5-31-2006

Limited Delegation (Without Sharing Secrets) in Web Applications

Nicholas J. Santos
Dartmouth College

Limited Delegation (Without Sharing Secrets) for Web Applications

An Undergraduate Senior Thesis in Computer Science

Dartmouth Computer Science Technical Report TR2006-574

Nicholas Santos, Dartmouth College

Nicholas.J.Santos@Dartmouth.edu

Thesis Advisor: Sean W. Smith

May 31, 2006

Abstract

Delegation is the process wherein an entity Alice designates an entity Bob to speak on her behalf. In password-based security systems, delegation is easy: Alice gives Bob her password. This is a useful feature, and is used often in the real world.

But it's also problematic. When Alice shares her password, she must delegate all her permissions, but she may wish to delegate a limited set. Also, as we move towards PKI-based systems, secret-sharing becomes impractical. This thesis explores one solution to these problems. We use proxy certificates in a non-standard way so that user Alice can delegate a subset of her privileges to user Bob in a secure, decentralized way for web applications.

We identify how delegation changes the semantics of access control, then build a system to demonstrate these possibilities in action. An extension on top of Mozilla's Firefox web browser allows a user to create and use proxy certificates for delegation, and a module on top of the Apache web server accepts multiple chains of these certificates. This is done in a modified SSL session that should not break current SSL implementations.

Acknowledgments

No matter how smart you think you are, you have to have a place to start from: a name, an address, a neighborhood, a background, an atmosphere, a point of reference of some sort. All I had was typing on a crumpled yellow page...
In a situation like that the small man tries to pick the big man's brains.

-Raymond Chandler, *The Long Goodbye*

First, I'd like to thank the person who ultimately assigns this thesis a grade: my thesis advisor, Sean W. Smith. His guidance was invaluable throughout this research, and his teaching largely led me to this topic in the first place. Additionally, I thank my thesis committee members, Michael Fromberger and M. Douglas McIlroy, for their comments and input.

I also need to thank Chris Masone and Scout Sinclair, who provided some key insights. The other denizens of 045 Sudikoff should get some credit for being amusing, sharing their opinions often without being asked, and keeping the coffee machine running. They even had some fantastic advice when I least expected it. That list includes Naomi Forman, Allen Harvey, Bob Brentrup, Punch Taylor, Alex Iliev, Apu Kapadia, Nihal D'Cunha, Kevin Mitcham, and Massimiliano Pala. I'm also forgetting a large number of them, most notably John Baek.

Bart Blackburn, Liz Middleton, Mere Wilson, and Matt Latterner are not computer scientists, but they are end-users with interesting thoughts. For example, sharing passwords can be surprisingly useful. Sometimes your computer breaks down, and pen-and-ink signatures are easier to forge than CA signatures.

Lots of thanks definitely go to the development team of OpenSSL (`http://www.openssl.`

`org`), as well as the Mozilla Foundation (`http://www.mozilla.org`), and the Apache Software Foundation (`http://www.apache.org`). If not for their thorough documentation, I'd have a lot more gray hair than I do now.

It's always important to thank my parents. You wouldn't expect two philosophy-types to have insights into PKI certificate theory, but perhaps the philosophers are smarter than we think.

And lastly, this material is based upon work supported by the National Science Foundation under Grant No. 0448499. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Contents

Chapter 1

Introduction

Delegation is the process of passing my rights onto you. Or more rigorously, delegation is when an entity Alice authorizes an entity Bob to act on behalf of her, but with only a subset of her privileges. We may also say that Bob is acting as a **proxy** for Alice. The ultimate goal of this research is to provide an easy-to-use and generalizable framework that extends web-based access control to handle this delegation.

This document is divided into chapters by area of interest. Chapter 1 discusses delegation at the theoretical level. It contains an overview of how delegation is used today, how this thesis applies delegation in pursuit of a more practical Public Key Infrastructure (PKI), and how this application of delegation introduces new ways to think about how we make trust judgments. Chapter 2 presents a high-level specification of a system that implements web-based authorization based on delegated credentials, and a strategy for how such a system could be deployed. Chapter 3 discusses the particular tools chosen for our implementation. Chapter 4 is titled "Trials and Tribulations" for a reason—it describes the process of building such a system. It will summarize the changes to a modern web browser and web server that were required by this system[1]. Chapter 5 talks about the relation of this work to other research, and explores possible future research projects. Chapter 6 summarizes the major accomplishments of this research.

Lastly, never forget the story of Alice and Bob, or their respective roles as delegator and delegatee. Throughout this paper, whenever we wish to discuss the delegation relationship,

[1]This chapter will be of particular interest to anyone who wishes to modify the Mozilla and Apache software projects. In particular, it delves deeply into Mozilla Firefox, and comes out with some advanced ways to modify it via its extensions system. I hope to post a tutorial-based explanation of these methods in a more public space.

we will often use Alice and Bob as the names for these two roles. This is a convention in traditional security literature. Alice and Bob are very popular.

