Towards A Deployable Framework for Delegation of Authority in Network Applications

by

Will Stockwell

S.B., Massachusetts Institute of Technology (2005)

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

September 2006

Author .....

Department of Electrical Engineering and Computer Science

August 23, 2006

Certified by .....

Stuart Schechter

Research Scientist, MIT Lincoln Laboratory

Thesis Supervisor

Certified by .....

Hari Balakrishnan

Professor

Thesis Supervisor

Accepted by .....

Arthur C. Smith

Chairman, Department Committee on Graduate Students

This work is sponsored by the United States Air Force under Air Force Contract F8721-05-C-0002. Opinions, interpretations, conclusions and recommendations are those of the author and are not necessarily endorsed by the United States Government.

Towards A Deployable Framework for Delegation of Authority in Network Applications

by

Will Stockwell

Submitted to the Department of Electrical Engineering and Computer Science
on August 23, 2006, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The Delegation Framework is a collection of programs, network protocols and library interfaces that provide fine-grained delegation of authority to network systems. The design and implementation of the Delegation Framework focus on addressing some of the stumbling blocks that have prevented delegation systems from becoming widely deployed in real world network applications. The Delegation Framework makes it possible to integrate delegation into an existing client-server network application without modification to the network application protocol. A dynamic library interposition implementation also make it possible to integrate delegation into large classes of legacy client and service programs with minimal or no manual source code modification. An integration case study describes the process of grafting the Delegation Framework onto a simple Apache- and MySQL-based network application and analyzes the added overhead incurred by the application.

Thesis Supervisor: Stuart Schechter
Title: Research Scientist, MIT Lincoln Laboratory

Thesis Supervisor: Hari Balakrishnan
Title: Professor

This work is sponsored by the United States Air Force under Air Force Contract F8721-05-C-0002. Opinions, interpretations, conclusions and recommendations are those of the author and are not necessarily endorsed by the United States Government.

Acknowledgments

I would like to thank Stuart Schechter for advising me on this work. The seed of the project was his. I must also graciously thank Hari Balakrishnan for taking the official "thesis supervisor" role, providing valuable feedback, and donating his time to evaluating this work. Rob Figueiredo and Cynthia McClain were both helpful in dealing with various technical subtleties.

I cannot forget to give enormous credit where it is due to many folks over the years. Thanks to all my great teachers along the way: Hal Abelson, Hari Balakrishnan (again), Robert Berwick, Joseph Birzes, David Ciarlo, Munther Dahleh, Tony Eng, Robin Farr, D. J. Fromal, Joseph Carapucci, Eric Grimson, Frans Kaashoek, Steven Leeb, Walter Lewin, Arthur Mattuck, Robert T. Morris, Ron Rivest, Donald Sadoway, Jay Scheib, Edward Sherretta, Stephen Ward, Alan Willsky, Patrick Winston, Daniel Witzner, Victor Zue, and all my good TAs whose names I have forgotten.

Thanks to Gnarls Barkley, Hard-Fi, Interpol, Jamiroquai, John Legend, Massive Attack, Paul Oakenfold, Red Hot Chili Peppers, The Streets, Thievery Corporation, and all the other great music that I love (but will never fit on this page) for keeping me sane.

And last but not least, I would like to thank Chris, Maggie, and Hope Stockwell for being encouraging when times were tough. You guys are great, and I'll see you at "home" in Montana soon!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Access control mechanisms common to many of today's network services blur the distinction between the operations a user is authorized to perform and the operations that programs the user employs are authorized to perform. Operating systems control resource access based on a *user* identity assigned to a process. Even if a user's program infrequently or never requires any more than a particular subset of those user authorities, few access control systems provide a practical mechanism to limit the authority delegated to the program.

A distinction between a user's authorities and those of the programs the user employs often does not exist; the user must give a program either all of his authorities or none of them. This circumstance would be satisfactory for users' security needs if the various other authority-requiring software that users employ were well-behaved and perfectly resilient to attack by malicious parties. However, given the ability to maliciously manipulate a client or service program, an adversary can translate the lack of distinction between user and program authorities into an opportunity to capture a user's credentials. The opportunity to capture credentials is especially apparent when resources are distributed across a network. Many of today's networked systems require a user to transfer his identity credentials to any remote system that is to make a request on his behalf.

1.1 Delegation of Authority

Many security mechanisms today provide implicit delegation of coarse-grained authority, such as the authority to assume the user's identity. Making that delegation explicit and more finely-grained can alleviate the problems resulting from blurring the identities of users with the identities of their programs. Explicit delegation provides precise control over what authority is granted to a given program for a particular period of time, and consequently it becomes more challenging for malicious to abuse authority derived from a user.

1.1.1 Coarse-Grained Delegation is Ubiquitous

Though not typically called delegation, computer systems often employ security mechanisms that are implicitly delegation. Often, a client program (e.g., a web browser) receives a password typed by a user, which the client program presents as a credential to prove that it holds authority to access a service on the user's behalf. The user thus delegates the authority to assume his identity to the client.

Many web applications provide are front-end interfaces through which users access data stored on an underlying service. These web applications take the role of a *deputy*, a network service that makes requests of other network services on a user's behalf. For example, a web interface to an IMAP mail service forwards a password from a user to the service. In this case, the deputy web application is delegated the authority to assume the identity of the client.

Many database-driven web applications rely on code running on the web server to verify user credentials and make authorization decisions to control access to database tables. This requires complete trust in the web server's access control. All users implicitly delegate to the web application the full set of authority to access their underlying database records. This authority can be easily abused by any adversary able to compromise the web server.

The salient problem with password credentials and strong trust relationships is that they only allow coarse-grained (i.e., all or nothing) delegation. It is difficult to

guarantee that a client or deputy does not abuse its authorities without revoking all of the authorities it has been given, rendering it unusable. A delegation mechanism for defining and granting more fine-grained sets of authorities to different network services would benefit a great many of the systems that rely on passwords (and other client-held user credentials), and strong trust relationships across the interfaces between deputies and underlying services.

1.1.2 Precise and Explicit Delegation

A variety of research has posited fine-grained delegation of authority as a solution to limit the scope of potential authority abuses that may result from reliance on passwords and strong trust relationships in networked systems [13, 17, 18, 26]. Fine-grained delegation of authority enables an authority holder to grant specific authorities to another principal for a specified period of time. If allowed by the delegator, the delegate may re-delegate this authority further. Delegation and re-delegation allow authority to trickle down through a system from the users who hold authorities, to their client software, to the deputy services those clients rely on to access other services. A client or deputy may then delegate the authority further or use the credentials that prove that it has been delegated the authority to access the underlying resource. Fine-grained delegation enables more explicit control of the amount and duration of authority users make available to the software they use locally and over a network, thereby limiting the amount of trust placed in those software programs.

In contrast to the common password-based authorization scenario, credentials proving delegated authority need not indefinitely grant a client the full set of authorities held by the user. Instead, the user is able to specify which of his authorities the client program may use and for what duration they make be used. Delegation of authority enables a user to grant different sets of authority to client programs of varying levels of trustworthiness and different functional requirements. A user need not be concerned with a compromised client retaining authority after the specified expiration time.

Additionally, the strength of trust relationships between deputies and the underly-

ing services to which they provide access can be reduced or, in some cases, eliminated entirely. In a delegation-enabled version of the system consisting of a web application (a deputy service) and an underlying database service, a user can delegate the deputy the precise set of authorities required to perform the database operations for the task requested. The strong trust relationship between the deputy and database service can be eliminted. Instead of lavishing the web application with a large set of authorities and trusting the web application to mediate access, authorization checks can take place at the database that sufficiently verify the user authorized the web application to perform such requests. The web application (acting as the user's deputy) may continue to perform authorization checks as before.

Delegation of authority and capabilities systems have been studied for many years now. Research efforts have addressed many of the problems encountered on the path to making delegation of authority possible, and demonstrated some interesting applications. Useful structures of cryptographic certificates for delegation are now well-defined. Authority description languages make it possible to flexibly describe arbitrarily fine- or coarse-grained sets of authority. Usable systems designed with delegation functionality built-in have been constructed and used in laboratory environments. If these laboratory success could be carried over into real world systems, it would improve the security properties of a large class of the network applications in existence today. Nevertheless, despite these developments in the study delegation of authority, most network applications have not been successfully adapted to use delegation for authorization.

1.2 Deployability Problems for Delegation Systems

While delegation has been research extensively, effective delegation of authority mechanisms have not become widely deployed in real world systems. This is a classic problem of a good technology failing to be deployed because it was not designed to be easily deployable. Redesigning systems and reimplementing client and service software to include support for delegation has been impractical, and good methods for

integrating delegation into legacy applications and services have not been deployed[1].

1.2.1 Lack of Support in Application Protocols

Network applications enlisting the support of a delegation systems require protocol support for requesting and passing delegation credentials between hosts. *Application protocols*, the protocols spoken between clients and services at the application layer across the Internet (such as HTTP, SMTP, IMAP) are often well-specified, standardized protocols that have received the intense scrutiny of working group committees at standards organizations. Few, if any, standard application are pre-specified to carry delegated credentials or pair protocol requests with particular credentials. Amending these protocol specifications to support delegation would be a labor-intensive process that could take standards organizations years, a common stumbling block for new security technologies. A better approach to solving the protocol problem must be developed.

1.2.2 Integration with Legacy Code

Rewriting or modifying legacy software can be prohibitively expensive and error-prone. Rewriting could also require major changes to access control functions. Source code may be lengthy, poorly documented and difficult for unacquainted developers to decipher. Even with a strong understanding of the code, the challenges involved in the integration process are numerous: major restructuring of a program may be involved; support for underlying cryptographic operations may need to be added; and the proper authorization checks can easily be misplaced or omitted.

1.2.3 Usability

If users cannot understand how to use a delegation mechanism, or find using it tedious, integrating delegation may not actually improve the security of a network application.

[1]The term *legacy* as it is employed in this thesis refers to software that was not originally designed and implemented to have a delegation of authority mechanism and is not intended to make any other qualitative judgments about the software's age or sophistication.

It may instead make the application more difficult to use. The reasons a user must delegate authority may not seem obvious to him, and authorization policies specifying the authorities granted to a delegate are difficult for the novice to craft. Usable interfaces must be developed to assist users of the system in delegating authority easily while ensuring that the user's actually receive the security benefits a delegation system is intended to provide them. The delegation system must also provide whatever infrastructure support these usability mechanisms require.

1.2.4 Performance

Regardless of how desirable a given authorization mechanism's security benefits are, the mechanism must be implemented efficiently to be successfully adopted. High volume Internet services demand security mechanisms that do not severely degrade the quality of service offered to clients. Considerable effort must be spent designing and implementing a delegation system with acceptable performance characteristics.

1.3 Making Delegation Deployable

These practical issues have rendered the research successes of delegation of authority difficult to deploy in real world systems. The contribution of this thesis is the Delegation Framework, which makes strides in addressing many of the problems of integrating delegation of authority into legacy network applications. The framework consists of a collection of network protocols, an agent, and a set of library routines that work together to address various of the deployability challenges facing delegation of authority. The Delegation Framework makes automatic or near-automatic integration of delegation into a large class of network applications software a reality, thereby helping system administrators to more easily deploy delegation-enabled systems to their users.

Chapter 2 summarizes relevant previous work. Chapter 3 gives an overview of the Delegation Framework design. Chapter 4 describes the structure of delegation credentials. Chapter 5 discusses in detail the design of the protocols used by the Del-

egation Framework. Chapter 6 discusses the implementation of the Delegation Agent program, the cornerstone of the Delegation Framework. Chapter 7 describes the Delegation Framework Library, the interface via which legacy software participates in Delegation Framework-enabled application. Chapter 8 discusses the integration of the Delegation Framework into an Apache-served web application that utilizes an underlying MySQL database when generating webpages. Chapter 9 summarizes the contributions of this work and proposes future work on the Delegation Framework.

Chapter 2

Related Work

2.1 Distributed Authorization and Delegation

2.1.1 Theory

Butler Lampson *et al.* dedicated a portion of their seminal work on a general calculus for access control (the most notable contribution being the "speaks for" relation) to the theoretical underpinnings of delegation[2, 17]. These works explore formalisms of delegation as a basic authorization primitive. They also touch on some of the interesting fundamental characteristics of delegation, namely the importance of identifying a delegate principal as working on behalf of its delegator. A delegate is not simply Alice, but must use a special identity such as "Alice **for** Bob" specifying that Alice is working on Bob's behalf. This provides a theoretical construct for dealing with the confused deputy problem [14] and highlights the consequent care that must be taken to avoid the misapplication of authorities delegated from principal Bob when performing an operation on behalf of a separate principal Charles. The confused deputy is an fundamental problem that the Delegation Framework, like any reasonable delegation system should, go to great lengths to avoid.

Gasser and McDermott described the principles for the proper construction of a delegation architecture [13]. This work has much in common with that of Lampson *et al.*, but covers delegation exclusively whereas Lampson *et al.* treat delegation within

a larger discussion. The paper serves as a tutorial on the delegation primitive and explores the mechanics of delegation in more depth than Lampson *et al.*. Certificate structure, delegation expiration, and delegating subsets of authority are among the issues they consider. Many of the presented ideas are leveraged in the Delegation Framework.

2.1.2 Applications

Kerberos [21] is a distributed authentication system that uses a trusted server to authenticate clients on behalf of the services they use. Following an authentication protocol between the client and Kerberos service, the Kerberos service issues the client a credential called a ticket. The client then supplies this ticket to the other services it uses to prove its authority. The trusted server may specify a "OK-AS-DELEGATE" flag within the tickets granted to the client. With this flag set, the user may delegate the authority associated with their credentials using a proxy ticket. The primary drawback of delegation using Kerberos is that it requires a central trusted server. The central trusted server is considerable drawback for security, flexibility, and availability of services. The Delegation Framework can provide delegation without reliance on a central trusted online component; a delegation between a delegator and delegate requires no involvement on behalf of a trusted third party.

The Taos operating system [32] provides an operating system level implementation of the concepts devised in the authors' theoretical work [2, 17] including roles, groups, secure channels, and delegation. Each host includes a component called the authentication agent which manages knowledge of principals and their associated credentials. The authentication agent runs in user space, and each application is linked with the necessary means to communicate with it. Agents on different hosts are able to communicate with one another, allowing authentication primitives such as delegation to be applied across hosts. The Delegation Framework incorporates an authentication agent-like component, called the delegation agent, but allows that each user and each individual client and service program have its own authentication agent for clearer isolation of user authority from the authority held by individual

programs. The major drawback for the Taos approach is that today, more than ten years after the original research on Taos took place, widespread support for delegation primitives in popular operating systems has not materialized. Requiring operating system support has greatly hinders deployability. The Delegation Framework favors an approach implemented entirely in user space so that system administrators can enabled the delegation primitive in the clients and services running on their hosts without complex kernel modifications.

A number of systems with delegation or delegation-like mechanisms are implemented in userspace and operate successfully without operating systems (as in Taos) or the support of a trusted third party (as in Kerberos). These userspace mechanisms tend to have much better deployability characteristics. However, existing userspace implementations do not fully realize the delegation solution.

Once such system is the popular Secure Shell (SSH) protocol [4], most notably the OpenSSH implementation [22]. SSH has enjoyed wide deployment as an answer to the `telnet` remote login program's security issues. The SSH protocol specifies support for a delegation-like mechanism called agent forwarding, and OpenSSH users may employ a program called `ssh-agent` that manages the user's public keys and engages in authorization protocols on behalf of user's SSH clients. When a user logs into a remote host, the user may specify an optional flag that causes the client to delegate the user's authority, allowing the remote host to act as his deputy. The "delegation" technique is called *agent forwarding*, and allows any SSH clients executed on the remote host to ask the user's `ssh-agent` on local host to engage in an authorization protocol on its behalf. With agent forwarding, it is not possible for the user to delegate a subset of his authority to the deputy host, only all or none of his authority. Also problematic is the lack of an is no opportunity to the agent to later revoke an authorization of a remote login; a deputy may maintain a persistent connection to the host authorized by the agent indefinitely and use or abuse the user's authorities on that host.

REX [16] is a remote execution system that improves upon agent forwarding mechanism of SSH. Like SSH, REX allows the forwarding of access to a credential

management agent to the user's remote execution environment upon login to a remote machine. Unlike SSH, the REX agent will present a confirmation dialogue box to the user when a remote `rex` command attempts to use the forwarded agent. The dialogue box asks the user whether to trust the host in question. This allows a user to build a policy dictating which hosts the agent delegates the user's credentials over time. This is an improvement when compared to SSH, but the mechanism is still course grained; the user's decision is binary in that the user can only delegate all or none of his authority to any given host.

SSH and REX are powerful tools for remote execution with delegation-like mechanism. However, they are not architected to allow explicit fine-grained delegation of authority to deputy hosts for precisely defined periods of time. Nevertheless, some characteristics of SSH and REX have influenced the design of the Delegation Framework: no special operating system support requirements, credential management agents, and communication channels dedicated to passing credentials and request for credentials between client agents and deputies.

Other userspace-implemented systems resolve some problems with SSH and REX by employing explicit certificate-based delegation mechanisms. One such system, called PorKI [26], allows PDA users to store their credentials and delegate their authorities to client machines, such as computer lab terminals, of which the level of trustworthiness is less than ideal. Rather than enter a static password into a terminal machine, a user can delegate the terminal temporary credentials using *proxy certificates* created on the PDA and transmitted to the client machine via the Bluetooth wireless protocol. System administrators specify different trust levels for the client terminals they administrate. The user's proxy certificate specifies authorization policies for each trust level, thereby allow client machines in each trust level different sets of authorities. The delegated credentials are sufficient for the user to access his network resources reasonably while limiting the set of authorities leaked in the event of the client machine's compromise. The user's authorities may only be exploited temporarily whilst delegated credentials on a client machine remain valid. The user's cryptographic keys remain isolated on the PDA and the password protecting them

is never typed into an unfamiliar machine where it could be captured. PorKI is an improvement over the status quo, but it focuses on concerns of untrustworthy client machines and consequently allows only single-level delegations to client machines. It is not possible for programs on the client host to re-delegate authority to a deputy across a network, or for the deputy to re-delegate authority to another deputy, and so on. While allowing a somewhat more fine-grained mechanism than SSH and REX and allowing delegations for precise periods of time, the limited form of delegation provided by PorKI does not allow the large amount of trust placed in the deputies of many systems to be eliminated. The Delegation Framework allows for arbitrary levels of delegation and re-delegation to support whatever system architectures that may be interested in employing delegation, and thereby allow for strongly trusted deputies to be replaced with delegate deputies wherever they may exist.

The Grey System [5], a system similar in many respects to PorKI, uses smartphones for delegation of authority to access both physical and virtual resources and allows users to delegate their authority to one another. Users may delegate credentials authorizing access to client machines, network resources, and physical spaces to one another using smartphones that communicate via Bluetooth- or SMS-based protocols. Unlike PorKI, users employ the smartphone (or PDA) as a delegate for accessing any of the aforementioned resources, not just a delegating credential management agent. Provided that PDAs or smartphones become highly ubiquitous and can effectively secure the data they store, their physical, not merely virtual, separation for the potentially untrustworthy client machines make them excellent candidates as trusted devices on which users store their credentials and perform delegations. The Delegation Framework does not presently support PDA or smartphone devices, and therefore user cryptographic keys must be stored on any client machines they use. However, the Delegation Framework can support a PDA- or smarthphone-based implementation provided that a PDA or smartphone supports TCP/IP and can perform a few important cryptographic operations.

SPKI/SDSI [9] (a project resulting from the merger of two formerly separate projects, SPKI and SDSI) builds on the delegation theory work of Gasser and Mc-

Dermot [13]. The SPKI/SDSI work describes *attribute certificates* (mapping authority to identities) and *authorization certificates* (mapping authority to a public key). In an attribute (authorization) certificate, an issuing principal specifies some set of authority to be delegated to a subject principal for a specified period of time. Making an authorization decision involves following a chain of attribute (authorization) certificates back to a trusted issuer. In SPKI/SDSI, this trust issuer need not be globally trusted as in traditional PKI. Following the certificate chain, the set of authority ultimately held by the subject at the bottom of the chain must be the intersection of all the sets of authority specified by each attribute (authorization) certificate along the chain. The intersection operation prevents any principal from authorizing the use of authority it does not hold or is not authorized to assign to another principal. The certificates used to delegate authority in the Delegation Framework are very similar to attribute certificates of SPKI/SDSI.

The term "delegation" is used in a variety of topical areas in computer science. Other systems refer talk about delegating tasks to software agents. The C# programming languages offer a "delegate" keyword for specifying callback methods in event-driven systems. Specifically within the area of network- and security-focused research, the Delegation Framework shares something of a naming collision with other systems that describe "delegation" or "delegating architectures". The Delegation Framework uses the term delegation in the sense of passing authorities from one principal to another such that the recipient of the delegation becomes authorized to perform tasks on the issuer's behalf. It is worth noting the difference between the Delegation Framework and other "delegation" or "delegating" architectures.

The Delegation-Oriented Architecture (DOA) [29] is a research effort aimed at resolving many of the problems associated with network "middleboxes" such as packet filters, network address translators, and transparent caches. DOA reintroduces a Internet-wide flat name space for all hosts (a role once served by the IP address space that has been diminished due to prevalence of network address translators) using what are called *endpoint identifiers* (EIDs). A host may specify a vector of EIDs in packet headers to explicitly redirect its packets to a middlebox or sequence of mid-

dleboxes whose functionality the host's administrator desires. This allows for greatly flexibility in the architecture of networks with middleboxes, allowing the middleboxes to be removed from points inline between end hosts and the internet. DOA focuses on specifically on delegating network-level packet manipulation tasks to remote packet processing hosts such as network address translators and firewalls, whereas the Delegation Framework focuses on the delegation of tasks (and their associated authority) to remote hosts at the level of application protocols.

Ostia [12] employs a "delegating architecture" to produce more effective application execution sandboxing. The sandboxed application requests operating system resources by way of a controlling agent program, thereby implicitly delegating the task to the agent. While Ostia, like the Delegation Framework, is motivated by the desire to minimize the abuse of authorities, the manner in which it employs the idea of delegation is quite different. In the Delegation Framework, and many of the delegation systems described above, a principal holding a set of authorities delegates those authorities to a principal that is less privileged. Ostia's approach is the reverse; an unprivileged process delegates the task of obtaining authority to perform a particular operation to a more privileged controlling process. Further, Ostia focus on executing environment sandboxing limits the target set of authorities to those provided on the local host.

