we were not able to obtain as much data for this experiment, increasing the influence of outliers. In addition, with cooperation, if one node experiences extra CPU or bandwidth load, this can affect other clients as they attempt to fetch blocks from the loaded node's client. This happens more frequently with larger file sizes because the longer experiments increase the chances of hitting a load spike. Notably, checksumming does not significantly affect the size of these peaks, suggesting that the CPU overhead of checksumming some blocks is much smaller than that of encrypting all communication.

While the results for this experiment are a bit more erratic, fetching the file on the secure WheelFS deployment still takes approximately 1.5–2 times as long compared to the insecure deployment. The effects of checksumming are basically negligible; the real performance impact comes from the SSH connections and not from the checksumming or checksum verification. Despite the irregularity of this

data, this does suggest that the checksumming used in inter-client sharing does not add much additional overhead and that the overhead for cooperative fetching is in line with the results from the direct transfer experiment.

Chapter 6

Related Work

There is a rich history of past work on distributed file systems, and these systems include a wide variety of security measures motivated by various deployment scenarios and trust models. This thesis draws inspiration from much of this work.

Security in distributed file systems, as in distributed systems in general, is an interesting area of research because the necessary and desired properties vary greatly depending on the target environment and use cases. Some distributed file systems, for example NFSv3 [15] and Pangaea [14], are intended to operate in trusted environments and thus are able to trust clients to not misidentify themselves or perform invalid operations.

Most systems, however, need secure authentication and authorization mechanisms to enable their use with untrusted clients. While the simplest form of authentication is the traditional username/password combination, most systems use public-key cryptography (such as RSA [13]) to reveal knowledge of a secret token, thus proving identity. Since public-key cryptography allows a program to prove knowledge of a token without revealing the token itself, these systems allow for mutual authentication between server and client without allowing either to impersonate the other, reducing the impact of a server compromise. Variations on this mechanism are used in many file systems, such as modern versions of AFS [25]. Unfortunately, AFS uses a custom credential management system that can be confusing and difficult to manage for users, and it requires a centralized authentication infrastructure to grant credentials

to users. NFSv4 is an updated version of NFSv3 that provides authentication options along these same lines [17].

Many recent distributed file systems make use of public-key cryptography to provide more security than just authentication. An example is FARSITE [1], which gives the file system client direct access to private keys, allowing them to be used for encryption and decryption tasks to provide guarantees that private data remains private even in the face of prying server administrators. It also allows file data to be cryptographically signed to allow verification of its source. FARSITE avoids the need for a user to manually copy her private key to each computer she wants to access the file system from by storing a world-readable copy of each user's private key, encrypted with a password of the user's choosing, in FARSITE itself. This simplifies key distribution without needing a centralized authentication service, but it also enables offline attacks against any user's private key, a potential security hazard. While FARSITE makes sense in a local area network with some weak trust assumptions about everyone in the network, this makes deployment on a public network unwise. Without this private key maintenance method, manually maintaining a custom private key would be less convenient than WheelFS's SSH-based key usage.

The Self-certifying File System (SFS) [12] has an authentication system which uses private keys stored on the users' local workstations. WheelFS takes a similar approach to server authentication. SFS attaches a public key to each reference to a given site, whether in the SFS file path or in references to remote users on ACLs [10]. Unlike WheelFS, this allows separate SFS instances to share user and group information without mutual trust; however, it prevents an SFS instance from being securely distributed onto multiple servers, and requires all server public keys to be distributed out-of-band. Extensions of this protocol that support agent forwarding in a similar way to SSH agent forwarding also exist [9].

Shark [2] provides a similar security model, authenticating its single central file server with a public key, encrypting point-to-point communication, and using checksums to allow verifiable sharing between mutually untrusting clients. It differs in its use of server-supplied cryptographic tokens for one client to prove read permission to

another client, since its pooling of identical blocks interferes with a simple ACL-based sharing policy. Like SFS, it includes the server public key in the file system path instead of storing it in a file on the local machine; this is not practical for a system like WheelFS with many servers that can have different public keys.

Chapter 7

Conclusion

WheelFS is a simple and flexible distributed storage layer that allows practical wide-area applications to be built without the need for a custom storage layer. This thesis shows how strong security properties can be added to WheelFS while maintaining the scalability and simplicity of the base system. WheelFS can conveniently use SSH connections for secure communication, allowing re-use of PlanetLab's existing RSA authentication keys. This allows for privilege separation between mutually-distrustful applications running on the same WheelFS deployment and protects against malicious clients and network manipulation. Based on the experience of running this prototype on the PlanetLab testbed, this security model provides reasonable performance while providing meaningful security guarantees and as much as possible maintaining WheelFS's easy-to-use interface.

7.1 Future Work

While this current implementation satisfies the core requirements for security in WheelFS, there are many ways it could be expanded and improved in the future.