1.1 A Brief History of Delegation

For most computer applications, the delegation of privileges is surprisingly easy. Most security systems implicitly tie a set of privileges to a password. If a user wants to use her computer—or read her e-mail, or sign onto her favorite chat account—she types in her password. If she wants to let her friend check her e-mail for her, she gives him her password. Users are accustomed to this paradigm. And, by God, it's simple.

This is a form of "delegation." But it's not really delegation as security experts like to use the term, simply because this method is not really secure at all.

In a world with a public key infrastructure (PKI), Alice can delegate privileges to Bob by cryptographically signing a statement that asserts

Bob has subset S of my rights.

Love, Alice.

As you can see, the process is easy to understand in the abstract. Such a statement can be encoded in a PKI certificate, which is then signed by Alice.

But the process proved difficult to implement when researchers tried to shoehorn a format for delegation statements into current standards and protocols for certificates. The ruling certificate standard, X.509, is rigidly hierarchical and does not allow the average user to issue certificates. In X.509, there are certificate authorities (CAs), and there are end entities (normal users like Alice). CAs can issue certificates, but only to entities that are "subordinate" to the CA. End entities do not have the authority to issue any certificates—the reason that they are called "end entities" is because their certificates can only appear at the end of a certificate chain [HPFS02]. To delegate her privileges to Bob in the X.509 system, Alice would need to find a CA that she and Bob had in common, and ask this CA to sign her privileges over to Bob.

Such a common trusted CA might not even exist. And even if Alice does find a common CA, delegation may be difficult. CAs are the bureaucrats of the X.509 world—it can be cumbersome (and often financially expensive) to get their approval.

The Globus Toolkit (`http://www.globus.org/`) ran into this problem while building a secure framework for distributed computing. The purpose of the Globus Toolkit is best described by its web site:

> The open source Globus Toolkit is a fundamental enabling technology for the 'Grid,' letting people share computing power, databases, and other tools securely online across corporate, institutional, and geographic boundaries without sacrificing local autonomy. [glo06]

In short, it empowers people to organize large-scale distributed computations by recruiting a community of peers across the Internet. But, as stated above, part of its goal is to share "securely," and "without sacrificing local autonomy." A process sitting on a remote, autonomous machine may need access to restricted resources, so it needs mechanism to authorize this access dynamically. The CA approval process was unsatisfactory, for the exact reasons noted above. CAs were too cumbersome to be practical for authorizing short-lived processes[WFK+04].

Thus, the Globus Toolkit developers invented **proxy certificates** for delegation. This is probably the most widespread use of PKI-based delegation in real-world applications today. After some evolution, proxy certificates were standardized for X.509 in RFC 3820.

Proxy certificates are an extension to the X.509 certificate standard that allow end entities to sign certificate statements that delegate their own privileges to other entities. By the standard, an end entity generates temporary private and public keys, signs a short-lived proxy certificate that passes on some of her privileges to the temporary keypair, then gives those credentials to a third party entity. The identity of the proxy certificate is derived from the identity of the end entity. Because a proxy certificate can also testify to another proxy certificate, the identity of a chain of proxy certificates is the last non-proxy certificate in the chain (the end entity certificate).

Notice that when we say that these credentials are "temporary," this is merely a convention. There is no rigorous definition of how long a "temporary" period of time is [TWE+04]. This flexibility is intentional, because simplicity is of the essence. On the Grid, these proxy certificates can be issued to dynamically created processes without requiring the approval of a CA.

Delegation has also been explored as a clever strategy to enhance mobility and convenience without sacrificing security, as in John Marchesini's SHEMP work. Suppose we have a locked-down key-store that's difficult to access but very secure, and also a promiscuous key-store that's easier to access but more likely to contain vulnerabilities. Imagine the "secure" key-store is on a remote server located in Fort Knox, and the "insecure" key-store is on a portable Palm Pilot. The remote server can use its credentials to delegate privileges for a very short period of time to a temporary key on the portable device. If the portable device is stolen, or otherwise compromised, the short expiration on the certificate ensures that the damage done by an attack is minimized. This use of delegation enhances security, and reduces the overhead of certificate revocation mechanisms [MS05].

These projects demonstrate how security experts are thinking about delegation. First, much of the current research operates on the premise that users should not have to worry about managing their credentials. This is not an implicit assumption, but a conscious decision. It is much more secure to store credentials in a central repository. The user can then sign into the repository, and let the repository figure out which credentials he needs to access a given resource. This is the approach encouraged in the Grid and SHEMP. Second, the primary concern is with the "vertical" displacement of trust—how privileges propagate from an authority to the user of the privilege. The goal of this thesis is to investigate the opposing sides of these two premises.