2.2 Trust Management Systems

Trust and authorization policy issues are at the core of the debate on authorization in networked applications. A body of work on "trust management systems" has identified the need for flexible mechanisms able to express complex authorization policies and trust relationships that is independent of the needs of a particular system. This allows for general policies that apply across a range of applications rather than being geared toward a specific one. Trust management systems typically consist of policy description languages for the specification of trust relationships and trust-requiring operations, and compliance checking algorithms for the evaluation of authorization

statements written in the policy language.

Trust management systems are important in delegation of authority systems due to a need for a clear specification of delegated authority. The set of authorities delegated to a given principal must be unambiguously described, and the policy languages of trust management systems are useful for this task. Trust management systems suggest a useful way to think about the delegation of authority mechanism: delegation of authority enables users (holders of authority) to dynamically define and distribute policies pertaining to the authority they wield.

Blaze *et al.* engaged in the seminal work on trust management systems, building a system they called PolicyMaker [7]. PolicyMaker uses *filters* to specify under which conditions given policy assertions apply. The policy language itself is very simple, allowing 1) policy assertions with specific filters, and 2) query statements used to determine whether a given request is permitted by a given policy. PolicyMaker assertions and filters are implemented in a "safe" (i.e., does not have access to filesystem or network resources) version of the AWK interpreted language. The AWK variant is a language contained within the PolicyMaker policy language, and could be substituted by any "safe" interpreted language. While PolicyMaker's stated aim is to separate trust management mechanisms from the workings of the target system itself, its reliance on the AWK language programs leads to a situation in which assertions and filters become complex and application-specific. The PolicyMaker framework around this language is certainly application independent, but this "independence" (a heralded benefit of trust management systems) is not implicitly transfered to the assertions and filters written by system designers.

Usability is a major concern with trust management systems as they are applied to the Delegation Framework. If users can be expected to delegate authority in a fine-grained manner, there must exist a means for them to define policies to be contained in the certificates they pass to delegates. Trust management systems' policy languages are beyond the grasp of novice users, and even knowledgeable users will find them tedious to deal with. This problem is a major obstacle for the Delegation Framework which presently relies on a simple, home-grown policy definition language.

The Delegation Framework is not tightly bound in its design or implementation to any particular trust management system, leaving room for future adaptations.

2.3 Other Related Work

A essential part of making the Delegation Framework practically deployable requires that it be possible to easily integrate the Delegation Framework with existing software. A few key implementation techniques that have been adopted in the Delegation Framework, or show promise for future adoption, can drastically ease the challenge of integrating delegation into exist client and service software. These are essential to making the Delegation Framework truly practical.

2.3.1 Extensibility through Interposition

A great deal of previous work has explored system call and dynamic library interposition as a technique for extending the functionality of existing interfaces [8, 24, 12, 15, 25, 28]. Interposition mechanisms allow system and library interface calls to be intercepted and redirected to new routines, allowing the interface to be modified or replaced entirely with a new implementation. Consequently, it becomes possible to modify the behavior of programs using interposition without modifying the programs themselves. In the following examples, interposition mechanisms make it possible for to transparently retrofit software with the following functionality without modifying the software's source code:

- profiling and debugging [8]
- execution environment restriction mechanisms [24, 12]
- file and socket encryption and compression [15]
- insertion of protocol stack layers between transport and application [25]
- network distribution of computationally-intensive calculations [28]

The Delegation Framework leverages some of these ideas using dynamic library interposition, causing the target software to augment the data transmitted within the TCP sockets with useful metadata using ideas derived from Tesla [25], and to pass data back and forth with a local agent program that may initiate network communications of its own from based on ideas from Jones [15] and Thain & Livny [28].

2.3.2 Checking Authorization Policies in Legacy Code

An important and challenging task for integrating the Delegation Framework is properly instrumenting service software with authorization checks. Many code integration problems can be solved automatically with dynamic library interposition, but this authorization check instrumentation problem is not one of them. Dynamic interposition techniques offer the possibility of placing authorization checks at well-known library interfaces, but a useful correspondence between these library interfaces and the interfaces offered by many network application services may not exist in a given case. Instrumenting a service with Delegation Framework authorization checks via dynamic library interposition is a task that cannot be accomplished in a sufficiently general way for all service software.

Recent research suggests other difficulties facing this instrumentation task. Xiaolan Zhang *et al.* have conducted work using the static analysis tool CQUAL in which they analyze the placement of authorization checks within the Linux kernel by the Linux Security Modules [33]. Rather than place these authorization checks at a well-know interface such as the system call interface, the Linux Security Modules placed them internally deeply within the kernel. They developed techniques using CQUAL to verify automatically whether authorization hooks completely covered all access modalities. Zhang *et al.* found that the manual work performed by well-qualified individuals placing these authorization checks was error prone, difficult to verify manually, and left the kernel vulnerable to exploitation. Though Zhang *et al.* focused on kernel code, which is known for being particularly complex and difficult to understanding and for which there exist fewer experience developers, Integrating authorization checks into the Delegation Framework is likely similarly error-prone.

The current design of the Delegation Framework, having only made manual instrumentation of service source code as easily as possible, does not fully address these major challenges.

However, recent work has successfully retrofitted legacy software with authorization policy enforcement hooks using static program analysis techniques [11]. This work acknowledges the need for security solutions that are deployable and can be integrated into legacy software. Their work describes two techniques, one which identifies the location of "security-sensitive operations" in a program's executable code and, based on its results, one for subsequently instrumenting the code to include reference monitor calls into an authorization checking routine based. The identification technique looks for locations where the code performs "primitive operations on critical server resources." Provided that the identification technique is effective, its plausible that their instrumentation techniques could be adapted to insert Delegation Framework authorization checks based on this identification information. However, a study of the applicability of these techniques to the Delegation Framework has not yet been explored.

Chapter 3

Design Overview

The design of the Delegation Framework is informed by needs of a delegation system as outlined in theoretical work on delegation [2, 17, 13], the design choices employed in many research efforts in delegation systems [32, 16, 26, 5, 9], and deployability problems from which many technologies suffer (see Chapter 1). Much of the Delegation Framework design is derived from previous work. Where noted, design decisions have been influenced by the aim of improving deployability.

3.1 Delegation Credentials

A credential proving that a principal holds a particular set of authorities in the Delegation Framework is comprised of a sequence of cryptographically signed statements that prove that authority has been delegated. These signed statements take the form of X.509 certificates. Taken together, a sequence of these certificates forms a *proof of delegation.* A significant portion of the functionality the Delegation Framework is devoted to dealing with proofs: constructing and signing them, requesting and passing them between principals, verify their authenticity, and mapping the authorities they prove to authority-requiring operations. Effectively supporting proofs of delegation requires properly-structured certificates and a language in which the authorities to be delegated can be described.

3.1.1 Delegation Certificates

The Delegation Framework proofs are comprised of two types of certificates: *delegation certificates* that map authorization policies to identities, and *identity certificates* that map a principal's identity to a public key. An issuing principal produces a delegation certificate to assert an authorization policy describing how a subject principal may use the issuer's authority, signing it with a public key. A chain of delegation certificates form the backbone of a proof of delegation, each certificate authorizing the issuer of the next certificate in the chain.

A delegation certificate may actually assigned more authority that its issuer holds. This is permitted because a principal's authorities can change over time, and a delegation certificate may have been created prior to the expiration of a subset of those authorities. The expiration of one subset of authority does not invalidate other authorities held by the delegator. Therefore, a delegation certificate authorizes the intersection of the set of authority held by the delegator and the set of authority the certificate assigns to the delegate. When a sequence of delegation certificates is chained together (forming a proof of delegation), a whole series of these intersection operations occurs as each delegate re-delegates the authority it has been delegated.

Identity certificates, familiar from various PKI systems, are used to map identity to keys so that the signatures on delegation certificates can be check during the proof verification process. In addition to use in proofs, the Delegation Framework uses identity certificates during peer authentication. The Delegation Framework does not depend on any particular PKI system so long as it securely maps identities to public keys.

3.1.2 Authorization Policy Statements

Delegation certificates must include a statement of the authorities assigned to the delegate using an authorization policy language. The Delegation Framework design is not bound to any particular language. However, because the set of authorities authorized by a proof of delegation is generated by taking the intersection of the

authorization statements contained in each delegation certificate, it should be possible to represent the intersection of any two policy statements defined in the language.

If a tractable algorithm for computing the intersection of two policies does not exist, it is suitable for a service to retain all of the authorization statements in a given proof. In such a scenario, checking whether a given access request is authorized by the proof amounts to checking whether the required access authorities are authorized by each of the policies contained in each of the proof's delegation certificates. Therefore, any of the policy definition languages from the various trust management systems could function sufficiently in this role.

3.2 Network Protocols

Section 1.2.1 described issues related to network protocol support for delegation. To deal with these issues, the Delegation Framework provides two new protocols, the Delegation Protocol and the Speaks For Layer Protocol. These protocols provide the communication needs of the Delegation Framework without requiring modifications to application protocols. Chapter 5 describes the Delegation Protocol and Speaks For Layer Protocol in greater detail.

3.2.1 The Delegation Protocol

The Delegation Protocol provides peer principals with a separate channel through which they negotiate the client's authority to access the service via application protocol flows. The Delegation Protocol enables the following transactions:

- A principal delegates authority to an unprivileged principal
- A principal proves it has been delegated the authority to work on behalf of another principal
- A service requests evidence that a client holds an authority
- A principal requests that authority be delegated to it

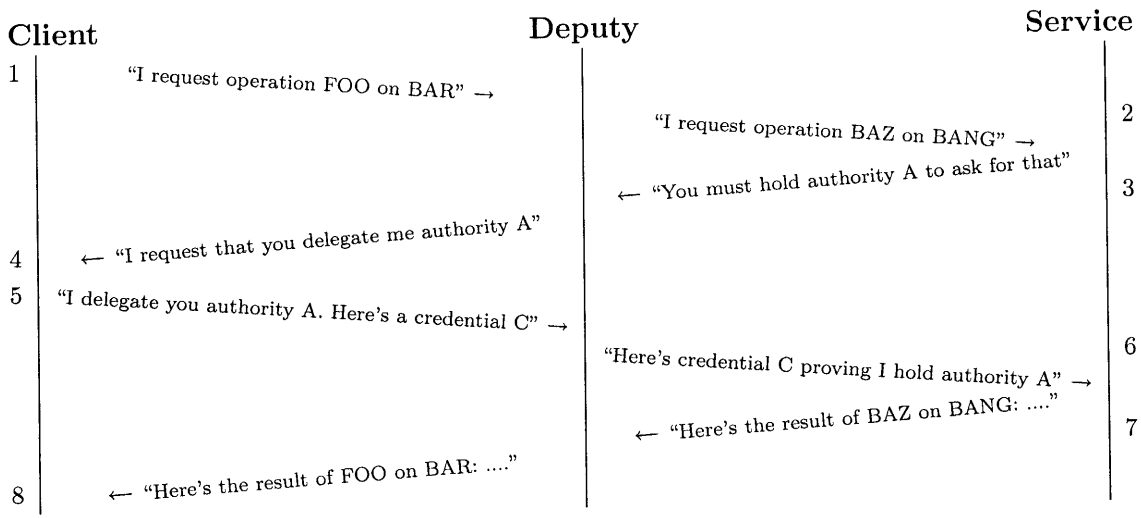

Figure 3-1: The conceptual model of the Delegation Protocol

Figure 3-1 depicts these different Delegation Protocol transaction at work. A client sends a request to a deputy asking that an operation be performed via an application protocol (step 1), causing the deputy in turn to make a request on behalf of the client asking a service to perform an operation (step 2). The service does not know whether the deputy is authorized to make its request, so it sends a Delegation Protocol request for evidence that the deputy holds some authorities (step 3). The deputy does not hold the authority, so deputy sends a Delegation Protocol request asking the client to delegate it the authority required to work on the client's behalf (step 4). The client, holding the required authority, delegates it to the deputy by passing the deputy a delegation credential via the Delegation Protocol (step 5). The deputy sends its delegation credential to the service via the Delegation Protocol to prove that it holds the required authority (step 6). The service verifies the delegation credential and, satisfied that the deputy now holds the required authority, responds to its application protocol request (step 7). Upon receiving a response to the request it performed on the client's behalf, the deputy responds to the client's request.

While the transactions enabled by the Delegation Protocol help the Delegation Framework avoid modifying application protocols, using the Delegation Protocol decouples delegation credentials from the application protocol requests to which they are intended to apply. The Delegation Protocol provides messages for negotiating principals' authority to access services, but these credentials only bind authority to identities and public keys. They provide no binding between a client's credentials sent via the Delegation Protocol and the client's application protocol requests sent via a separate channel. A mechanism must exist that binds a client's identity to application protocol requests.

This binding problem could be solved by creating credentials that bind authority directly to application protocol flows, but that is problematic because some clients will employ multiple application protocol flows when accessing a service. For instance, web browsers often open multiple connections to web servers when downloading content. Credentials would have to bind authorities to each of these flows individually. When authorizing a client to access a service, a user would have to include flow information

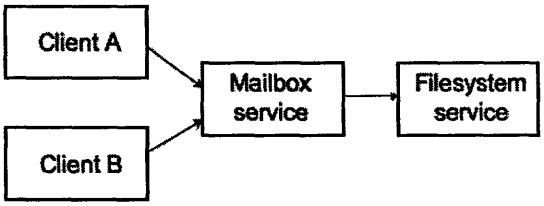

Figure 3-2: A deputy acts on behalf of two clients when communicating via one application protocol flow

in those credentials, requiring that new credentials be regenerated each time a client initiates a new application protocol flow.

Binding authority directly to flows is also problematic because a deputy can send application protocol requests via a single flow on behalf of multiple client principals. Figure 3-2 illustrates a network application in which this occurs. A mailbox server acts as a deputy by accessing a user's mailbox files stored on an underlying filesystem service on the user's behalf. Different credentials should apply to the flow at different times depending on whose behalf the deputy is accessing the underlying service (e.g., the deputy acts as "*deputy* **for** *clientA*" or "*deputy* **for** *clientB*" depending on which client it serves at a given time).

Instead of binding authorities directly to flows, a client could send flow-to-identity bindings via the Delegation Protocol. However, this would introduce significant network overhead; being transmitted via separate channels, application protocol and Delegation Protocol messages cannot be expected to arrive in a particular order. Therefore, ensuring that a flow-to-identity binding was correct would require the client to block its application protocol request until it received a confirmation that its binding had been received by the service. In the network application depicted in Figure 3-2, this could happen frequently due to the mailbox service acting as a deputy.

3.2.2 The Speaks For Layer Protocol

The Speaks For Layer Protocol address the problem of mapping authorities onto the flows to which they apply. Delegation credentials sent via the Delegation Protocol

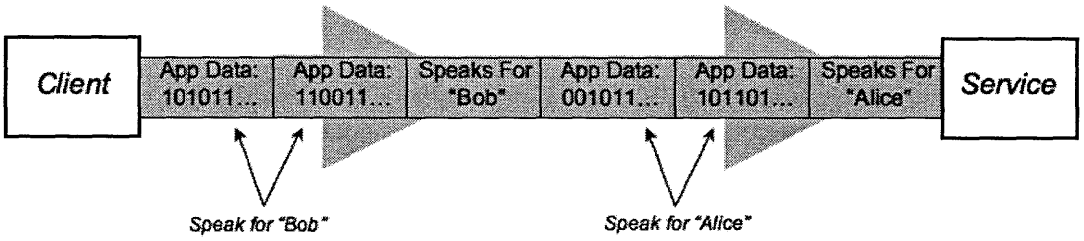

Figure 3-3: The contents of a client's Speaks For Layer Protocol-encapsulated application protocol flow

bind authority to identities (as is done with attribute certificates in SPKI/SDSI). The Speaks For Layer Protocol provides the other piece of the puzzle, binding an application protocol flow to the identity for which it speaks. The Speaks For Layer Protocol is a protocol in which application protocol data is encapsulated by a client before it is sent to a service. Normal application protocol data is sent wrapped in a "Data" message. When necessary, the client may insert a SpeaksFor message binding the flow to the identity for which the flow now speaks. The service receives these bindings, remembering the most recently received flow-to-identity binding. The service makes authorization decisions regarding individual application protocol access requests based on the identity given in the most recently received SpeaksFor message.

Figure 3-3 depicts the contents of the Speaks For Layer Protocol messages that a client sends to a service. The first message is a SpeaksFor message specifying that subsequent application protocol messages speak for the identity Alice. The client then sends two Speaks For Layer "Data" messages containing some application protocol request data. The service makes authorization decisions for these application protocol requests by checking the authority held by Alice. Following the two "Data" messages, the client sends sends another SpeaksFor message indicating that subsequent application protocol messages now speak for the identity Bob. The SpeaksFor message is followed by two more "Data" messages containing application protocol, for which the service makes authorization checks of authority held by Bob.

A client would not typically switch from speaking on behalf of one identity to speaking on behalf of an entirely other identity as depicted in Figure 3-3. A client, speaking on behalf of one user, will typically only send one SpeaksFor messages during

the lifetime of a given application protocol flow. However, a deputy that constantly switches between handling requests from multiple clients such as in Figure 3-2 will send a SpeaksFor message each time it switches between sending requests on behalf of one client to sending requests on behalf of another client (i.e., when the deputy switches between its identity "*deputy* **for** *clientA*" and "*deputy* **for** *clientB*").

3.3 Framework Architecture

Delegation credentials are essential to realizing delegation in the Delegation Framework and the proposed protocols facilitate deployment by avoiding application protocol modifications, but the problem remains that delegation credentials and the Delegation Protocol and Speaks For Layer Protocol must be integrated with legacy client and service programs. The Delegation Framework must be architected around the needs of the legacy programs with which it is to be integrated.

Integrating support for handling delegation credentials and two new network protocols into a legacy client or service program is a considerable challenge for a developer. Legacy code may be a challenge to understand and integrating the Delegation Framework may require an expensive and error-prone development process.

3.3.1 The Delegation Agent

Rather than require developers to port all the delegation functionality directly into the legacy software, the Delegation Framework introduces an agent program, the Delegation Agent, which performs much of the work of delegation on behalf of a legacy program. Each legacy client and service is assisted by its own Delegation Agent[1]. Each Delegation Agent is assigned an identity, and a corresponding identity certificate maps this identity to a public key. Using this identity and public key, the Delegation Agent manages principal credentials, much like Taos' authentication agent and OpenSSH's `ssh-agent` program, and speaks the Delegation Protocol on

[1]In legacy software implemented using multiple processes or threads., a single Delegation Agent serves all of the processes or threads of a given client or service.

behalf of the legacy process it is assisting, eliminating the need for the legacy software to understand the details of each of these tasks. Though the legacy program code must be modified to interact with its Delegation Agent, the Delegation Framework's agent-assisted design simplifies the task of integrating the Delegation Framework into legacy code considerably.

In the interest of maintaining separate notions of the authorities assigned to users and the authorities assigned to the clients they use, each user also maintains a separate Delegation Agent that is distinct from the Delegation Agents of each of the client programs he uses. When needed, a client program's Delegation Agent asks the user's Delegation Agent to delegate authority the client requires to perform its function. The user Delegation Agent provides a delegation user interface through which the user decides whether to delegate authority to a client when a client makes such a request.

3.3.2 The Delegation Framework Library

The Delegation Agent considerably reduces the amount of functionality that each legacy program must support directly within its code, but nevertheless the legacy code must be modified to support the Delegation Framework. The set of calls that the legacy software must perform comprise the Delegation Framework Library. This library provides calls to support the Delegation Agent's functions, calls to check whether a client is properly authorized, and calls for communicating via the Speaks For Layer Protocol.

In the interest of making modifications to legacy programs as easy as possible, the interface of the Delegation Framework Library should be conducive to being implemented using dynamic library interposition techniques [25, 15, 28]. For instance, a considerable amount of information can be derived from the pattern of socket I/O operations the legacy process performs. If the Delegation Framework Library interface is designed appropriately, it will be possible to reroute socket I/O calls to alternate implementations of the call that perform Delegation Framework Library calls without requiring a programmer to modify the legacy program manually. If calls to the Del-

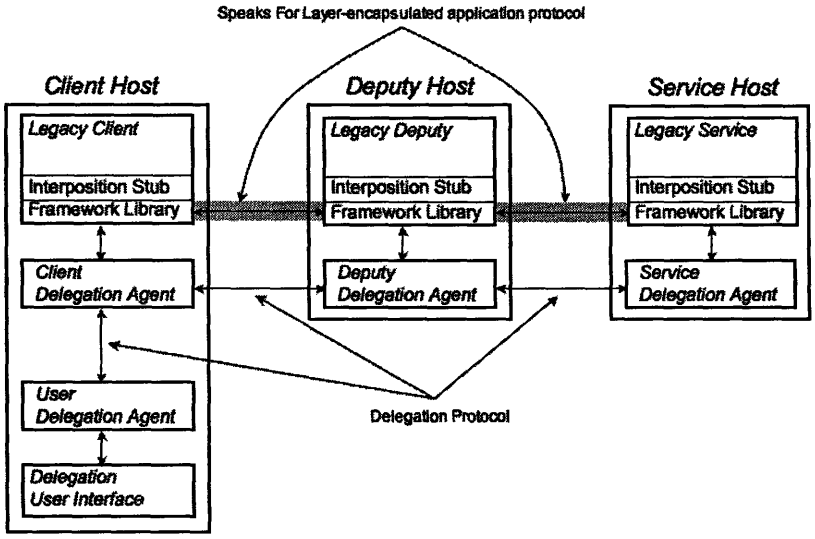

Figure 3-4: The Delegation Framework architecture

egation Framework Library can be implemented successfully using dynamic library interposition as a *stub*, it would considerably reduce the difficulty of integrating the Delegation Framework into legacy software.