7.1.1 Protocol Evolvability

Proper security is a moving target: new attacks against popular encryption and hashing algorithms are continually being developed, and new and improved algorithms are created in response. Currently, the algorithms used for encryption, authentication, and checksumming in WheelFS are fixed at compile time. For production use, however, WheelFS should support multiple algorithms for encryption, authentication, and checksumming and have nodes negotiate with each other over which algorithms to use. This would make it possible to upgrade nodes to new, more secure algorithms without breaking compatibility with older nodes, which would be necessary to upgrade a WheelFS instance without system downtime. Along with this, a protocol versioning system should be implemented to allow nodes to negotiate which version of the WheelFS protocol to use, in case new technical or security improvements necessitate changing the core protocol or additional types of channel are added.

7.1.2 Performance Optimizations

The superior performance of the SCP SSH-based file copy program relative to WheelFS suggests that WheelFS could do better in optimizing its performance. One potential option would be to, instead of requesting individual blocks from storage nodes or clients via RPC, to instead use a message-based streaming protocol. A client would send a single message listing blocks it needs, and the remote client or storage node would stream those blocks back as available, effectively pipelining multiple requests into a single exchange. The local client could use incoming data as soon as it had the right blocks to fulfill the application request, while continuing to accept and cache blocks that arrive later in the stream. Details would need to be worked out, but this pipelining approach has potential to hide the latency of encryption operations and improve performance.

A second approach would be to find a middle ground between the TCP and SSH channels, encrypting most data but leaving data unencrypted when it is safe to do so. For example, a file that has the ACL entry "anyuser: read" could be safe to

send unencrypted, as long as a checksum or signature were present to prevent an adversary from modifying the data in transit.[1] Similarly, modifications to such a file could be transmitted unencrypted as long as they were signed. Limiting the amount of encrypted traffic could provide a significant performance benefit when CPU is a limiting factor and large publicly-readable files are used frequently. However, this would require making significant modifications to WheelFS's SSH-based protocol.

7.1.3 Enhanced ACL Support

There is currently no way to examine the full ACL for a file except by iterating over all WheelFS users. One way to give more convenient access to the whole ACL would be via a **.Acl** cue that would present the access control list as a file. For example, reading **`.Acl`**`/foo` would return something like '`bob: rwa; anyuser: r`'. This would be convenient for examining unfamiliar files or for large-scale ACL changes.

It would also be useful to be able to read and modify the access control list as an extended file attribute on systems which support such attributes. The current interface is awkward, and this would provide more convenient ACL access in many cases. However, not all operating systems have the command-line utilities to access extended attributes easily available, and user familiarity with these tools is low. Providing compatibility with the current interface would likely be necessary.

An alternative to using a custom extended attribute would be to use POSIX ACLs [8]. While not formally part of the POSIX standard and not widely used, this form of ACL is the closest thing to a standard ACL interface for POSIX systems. However, this system has its own complexities, and it is unclear if providing access to WheelFS ACLs in this way would actually be easier to use than the current system. In addition, like extended attributes not all operating systems have command-line tools to manipulate POSIX ACLs easily available.

Regardless of the technical ACL interface, it would improve usability to make WheelFS-specific scripts that users could use to manage ACLs more conveniently.

[1] The assumption that 'anyuser: read' files aren't private would not be true in all deployments, so this would have to be a configuration option.

These could easily be implemented as shell scripts either on top of the current interface or one of the proposed interfaces.

7.1.4 Enhanced PlanetLab Integration

To make WheelFS more transparently usable by the PlanetLab community, it would be possible to write a synchronization script that would periodically write the list of RSA public keys associated with each PlanetLab slice to the user key lists in WheelFS. This would take advantage of PlanetLab's existing public key maintenance infrastructure, simplifying administration of a PlanetLab WheelFS instance. If the script also created personal 'home' directories for each new slice, this would provide convenient and transparent access to WheelFS to all PlanetLab users, and would make widespread use of WheelFS more practical.

7.1.5 Configuration Key Distribution

It would simplify WheelFS administration to be able to add and remove server identities from the system without needing to make an out-of-band change to each configuration node. However, it is difficult to design a protocol that makes these changes easier for an administrator without significantly increasing the impact of a compromised key.

7.1.6 Signed Checksums

Currently, all information about which blocks cached clients have needs to come from the storage node to obtain secure checksums. However, in the original WheelFS design, clients that fetch data from each other update each other about what blocks they have at the same time, leading to less load on the storage node. To be able to do this with secure checksums, a client needs a way to verify the checksums given to it by another client without communicating with the storage node. This could be done by having the storage node sign a checksum record with its private key, which the

client could then verify with the server's public key. The current prototype ignores client-to-client block information when using secure connections.