3.3.3 The Components Working in Concert

Figure 3-4 illustrates the interactions that occur in a Delegation Framework-enabled network application between legacy processes, different instances of the Delegation Framework Library, and Delegation Agents. The figure depicts three hosts on each of which a client, deputy, and service program executes. An instance of the Delegation Framework Library resides within the memory of each legacy process. An interposition stub, also resident within each legacy process's memory, performs the appropriate Delegation Framework Library automatically for the legacy process when it is able. Legacy programs communicate with one another using their normal application protocol, but they encapsulate the application protocol data within Speaks For Layer Protocol using calls in the Delegation Framework Library.

For each legacy program, there exists a Delegation Agent that manages the legacy program's delegation credentials and communicates via the Delegation Protocol with other Delegation Agents on behalf of the legacy program. The legacy program com-

municates with its Delegation Agent using its instance of the Delegation Framework Library, which opens a communication channel with the local Delegation Agent via an operating system-provided domain socket. In a legacy program involving multiple threads or processes, there exists only one Delegation Agent, but each legacy thread or process maintains its own separate communication channel with the Delegation Agent.

In addition to a Delegation Agent working on behalf of each legacy program, there exists a separate Delegation Agent working on behalf of each user. The user's Delegation Agent communicates with the Delegation Agents of the legacy client programs that the user uses. A delegation user interface, through which the user makes authorization decisions, is attached to each user Delegation Agent. The user's Delegation Agent and the corresponding delegation user interface may reside on the same host as the user's client programs (as depicted in Figure 3-4) or, for added security, on a separate host or portable hardware device as has been done in previous delegation applications [26, 5].

Chapter 4

Delegation Credentials

Delegation credentials provide the mechanism by which one principal asserts that it delegates authority to another principal. A delegate uses these assertions to prove its authority when accessing a service on behalf of the delegator. This chapter details the structure of delegation credentials used in the Delegation Framework.

4.1 Delegation and Identity Certificates

The Delegation Framework employs two types of certificates: identity certificates which allows a principal to assert mappings between an identity's name to public key, and delegation certificates which allow a principal to assert a delegation of its authority to another principal. These certificates are the building blocks of delegation credentials.

4.1.1 Identity Certificates

Using an identity certificate, an *issuer* asserts that a *subject* holds a particular public key for a specified period of time. An identity certificate contains the following fields:

- `issuer` - the name of the issuing identity
- `subject` - the name of the subject identity

- `subject_key` - the subject identity's public key
- `not_before` - the time before which the certificate is not valid
- `not_after` - the time after which the certificate is not valid

In the prototype implementation of the Delegation Framework, the `issuer` identity for all identity certificates is a well-known, globally-trusted certificate authority principal whose public key and name are stored on all participating hosts. This amounts to a simple, single-tiered PKI system. The single-tiered PKI approach would not effective for internet-wide deployments of the Delegation Framework. A real world deployment should employ a true hierarchical PKI-based or other infrastructure for mapping an identity's name to key, such as DNSSEC. Provided it can be successfully deployed, DNSSEC servers have potential to be convenient places to store identity certificate-like mappings; when a client looks up the IP address of a given service with which it is about to initiate an application protocol stream, the query could also return the identity certificates, or equivalent identity certificate-like mappings, for all Delegation Framework-based services on the host.

The `subject` identity and the `subject_key` represent the name-to-key mapping that the certificate provides. The `not_before` and `not_after` fields are timestamps specifying the period of time during which the certificate is valid. The use of these timestamps requires all hosts that are part in a given network application using the Delegation Framework to maintain reasonably consistent clocks within a few seconds of one another.

4.1.2 Delegation Certificates

Using a delegation certificate, an issuing delegator asserts that it grants a subject delegate a set of authorities that the issuer presently holds or may hold at some time during the delegation certificate's validity. A delegation certificate includes the same fields as an identity certificate (thereby making it possible to include a name-to-key mapping) along with one additional field:

- `auth_policy` - an authorization policy assertion specifying the delegated set of authorities

As described in Section 3.1.2, the design of the Delegation Framework is not bound to a particular authorization policy language. The Delegation Framework prototype implementation uses the language described in Section 4.3.

A problem with relying exclusive on global name-to-key mapping infrastructure is that they tend to produce mappings with long lifetimes. When a principal's private key credential is compromised, the long-lived mappings between its name and public key must be revoked and replaced with new mappings. To help deal with this issue, delegation certificates include a `subject_key` field, allowing a principal to provide an name-to-key mapping for the principal to whom it delegates authority, making it possible to delegate authority to a temporary private key that does not have an corresponding name-to-key mapping in the global PKI infrastructure, DNSSEC, or whatever other name-to-key mapping mechanism that may be used.

Having the `subject_key` field in delegation certificates allows a deputy administrator to store the deputy's private key on a secure (perhaps offline) host separate from the deputy's service-providing host, and regularly regenerate and push temporary keys and short-lived delegation certificates to the online deputy host. The authorization policy specified in these delegation certificates is of a special form that allows the temporary keys to inherit all authority delegated to the globally-mapped identity. However, in the event of a deputy's compromise (provided that the compromise is discovered), the temporary credentials hijacked from the host only authorize the authorities delegated to service for the window of time during which the hijacked temporary credentials remain valid. Global identity-to-key mappings remain valid, allowing the administrator to go to the trouble of having a certificate authority produce a long-lived mapping to a new key pair.

The Delegation Framework implements certificates using the OpenSSL's X.509 and cryptographic library routines to construct, sign, and verify certificates. The Delegation Framework is not bound to any particular public key algorithm or specific parameters of that algorithm. Any secure public key algorithm with appropriately-

sized keys will suffice, and the OpenSSL library implements several. The Delegation Framework prototype uses 2048-bit RSA keys.

4.2 Identity Names

The Delegation Framework uses strings with an email address-like naming format to identify principals in the `issuer` and `subject` fields of identity and delegation certificates. The following of some examples of the name format:

- `alice@foo.anydomain.com`
- `mailbox@bar.somedomain.com`
- `filesystem@baz.otherdomain.com`
- `database@bang.otherdomain.com`

Each principal in the Delegation Framework (be it a user, client, deputy, or service) has an associated name assigned by its administrator.

Maintaining a strict mapping between the domain name specified in an identity and the actual domain name of the host on which a principal's legacy software and Delegation Agent processes reside. For example, users who may employ multiple client machines or whose client machines are part of a DHCP pool in which domain names can change. Additionally, a single service may be served by a dynamic pool of hosts. Therefore, the Delegation Framework does not rely on a strict mapping being a host's domain name and the one specified in an identity string. However, in the case of services, it is useful to maintain this mapping because it will help users and clients locate services identified by these strings. This would be particularly important in deployment of the Delegation Framework where DNSSEC is used to map keys to identities. As a guideline, a service principal's domain name specified in an identity string should smallest domain in which the service can be expected to remain. User and client Delegation Agents should be suspicious of services that do not follow this guideline, and maintain a policy for dealing with them.

4.3 Authorization Policies

The Delegation Framework prototype supports a simple language for defining authorization policies contained in delegation certificates. Authorization policy are specified by strings of the following form:

- `<identity>:<operation>:<subject>`

The `<identity>` field is of the form described in Section 4.2. This identity defines the principal of the service on which the operation and argument fields are relevant, *not* the principal to whom the authority has been given (the holder of the defined authority is specified within a delegation certificate). The `<operation>` and `<subject>` fields specify the access operation attempting to be performed the subject object of that operation. The precise contents of the `<operation>` and `<subject>` fields are specific to the service for which the authorization policy applies.

Each of the individual fields may include a set definition, allowing an authorization policy definition to specify multiple combinations of field values. A set definition is represented either as a comma-separated list of string values or as a string with an asterisk. An asterisk defines that an arbitrary length string may take its place. Asterisks are only allowed as the last character of the `<operation>` and `<subject>` fields. They are allowed in the `<identity>` field in two locations: the end of the string preceding the '@' symbol, and the beginning of the string following the '@' symbol. These restrictions on the location of the asterisk make it possible to compute the intersection of two authorization policies using a simple tractable algorithm while still providing a fairly descriptive authorization policy language.

The following are examples of authorization policies that might be employed in a network file service:

1. `filesystem@foo.somedomain.com:*:/home/alice/*`
2. `filesystem@foo.somedomain.com:read:/home/alice/www/*`
3. `filesystem@*.somedomain.com:read,execute:/usr/bin/*`

4. `*@*:*:*`

Example 1 specifies the authority to perform any operation on a file or directory within the `/home/alice` file hierarchy. This defines the set of authority that the file service would logically granted to the user `alice@foo.somedomain.com` to access her own home directory.

Example 2 specifies the authorities required to read the web files of the principal named `alice@foo.somedomain.com`.

Example 3 defines the authority to read and execute files and directories within the `/usr/bin` directory of any service named `filesystem` on any host in the `somedomain.com` domain name space. This is an example of authority that might be delegated to users of a distributed computing environment. Programs of interest can be maintained and made available by different sub-organizations within a given administrative domain on their own file servers such as is common in MIT's Athena system.

Example 4 defines the set of all authority. This defines the all-inclusive authorization policy that may define the authorities a deputy's administrator delegates to temporary online keys as described in Section 4.1 such that the temporary keys inherit any authority delegated to the deputy's identity.

A production deployment of the Delegation Framework would likely require a sophisticated authorization policy definition language, such as SAML or that provided by Keynote. These language would provide more descriptive capabilities than our scheme. The Delegation Framework prototype employs this simple scheme to allowing for experimentation with the architecture of the framework without requiring the prototype to implement a complex authorization policy language.

4.4 Credential Structure and Validity

Work on general purpose algorithm for distributed proving of authorization have proven to be complex [6]. These algorithms often go beyond the needs of the Delegation Framework. A simpler algorithm will suffice provided that clients and deputies

store the delegation credentials providing evidence of their authority, and that those credentials are structured appropriately.

The Delegation Framework requires delegation credentials to adhere to a structure. Each delegation credential includes two sequences of certificates: one consisting of delegation certificates, and the other consisting of identity certificates. The ordering of these sequences reflects the natural structure of the chain of delegation. The name of the issuer of each delegation certificate must match the name of the subject of the previous delegation certificate in the chain (no constraints are place on the issuing identity of the first delegation certificate or the subject identity of the last delegation certificate). The sequence of identity certificates must follow the same ordering. The subject of the next identity certificate must match the issuer of the next delegation certificate. This mapping requirement enables the credential verifier to quickly determine the public key used to sign the delegation certificate. If the i'th delegation certificate contains a non-empty `subject_key` field, it provides the identity-to-key mapping that the $(i+1)$'th identity certificate would have if the `subject_key` field were empty. The $(i+1)$'th position in the sequence of identity certificates is left empty.

4.4.1 Credential Structure

Figure 4-1 illustrates the structure of credentials using an example constructed according to the requirements of the Delegation Framework. Signature values are omitted from the figure, but are normally contained in each certificate. The 1st identity certificate must always be included in the credential because there does not exist a delegation certificates previous to the 1st delegation certificate in which an identity-to-key mapping could be included. Delegation certificates 1 and 3 leave the `subject_key` field empty, and consequently the 2nd and 4th identity certificates must be non-empty. However, the 2nd delegation certificate includes a non-empty `subject_key` (highlighted in bold font in Figure 4-1), providing the mapping that would otherwise be provided by the 3rd identity certificate.

i	**Delegation Certs**	**Identity Certs**
1	issuer=Alice subject=Bob subject_key=$\emptyset$ not_before=T_1 not_after=T_8 auth_policy=A_1	issuer=CA subject=Alice subject_key=1FB0... not_before=T_2 not_after=T_9
2	issuer=Bob subject=Charles **subject_key=7B42...** not_before=T_3 not_after=T_{10} auth_policy=A_2	issuer=CA subject=Bob subject_key=4C25... not_before=T_4 not_after=T_{11}
3	issuer=Charles subject=Diane subject_key=$\emptyset$ not_before=T_5 not_after=T_{12} auth_policy=A_3	$\emptyset$
4	issuer=Diane subject=Edward subject_key=$\emptyset$ not_before=T_6 not_after=T_{13} auth_policy=A_4	issuer=CA subject=Diane subject_key=A289... not_before=T_7 not_after=T_{14}

Figure 4-1: A sample delegation credential in the Delegation Framework

4.4.2 Credential Validity

To be considered valid, the certificate chain in each delegation credential must adhere to the following three constraints:

- The signature of each certificate must be valid
- The `not_before` fields of all certificates must not be after the current time and the `not_after` fields of all certificates must not be before the current time (the *temporal constraint*)
- Each authority granted to the subject of a delegation credential must be specified in each the `auth_policy` specifications of each delegation certificate (the *spatial constraint*)

The most basic of these is the constraint on signatures. For a delegation credential to be considered valid, each of its certificates must be considered valid. Each identity certificate must be validly signed by the key of the globally trusted certificate authority principal. Each delegation certificate must be validly signed by the key that has been mapped to the certificate's issuing identity, whether the key was derived from an identity certificate or from a previous delegation certificate.

The temporal constraint is satisfied if every certificates is valid at the present time. Therefore, the present time must be later in time the most recent time specified in the `not_before` field values of all the certificates in the credential. Correspondingly, the present time must be earlier than the earliest time specified in the `not_after` field values of all the certificates in the credential. The Delegation Framework uses a common representation for time: the integer number of seconds have passed since an epoch. Therefore, determining the latest and earliest time values from among a set of time values involves computing the maximum and minimum of the integer time values, respectively. In the example depicted in Figure 4-1, the delegation credential is valid from the time $max(T_1, T_2, T_3, T_4, T_5, T_6, T_7)$ until the time $min(T_8, T_9, T_{10}, T_{11}, T_{12}, T_{13}, T_{14})$.

To satisfy the spatial constraint, every delegation certificates in the chain must contain each of the authorized authorities and the original issuer must be known to possess each of these authorities as well. Each access-controlled service (where the credential is ultimately verified) maintains an access control list of the authority it has granted to users of the service, the principals who will be the original delegators of authority in any given credential.

Computing the set of authorities authorized by a delegation credential in order to meet the spatial constraint is straightforward. As described in Section 3.1, the set of authorities authorized by a delegation certificate is defined to be the intersection of the authority known to be held by the issuer with the set of authorities defined within the delegation certificate. In the example from Figure 4-1, the first delegation certificate in the chain specifies that "Alice" delegates "Bob" the set of authorities A_1 defined by the authorization policy in the `auth_policy` field of the certificate. Therefore, if the access control list on a given service to which a credential applies authorizes "Alice" to use the set of authorities A_0, the actual set of authorities granted to "Bob" by the first delegation certificate is $A_0 \cap A_1$. "Bob", holding the set of authority $A_0 \cap A_1$, then re-delegates a subset of his authorities to "Charles" with the second delegation certificate depicted in Figure 4-1. The authority held by "Charles" is the intersection of the set of authorities held by "Bob" and the set of authorities A_2 held specified by the `auth_policy` field of the second delegation certificate. Consequently, "Charles" is granted the set of authorities $A_0 \cap A_1 \cap A_2$. This pattern of intersection operations continues through the remainder of the delegation credential, finally authorizing the principal named "Edward **for** Diane **for** Charles **for** Bob **Alice**" with the set of authorities $A_0 \cap A_1 \cap A_2 \cap A_3 \cap A_4$.

4.4.3 Credential Verification

Verifying the validity of a proof is a matter of checking the ordering requirements and other constraints on the proof. Having imposed restrictions ordering constraints on the proofs in the Delegation Framework, it is possible to avoid complex proving algorithms. Verifying Delegation Framework proofs requires only a single linear pass

over the sequences of delegation and identity certificates contained in a proof. Naturally, the certificate signatures can be checked one at a time. The values required to check the temporal and spatial constraints can be computed iteratively as the verifier passes over each certificate.

The proof algorithm proceeds as follows. The algorithm first initializes values T_{not_before} to 0, and T_{not_after} to infinity[1]. The algorithm must also initialize A_{proof} to the authorization policy contained in the local access control list that describes the authorities granted to the issuer of the first delegation certificate in the chain.

After initializing this state, the algorithm examines each of the certificates in the proof. Assuming there are n delegation certificates (and thus n identity certificates), for each value i from 1 to n a proof verification algorithm does the following:

1. If i'th identity certificate is non-empty:
   (a) verify its signature. If invalid, report an invalid proof
   (b) check that its subject matches the issuer of the i'th delegation certificate. If they do not match, report an invalid proof
   (c) Compute the maximum of T_{not_before} and the value contained in its `not_before field` , and assign the result to T_{not_before}
   (d) Compute the minimum of T_{not_after} and the value contained in its `not_after field` , and assign the result to T_{not_after}
2. Verify the signature of the i'th delegation certificate using the public key derived from the `subject_key` field of the $(i-1)$'th delegation certificate (if the `subject_key` field was non-empty), or the i'th identity certificate. If invalid, report an invalid proof.
3. If $i > 1$, check that the issuer of the i'th delegation certificate matches the subject of the $(i-1)$'th delegation certificate. If they do not match, report an

[1]Under the epoch-based time representation, the time value 0 represents the earlier possible time that can be represented using this convention, the epoch itself (12 AM January 1, 1970 on UNIX systems) and time value infinity represents the latest possible time that can be represented (actually limited by the number of bit used to store time values by the operating system).

invalid proof.

4. Compute the maximum of T_{not_before} and the value contained in the `not_before` field of the i'th delegation certificate, and assign the result to T_{not_before}

5. Compute the minimum of T_{not_after} and the value contained in the `not_after` field of the i'th delegation certificate, and assign the result to T_{not_after}

6. Computer the intersection of A_{proof} and the authorization policy contained in the i'th delegation certificate's `auth_policy` field, and assign the result to A_{proof}

After completing the n passes through this sequence of steps, T_{not_before} and T_{not_after} will respectively contain the earliest time and latest time that the whole proof is valid, and A_{proof} will contain the authorization policy describing the authorities assigned to the subject of the last delegation certificate in the proof. If T_{not_before} is greater than T_{not_after} or if A_{proof} is the empty set, the proof should be considered invalid[2].

[2]These conditions can be checked after each of the n passes through the looping portion of the algorithm in order to avoid wasting time in the case of an invalid proof.

Chapter 5

The Delegation and Speaks For Layer Protocols

Application protocols cannot be assumed to support delegation of authority. The conventional approach to including support for new security features in a protocol is wrought with complications. Application protocol specifications are often drafted by standards organizations, and achieving alteration to their specifications requires a great deal of lengthy debate. Further, the adoption of new protocol specifications requires each protocol implementation to be modified which can also be a length and expensive process. The Delegation Framework is designed to avoids these complication by providing two novel protocols, the Delegation Protocol and Speaks For Layer Protocol. These protocols allow the Delegation Framework to provide delegation of authority in network applications without modification to the existing application protocol specifications used in the network application.

5.1 Application Protocol Model

The Delegation Framework relies on some assumptions about the application protocols to which it is being applied. Application protocols must adhere to the client-server model. Application protocols associated with many important network applications fit this model: HTTP, IMAP, database protocols, and NFS.

The typical order of events that occurs in the client-server model is that the client host sends a *request* message via the application protocol to the server asking that one or more specified operations be performed on one or more specified objects. Upon receiving the request message, the server may make one or more authorization decisions regarding each of the individual operations on each of the specified objects. If the client is authorized, the server performs the requested operation and sends a *response* message to the client via the application protocol with the result. If the client is unauthorized, the server responds with a message indicating its denial of the request.

In addition to adhering the client-server model, an additional assumption must be made about the application protocol. The Speaks For Layer Protocol allows the client to periodically "tag" an application protocol flow with SpeaksFor messages (see Section 3.2.2). This SpeaksFor message applies to all application protocol messages transmitted subsequent to it until the client tags the application protocol stream with a new SpeaksFor message. If application protocol messages and SpeaksFor messages are not delivered in order, it will be impossible for the server receiving them to conclusively determine the correct identity for which a particular application protocol message speaks. To support the Speaks For Layer, the transport over which application protocol messages are carried must be reliable and provide in-order delivery. For practical purposes on the internet today, this means the application protocol must be transported over TCP, though the design of the Delegation Framework is not bound to TCP[1].

The reliable, in-order delivery requirement eliminates the possibility of using UDP-transport application protocols with the Delegation Framework, whether or not they adhere to the client-server model. The NFS protocol is an example of a common application protocol transported over UDP, and NFS-based services are a strong potential target for a delegation-based authorization mechanism. Fortunately, implementations

[1]In recent years, a reliable, in-order transport protocol called Stream Control Transmission Protocol (SCTP) has been proposed as a more feature-rich substitute for TCP [27]. Though an examination using SCTP with the Delegation Framework is beyond the scope of this thesis, its probable that SCTP would be a suitable transport for application protocols in the Delegation Framework.

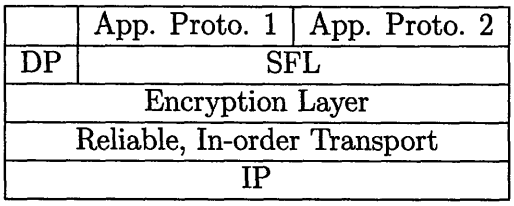

Figure 5-1: Delegation Framework protocol stack

of NFS that are transported over TCP exist, though the use of TCP versus UDP can considerably affect the protocol's performance.

Peer-to-peer and group protocols may also benefit from the delegation authorization paradigm, but may or may not function such that the Delegation Framework can be integrated as it is presently designed and implemented. This thesis will not consider the applicability of the Delegation Framework to these types of protocols.

5.2 Protocols Working Together

The Delegation Framework is able to achieve delegation across a network for a large class of application protocols not by substituting a new protocol for the existing application protocol, but rather by using the Delegation Protocol and Speaks For Layer Protocol in conjunction with a given application protocol. Together, the protocols form a *meta*-protocol, a version of the application protocol that supports the delegation authorization mechanism. Figure 5-1 illustrates the organization of the protocol stack that the Delegation Framework employs.

The Speaks For Layer Protocol (SFL) must be carried over a reliable, in-order transport in order the SpeaksFor messages to be properly ordered with the application protocol requests that they attach to specific identities (see Section 5.1). The Delegation Protocol is used to pass potentially large delegation credentials (containing arbitrarily many delegation certificates), and so its messages are not guaranteed to fit within a single maximum transmission unit specified for a given data link that may lie between a client and service[2]. To ensure that its large messages are received

[2]The maximum transmission unit by standard gigabit ethernet is 1500 bytes. A sequence of

properly, the Delegation Protocol must also be carried over a reliable, in-order transport. Though it is not an express requirement of the design, the prototype Delegation Framework implementation relies on TCP for reliable, in-order transport.

Further, the Delegation Protocol and Speaks For Layer Protocol must be carried within an encrypted channel. The Delegation Protocol carries a considerable amount of information regarding the authorities held by individual principals that could be discovered by an eavesdropper, and the Speaks For Layer Protocol contains authenticating information that could be leveraged by adversaries to abuse a principal's delegation credentials (described in further detail in Section 5.5). Encapsulating both protocols in an encryption layer prevents most information leakage attacks.

The remainder of this chapter describes the facilities of the Delegation Protocol and the Speaks For Layer Protocol, and details the meta-protocol they produce in conjunction with a generic application protocol that adheres to the client-server model (see Section 5.1).

5.3 Diagram Notation

The remainder of this chapter employs network protocol diagrams. Figure 5-2 is a simple example of one such diagram. Each vertical bar in a given diagram corresponds to a single principal participating in the protocol. These vertical bar are each labeled with the principal's name or role in the protocol. The sloped lines of text between the vertical bars represent protocol messages transmitted between the principals, and and the corresponding arrows point toward the recipient principal's vertical bar. The diagrams depict a temporal dimension, which advances downward such that MessageTwo follows MessageOne in time in Figure 5-2.

identity certificate containing the 2048-bit (512-byte) public keys used in the Delegation Framework can easily exceeds this capacity.

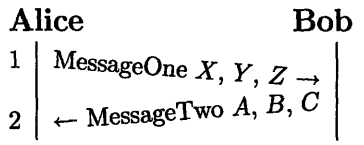

Figure 5-2: A generic protocol example

5.3.1 Protocol Messages

The particular protocol that a message is a part of is explicitly noted for clarity in the network protocol diagrams.

Delegation Protocol messages are colored red and enclosed in square brackets as follows:

- ^{DP}[Message]

Application protocol messages are colored blue and enclosed in square brackets as follows:

- $^{(p,i)}$[Request]
- $^{(p,i)}$[Response]

With regard to application protocol message notation, p specifies the application protocol in which a message is communicated, and i specifies a particular session of the application protocol p. For instance, messages as part of the 1st session of the SMTP protocol would be denoted as $^{(SMTP,1)}$[.]. This notation allows diagrams to distinguish between different sessions of the same protocol (i.e., $^{(IMAP,i)}$[.] versus $^{(IMAP,j)}$[.]) and sessions of different protocols (i.e., $^{(IMAP,i)}$[.] versus $^{(POP,i)}$[.]).

Speaks For Layer Protocol messages are colored green and enclosed in square brackets as follows:

- ^{SFL}[Message]

5.4 Delegation Protocol Channel Setup

Before a legacy client process initiates an application protocol session with a service, the client's Delegation Agent must first attempt to establish a Delegation Protocol channel with the service's Delegation Agent. This process indicates whether the legacy service is Delegation Framework-enabled. If the client Delegation Agent determines that the service supports the Delegation Framework, then the client can encapsulate application protocol requests within Speaks For Layer Protocol messages. If the service is not Delegation Framework-enabled, the legacy client process can transmit application protocol requests without Speaks For Layer Protocol encapsulation.

5.4.1 Finding a Service's Delegation Agent

The Delegation Agent for each Delegation Framework-enabled legacy service running on a given host is expected to listen on a distinct port in order to prevent the Delegation Agents of different services from colliding on a particular port. To avoid port collisions in the Delegation Framework prototype, a legacy service's port number maps to an arbitrary, but well-known, port number on which the service's Delegation Agent listens. For instance, if the legacy service is a web server listening on port 80, its Delegation Agent listens for Delegation Protocol connections on port 3489.

This approach is suitable for the Delegation Framework prototype, but the choice of ports in the prototype's convention may collide with services offered on real world hosts. Instead of relying on a convention that may still result in port conflicts, real world deployments could use a service similar to RPC's `portmapper`. This `portmapper`-like service would listen on a single well-known port, and would provide mappings between the ports of Delegation Framework-enabled services and the ports on which those services' Delegation Agents listen.

5.4.2 Encrypted Channel Setup

Given that the service is Delegation Framework-enabled, the client Delegation Agent opens a TCP connection to the service Delegation Agent via which the two Delegation

Agents will communicate using the Delegation Protocol. Immediately following the channel setup, the two Delegation Agents negotiate a shared secret key and encrypt all connection data with it. The specific key negotiation protocol and encryption algorithm used are not important provided that they produce an encrypted channel that is resilient to eavesdropping. Though a deployed Delegation Framework implementation should use an encrypted channel, the current prototype of the Delegation Framework does not implement an encryption layer. Producing an encrypted channel is a well understood problem and an implementation is non-essential to validating the Delegation Framework design.

5.4.3 Mutual Authentication

After the two Delegation Agents have established an encrypted channel, they must mutually authenticate one another. The mutual authentication protocol in which they engage is a variation of the Needham-Schroeder public key mutual authentication protocol [20] that is resilient to the Lowe attack [19] and has been proven to be cryptographically sound [3]. The Needham-Schroeder protocol is structured such that, following the successful completion of the protocol, each party is convinced that its peer holds the public key it claims to hold. In the Delegation Framework variation of the protocol, that key is also mapped to the peer's identity. This mapping is provided by an identity certificate or a delegation delegation certificate with a non-empty `subject_key` field (see Section 4.1). Finally, the peers also agree on a *session nonce* that will be used later to authenticate Speaks For Layer Protocol messages.

The Delegation Protocol messages that compose this protocol are the Hello and HelloResponse messages. During mutual authentication, each Delegation Agent sends its peer a Hello message. With the Hello message, a Delegation Agent transmits its identity certificate C_{id} or, if the principal is using a temporary key that does not have an identity certificate, the identity certificate C_{id} of the principal on whose behalf it is working. If the Delegation Agent is using a temporary key, It also transmits a delegation certificate C_{temp} that contains its temporary key and the authorization policy defining the authorities delegated to the temporary key. The Hello messages

also contains a randomly-generated nonce N that the sending Delegation Agent signs using it private key (whether its a long-term or temporary key), which is essential for the Needham-Schroeder protocol. Protocol diagrams denote the Hello messages as follows:

- Hello $C_{id}, C_{temp}, \{N\}_{PK}$

In order to correctly perform mutual authentication under Needham-Schroeder, each peer must respond to the Hello message it receives with a HelloResponse message containing the nonce that it received from its peer, this time signing the nonce with its own private key. Protocol diagrams denote the HelloResponse message as follows:

- HelloResp $\{N\}_{PK}$

Figure 5-3 depicts the Delegation Protocol channel setup dialogue. All together, the steps of this dialogue are as follows:

1. The client Delegation Agent transmits a Hello message containing the identity certificate $C_{id,cli}$ containing the permanent identity-to-key mapping for the principal on whose behalf its temporary principal operates, and the delegation certificate $C_{temp,cli}$ which authorizes the temporary principal and provides its identity-to-key mapping. The client includes the randomly generated nonce N_1 in the message signed in this case with a temporary private key $PK_{temp,cli}$. The service Delegation Agent verifies the signatures on both certificates and the nonce, and closes the Delegation Protocol connection if any signature is invalid.

2. The service Delegation Agent transmits a Hello message containing the identity certificate $C_{id,srv}$ containing the permanent identity-to-key mapping for the principal on whose behalf its temporary principal operates, and the delegation certificate $C_{temp,srv}$ which authorizes the temporary principal and provides its identity-to-key mapping. The service includes the randomly generated nonce N_2 in the message signed in this case with a temporary private key $PK_{temp,srv}$. The client Delegation Agent verifies the signatures on both certificates and the nonce, and closes the Delegation Protocol connection if any signature is invalid.

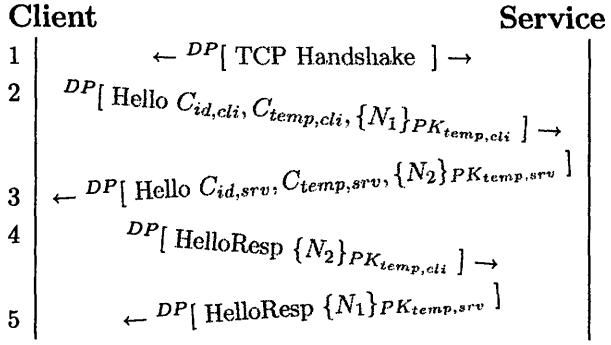

Figure 5-3: The Delegation Protocol channel setup dialogue

3. After receiving the service Delegation Agent's Hello message in step 2, the client Delegation Agent extracts the nonce N_2 and signs the nonce using its temporary private key $PK_{temp,cli}$, marshalls the signed nonce into a HelloResponse message, and transmits the message to the service Delegation Agent. Upon receiving the HelloResponse message, the service Delegation agent extracts the signed nonce $\{N_2\}_{PK_{temp,cli}}$ and uses the temporary public key contained in $C_{temp,cli}$ to verify the signature on the encrypted nonce. If the signature is valid, the service considers the client to be authenticated. If the signature is invalid, the service Delegation Agent closes the Delegation Protocol connection.

4. After receiving the client Delegation Agent's Hello message in step 1, the service Delegation Agent extracts the nonce N_1 and signs the nonce using its temporary private key $PK_{temp,srv}$, marshalls the signed nonce into a HelloResponse message, and transmits the message to the client Delegation Agent. Upon receiving the HelloResponse message, the client Delegation agent extracts the signed nonce $\{N_1\}_{PK_{temp,srv}}$ and uses the temporary public key contained in $C_{temp,srv}$ to verify the signature on the encrypted nonce. If the signature is valid, the client Delegation Agent considers the service to be authenticated. If the signature is invalid, the client Delegation Agent closes the Delegation Protocol connection.

After the client and service have authenticated one another, each peer remembers the nonce N_1 sent by the service. N_2 becomes the *session nonce*, which is used later to authenticate SpeaksFor messages.

The messages in steps 1 and 2 may be exchanged in any order, but will always precede the messages in steps 3 and 4. Steps 3 and 4 may also be exchanged in any order, but will always follow the messages in steps 1 and 2. Once the channel setup dialogue is completed, each Delegation Agent is prepared to exchange Delegation Protocol messages with one another as described through the remainder of this chapter. Network protocol diagrams throughout the remainder of this chapter will denote an instance of the dialogue shown in Figure 5-3 as follows:

- $\leftarrow {}^{DP}[\text{ Setup }] \rightarrow$

Following the Delegation Protocol connection setup, the client will complete the intended application protocol connection setup, performing whatever session negotiation that the application protocol requires of the legacy client and service processes. This may amount to nothing more than a TCP handshake, or it may involve the authentication of a client by a standard mechanism such as a password or challenge-response protocol and the negotiation of a number of session parameters. Whatever the exchange entails, the data is exchanged is encapsulated in the Speaks For Layer Protocol.

5.5 The Speaks For Layer

The Speaks For Layer's role is to provide mappings between identities and application protocol flows. All application protocol requests must be mapped to an identity so that the requesting client's authorities can be properly to application protocol requests. These mappings state that an application protocol flow SpeaksFor a given identity. Authorities provides by delegation credentials for the given identity subsequently apply to the requests transmitted via the application protocol flow. Without proper identity-to-flow mappings, it is impossible for a service to properly authorize any application protocol requests.

The Speaks For Layer Protocol (SFL) is a simple protocol used by a client to mark its application protocol flows with information identifying on whose behalf application protocol messages speak. SFL is not a modification to application protocols

themselves, but an additional protocol stack layer inserted above TCP and an encapsulating encryption layer and below the application protocol as depicted in Figure 5-1. These tags are only included by clients; a service does encapsulate the data it transmits to clients within SFL messages; clients in the Delegation Framework do not attempt to authorize the responses that they receive from services and therefore do not require that each application protocol response by tagged with SpeaksFor information. Thus, the Speaks For Layer Protocol only applies to the client-transmitted half of the flow. Services transmit application protocol data as it normally would without any SFL encapsulation.

Figure 5-4 depicts a series of messages transmitted via the Speaks For Layer Protocol. With message 1, a SpeaksFor message, the client informs the service that subsequent application protocol requests speak for the principal "Alice **for** Charles". The client includes the session nonce N_s. The legacy service process receiving this SpeaksFor message reports its contents to its local Delegation Agent which verifies that N_s is the correct session nonce. If the session nonce is incorrect, the service will deny any subsequent requests from the client. The client then transmits an arbitrary application protocol request in message 2 within an SFL "Data" message, which identifies the request as application protocol data. If the service performs an authorization check following the reception of message 2, it will evaluate the request's authority with respect to the principal identified in the most recent SpeaksFor message, "Alice **for** Charles". Message 3 is the service's application protocol response, transmitted normally without being encapsulated in the Speaks For Layer Protocol. Message 4 through 6 repeat the protocol, but this time around the client informs the service that it speaks for the principal "Alice **for** David". Again, the service Delegation Agent will verify that N_s is the correct session nonce, and any subsequent authorization checks that occur following message 5 are evaluated with respect to the newly specified principal "Alice **for** David".

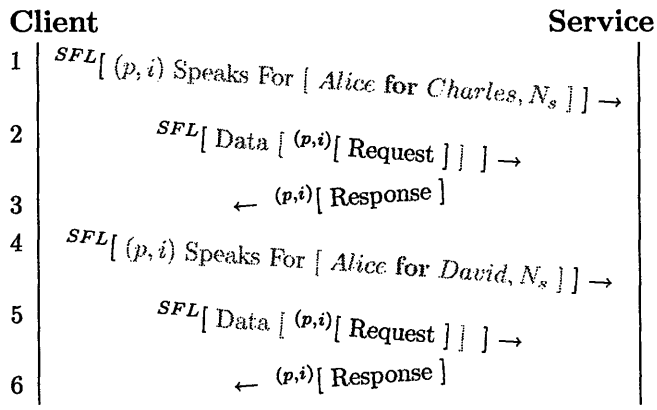

Figure 5-4: A Speaks For Layer session

5.5.1 Authenticating SpeaksFor Messages

A SpeaksFor message contains a session nonce because the SpeaksFor message cannot be authenticated exclusively based on knowledge of the host from which it were received. The Delegation Framework allows multiple client principals (e.g., a mail client and a web browser) to exist on a single host, each having separate identities and having obtained different delegation credentials from their respective user(s). If multiple client principals on a single host are communicating with a single legacy service, the legacy service cannot determine whether an application protocol flow actually speaks for a given client principal using only the identity contained in a SpeaksFor message. If they did, a malicious client could claim one of its flows speaks for another client principal on the same host.

Therefore, each SpeaksFor message contains the session nonce, N_s, negotiated between a given client and service Delegation Agent during the Delegation Protocol channel setup. The client Delegation Agent can pass the appropriate session nonce for a given service to a legacy client process when the legacy client process reports that it is initiating a new application protocol session. With the correct session nonce in hand, the legacy client process can include it in subsequent SpeaksFor messages sent via the application protocol flow. The receiving legacy service process passes the SpeaksFor message to its Delegation Agent so that subsequent authorization checks can be made with respect to the correct client identity, giving the Delegation Agent

the opportunity to check the session nonce as well.

5.6 Protocols in Two-level Transactions

When a legacy service process receives an application protocol request from a client, it performs an authorization check by making one or more reference monitor calls into its Delegation Agent, passing the Delegation Agent an authorization policy definition describing the authority that the client issuing the request must hold in order to have the request honored. If the service Delegation Agent does not know whether the identity for whom the request speaks holds the required authority, the Delegation Agent sends the a Delegation Protocol RequireAuthority message to the client's Delegation Agent. The RequireAuthority message specifies the authorization policy A defining the authority required, and the identity ID for whom the application protocol message in question speaks (the identity that must be proven to hold the authority). The RequireAuthority message also contains a nonce N. A later message includes this nonce to identify itself as response to the RequireAuthority message. Network protocol diagrams denote the RequireAuthority message as follows:

- ReqAuth A, ID, N

Upon receiving a RequireAuthority message, the client's Delegation Agent checks its local knowledge of the authority it holds. If it holds the authority set A and can present a corresponding credential providing evidence that it holds the authority, it presents this credential to the service Delegation Agent using a DemonstrateAuthority message. A DemonstrateAuthority message contains a delegation credential C providing evidence that ID holds the authority set A required by the service. The DemonstrateAuthority message contains the value of the nonce N included in the RequireAuthority message to which it responds. The DemonstrateAuthority message need not contain explicit references to A or ID as this will be apparent to the service Delegation Agent upon examining N. The credential C, if valid, will also contain information about ID and A. Upon receiving the Demonstrate Authority message, the

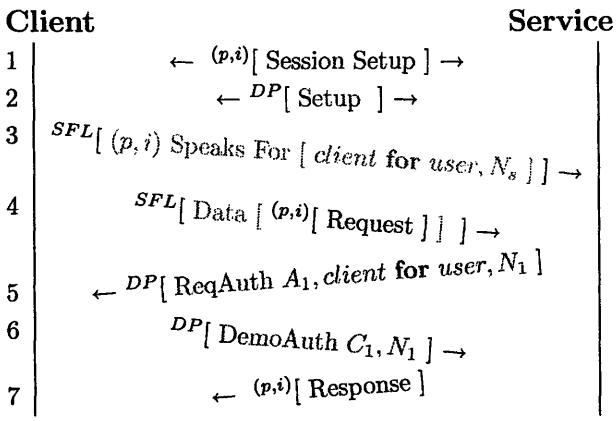

Figure 5-5: A client-service dialogue in the Delegation Framework.

service Delegation Agent verifies the credential according to the algorithm described in Section 4.4. If the credential is valid and does in fact show that ID holds the authority set A, the service Delegation Agent informs the legacy service process that the request is authorized. Network protocol diagrams denote the DemonstrateAuthority message as follows:

- DemoAuth C, N

Figure 5-5 depicts the RequireAuthority and DemonstrateAuthority as they fit into a dialogue between a client and a service. The steps that take place in this dialogue are as follows:

1. The legacy client process is about to initiate a TCP connection to legacy service process over which they will communicate using an application protocol. Before allowing the connection to be initiated, legacy client process informs its Delegation Agent of its intentions to open an application protocol connection. If a Delegation Protocol channel has not already been established between the client and service, the client's Delegation Agent establishes a TCP connection to the service's Delegation Agent over which they will communicate via the Delegation Protocol. The Delegation Agents engage in a setup dialogue (see Section 5.4).

2. The legacy client process initiates the application protocol TCP connection.

3. Immediately following the completion of the application protocol connection's TCP handshake, the legacy client process informs the service of the principal for which the application protocol flow speaks. The legacy service process receives the SpeaksFor message and reports that the identity of the principal for whom the application protocol flow speaks to its local Delegation Agent.

4. The legacy client process sends a Speaks For Layer-encapsulated application protocol request.

5. Upon receiving the application protocol request, the legacy service process makes a reference monitor call into its Delegation Agent specifying the authority needed that the client must demonstrate in order to its request serviced. The Delegation Agent finds that it has no evidence that the principal for whom the application protocol request speaks (as identified by the SpeaksFor message received in step 3) possesses the required authority, and consequently sends the client's Delegation Agent a RequireAuthority message specifying the authority set A_1 that the principal must demonstrate.

6. Upon receiving the RequireAuthority message, the client examines the delegation credentials that it holds to determine whether any of them provide him with the authority set A_1. If any of the credentials the client holds do provide the client principal with A_1, the principal will marshall that credential C_1 into a DemonstrateAuthority message and send it to the service Delegation Agent. Otherwise, the client's Delegation Agent will send a request to the user's Delegation Agent asking whether to delegate it the required authority. The dialogue with the user's Delegation Agent is described in Section 5.7. If the user delegates the authority set A_1 to the client, it provides the client Delegation Agent the credential C_1. The client Delegation Agent marshalls C_1 into a DemonstrateAuthority message and sends the message to the service Delegation Agent.

7. Upon receiving the DemonstrateAuthority message, the service's Delegation

Agent verifies the credential C_1. If C_1 is valid, it proves that the The Delegation Agent responds to the service software's reference monitor call, indicating whether the the client holds the required authority (i.e., whether the credential C_1 is valid and the set of authorities A_2 proven to be held by the client contains A_1 as a subset). Given that the reference monitor call returns affirmatively, the service software handles the application protocol message as it would normally prior to the introduction of the Delegation Framework.

5.6.1 When a Client is Unable to Demonstrate Authority

If the client does not hold and cannot obtain delegation credentials from the user providing the client with authority set A_1, the client responds with a NoAuthority message. This message appears in the dialogue depicted in Figure 5-5 at step 6. The NoAuthority message contains only the nonce N indicating the RequireAuthority message to which the NoAuthority message is being sent in response:

- NoAuth N

The service receives the NoAuthority message, checks the nonce and responds to the legacy service process's corresponding reference monitor call indicating the client principal's lack of authority. The application protocol response sent by the legacy service process in step 7 indicates a failure due to lack of client authority.

5.7 Requesting Credentials from the User

If a legacy client program requires some authority that it does not hold, it must have a mechanism by which it may request the required authority from its user. Between steps 6 and 7 in the dialogue depicted in Figure 5-5, such a request may occur. The client Delegation Agent engages in a dialogue with the Delegation Agent of the client's user, asking the user to delegate the client some authority. This dialogue looks similar to the RequireAuthority-DemonstrateAuthority exchange, expect now the client requests a delegation of authority rather than a demonstration. The client Delegation

Agent sends a RequireDelegation message requesting that the user Delegation Agent with identity ID delegate it authority set A. Like the RequireAuthority message, the RequireDelegation message includes a nonce N that a user Delegation Agent message will include to indicate it is a response to the RequireDelegation message. Network protocol diagrams denote the RequireDelegation message as follows:

- ReqDel A, ID, N

The Delegation Protocol channel through which this dialogue occurs is initiated by the user Delegation Agent prior to the user actually interacting with a given Delegation Framework-enabled client program. Just like a client initiating an exchange with a service, the user Delegation Agent initiated a Delegation Protocol channel and engaged in the Delegation Protocol channel setup dialogue described in Section 5.4. This Delegation Protocol channel is a available for the client Delegation Agent to transmit the RequireDelegation message to the user Delegation Agent.