Bibliography

[1] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer and Roger P. Wattenhofer. “FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment”. *5th Symposium on Operating Systems Design and Implementation*. (December, 2002).

[2] Siddhartha Annapureddy, Michael J. Freedman and David Mazières. “Shark: Scaling File Servers via Cooperative Caching”. *Proceedings of the 2nd NSDI*. (May, 2005).

[3] Jean-Philippe Garcia Ballester, Norbert Kiesel and Laurent Bigonville. “The SSH Library”. `http://0xbadc0de.be/wiki/libssh:libssh`.

[4] Andy Bavier, Scott Karlin, Steve Muir, Larry Peterson, Tammo Spalink, Mike Wawrzoniak, Mic Bowman, Brent Chun, Timothy Roscoe and David Culler. “Operating System Support for Planetary-Scale Network Services”. *Proceedings of the 1st NSDI*. (March, 2004).

[5] Bram Cohen. “Incentives build robustness in BitTorrent”. *Workshop on Economics of Peer-to-Peer Systems*. (May, 2003).

[6] Frank Dabek, Russ Cox, M. Frans Kaashoek and Robert Morris. “Vivaldi: A Decentralized Network Coordinate System”. *Proceedings of the 2004 SIGCOMM*. (September, 2004).

[7] “FUSE: Filesystem in Userspace”. `http://fuse.sourceforge.net/`.

[8] Andreas Grünbacher. “POSIX Access Control Lists on Linux”. *Proceedings of the FREENIX Track: 2003 USENIX Annual Technical Conference*. (June, 2003).

[9] Michael Kaminsky, Eric Peterson, Kevin Fu, David Mazières and M. Frans Kaashoek. “REX: Secure, modular remote execution through file descriptor passing”. *MIT Laboratory for Computer Science Technical Report*. Number MIT-LCS-TR-884 (January, 2003).

[10] Michael Kaminsky, George Savvides, David Mazières and M. Frans Kaashoek. “Decentralized User Authentication in a Global File System”. *19th ACM Symposium on Operating Systems Principles*. (October, 2003).

[11] Leslie Lamport. "The Part-time Parliament". *ACM Transactions on Computer Systems*. Volume 16, Issue 2 (May, 1998). pp. 133–169.

[12] David Mazières, Michael Kaminsky, M. Frans Kaashoek and Emmett Witchel. "Separating key management from file system security". *Proceedings of the 17th ACM Symposium on Operating Systems Principles*. (December, 1999). pp. 124–139.

[13] Ron Rivest, Adi Shamir and Leonard Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems". *Communications of the ACM*. Volume 21, Issue 2 (February, 1978). pp. 120–126.

[14] Yasushi Saito, Christos Karamanolis, Magnus Karlsson and Mallik Mahalingam. "Taming aggressive replication in the Pangaea wide-area file system". *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*. (December, 2002).

[15] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh and Bob Lyon. "Design and Implementation of the Sun Network Filesystem". *Proceedings of the Summer 1985 USENIX*. (June, 1985).

[16] "Secure Hash Standard (SHS)". *Federal Information Processing Standards Publication*. Number 180-3 (October, 2008).

[17] Spencer Shepler, Brent Callaghan, David Robinson, Robert Thurlow, Carl Beame, Mike Eisler and Dave Noveck. "Network File System (NFS) version 4 Protocol". *RFC 3530*. (April, 2003).

[18] Jeremy Stribling, Yair Sovran, Irene Zhang, Xavid Pretzer, Jinyang Li, M. Frans Kaashoek and Robert Morris. "Flexible, Wide-Area Storage for Distributed Systems with WheelFS". *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*. (April, 2009).

[19] David Wagner and Bruce Schneier. "Analysis of the SSL 3.0 protocol". *The Second USENIX Workshop on Electronic Commerce Proceedings*. (November, 1996).

[20] Tatu Ylönen and Chris Lonvick, Ed. "The Secure Shell (SSH) Authentication Protocol". *RFC 4252*. (January, 2006).

[21] Tatu Ylönen and Chris Lonvick, Ed. "The Secure Shell (SSH) Protocol Architecture". *RFC 4251*. (January, 2006).

[22] Tatu Ylönen and Chris Lonvick, Ed. "The Secure Shell (SSH) Transport Layer Protocol". *RFC 4253*. (January, 2006).

[23] Tatu Ylönen. "SSH: Secure Login Connections over the Internet". *Proceedings of the Sixth USENIX UNIX Security Symposium*. (July, 1996).

[24] Eric A. Young, Tim J. Hudson and Ralf S. Engelschall. "OpenSSL". `http://www.openssl.org/`.

[25] Edward R. Zayas. "AFS-3 Programmer's Reference: Architectural Overview". *Transarc Corporation*. (September, 1991).