Upon receiving the RequireDelegation, the user Delegation Agent determines whether the user holds the requested authority and, if the user does, will prompt the user asking whether to delegate the requested authority. If the user refuses, the user Delegation Agent sends a NoAuthority message containing N back to the client Delegation Agent. if the user wishes to the delegate the authority, the user Delegation Agent generates a delegation certificate providing the client principal with the required authority set and constructs a credential from the delegation certificate. The user Delegation Agent marshalls this credential C and the nonce N from the RequireDelegation message to which the Delegation Agent is responding into a Delegate message and sends the message to the client Delegation Agent. Network protocol diagrams denote the Delegate message as follows:

- Delegate C, N

Figure 5-6 depicts the simple exchange between the user and client Delegation Agents that would occur between steps 6 and 7 of the dialogue depicted in Figure 5-5 if the client did not already hold delegation credentials proving its authority.

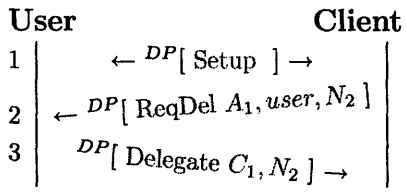

Figure 5-6: A user-client dialogue in the Delegation Framework

Upon completion of the dialogue, the client Delegation Agent holds the delegation credential P_1 proving its authority.

5.8 Protocols in Three-level Transactions

Clients are not the only principals that may require authority to be delegated to them, and users are not the only principals who may delegate their authority. If a client employs a deputy to access an underlying service on its behalf (e.g., a web browser employs a deputy web service to access an underlying database service), it may be necessary for the client to delegate some authority that it has received from its user to a deputy. Doing this does not require any additional messages that the Delegation Protocol does not already provide; the RequireDelegation, Delegate, and NoAuthority messages can be reused in a dialogue between the client and deputy.

The dialogue depicted in Figure 5-7 is an expansion around the client-service dialogue depicts in Figure 5-5. In this dialogue, the deputy principal takes up the role that the client had in Figure 5-5.

The first five steps of the dialogue in Figure 5-7 are identical to the first five steps of the dialogue which takes place in Figure 5-5. Instead of performing a reference monitor call after step 5, the legacy deputy process initiates a client application protocol flow of its own in order to service the client's request. In order to initiate the application protocol flow, the deputy takes the role of the client in Figure 5-5, and the same dialogue occurs between the deputy and service. After receiving the RequireAuthority message in step 9, the deputy sends a RequireDelegation message to the client if it does not hold the authority required to make the request sent in step 8.

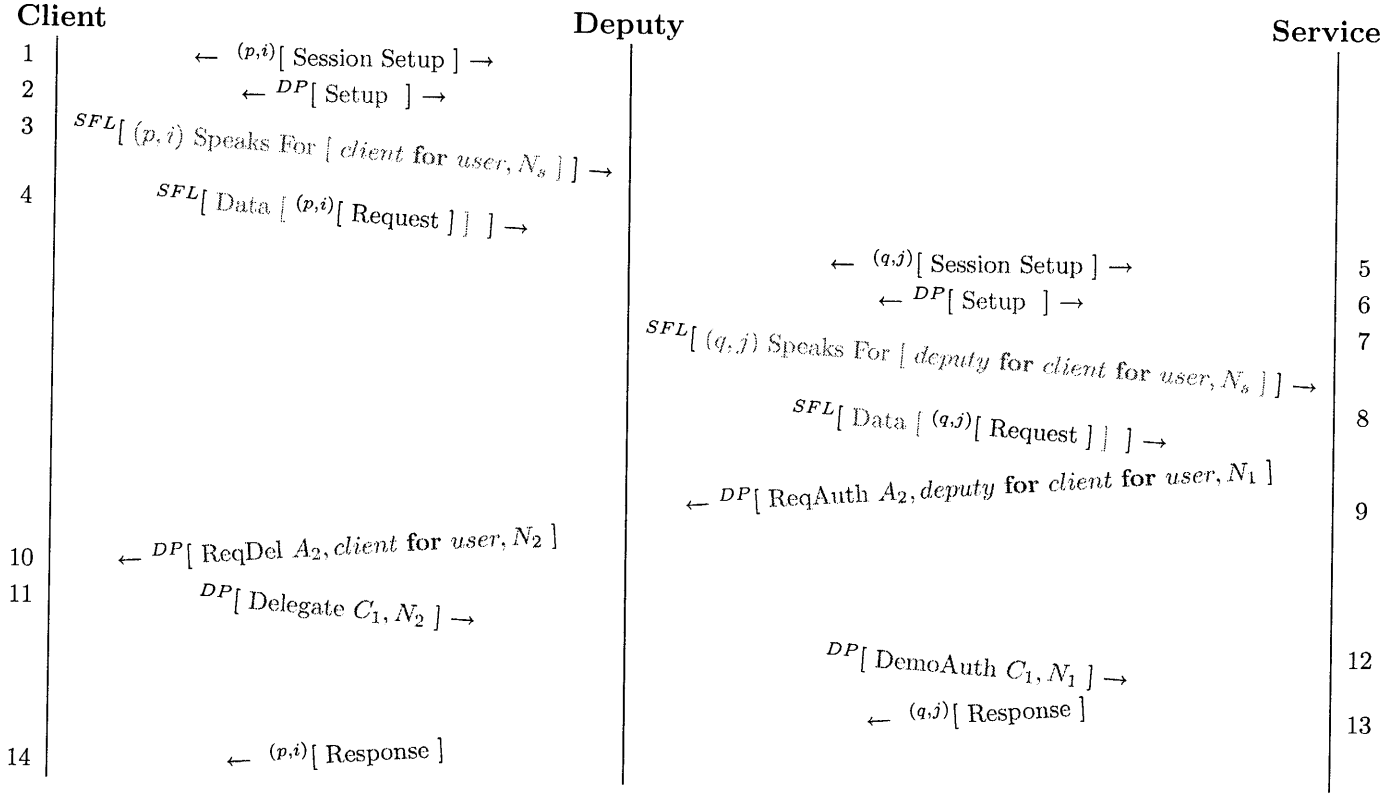

Figure 5-7: A client-deputy-service dialogue in the Delegation Framework.

If the client does not hold the authority it has been asked to delegate, the client handles the RequireDelegation in step 10 by engaging a dialogue such as that in Figure 5-6 before it can send the Delegate message in step 11. Once the client Delegation Agent holds the authority it must delegate to the deputy, the client Delegation Agent generates a delegation certificate and crafts a delegation credential for the deputy using this delegation certificate and the delegation credential proving the client's authority. This is done by appending the new delegation certificate to the end of his own delegation credential. The client Delegation Agent sends a Delegate message containing this new delegation credential to the deputy in step 11. Like step 6 in Figure 5-5, the deputy passes this delegation credential to the service in step 12 to prove that it holds the needed authority. In step 13, the service Delegation Agent returns an affirmative response to the legacy service process's reference monitor call following step 12 that causes the legacy service process to handle the deputy's request in step 13. Upon receiving a response to its application protocol request in step 13, the deputy is now able to handle the client's request. The deputy sends its application protocol response in step 14.

5.9 A Priori Delegation and Demonstration

If a client knows what authority it must delegate to a deputy or demonstrate to a service before initiating a transaction, it is possible to reduce the request's latency. Figure 5-8 depicts a dialogue similar to that which occurs in Figure 5-7, in which the client and deputy delegate and demonstrate their authority prior to sending their application protocol requests. If a client or deputy had a mechanism for remembering which authority was needed to make a given application protocol request, this could significantly reduces the delay incurred in transaction by avoiding the RequireDelegate and RequireDemonstrate messages altogether. The service software's reference monitor call into the Delegation Agent following step 12 returns affirmatively instead of causing the Delegation Agent to send a RequireDemonstrate message to the deputy. The deputy need not send a RequireDelegate message since it received the needed

authority during step 4 and demonstrated that authority in step 9 before sending its application protocol request in step 10.

The Delegation Framework prototype does not presently implement such a mechanism. In practice, it is difficult to know what authority a given application protocol request will require. An examination of methods for ascertaining this information in support of a priori delegation and demonstration is beyond the scope of this thesis.

5.10 Protocols in n-level Transactions

As an extension to the protocols depicted in Figures 5-7 and 5-8, the Delegation Framework supports an arbitrary number of deputies in between the leftmost client and the rightmost service in a given transaction. Passing of credentials from the leftmost client to the rightmost deputy would involve additional exchanges of RequireDelegation and Delegation messages.

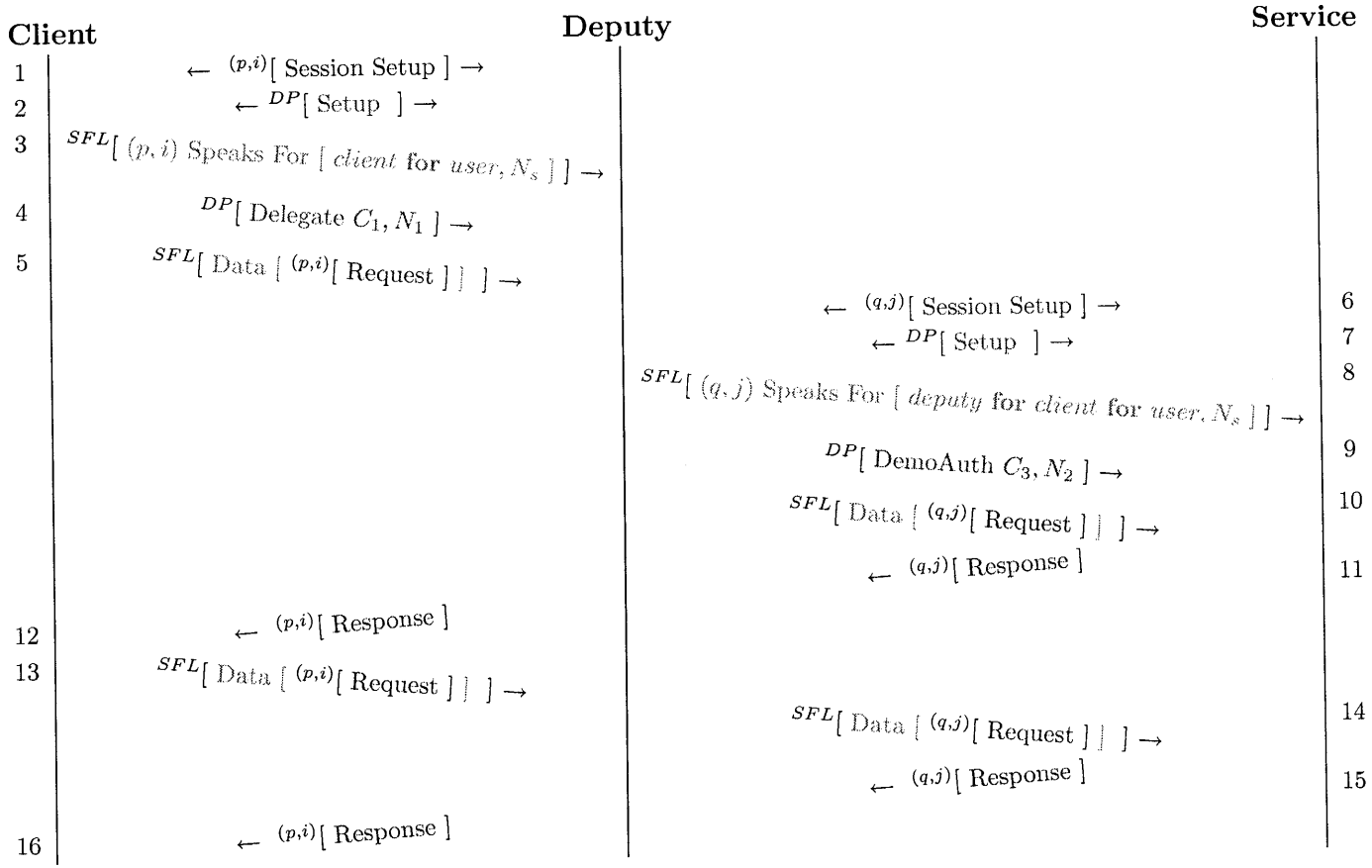

Figure 5-8: A client-deputy-service dialogue in the Delegation Framework with a priori delegation.

Chapter 6

The Delegation Agent

The Delegation Agent performs a couple of different functions depending what type of software that it is assisting. When assisting a user or client principal, it acts as a credential manager demonstrating and delegating authority in much the same way that OpenSSH's `ssh-agent` manages credentials for `ssh` clients [22]. When assisting a service process, the Delegation Agent acts as a reference monitor into which a service processes make calls to check whether a client holds some required authority. When assisting a deputy (i.e., a service that may act as a client when communicating with an underlying service on behalf of its own clients), the Delegation Agent performs both of these roles.

One instance of the Delegation Agent is run for each user and each client or service in a Delegation Framework-enabled network application. A process communicates with its Delegation Agent when the process:

1. Initiates a new outgoing application protocol flow (clients and deputies only)
2. Changes an identity-to-application protocol flow mapping (deputies only)
3. Closes an application protocol flow
4. Receives a SpeaksFor message from a client (services only)
5. Receives a request that requires authorization (services only)

Much of the interaction between a client or service process and its Delegation Agent takes place so that Speaks For Layer Protocol messages properly identify the principal on whose behalf each application protocol speaks. A Delegation Agent must be capable of handling all of these interactions with the legacy client or service on whose behalf it operates while the client or service is interacting with any number of peers in a Delegation Framework-enabled system. I will later describe how legacy processes can be made to interact with the Delegation Agent automatically in Chapter 7.

6.1 Delegation Agent architecture

The Delegation Agent is an event-driven program implemented using a `select()`-based event loop. It is activated by Delegation Framework Library calls from from the local process it assists, and other Delegation Agents. Each type of event corresponds to a different type of message received by the Delegation Agent, and each message has an associated handling routine to which the Delegation Agent dispatches in order to properly handle the message.

The Delegation Framework specifies that each instance of a service or client on a given host should have its own identity and, correspondingly, its own Delegation Agent. However, due to the multiple-process and multi-threaded architecture of many client and service processes, a single Delegation Framework principal may have multiple processes with which it must communicate. Each of these process with which a Delegation Agent communicates maintains a separate UNIX domain socket with the Delegation Agent.

Each Delegation Agent maintains an instance of each of the data structures listed in 6.1. These data structures are introduced at appropriate times throughout the discussion of this chapter.

Structure	Key	Value
peer_da_sock_map	ID_{peer}	peer_da_fd
peer_id_map	peer_da_fd	ID_{peer}
peer_nonce_map	ID_{peer}	$N_{s,peer}$
flow_parent_map	$flow_{child}$	$flow_{parent}$
client_speaks_for_map	$flow$	ID_{client}
pending_resp_map	N_{req}	$(resp\_type, resp\_fd, A_{resp}, ID_{resp}, N_{resp})$
known_authority_map	ID	$\{P_0, P_1, ..., P_n\}$

Table 6.1: The Delegation Agent data structures

6.2 Handling Events

6.2.1 Handling a Legacy Client's New Connection Report

Section 5.6 discusses the steps taken between a client and service in a two-level transaction. Immediately after the legacy client process initiates a new application protocol channel in the first step of the transaction, the legacy client calls into its Delegation Agent to report the new application protocol TCP flow's *flow identifier*, a 4-tuple identifier comprised of the four pieces of information that uniquely the flow by its endpoints: (client IP, client port, service IP, service port).

Maintaining Peer Authentication Knowledge

If the client's Delegation Agent has not already opened a Delegation Protocol channel with the Delegation Agent of the service to which the client is connecting, it initiates a new Delegation Protocol connection to the service's Delegation Agent.

During the Delegation Protocol channel setup dialogue, each of the Delegation Agents learns the identity and public key of its peer, along with a session nonce which each must remember for the duration of the Delegation Protocol session. Each Delegation Agent inserts an entries into three separate data structures containing information about the session nonce, and identity and public key of the peer Delegation Agent. A Delegation Agent inserts an entry in the peer_nonce_map data structure, mapping the peer name to the session nonce of the Delegation Protocol session between the two peers. The Delegation Agent also inserts an entry in the

`peer_da_sock_map` data structure, mapping the name of the peer identity to the TCP socket descriptor of the Delegation Protocol channel that connects the local Delegation Agent to the peer Delegation Agent. Finally, the Delegation Agent inserts an entry in the `peer_id_map` data structure, mapping the Delegation Protocol socket descriptor to a pairing of the identity and key of the peer Delegation Agent. These entries become useful later for identifying the proper descriptor of the socket via which a particular Delegation Protocol message should be sent, and for determining the identity of Delegation Protocol peer who sent a message that was received via a particular socket.

Determining the Correct "Speaks For" Identity

Following the Delegation Protocol channel setup, the client must sends a message through the application protocol TCP flow that it just setup. After completing the mutual authentication protocol, the client's Delegation Agent determines the proper identity for which the application protocol request presently speaks, and sends the identity and the session nonce to the legacy client as the response to the new connection report that caused the Delegation Agent to setup the Delegation Protocol session. In a two-level transaction (such as that discussed in Section 5.6), the identity for which the application protocol is of the form "*client* **for** *user*" where *client* is the name of the principal on whose behalf the Delegation Agent is working, and *user* is the name of the user principal currently using the *client* principal.

The Correct "Speaks For" Identity in a Three-Level Transaction

However, if the new application protocol connection is reported by a deputy in a three-level transaction (see Section 5.8), the correct identity is that of the deputy speaking on behalf of its client (who is in turn speaking on a user's behalf): "*deputy* **for** *client* **for** *user*". Correctly reporting the deputy's identity requires knowledge of the principal on whose behalf the new application protocol is initiated. This cannot be determined with the 4-tuple of the newly setup application protocol flow alone; the legacy deputy's report of the application protocol flow must also contain the flow

identifier of the flow from which it read the application protocol request on whose behalf the new flow is being setup.

Using this additional flow identifier, the Delegation Agent can lookup the identity of the deputy's "*client* **for** *user*" principal in a data structure called the `client_speaks_for_map`, which maps a serivce's client application protocol flows to the identities for which they speak. A service (including deputy services) populates its Delegation Agent's `client_speaks_for_map` by reporting when it has received a message to the Delegation Agent. In the case of a three-level transaction, the entry, for an application protocol flow connecting a client to the deputy initiating the new application protocol flow, will contain a principal identity of the form "*client* **for** *user*".

Remembering a Child Flow's Parent

Later, the deputy must be able to determine to whom it should send a RequireDelegation message upon receiving a RequireDemonstration or RequireDelegation message from a service. To support this, a deputy's Delegation Agent must maintain knowledge of the mapping between each outgoing application protocol flow that the deputy has initiated on behalf of a client, the *child flow*, and the incoming client flow on whose behalf the child flow was initiated, the *parent flow*. The Delegation Agent maps child flows to parent flows using the `flow_parent_map` data structure.

Returning to the Legacy Client or Deputy

Now that the client or deputy Delegation Agent has determined the identity of the principal for which the new application protocol flow speaks, it returns the identity to the legacy client or deputy process for inclusion in a message along with the corresponding session nonce determined from the `peer_nonce_map` data structure. The legacy client or deputy process sends this information via a message (see Section 5.5).

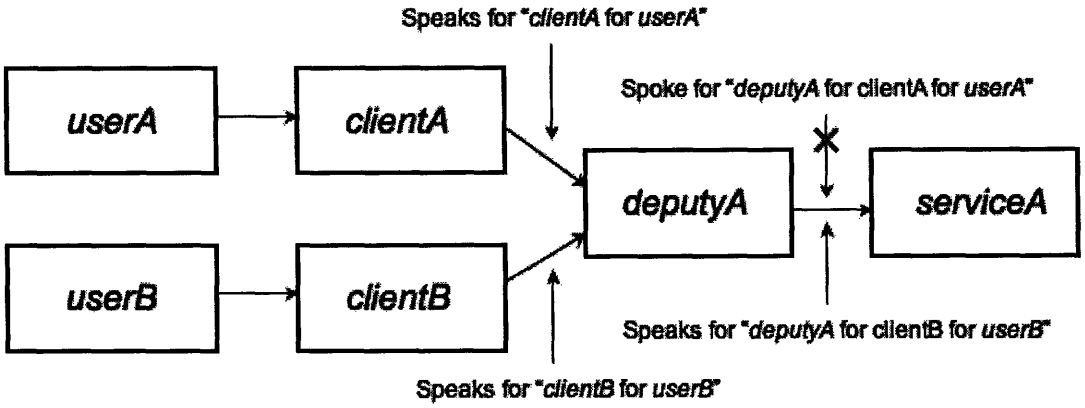

Figure 6-1: A deputy switches from working on behalf of one client to working on behalf of another

6.2.2 Handling a New Parent-Child Relationship

When a deputy legacy process is working on behalf of multiple clients, it is possible that it may send client requests to an underlying service for multiple clients via a single application protocol flow. The identity of the application protocol flow to the underlying service must change when a deputy switches from working on behalf of one client to working on behalf of another client.

Figure 6-1 illustrates a network application with this problem. *deputyA* switches from working on behalf of *clientA* to working on behalf of *clientB*. The application protocol flow via which it communicates with *serviceA* much now speak on behalf of the new deputy identity "*deputyA* **for** *clientB* **for** *userB*". In abstract terms, this may appear to be a contrived case, but this sort of configuration is not improbable in real world systems. For example, *deputyA* might be a web front-end interface to an IMAP service *deputyB* that accesses mailbox files on *serviceA* and e-mail directory entries on *serviceB* on a user's behalf.

The deputy process must report to its Delegation Agent when such a change occurs, passing the Delegation Agent the child flow identifier and the identifier of the parent flow on behalf of which the child flow is being used. The Delegation Agent handles this change much as it would a new application protocol flow connection (see Section 6.2.1). The Delegation Agent need not setup a new Delegation Protocol session because one already exists. However, the Delegation Agent must determine

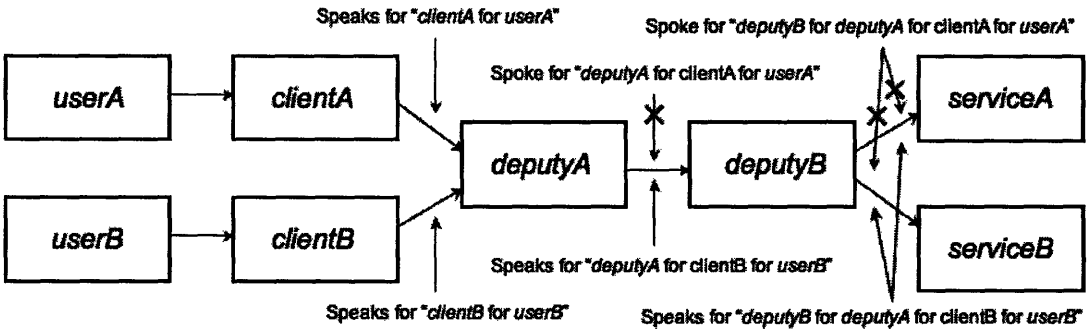

Figure 6-2: A network application requiring the forwarding of messages

the correct identity for which the child flow now speaks by looking up the parent flow identifier in the `client_speaks_for_map` data structure, enter the new child-parent relationship in the `flow_parent_map`, and return the new identity for whom the child flow speaks and the corresponding session nonce from `peer_nonce_map` for inclusion in the legacy deputy process's message.

6.2.3 Handling a "Speaks For" Report

Upon receiving a message, a legacy service reports the message's contents to its Delegation Agent. The Delegation Agent populates its `client_speaks_for_map` data structure using the reports. These reports include the session nonce identity contained in the message, the flow identifier of the application protocol TCP flow from which the message was received. The Delegation Agent looks up the session nonce stored in its `peer_nonce_map` using the identity contained in the message. If the `peer_nonce_map` value does not match that in the message, it rejects the "Speaks For" mapping. This prevents the client from having its application protocol request authorized using any principal's delegation credentials. If the nonce values match, the Delegation Agent inserts an entry mapping the application protocol flow identifier to the identity of the principal for which the application protocol flow speaks into the `client_speaks_for_map`.

Forwarding "Speaks For" Information

If the service receiving a "Speaks For" report is a deputy, there may exist flows which are the children of the flow for which a new "Speaks For" identity has been specified. This means the identities of each child flow have changed as well. For instance, Figure 6-2 illustrates a network application which such a situation might occur. A deputy, *deputyB*, may receive a "Speaks For" report regarding a parent flow, which previously spoke for principal "*deputyA* **for** *clientA* **for** *userA*", declaring that the parent flow now speaks for "*deputyA* **for** *clientB* **for** *userB*". The parent's child flows that spoke for "*deputyB* **for** *deputyA* **for** *clientA* **for** *userA*" now speak for "*deputyB* **for** *deputyA* **for** *clientB* **for** *userB*". The *deputyB* Delegation Agent returns this identity to the deputy legacy process, which must send messages via each of the parent's child application protocol flows[1].

6.2.4 Handling a Reference Monitor Call

When a service process makes a reference monitor call into its Delegation Agent asking whether a given application protocol flow's principal satisfies some authorization policy, the Delegation Agent begins handling the reference monitor call by looking up the identity of the principal for whom the request speaks in the `client_speaks_for_map`. Given this identity, the Delegation Agent looks up the credentials of the identified principal in the `known_authority_map` data structure, which maps a principal's identity to the set of *valid* delegation credentials that the Delegation Agent has verified for the principal. The Delegation Agent examines the principal's credentials to determine whether any of them prove that the principal satisfies the authorization policy required by legacy service process's reference monitor call.

If the principal's `known_authority_map` entry contains a credential satisfying the reference monitor call's authorization policy, the Delegation Agent sends an affirmative response back to the legacy service process. If the principal's `known_authority_map` entry does not contain credentials satisfying the reference monitor call, it sends a Re-

[1]The task of remembering a given flow's children is left to the legacy deputy and implemented with dynamic library interposition techniques (see Chapter 7).

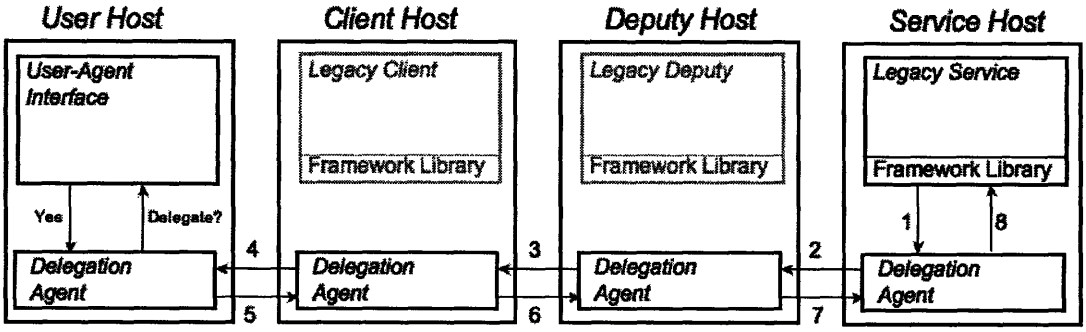

Figure 6-3: The chain of Require* messages and corresponding responses

quireDemonstration to the client's Delegation Agent. The Delegation Agent looks up the socket descriptor of the Delegation Protocol channel connect the Delegation Agent to the appropriate client's Delegation Agent in the `peer_da_sock_map`.

Handling the Require* Chain

Sending a RequireDemonstration message to a deputy can result in the deputy Delegation Agent sending a RequireDelegation message to another client, initiating an arbitrarily-long chain of RequireDelegation messages sent from service to deputy, deputy to deputy, ..., deputy to client, and client to user. Eventually, a RequireDelegation message will reach a user's Delegation Agent who may send a Delegate message back up the chain. Each Delegation Agent must maintain state remembering each outstanding RequireDelegation and RequireDemonstration that it has sent, and the parameters of the response it must send upon receiving a response to its RequireDemonstration or RequireDelegation mesage.

Figure 6-3 gives an illustration of the chain of Require* messages and corresponding chain of Delegate and DemonstrateAuthority responses that must occur when a user decides to delegate the authority that has been requested of him via a RequireDelegation message. The arrows in the figure represent the transmission of following messages (numbered in the order in which they are transmitted) sent between each of the depicted processes:

1. Reference monitor call asking whether "*deputy* **for** *client* **for** *user*" holds A_1

2. ReqAuth A_1, *deputy* **for** *client* **for** *user*, N_1

3. ReqDel A_1, *client* **for** *user*, N_2

4. ReqDel A_1, *user*, N_3[2]

5. Delegate P_1, N_3

6. Delegate P_2, N_2

7. DemoAuth P_3, N_1

8. Affirmative reference monitor call response

From the viewpoint of service Delegation Agent depicted in Figure 6-3, the following series of events must occur:

1. The service Delegation Agent receives a reference monitor call, message 1, from a legacy service process

2. The service Delegation Agent sends a RequireDemonstration, message 2, because it has of whether the requesting principal authority has authority

3. The service Delegation Agent receives a DemonstrateAuthority, message 7, in response to message 2

4. The service Delegation Agent sends a reference monitor call response, message 8, in response to the original message 1

Without some kind of state, the service Delegation Agent will fail to send message 8. Each of the other Delegation Agents in the system will similarly fail to send messages 5 through 7.

[2]Upon receiving the RequireDelegation, the user Delegation Agent prompts the user asking whether to delegate "*client*" the authority set defined by authorization policy A_1.

Maintaining Pending Response State

After sending a RequireDemonstration or RequireDelegation message, a Delegation Agent uses the `pending_response_map` data structure to maintain knowledge of the responses it must send; either a Delegate, DemonstrateAuthority, or reference monitor call response. The `pending_response_map` data structure maps the nonce N_{req} from its RequireDemonstration or RequireDelegation messages to a vector containing the following information about the message that must be sent once a response to the RequireDemonstration or RequireDelegation message with nonce N_{req} is received[3]:

- The message type `resp_type`: a Delegate message, DemonstrateAuthority message, or reference monitor response
- The authorization policy A_{req} required by the RequireDemonstration or RequireDelegation message
- The identity ID of the principal for whom the authority set A_{req} is required
- The socket descriptor `resp_sd` via which the response should be sent[4]
- The nonce N_{resp} that must be included in the response[5]

Handling a RequireDelegation for the User

The chain of RequireDelegation messages stops at the user's Delegation Agent. Instead of checking the `known_authority_map` and possibly sending an additional RequireDelegation message, a user's Delegation Agent prompts the user asking whether

[3]The DemonstrateAuthority or Delegate message sent in response will contain the nonce N_{req} (see Sections 5.6 and 5.7)

[4]A service Delegation Protocol channel descriptor if the response is a Delegate or DemonstrateAuthority message, or the UNIX domain socket of a local legacy process if the response is a reference monitor response.

[5]This may be empty if the response is a reference monitor call response because this is important only for Delegate and DemonstrateAuthority responses.

to delegate the set of authority defined by the authorization policy contained in the RequireDelegation.

In the current Delegation Framework prototype, the Delegation Agent prompts the user with a very simplistic textual interface that might look the following[6]:

```
Would you like to delegate the following authority set defined by
the authorization policy:

web_banking@somebank.com:view_balance,transfer_funds:my_accounts

to the principal ''firefox@foo.somedomain.tld''? [Y/N]
```

If the user responds affirmatively, the Delegation Agent constructs a delegation credential P consisting of a single delegation certificate in which the user delegates the identity displayed in the user prompt to the set of authority defined by the authorization policy listed in the prompt. It sends the Delegate message containing P and the nonce N_{req} contained in the RequireDelegation message via the socket from which the RequireDelegation message was received. If the user responds negatively, the Delegation Agent sends a NoAuthority message containing the nonce N_{req} contained in the RequireDelegation message via the socket from which the RequireDelegation message was received.

6.2.5 Handling the Delegate or DemonstrateAuthority Chain

After prompting the user, the user Delegation Agent sends a Delegate message, initiating a chain of responses to the RequireDelegation and RequireAuthority messages sent earlier. Upon receiving a Delegate or Demonstrate Message, a Delegation Agent verifies the delegation credential C contained in the Delegate or DemosntrateAuthority Message[7]. If the credential is valid, the Delegation Agent inserts an entry mapping

[6]Despite being an important issue for a successfully-deployable Delegation Framework, the development of an effective and usable user interface is a considerably difficult task and is outside the scope of this thesis.

[7]If the message is a DemonstrateAuthority message, the credential C provides evidence that the authority of the client sending the message. When verifying the delegation credential, the service

the subject of the credential to the credential C into its known_authority_map data structure. The known_authority_map entry allows the Delegation agent to handle subsequent RequireAuthority, RequireDelegation, and reference monitor calls with an immediate response instead of initiating a potentially long series of Delegation Protocol messages.

Using Pending Response State

After verifying the credential and storing it in the known_authority_map data structure, the Delegation Agent looks up the nonce N_{req} contained in the Delegate or DemonstrateAuthority message in the pending_response_map to identify the correct outstanding response parameters that must be sent. If an entry exists in the pending_response_map, the Delegation Agent checks whether the authorization policy A_{req} contained in the entry is satisfied by the delegation credential C.

If the policy is not satisfied and resp_type specifies a reference monitor call response, the Delegation Agent send a reference monitor call response via resp_sd indicating that the application protocol request is not authorized. If resp_type specifies a Delegate or DemonstrateAuthority response, a NoAuthority message containing nonce N_{resp} is sent via resp_sd.

If the policy is satisfied and resp_type specifies a reference monitor call response, the Delegation Agent send an affirmative reference monitor call response via resp_fd indicating that the application protocol request is not authorized. If resp_type specifies a DemonstrateAuthority response, the Delegation Agent sends a DemonstrateAuthority message containing the delegation credential C and nonce N_{resp} via resp_fd.

If resp_type specifies a Delegate response, the proper response requires the Delegation Agent to craft a new delegation certificate and update the delegation credential C to prove the delegate's authority. First, if the Delegation Agent uses a

Delegation Agent also verifies that the issuer of the first delegation certificate holds the specified authority in its local access control list (see Section 4.4).

temporary key, it appends the delegation certificate that delegates it the permanent principal's authority to C[8]. Second, the Delegation Agent crafts a new delegation certificate in which it delegates the authority A_{req} to the identity ID values from the `pending_response_map` entry, and appends this certificate to C. Next, the Delegation Agent sends a Delegate message containing the newly-constructed delegation credential C' and nonce N_{resp} via `resp_fd`.

Finally, in all cases, the `pending_resp_map` entry for nonce N_{req} is deleted.

6.2.6 Handling a NoAuthority Message Response

A Delegation Agent also handles a NoAuthority messages by looking up its nonce N_{req} in the `pending_resp_map`. If the `resp_type` value specifies reference monitor call response, the Delegation Agent sends the reference monitor call response indicating that the application protocol flow's principal does not hold the required authority. If the the `resp_type` value specifies a Delegate or DemonstrateAuthority message, the Delegation Agent sends a NoAuthority message containing the nonce N_{resp} via `resp_fd`.

6.2.7 Handling a Connection Shutdown

A legacy client or service process must inform its Delegation Agent when it closes an application protocol flow. This allows the Delegation Agent to clear state that is no longer necessary. The legacy process reports the flow identifier of the flow being closed. The Delegation Agent clears any entries that may exist in the `flow_parent_map` and `client_speaks_for_map` data structures. The Delegation Agent does not return any information to the legacy process in response to the flow shutdown report.

[8]Typically, the delegation certificate contains the authorization policy `*@*:*:*` (see Section 4.3).

Chapter 7

The Delegation Framework Library

Client and service software requires an interface through which it may participate in the Delegation Framework. This interface must enable the legacy software to:

1. Report important application events to its Delegation Agent
2. Superimpose the Speaks For Layer protocol beneath application protocols
3. Determine whether a client holds some authorities

The library functions are divided into three corresponding functional categories:

- Speaks For Layer Protocol implementation
- Socket event reporting
- Reference monitoring

The library is implemented in C and consists of the eight functions summarized in Table 7. The `check_authority()` function provides the reference monitoring interface. The socket event reporting interface is comprised of the `report_socket_*()` functions. The `sfl_*()` functions implement the Speaks For Layer.

Library Call	Return Value	Callers	Automated
`check_authority()`	boolean	service	
`report_socket_setup()`	identity name	client	X
`report_socket_dependency()`	identity name	deputy	X
`report_socket_speaks_for()`	identity name	service	X
`report_socket_shutdown()`	none	all	X
`sfl_write_speaks_for()`	boolean	client	X
`sfl_write_data()`	boolean	client	X
`sfl_read()`	(identity name, int)	service	X

Table 7.1: Summary of the Delegation Framework Library functions

7.1 Automating the Library Calls

An important goal of the Delegation Framework is to increase the ease with which delegation can be integrated into legacy network applications. As is apparent from the topics discussed in Chapters 5 and 6, the Delegation Framework has gone to considerable ends to avoid modifying application protocols and to handle delegation credentials on behalf of legacy programs. However, it remains apparent that a considerable number of modifications must be made to the code of a legacy program before it can participate in the Delegation Framework.

Fortunately, it is possible to perform all or most of the Delegation Framework Library functions automatically for large classes of legacy client and service software using synchronous I/O. Seven of the eight Delegation Framework Library functions can be implemented automatically by a dynamic interposition library that catches various C library routines (indicated by an 'X' in the "automated" column in Table 7). Therefore, a programmer need not manually insert function calls into the legacy software source code. The only function that cannot at present be performed automatically is the `check_authority()` function. This eases integration of the Delegation Framework with legacy clients (which never perform reference monitor calls) and deputies that opt not to perform access control based on delegation credentials. All *end services* (i.e., services that do not act as deputies) protecting privileged resources

must control access based on delegation credentials if the network application is to derive any benefit from the Delegation Framework. Therefore, `check_authority()` calls will have to be included in source code.

7.1.1 Authorization Checks in Deputies

Depending on the characteristics of a given network application into which the Delegation Framework is being integrated, services acting as deputies may also include delegation credential-based authorization checks. For instance, an application may be structured such that sensitive information is stored both on a deputy service and on an underlying service used by the deputy. This may often be the case for some kinds of database-driven web applications (i.e., the web service is the deputy and the database is the underlying service).

In other circumstances, the authority to perform a fine-grained operation at the interface provided by a deputy might require the deputy to be delegated the authority to perform a more coarse-grained operation on an underlying service. For example, the deputy is an IMAP-based mailbox service that accesses mail files on an underlying file service on behalf of users. A user wishes to delegate a spam filtering client program the authority only to read messages in his inbox and mark them as spam. However, the fine-grained authority to mark messages as spam depends on the coarser-grained authority to perform filesystem write operations on the user's inbox file. Authority to perform write operations on the user's inbox file is sufficient to delete messages from the inbox as well as mark them as spam. In this network application, it is not possible to enforce the desired fine-grained authority at the interface of the filesystem service. Therefore, an authorization check must be performed at the interface of the mailbox system to properly prevent the spamfilter doing anything more than marking messages as spam.

7.2 The Speaks For Layer Protocol Interface

Three functions implement the Speaks For Layer Protocol. Clients encapsulate application protocol requests in the Speaks For Layer Protocol so that they are able to intermittently transmit messages tagging the application protocol flow with the name of the identity for whom the application protocol flow speaks. All of the functions provided by the Speaks For Layer interface are made automatically by a dynamic interposition library.

7.2.1 `sfl_write_speaks_for()`

```
int sfl_write_speaks_for(int srv_socket, Identity *id, char *nonce)
```

A process acting as a client calls `sfl_write_speaks_for()` when it:

- Opens a new socket connection to a service
- The identity on behalf of which the socket speaks has changed

The function writes a SpeaksFor message into the socket `srv_socket` informing the recipient service of the identity name (specified by the Identity object pointed to by `id`) for which subsequent requests written to the socket speak[1]. `sfl_write_speaks_for()` is only called on a socket connecting the caller to a legacy service process, and never called on a socket for which the caller behaves as a legacy service.

`sfl_write_speaks_for()` marshalls a SpeaksFor message containing the string representation of the Identity object pointed to by `id`, and the session nonce `nonce` and writes this message into `srv_socket` using the C library `write()` function[2]. `sfl_write_speaks_for()` returns the result of the `write()` function.

[1]The Identity object is a C `struct` implemented in the Delegation Framework. Any naming collision that Identity encounters with another software package is coincidental.

[2]The Delegation Framework prototype uses 8-byte nonces.

7.2.2 sfl_write_data()

```
int sfl_write_data(int srv_socket, char *buf, int size)
```

When writing on a socket on which it is acting as a client, a process calls `sfl_write_data()` on a socket to write application protocol data carried via the Speaks For Layer in place of a C library `write()`. The function marshalls a Speaks For Layer "Data" message containing `size` bytes of data pointed to by `buf`, and writes this message into `srv_socket` using the C library `write()` function. `sfl_write_data()` returns the result of the `write()` function.

7.2.3 sfl_read()

```
int sfl_read(int cli_socket, char *buf, size_t bytes, Identity **id,
             char **nonce)
```

When reading from a socket on which it acts as a service, a process calls `sfl_read()` in place of `read()`. `sfl_read()` reads SpeaksFor and "Data" Speaks For Layer Protocol messages from `cli_socket`. Having no knowledge of whether a SpeaksFor or "Data" message will be received next, the caller passes three return parameters, `buf` that is `bytes` bytes long and return parameters Identity object pointer `id` and nonce byte string pointer `nonce`. `sfl_read()` performs a C library `read()` function on `cli_socket` to receive the next Speaks For Layer message.

If the received message is a SpeaksFor message, the an Identity object is allocated containing the name of the identity contained in the SpeaksFor message. Memory is also allocated for the session nonce contained in the SpeaksFor message and returned via the `nonce` return argument. The memory pointed to by both the `id` and `nonce` must be freed by the caller. It does not modify `buf` and returns 0.

If the received message is a "Data" message, `sfl_read()` fills `buf` with `bytes` worth of data and sets `id` and `nonce` to `NULL`. If the "Data" message carried more

than `bytes` bytes of data, the excess is left in the socket buffer for reading during a subsequent call to `sfl_read()` and internal state is marked to reflect this. Future calls to `sfl_read()` treat the left over data as a new "Data" message and will return the data. If the "Data" message carried less than `bytes` bytes, `sfl_read()` will not attempt to read additional Speaks For Layer messages. `sfl_read()` returns the number of bytes read into memory starting at `buf`.

7.3 The Socket Event Reporting Interface

The four `report_socket_*()` functions comprise the portion of the Delegation Framework Library through which both clients and services communicate with the Delegation Agent. These functions help the Delegation Agent properly assist the calling process via the Delegation Protocol and provide the appropriate contents of Speaks For Layer Protocol messages.

All calls to the functions described in this section can be handled automatically by the dynamic interposition library (see Section 7.5). Therefore, a developer integrating the Delegation Framework into an existing system need not be familiar with these functions beyond understanding their purpose and knowing that they are handled automatically by a dynamic interposition library.

Socket descriptors passed to these calls are marshalled into strings representing unique *flow identifier* of the TCP flow associated with the socket descriptor. The strings are represented as the 4-tuple (client IP address, client TCP port, service IP address, service TCP port).

7.3.1 `report_socket_setup()`

```
Identity *report_socket_setup(int new_socket, int cli_socket,
                              char **nonce)
```

A client or deputy calls `report_socket_setup()` after initiating a new application protocol TCP connection to a legacy service, but before writing any data to the socket. The `new_socket` argument specifies the socket descriptor associated with this newly created TCP flow. If the calling program is a deputy and initiates this new socket on behalf of an application protocol request received from a client socket, it should indicate this by specifying the client socket's descriptor `cli_socket`. If the calling program is not a deputy, it should indicate that it is not creating the new socket on behalf of a request received from another principal by setting the `cli_socket` argument to -1.

`report_socket_setup()` marshalls the identifier of the TCP flow associated with the `new_socket` and, if one is specified, the flow associated with `cli_socket`. The function sends the identifiers via UNIX domain socket to the caller's Delegation Agent. `report_socket_setup()` then receives via the domain socket a response from the Delegation Agent containing the identity name for which the flow associated with `new_socket` speaks and the flow's session nonce, returning the name of the identity in the form of an Identity object pointer and the session nonce by setting the `nonce` return parameter.

The caller must subsequently pass `new_socket` and the returned Identity object and session nonce as the arguments to the `sfl_write_speaks_for()` function (described in Section 7.2.1). Also, the caller must remember the session nonce for any subsequent SpeaksFor messages that may need to be written regarding the flow associated with the socket descriptor `new_socket`.

7.3.2 `report_socket_dependency()`

```
Identity *report_socket_dependency(int srv_socket, int cli_socket)
```

A deputy calls `report_socket_dependency()` to indicate that an application protocol request is about to be written to the socket `srv_socket` on behalf of an appli-

cation protocol request received from the socket `cli_socket`. Unlike in `report_socket_setup()`, `cli_socket` cannot be set to -1.

`report_socket_dependency()` marshalls the TCP flow identifiers associated with `srv_socket` and `cli_socket` into a message and sends the message via UNIX domain socket to the caller's Delegation Agent. `report_socket_dependency()` then receives via the domain socket a response from the Delegation Agent containing the name of the identity for which the flow associated with `srv_socket` speaks, returning the name of the identity in the form of an Identity object pointer.

The caller must subsequently pass `srv_socket` and the returned Identity object pointer as the arguments to the `sfl_write_speaks_for()` function along with the session nonce that was saved following the preceding call to `report_socket_setup()` (see Section 7.2.1).

7.3.3 `report_socket_speaks_for()`

```
Identity *report_socket_speaks_for(int cli_socket, Identity *id,
                                   char *nonce)
```

A deputy or service calls `report_socket_speaks_for()` after having been returned an Identity object pointer and session nonce by the `sfl_read()` function (described in Section 7.2.3) on a client socket. The `cli_socket` argument must be the socket argument passed to the `sfl_read()` call that returned the Identity object.

`report_socket_speaks_for()` marshalls the identifier of the TCP flow associated with `cli_socket`, the Identity object pointed to by `id`, and the session nonce pointed to by `nonce` into a serial message. It sends the message via UNIX domain socket to the caller's Delegation Agent. The function then receives via the domain socket a response from the Delegation Agent containing the name of the identity for which flows initiated on behalf of requests received from `cli_socket` speak. This return value will only be important in the case of a deputy who initiates socket connections

and makes application protocol requests on behalf of its clients.

The deputy must maintain state identifying the child sockets working on each client socket's behalf, and correspondingly call `sfl_write_speaks_for()` once for each one of these sockets, passing the socket descriptor and the Identity object pointer returned by `report_socket_speaks_for()` along with the session nonces that were saved following the preceding calls to `report_socket_setup()` for each of the child sockets.

7.3.4 `report_socket_shutdown()`

```
void report_socket_shutdown(int socket)}
```

A client, deputy, or service calls `report_socket_shutdown()` immediate after it closes a socket. No socket data should be read of written between the call to `report_socket_shutdown()` and the C library `close()` function.

The function generates a string representation of the 4-tuple that uniquely identifies the TCP flow associated with `socket`. It marshalls this string into a message buffer, and sends the message via UNIX domain socket to the caller's Delegation Agent. The Delegation Agent does not send any messages in return, and the function has no return value.

7.4 The Reference Monitoring Interface

The reference monitoring interface consists of a single function, `check_authority()`.

7.4.1 `check_authority()`

```
int check_authority(int cli_socket, char *operation, char  *subject)
```

A service calls `check_authority()` to determine whether the sender of a client request received via the socket `cli_socket` has the authority to request an operation

on a given subject. The strings contained in the `operation` and `subject` argument specific to the authorization policies that the service has granted to its users.

`check_authority()` marshalls the TCP flow identifier associated with `cli_socket`, and an authorization policy definition (see Section 4.3) formed by concatenating the local deputy or service's identity name with the contents of `operation` and `argument`. The function marshalls a message containing the flow and authorization policy definition and sends it to the caller's Delegation Agent. It then receives a response from the Delegation Agent signifying whether the identity for whom the client socket `cli_socket` speaks holds the authorities defined by the authorization policy definition passed to the Delegation Agent. If the principal for whom the client flow speaks holds the authority, `check_authority()` returns 1. if the principal does not, `check_authority()` returns 0.

Responding to a `check_authority()` may require the Delegation Agent to engage in a Delegation Protocol dialogue (see Chapter 5). `check_authority()` will block until the Delegation Agent responds. The duration for which the function blocks depends on the delay characteristics of the network and the promptness with which a user responds to a request for delegation.

7.5 Dynamic Interposition Library

A dynamic interposition library automatically implements most of the Delegation Framework Library functions that a legacy process must perform. The dynamic interposition library intercepts a set of five functions at the library functions. The C library interface was chosen as the point of interception because it is both ubiquitous across UNIX platforms and provides the opportunity to automate the correct calling of most of the Delegation Framework Library functions. The interposition library simplifies the integration process, and reduces the possibility of programming errors

during the process of integrating the Delegation Framework into a legacy program.

7.5.1 Loading the Interposition Library

Each legacy client and service process participating in a Delegation Framework-enabled system must have the dynamic interposition library loaded into its address space. The interposition library is loaded into a target process's memory using the LD_PRELOAD environment variable[3]. LD_PRELOAD instructs the operating system's dynamic linker to load the dynamic library into a program's address space prior to all other dynamic libraries, including the dynamic C library. When the linker begins linking any unresolved symbols in the program's code segment, the linker looks through the dynamic libraries for the symbols in the order in which the libraries were loaded.

7.5.2 Inferring Socket Dependencies

Within a deputy legacy process, the `report_socket_setup()` and `report_socket_dependency()` functions described in Section 7.2 both require the caller to report knowledge of instances when a socket is setup or written to by the deputy on behalf of a request received from a client. A deputy program developed from the ground up could handle this requirement straightforwardly, but understanding existing code and modifying it manually will often be a complex and error-prone task. If the interposition library could handle this automatically for all types of deputy programs, it would make integrating the Delegation Framework significantly easier.

While not a complete model of all deputy behavior, the Delegation Framework interposition library employs a simple model that correctly infers socket dependencies

[3]The appropriate loading mechanism may differ on some UNIX platforms and consequently the prototype implementation of the library may not be compatible with all UNIX platforms.

for a class of deputy programs:

A deputy process sends an outgoing application protocol request (i.e., writes to a socket that it initiated) on behalf of the client from which it has most recently received an application protocol request (i.e., reads data from a socket open by a `listen()` *function).*

This model works correctly for the class of deputy programs in which one or more processes each handle requests from one client at a time using synchronous I/O.

The Apache web server is a notable example of a deputy program from this class. Apache assigns each incoming client connection to one of a pool handling processes. These handling processes read client requests and, when authorized, perform whatever handling is necessary to fulfill them. Dynamic web content is often database-driven, requiring the handling process to subsequently write requests to a database service socket. Because an Apache process handles at most one client at a time, it is straightforward to determine the client on whose behalf any writes to the database service socket were performed.

7.5.3 The Interposition Functions

This section describes a set of five functions that we insert into the C library of legacy applications to replace five existing C library functions. The goal of these functions is to properly communicate via the Speaks For Layer Protocol, and to communicate with the Delegation agent when necessary. The five re-implemented C library functions are:

- `accept()`
- `connect()`
- `read()`
- `write()`
- `close()`

Structure	Key	Value
client_sockets	n/a	cli_sd
service_sockets	n/a	srv_sd
current_cli_sd	n/a	n/a
socket_parent_map	srv_sd	cli_sd
socket_child_map	cli_sd	Set(srv_sd)
socket_nonce_map	srv_sd	nonce

Table 7.2: Summary of the interposition library data structures

Instead of completely replacing the implementation of a function, the replacement implementations of these functions call into the actual C library functions whose place they take. The re-implemented versions of these functions are referred to with a `new_` prefix. Calls to the C library implementations of the functions have the prefix `original_`[4].

The interposition library implementation maintains the data structures in Table 7.2. `client_sockets` is the set of socket descriptors for which the socket connection peer is a client. `service_sockets` is the set of socket descriptors for which the socket connection peer is a service. `current_cli_sd` is the socket descriptor within `client_sockets` on which the application has most recently performed a `read()` function. If a socket descriptor in `service_sockets` was initiated on behalf of a socket descriptor in `client_sockets` (as would occur in a deputy application), `socket_parent_map` maintains a mapping from the "child" service socket to the "parent" client socket on whose behalf it was initiated. `socket_child_map` maintains the reverse of the mapping in `socket_parent_map`. `socket_nonce_map` maintains a mapping between a service socket and the session nonce that must be written into Speaks For Layer Protocol messages (see Section 5.5.1 for a description of session nonces). These data structures are discussed where appropriate in the discussion of this section.

[4]The Delegation Framework interposition library resolves the memory address of a given function's original C library implementation using the `dlsym()` interface.

new_connect()

new_connect() is implemented as follows:

```
 1: ret ← original_connect(sd, name, namelen)
 2: if ret < 0 then
 3:   return ret
 4: end if
 5:
 6: service_sockets ← service_sockets ∪ {sd}
 7: id ← report_flow_socket_setup(sd, current_cli_sd, &nonce)
 8: sfl_write_speaks_for(sd, id, nonce)
 9: socket_nonce_map[sd] ← nonce
10: if current_cli_sd ≠ −1 then
11:   socket_parent_map[sd] ← current_cli_sd
12:   S ← socket_child_map[current_cli_sd] ∪ {sd}
13:   socket_child_map[current_cli_sd] ← S
14: end if
15: return ret
```

new_connect() is the routine used by a client or deputy to initiate a new service socket connection. The original_connect() returns 0 on success, or -1 on error. If ret does not reflect an error condition following the call in line 1, the new socket descriptor is inserted into the service socket descriptor set service_sockets. Subsequent new_write() calls use service_sockets to identify when data should be encapsulated within Speaks For Layer Protocol messages before being written to the socket. Having setup a new socket, new_connect() routine reports the new socket to the local Delegation Agent at line 7. The argument current_cli_sd value is either -1 (the initialized value), or a value set by the new_read(). The report_flow_socket_setup() function returns the identity name for which the new socket speaks and the appropriate session nonce for the writing of a SpeaksFor message into the socket at line 8. The nonce must be stored in the socket_nonce_map data structure for subsequent SpeaksFor messages. new_connect() call then checks whether the current_cli_sd value is set to a valid socket descriptor number. current_cli_fd stores the socket descriptor of the most recently read client socket, the socket which (according to the

model described in Section 7.5.2) any subsequent application protocol requests must speak for. The interposition library stores the parent-child relationships (i.e., a child socket is a socket that a deputy uses to speaks on behalf of a parent socket) between sockets in the socket_parent_map and socket_child_map data structures.

new_write()

new_write() is implemented as follows:

```
 1: if sd ∉ service_sockets then
 2:   return original_write(sd, buf, bytes)
 3: end if
 4:
 5: if current_cli_sd ≠ −1 and socket_parent_map[sd] ≠ current_cli_sd then
 6:   id ← report_socket_dependency(sd, current_cli_fd)
 7:   sfl_write_speaks_for(sd, id, socket_nonce_map[sd])
 8:   socket_parent_map[sd] ← current_cli_sd
 9:   S ← socket_child_map[current_cli_sd] ∪ {sd}
10:   socket_child_map[current_cli_sd] ← S
11: end if
12: return sfl_write_data(sd, buf, bytes)
```

new_write() properly encapsulates application protocol data in the Speaks For Layer Protocol. If the sd argument is not included in service_sockets, data written to the descriptor need not be encapsulated in Speaks For Layer Protocol messages. Otherwise, the data writing to the service socket must be encapsulated in the Speaks For Layer Protocol using the sfl_write_data() Delegation Framework Library call. Before the data can be written, the interposition code checks whether the current_cli_sd (i.e., the socket from which the most recent application protocol request was received) is the known parent socket of the child service socket to which the write is about to occur. If it is not, the new child-parent relationship must be reported to the local Delegation Agent, which is done at line 6. The name and session nonce returned by the dependency report function are then sent to the legacy service process in a SpeaksFor message. The procedure then updates the socket_child_map

and socket_parent_map to reflect to current parent-child socket relationships. Finally, the interposition library writes the Speaks For Layer Protocol-encapsulated request to the socket.

new_accept()

new_accept() is implemented as follows:

```
1: ret ← original_accept(ld, name, namelen)
2: if ret < 0 then
3:   return ret
4: end if
5:
6: client_sockets ← client_sockets ∪ {ret}
7: return ret
```

A deputy or service process uses the new_accept() to accept a new client socket connections. original_accept() returns the newly accepted client socket (ld is the listening socket), or -1 on error, at line 1. If ret does not reflect an error condition, the socket descriptor must be inserted into the client socket descriptor set client_sockets. Subsequent new_read() functions use client_sockets to identify when to expect data carried within Speaks For Layer messages.

new_read()

new_read() is implemented as follows:

```
1: if sd ∉ client_sockets then
2:   return original_read(sd, buf, bytes)
3: end if
4:
5: current_cli_fd ← sd
6: ret ← 0
7: while ret = 0 do
8:   ret ← sfl_read(sd, buf, bytes, &id, &nonce)
9:   if id ≠ NULL then
10:    id_dep ← report_socket_speaks_for(sd, id, nonce)
11:    for each child_sd ∈ socket_child_map[sd] do
```

```
12:       sfl_write_speaks_for(child_sd, id_dep, socket_nonce_map[sd])
13:     end for
14:   end if
15: end while
16: return ret
```

Descriptors not contained in `client_sockets`, both for files and service sockets (services do not encapsulate their application protocol response in Speaks For Layer Protocol messages), will be handled normally using the C library implementation of `new_read()`. After being created by the `new_accept()` call and included in `client_sockets`, a `new_read()` call on a client socket causes the `current_cli_fd` to be updated to reflect the most recently read from client socket in accordance with the model described in Section 7.5.2. When reading from a client socket, the `sfl_read()` call may return a SpeaksFor message. The contents must be reported to the local Delegation Agent, and the new identity name for which the client socket's children speaks must be forwarded along via the child sockets. `new_read()` loops until it receives Speaks For Layer-encapsulated application protocol request data at which point it returns to the caller.

`new_close()`

When a socket is closed, the corresponding state in the interposition library must be cleaned up. `new_close()` function removes all state relevant to the closed socket from the interposition implementation data structures. The function is implemented as follows:

```
1: original_close(d)
2: if d ∈ service_sockets or d ∈ client_sockets then
3:   report_socket_shutdown(d)
4: end if
5: if d ∈ service_sockets then
6:   socket_parent_map[d] ← ∅
7:   service_sockets ← service_sockets / {d}
8: end if
9: if d ∈ client_sockets then
```

```
10:    socket_child_map[d] ← ∅
11:    client_sockets ← client_sockets / {d}
12: end if
13: if current_cli_fd = d then
14:    current_cli_fd ← −1
15: end if
```

Chapter 8

Integration Study of Apache and MySQL

Though the Delegation Framework Library implementation makes it possible to integrate the Delegation Framework with minimal or no manual source code modification, it remains unclear whether the remaining level of effort to make a system delegation-enabled is practical. Further, it is unclear whether it is possible to integrate the Delegation Framework into a network application without incurring unreasonable overhead.

This chapter presents a proof-of-concept integration of the Delegation Framework into an existing network system. The system consists of a clients and an Apache web server which provides content derived from queries on a MySQL database. In the application of the Delegation Framework, Apache takes the role of a deputy working on behalf of its clients to access the MySQL database. The MySQL database acts as an end service which grants end users the authority to access tables and records. Apache holds no authority to access the MySQL database that it is not explicitly delegated by its users. A web client, the `wget` utility, is used to access the MySQL database indirectly via an Apache-served PHP web application to which it delegates

the authority necessary to perform the requested task on the underlying database.

Following a description of the integration process that took place, this chapter presents a measurement of the additional Delegation Framework overhead observed at the client.

8.1 MySQL

If MySQL is to control access to its tables using the Delegation Framework, it will need to perform reference monitor calls into its Delegation Agent. Reference monitor calls are not automatically integrating by the dynamic interposition implementation of the Delegation Framework Library and, therefore, must be included manually in the MySQL source code. This requires an understanding of MySQL's existing authorization mechanisms and how these mechanisms will interact with Delegation Framework reference monitor calls. MySQL version 5.0.22 was used for the integration study.

8.1.1 Existing Authorization Mechanisms

The MySQL documentation provides a detailed description of the database software's access control mechanisms [1]. MySQL checks for authority at two times: when a client connects to a database service, and when the database service receives a client request (i.e., receives an SQL query).

When a client connects to the database, MySQL attempts to authenticate the client. The client presents a username and password, and may be authorized to connect only from a specified range of hosts. The username, password, and allowed address range are stored in the `user` table within the `mysql` database stored on a given MySQL database service.

After being authenticated and authorized to connect to the MySQL service, a

client may send requests to the database service in the form of SQL queries. MySQL performs authorization checks on each of these queries using records from a number of tables in its `mysql` database. The `user` table, mentioned above, stores grants of authorities on a global basis (e.g., grant the authority to perform SELECT queries on all tables in a database). The `db` table provides authority at the coarse granularity of a given database served by the MySQL service (e.g., grant INSERT and UPDATE authority on all tables in a database). `tables_priv` allows authorities to be granted for specific queries on tables in specific databases (e.g., grant INSERT, UPDATE and SELECT on `tableA` but only SELECT on `tableB`). Another table, `procs_priv` defines authorities to use different stored procedures offered by a given MySQL service. The most fine-grained authorities are specified in the `columns_priv` table, which allows query authorities to be granted on a per-column basis.

The entries in these tables are keyed on the identity (i.e., username) of that the client as authenticated at the time the client connected to the MySQL service. When determining whether to authorize a particular query, the MySQL service examines the entries that match the client's identity in each of these tables. A client principal's full set of authority is the union of the sets of authority granted in each of these tables; if any one of these tables grants authority that matches the query's requirements, the database service will execute the query and send the results back to the client.

8.1.2 Crafting MySQL Authorization Policies

For the purposes of the integration study, the goal was to have the ability to control access on a per-table basis using delegation credentials as is provided by MySQL's `tables_priv` table. This provides an administrator the ability to grant authorities to perform specific types of queries on specifies tables. Defining a useful Delegation Framework authorization policy for these authorities was straightforward. The allowed operations correspond to the different query types (a boolean column for

each of which exists in the `tables_priv`) and the allowed objects correspond to the database and table names on which the query may be performed. For instance, the following authorization policy defines the authority to perform a SELECT query on the table `tbl` in the database `data` on the database service with identity named `database@foo.example.com`:

```
database@foo.example.com:select:data.tbl
```

The database name and table name are concatenated together to form the complete subject field value.

Alternatively, an administrator can grant authorities to perform any query on any table in the `data` using the following authorization policy:

```
database@foo.example.com:*:data.*
```

Thought not implemented during the integration study, it is also straightforward to define authority to perform a particular query on a per-column level. The following authorization policy defines the authority to perform the INSERT query only on the column `colA` in the table `tbl` in databse `data`:

```
database@foo.example.com:insert:data.tbl.colA
```

8.1.3 Adding Delegation Framework Authorization Checks

Understanding and modifying the MySQL code required about 15 hours of effort (with some of that time spent resolving bugs in the Delegation Framework which presented themselves while testing the MySQL source code modifications).

Understanding the Code

Given that MySQL has a sophisticated system for defining authorities, integrating Delegation Framework authorization policy check was a conceptually straightforward

task. With a few hours of examining the MySQL source code, I was able to identify the `sql_acl.cc` source file in which MySQL's implementation of authorization checks using the internal grant tables exists. A number of routines exists in this file, with one corresponding to each of levels of granularity provided by each of the different grant tables that MySQL uses (i.e., `tables_priv` and `columns_priv`).

The `check_grant()` routine provides the interface through which MySQL threads check whether a query is authorized by an table-level authorization entry in the `tables_priv` table. This is the routine that must be modified to perform the appropriate Delegation Framework reference monitor calls containing authorization policies that the client principal must satisfy.

The caller passes the `check_grant()` routine a list of the tables to which a query requires access and a bit vector specifying the types of queries that the client wishes to perform on those tables. `check_grant()` verifies whether the client is authorized to perform each of the queries specified in the bit vector on all of the listed tables.

Though not modified for this integration study, the other authorization checking functions in the `sql_acl.cc` source file are similarly structured and lend themselves to analogous modifications, thereby making it possible to use delegation credentials for authorization of SQL queries at all levels of granularity provided by MySQL's authorization mechanisms.

Modifying the Code

Instead of eliminating the MySQL authorization mechanism in favor of a purely Delegation Framework-based approach, my approach to modifying `check_grant()` alters the MySQL authorization mechanism such that a query must satisfy both the requirements of the standard MySQL `tables_priv` grant table *and* the Delegation Framework-equivalent authorization policy. The following modifications to `check_grant()` routine could be applied to the corresponding routines for each of

MySQL's other grant tables (i.e., `columns_priv`, etc.), thereby creating Delegation Framework analogues to each grant table provide by the standard MySQL mechanism.

Modifications to the `check_grant()` function totaled an additional 34 lines of code to the its original implementation of 71 lines[1]. An additional 14 lines referencing the Delegation Framework Library header files and a static data structure were also included.

As `check_grant()` loops over the list of tables to check whether the client is authorized to perform all required queries on each table, the modified source constructs a Delegation Framework authorization policy matching the authorization requirements that `check_grants()` looks up in `tables_priv`. After `check_grant()` has checked that the client satisfies all `tables_priv` requirements, a complete Delegation Framework authorization policy has been defined. `check_grant()` calls the Delegation Framework function `check_authority()` (see Section 7.4), passing the constructed authorization policy and the socket descriptor for the socket from which the query was read. The socket descriptor argument is derived from an argument passed to `check_grant()` called the *thread handle*. The thread handle stores information relevant to the context in which the calling thread is handling client requests. The thread handle includes the descriptor of the socket through which the handler is connected to its client. The call to `check_authority()` may initiate a Delegation Protocol dialogue (see Chapter 5) and, if it does, will not return until the dialogue is completed. The reference monitor call returns a boolean result indicating whether the client has satisfied the necessary authorization policy and, if it has, `check_grant()` returns a result indicating that the client's SQL query is authorized.

[1]Source code lines were counted using the "physical source lines of code" method [31].

Building `mysqld`

Properly building MySQL required that the necessary Delegation Framework Library shared library object, `libdf.so`, be linked into the `mysqld` binary. This requires an additional argument to the MySQL package's `configure` script specifying additional linking flags to use when building the `mysqld` binary. The Delegation Framework Library and OpenSSL library must be included in the binary[2]:

```
$ ./configure --with-mysqld-ldflags=-L<Framework library path> \
  -ldf -lssl
```

Executing `mysqld`

When executing the `mysqld` binary, a number of parameters must be specified via environment variables at the time of execution. These serve to properly load the dynamic interposition code in the shared library object `poser.so`, and to provide the Delegation Framework Library with configuration information. The location of the UNIX domain socket for `mysqld`'s Delegation Agent and its identity name are passed in the `DA_SOCK` and `DA_IDENTITY` variables respectively. The library files specified in `LD_PRELOAD` cause the interposition functions, contained in `poser.so`, to be loaded. The following is an example of the environment variables as they might be setup in the environment `mysqld` when the binary is executed:

```
LD_PRELOAD=poser.so:libdf.so
DA_SOCK=/tmp/da.sock.A5g21F
DA_IDENTITY=database@foo.example.com
```

Within this environment, the `mysqld` binary can be executed normally without any special arguments, causing the interposition library to be loaded and intercept the appropriate function calls.

[2]The Delegation Framework Library implementation relies on OpenSSL for cryptographic functions and random number generation.

8.2 Apache

Apache plays the role of a deputy when serving a database-driven web application. For the integration study, the Apache web service does not perform any authorization checks and therefore can have the Delegation Framework integrated automatically.

8.2.1 Automatic Integration

Integrating the Delegation Framework into Apache was very straightforward, but did require some modification of the interposition library for support.

Forcing `connect()` to be Synchronous

Though each Apache process handles one request at a time from one client, the Apache web server using non-blocking I/O when performing `connect()` calls on behalf of a client. This is a violation of the assumptions that socket I/O operations are performed synchronously. This violation makes it impossible for the interposition library's `connect()` function to behave properly because the C library `connect()` call may return a result indicating an asynchronous I/O operation is in progress. The interposition library's `connect()` function cannot report the flow identifier of the new flow to its Delegation Agent in this case, and also cannot write a SpeaksFor message into the socket until the C library `connect()` function call returns.

This requires the interposition library's `connect()` function to force the C library `connect()` call to be blocking by setting its socket descriptor argument as blocking[3]. Due to the multiprocess, synchronous architecture of Apache, this does not constitute a major reduction in the web application's performance; in our system a socket must be setup before a database query can be sent and therefore, blocking or not, the C library `connect()` call must be completed before a MySQL-driven webpage can be served to a client.

[3]This is done using the `fcntl()` routine.

Executing Apache's `httpd` Binary

Because Apache does not perform any reference monitor calls, integrating the Delegation Framework into Apache requires only that the Apache `httpd` binary is executed within the proper environment. This environment will ensure that the interposition library is properly loaded and the Delegation Framework Library parameters are available to it. These parameters are similar to those used when executing `mysqld`. For the Apache web service, this might look like the following:

```
LD_PRELOAD=poser.so:libdf.so
DA_SOCK=/tmp/da.sock.G54x2D
DA_IDENTITY=web@bar.example.com
```

8.2.2 Challenges for Web Application Authorization Policies

Though this integration study did not involve performing authorization checks at the Apache web service, it is important to consider the ease with which this might be done if system administrators find it desirable to perform authorization checks from within Apache. The Apache web server may not be able to define an authorization policy domain that seems reasonable for all Apache-based web applications. Unlike a database service, an Apache web service can serve a variety of very different applications with very different interfaces; an arbitrary CGI binary or HTML-embedded PHP program can be executed to serve a page. Each of these types of web applications is accessed using the HTTP GET and POST methods, but these methods are too general operations to perform fine-grained authorization checks for the operations offered by a particular web application. Therefore, it seems unlikely that a general Delegation Framework authorization checking implementation for Apache will be useful. If authorization checks are desirable at the Apache interface, each CGI binary or PHP program will have to implement its own reference monitor calls.

8.3 Client Programs

The `wget` utility [30] served as the legacy client program for the Apache/MySQL-based network application. Like Apache, `wget` required no manual modifications, only that the loading of the interposition library and the corresponding Delegation Framework Library configuration are specified in the execution environment. For `wget` or other client program, the environment might look like the following:

```
LD_PRELOAD=poser.so:libdf.so
DA_SOCK=/tmp/da.sock.G54x2D
DA_IDENTITY=client@baz.example.com
```

In addition to the `wget` client, the `mysql` client program was used during testing of the MySQL source modifications. Automatic integration worked well with both clients without additional tweaking of the dynamic library implementation.

8.4 Delegation Framework Overhead

The implementation of the Delegation Framework was focused on serving as a proof of concept, not as a well-optimized, high performance implementation. Therefore, no effort was put into improving the performance of the Delegation Framework implementation. Nevertheless, it is important to understand the magnitude of the overhead incurred due to the integration of the Delegation Framework. Future work on the Delegation Framework can improve upon the shortcomings of the current prototype implementation.

It is possible to measure the overhead incurred due to the Delegation Framework by comparing the delay observed from the perspective of a client with and without the Delegation Framework integrated into the network application. To perform this comparison, one simply measures the amount of time from the instant a client sends its request to a service until the instant when the client receives a response from the

service. Comparing the delay incurred with and without the Delegation Framework integrated into the `wget`, Apache and MySQL programs gives us a sense of whether the overhead experienced due to the Delegation Framework will be reasonable in real world system.

8.4.1 Experimental Setup

To measure the additional delay experienced by a client in a Delegation Framework-enabled network application, a toy web application was constructed. The web application, `select.php`, consisted of a simple PHP program executed within the Apache deputy that connects to the MySQL database service and issues a SELECT query on a table with a single column of integer type values. `select.php` responds to the client indicating whether the query succeeded or failed. During the experiment, the table was populated with 20 rows. This simple web application can be considered a reasonable baseline for the minimum delay experienced in most any database-served PHP web application; the performance of the average PHP-implemented web application will likely exceed that of the web application used in these tests.

The delay between initiating an HTTP request and receiving an HTTP response was measured from the perspective of the `wget` client. The delay was measured both with and without the Delegation Framework integrated into the legacy software components. The average difference in the observed delays with and without the Delegation Framework provide an estimate of the performance overhead that a network application can be expected to incur when integrated with the Delegation Framework.

To measure the delay, the `wget` client program is instrumented with timing code capable of microsecond precision[4]. The timing code measures the amount of time that the program's `retrieve_url()` routine takes to execute and return. The

[4]Microsecond precision is available using the `gettimeofday()` C library routine, though the clock in a given hardware platform may not support microsecond timing.

`retrieve_url()` sets up the application protocol socket setup with Apache if necessary, retrieves the specified URL, and writes the output to a file. If a TCP socket has already been setup with a specific web service (i.e., during the first call to `retrieve_url()`), it will be reused for subsequent retrievals from the same web service.

In a real world system, a user would interact with its Delegation Agent during a Delegation Protocol dialogue (see Section 6.2.4). In order to eliminate the human response delay factor from the experiment, the user Delegation Agent was modified not to prompt the user. Instead, the user Delegation Agent behaves as though its user had authorized the delegation without waiting for a user response. This allows the experiment to focus on the measurement of delay derived from parameters excluding the user involved (i.e., implementation performance).

For the experiment, the user Delegation Agent and `wget` along with its Delegation Agent were executed on the same network host. The MySQL and Apache services were executed on two additional independent network hosts. The three machines were connected via a dedicated network switch carrying only the traffic between the three hosts involved in the test and remote login traffic from a fourth machine.

Four separate trial sets were run; one or two `wget` clients were used and the Delegation Framework was or was not integrated in a given trial, constituting the total of four sets of trials. In each trial set, five individual trials in which each client issued 100 successive HTTP requests for the `select.php` page served by the Apache web deputy (requiring Apache to initiate a TCP connection to the underlying MySQL database service to which `select.php` program connects). The length of time for which the `retrieve_url()` executed was recorded for each of the HTTP requests issued by `wget`.

# clients	1	2
Average Normal (ms)	12.72	10.48
Std. Dev. Normal (ms)	2.33	2.65
Average with Framework (ms)	323.71	634.56
Std. Dev. with Framework (ms)	1.53	199.08
Average Added Delay (ms)	310.99	624.07
% Average Delay Increase	2545	5953

Table 8.1: Delay statistics for initial HTTP query

8.4.2 Results and Analysis

For the analysis of results, the initial query sent by each instance of `wget` during a given trial set is treated as distinct from all subsequent trials in the same set. Due to the implementation of `wget`, the initial request is the only request for which a new HTTP socket is opened. All subsequent queries reuse the HTTP socket. Therefore, the measured delay of the initial query will include the delay incurred due to the TCP handshake between `wget` and Apache. Additionally, the initial query during a given set of trials with the Delegation Framework will include the delay due to the initial negotiations of Delegation Protocol sessions between the `wget` and Apache, as well as Apache and MySQL. This allows us to consider the Delegation Framework setup overhead distinctly from the steady state overhead.

Initial Query Delay

Table 8.1 lists the statistics gathered regarding the initial queries across the one and two client trial sets, both with and without the Delegation Framework integrated. The delay measurements are given in milliseconds. The "normal" values represent those taken without the Delegation Framework integrated. In the single client trials, the initial query took 369 milliseconds with the Delegation Framework involved where they took 13 milliseconds without the Delegation Framework, indicating an average of 356 milliseconds of additional delay. In the two client trial sets, the initial query

of each client took 635 milliseconds on average with the Delegation Framework integrated as compared to an average of 10 milliseconds without, representing an average additional delay of 624 milliseconds.

Though the figures of a 28.0 times and 59.5 times increases in delay (in the one and two client cases respectively) seem alarming, they may not seem unreasonable in the context of more complex web applications. a real world web applications is typically much more complex than the toy web application used for these performance tests and would incur similar amounts of Delegation Framework-related overhead as the toy application would. Nevertheless, this additional delay is considerable.

Little of the delay can be blamed on the network latency in the Delegation Protocol channel. Table 8.2 lists the pairwise round trip time measured over ten consecutive `ping` queries between each of the host pairs that must exchange Delegation Protocol message[5]. Each of the pairs of principals listed in the table performs the mutual authentication protocol during Delegation Protocol session setup (one round trip time for each host pair) and Delegation Protocol dialogue initiated during the handling of the initial HTTP request (a second round trip time for each host pair), but this does not account for much of the measured additional delay:

$2RTT * (0.016 + 0.165 + 0.223)ms/RTT = 0.808ms.$

Most of the additional delay is incurred due to processing overhead within the Delegation Framework implementation. The considerable amount experienced during the initial query is due to the cryptographic processing of multiple certificates during the Delegation Protocol session setup dialogue (see Section 5.4) and the subsequent dialogue in which authority requirements and delegation credentials are transmitted (see Section 5.8). A considerable amount of time is spent in the prototype implementation marshalling and unmarshalling the large Delegation Protocol messages as well as encapsulating and de-encapsulating application data in the Speaks For Layer

[5]The user and client Delegation Agents are executed on the same host, but communicate using the TCP/IP stack as they would if they were communicating via a network.

Host Pair	Avg. (ms)	Dev. (ms)
user-client	0.016	0.006
client-deputy	0.165	0.035
deputy-service	0.223	0.040

Table 8.2: Round trip times between the network application's host pairs

# clients	1	2
Average Normal (ms)	5.82	10.10
Std. Dev. Normal (ms)	0.47	4.03
Average with Framework (ms)	46.53	160.24
Std. Dev. with Framework (ms)	4.05	86.26
Average Added Delay (ms)	40.71	150.14
% Average Delay Increase	699	1486

Table 8.3: Delay statistics for subsequent HTTP queries

Protocol. The client experiences additional overhead due to communication between each process and its local Delegation Agent via the Delegation Framework Library (some of which is also experienced in subsequent queries).

Subsequent Query Delay

Table 8.3 lists the statistics gathered regarding the non-initial "steady state" queries in each of the four trial sets. The delay experienced by these queries is significantly lower given that the TCP socket between `wget` and Apache is setup (`select.php` sets up a new one each time it is run, but the resulting delay is experienced both with and without the Delegation Framework), and the necessary delegation credentials have already been delegated and presented. The delay times are given in milliseconds. The "normal" values represent those taken without the Delegation Framework integrated. In the single client trials, the initial query took 77 milliseconds on average with the Delegation Framework integrated where they took 6 milliseconds on average without the Delegation Framework, indicating an average of 71 milliseconds of additional delay. In the two client trials, the the average subsequent query from each client

took 160 milliseconds with the Delegation Framework integrated as compared to an average of 10 milliseconds without, representing an average additional delay of 150 milliseconds. These figures represent 12.2 times and 14.9 times increases in delay for non-initial HTTP queries in the one and two client cases respectively as compared to the network application excluding the Delegation Framework.

Given that the Delegation Protocol sessions have been setup between peer Delegation Agents and that all necessary credentials have been delegated and demonstrated, the delay experienced by steady state queries consists exclusively of Speaks For Layer overhead and communication between each client, deputy, or service process and its Delegation Agent.

Chapter 9

Conclusion

The Delegation Framework provides legacy client and service software with a delegation mechanism for the passing of authority from one principal to another. The framework, designed with deployability in mind, makes delegation possible in network applications without re-specification of application protocols or the re-design and re-implementation of legacy software. The Framework's prototype implementation further minimizes the effort required to integrate delegation into legacy software.

9.1 Contributions

9.1.1 Protocol Support for Delegation of Authority

The Delegation Framework provides two novel protocols, the Delegation Protocol and Speaks For Layer Protocol, that make it possible to delegate authority for the class of network applications which adhere to the client-server protocol model and are carried via TCP. These protocols, together with a target application protocol, allow a service to mediate requests made via the application protocol using credentials delegated via the Delegation Protocol. This approach enables systems to support delegation of authority without a redesign and re-implementation of the myriad application-level

protocols on which they rely.

9.1.2 Integration of Delegation into Legacy Software

The design and implementation of the Delegation Framework simplifies the integration process by which a network application features the delegation of authority. Past delegation systems implemented the mechanism in ways that were not conducive to easy deployment in legacy systems. The Delegation Framework succeeds in implementing a delegation mechanism with a realistic integration path into legacy software. For many legacy programs, integration is possible without manual modification to the program. At worst, a developer must include authorization checks in services that handle authority-requiring requests.

9.2 Future Work

While the Delegation Framework has improved the ease with which delegation can be integrated into legacy network applications, a number of challenges remain which affect whether the Delegation Framework can realistically be integrated into existing systems.

9.2.1 Fully-automated Integration

Currently, it is not possible to automatically integrate the Delegation Framework into legacy applications. To benefit from the the Delegation Framework, services must be manually modified to use the framework's reference monitor. Additionally, legacy applications that perform non-blocking I/O are not supported. Future work on the Delegation Framework should attempt to address these shortcomings.

Automatic Reference Monitoring

While the Delegation Framework has made significant strides in reducing the effort required to integrate delegation into legacy software, it remains unclear whether it is possible to automatically integrate the authorization checks that must currently be integrated manually. Recent research has demonstrated some promising code analysis techniques for automatically inserting reference monitor calls into existing code [11]. However, it is not immediately clear whether their techniques are applicable to the Delegation Framework. If the authors' techniques (or extensions thereof) are indeed applicable, a future implementation of the Delegation Framework could be integrated without any manual modification to the source code of legacy software.

Automatic Integration for Asynchronous Programs

Services that employ asynchronous I/O typically outperform their synchronous counterparts [23, 10] and, therefore, are desirable choices for service providers. However, the Delegation Framework implementation does not properly support asynchronous socket I/O. Two aspects of the Delegation Framework's dynamic interposition implementation make dealing with asynchronous I/O difficult.

First, the interposed C library routines rely on the successful completion of synchronous calls to the real underlying C library routines. For instance, the `new_connect()` call requires that the socket be completely setup because it must write a SpeaksFor message into the socket before it returns. The interposed routines rely on the full completion of `new_accept()`, `new_read()`, and `new_write()` calls as well. Versions of these interposed routines that are robust in the face of asynchronous I/O must be developed.

Second, the socket dependency model used by the interposition library is not suited for correctly inferring socket dependencies in legacy deputies that perform asynchronous I/O. An event-driven deputy frequently switches between handling different

requests from multiple client when it is waiting for one or more asynchronous I/O operations to complete. When this type of frequent switching occurs, it is no longer reasonable to assume that an application protocol request was written to a socket on behalf of the most recently read socket. A more sophisticated model must be developed to handle asynchronous I/O before the interposition library can successfully implement the Delegation Framework Library socket dependency calls automatically for asynchronously programmed deputies.

9.2.2 Usability

The usability of a delegation system is just as important to its adoption as making it easy to integrate with existing software. The current prototype of the Delegation Framework has addressed some of the issues inhibiting the deployment of delegation in legacy software, but it has not addressed the usability issues that remain. If the Delegation Framework is to become widely deployed *and* used properly once it is deployed, a sensible and intuitive user interface must be developed. The current prototype provides only a basic functional console-based user interface. The authorization policies used in the Delegation Framework may not make sense to users, especially novices. They must be presented to users in a palatable manner. Research on user interfaces for managing authorization policies must be conducted and incorporated into the Delegation Framework. Further, providing usable interfaces may require modifications to parts of the Delegation Framework supporting the interface. These issues must be explored along with corresponding studies of the usability of any interfaces that may be proposed.

Bibliography

[1] MySQL 5.0 reference manual. http://dev.mysql.com/doc/refman/5.0/en/index.html.

[2] Martín Abadi, Michael Burrows, Butler Lampson, and Gordon Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, September 1993.

[3] M. Backes and B. Pfitzmann. A cryptographically sound security proof of the needham-schroeder -lowe public-key protocol. December 2003.

[4] Daniel J. Barrett and Richard E. Silverman. *SSH, the Secure Shell: The Definitive Guide.* O'Reilly Media, Inc., Sebastopol, CA, February 2001.

[5] Lujo Bauer, Scott Garriss, Jonathan M. McCune, Michael K. Reiter, Kason Rouse, and Peter Rutenbar. Device-enabled authorization in the grey system. Technical Report CMU-CS-05-111, CMU, February 2005.

[6] Lujo Bauer, Scott Garriss, and Michael K. Reiter. Distributed proving in access-control systems. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, May 2005.

[7] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized trust management. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 164–173, May 1996.

[8] Timothy W. Curry. Profiling and tracing dynamic library usage via interposition. In *USENIX Summer*, pages 267–278, 1994.

[9] Carl Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian Thomas, and Tatu Ylonen. SPKI certificate theory. RFC 2693, September 1999.

[10] Khaled Elmeleegy, Anupam Chanda, Alan L. Cox, and Willy Zwaenepoel. Lazy asynchronous I/O for event-driven servers. In *Proceedings of the 2004 USENIX Technical Conference*, Boston, MA, June 2004.

[11] Vinod Ganapathy, Trent Jaeger, and Somesh Jha. Retrofitting legacy code for authorization policy enforcement. May 2006.

[12] Tal Garfinkel, Ben Pfaff, and Mendel Rosenblum. Ostia: A delegating architecture for secure system call interposition. In *Proceedings of the 2004 Network and Distributed Systems Security Symposium*, February 2004.

[13] Morrie Gasser and Ellen McDermott. An architecture for practical delegation in a distributed system. In *Proceedings of the 12th IEEE Symposium on Security and Privacy*, pages 20–30, October 1990.

[14] Norm Hardy. The confused deputy (or why capabilities might have been invented). 1988.

[15] Michael B. Jones. Interposition agents: Transparently interposing user code at the system interface. *Operating Systems Review*, 4(5):80–93, 1993.

[16] Michael Kaminsky, Eric Peterson, Daniel Giffin, Kevin Fu, David Mazires, and M. Frans Kaashoek. REX: Secure, extensible remote execution. In *Proceedings of the 2004 USENIX Annual Technical Conference*, June 2004.

[17] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: theory and practice. *ACM Trans. Comput. Syst.*, 10(4):265–310, 1992.

[18] Ninghui Li, Benjamin N. Grosof, and Joan Feigenbaum. Delegation Logic: A logic-based approach to distributed authorization. *ACM Transaction on Information and System Security (TISSEC)*, February 2003.

[19] Gavin Lowe. An attack on the needham-schroeder public-key authentication protocol. *Information Processing Letters*, 56(3):131–133, 1995.

[20] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, December 1978.

[21] Clifford Neuman, Tom Yu, Sam Hartman, and Kenneth Raeburn. The kerberos network authentication system. RFC 4120, July 2005.

[22] http://www.openssh.org/.

[23] Vivek Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An efficient and portable web server. In *Proceedings of the 1999 USENIX Technical Conference*, Monterey, CA, June 1999.

[24] Niels Provos. Improving host security with system call policies. In *Proceedings of the 12th USENIX Security Symposium*, August 2003.

[25] Jon Salz, Alex C. Snoeren, and Hari Balakrishnan. TESLA: A transparent, extensible session-layer architecture for end-to-end network services. In *4th USENIX Symposium on Internet Technologies and Systems*, Seattle, WA, March 2006.

[26] Sara Sinclair and Sean W. Smith. PorKI: Making user PKI safe on machines of heterogeneous trustworthiness. In *Proceedings of the 21st Annual IEEE Computer Security Applications Conference*, December 2005.

[27] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. Stream control transmission protocol. RFC 2960, The Internet Society, October 2000.

[28] Douglas Thain and Miron Livny. Multiple bypass: Interposition agents for distributed computing. *Cluster Computing*, 4(1):39–47, March 2001.

[29] Michael Walfish, Jeremy Stribling, Maxwell Krohn, Hari Balakrishnan, Robert Morris, and Scott Shenker. Middleboxes no longer considered harmful. In *Proceedings of the 6th USENIX Symposium on Operating System Design and Implementation*, San Francisco, CA, December 2004.

[30] http://www.gnu.org/software/wget/.

[31] David Wheeler. More than a gigabuck: Estimating gnu/linux's size. http://www.dwheeler.com/sloc/redhat71-v1/redhat71sloc.html, June 2001.

[32] Edward Wobber, Martín Abadi, Michael Burrows, and Butler Lampson. Authentication in the taos operating system. In *Proceedings of the 14th ACM Symposium on Operating System Principles*, pages 256–269. ACM Press, 1994.

[33] Xiaolan Zhang, Antony Edwards, and Trent Jaeger. Using CQUAL for static analysis of authorization hook placement. In *Proceedings of the 2002 USENIX Technical Conference*, San Francisco, CA, August 2002.