1.2 Delegation from A to B

Suppose Alice wants to grant a subset of her privileges to another user, Bob, to act on her behalf. We call this user-to-user delegation. It's a less sophisticated case than user-to-process delegation, or user-to-promiscuous-keystore delegation. Yet this case has not been analyzed as often as these other modes, perhaps because common wisdom suggests that it doesn't happen as often.

However, I submit that it does happen very often. The best illustration is by example.

I once spoke with an IT administrator at a hospital. He loved the "delegation" feature of the Microsoft Exchange Server, an application that primarily handled their e-mail. This feature allows one user to "delegate" another to read and write e-mail on her behalf. The feature is activated through the Exchange Server, and both users must have accounts.

Notice that this isn't fully delegation in the sense that we're talking about it here, in the context of PKI. For us, delegation is a decentralized way for Bob to securely act on behalf Alice in a limited capacity. For the Microsoft Exchange Server, they toss out the "decentralized" qualifier on that definition. The server manages all permissions, and provides an application-centric interface for users to delegate privileges to other users, in a centralized way.

However, this certainly demonstrates what delegation can offer to the end-user experience. For example, executives can delegate access to their e-mail to their secretaries. When an employee goes on vacation, he can delegate temporary access to his resources for his co-workers—for the benefit of the co-workers, and of the employee who doesn't want to be bothered while on vacation! One can think of many relationships in a corporate, medical, legal, and even military environment where carefully-restricted access to another individual's e-mail could be a powerful and desirable tool.

Consider a second example. Here at Dartmouth, I contributed a little to the Greenpass project, which touted a delegation-based solution to wireless networking. In the problem statement addressed by Greenpass, there is a wireless network that we would like to restrict to only allow people that we know on the network. That's easy enough if the network is in a private home because the scale is small; but what do we do in the case of a large institution, like a university, where many guests are coming and going on a regular basis? Greenpass solved this problem by using delegation to propagate network-access privileges. The wireless network provided a service that allowed regular users to sign temporary SDSI/SPKI delegation certificates that granted network access to guests. This allowed the network to support the dynamic propagation of access—without the hassle or scaling problems of a central authority [GKS$^+$04]. (It also frees up the sysadmins to deal with more important matters!)

However, it's important to point out that the Greenpass project is also using the word "delegation" in a way that's a special case of how we mean it here. In the Greenpass solution, one entity is not acting on behalf of another entity. Because there are two indivisible privileges involved in the delegation process (network access, and the ability to delegate further), it doesn't even make sense to talk about one user acting on behalf of another. When the Greenpass people talk about "delegation," they are primarily interested in its advantages as decentralization: when any regular user in the system can grant privileges to a guest.

Notice that between these two examples—the hospital, and Greenpass—we can see that delegation has two core qualities: decentralization, and limited access to the resources of another user. Both of these qualities are useful even when taken by themselves. If we can provide a framework for *generalized* delegation, we take advantage of both.

But delegation is not only helpful. It's necessary for secure systems. Consider a final example of a security system that does not explicitly offer delegation. At Dartmouth College, as well as many other colleges, administrative tasks are moving on-line. Students update their paper mail address, register for classes, and renew library books via a web portal. But students are people too, and sometimes people run into difficulties. The Internet connection at their home goes down, or they are sick in the hospital, or on the road, or their Internet Service Provider has a firewall that prevents them from accessing this particular web portal. In short, shit happens.

As an undergraduate, several friends have willingly given me passwords so that I could take care of business on their behalf. In each case, the student knew in advance that they would be unable to act on their own behalf due to extraneous circumstances. They would be fined if they took no action, and it was too cumbersome to deal with the administrative bureaucracy. Thus, they forced their own form of delegation into the system—by sharing their secrets, they delegated unrestricted access to their student accounts to me.

All of them hesitated before they shared their passwords. All would have preferred to delegate a small subset of privileges for a short period of time, just enough to accomplish the task. But they shared their passwords because it was the easiest way to get work done. At Dartmouth, it's common knowledge that students share their passwords for just this reason. At the end of the day, users will circumvent security if there is no simple path to do what they need to do.

1.3 The Main Idea

Security systems appear to be moving away from password-based authentication. When computers become faster, a password must be longer to be "strong," but the average user's patience for memorizing long sequences of characters will not grow indefinitely.

Security experts are looking at PKI-based systems as a possible replacement for passwords. But if PKI does not offer a way for users to delegate permissions, users will again force their own form of delegation into the system. PKI advocates may shudder to imagine one user lending another her *private key*, but unless there's an easy-to-understand way to delegate rights, that might be her only option.

With this in mind, a usable PKI system should admit a means of delegation that is secure and generalizable. This delegation mechanism should be decentralized, to avoid the cost and hassle of a central authority with a central database that needs to keep track of every user. It should empower Alice to unambiguously specify a limited subset of her privileges to pass to Bob, so that he can take care of business on her behalf.

This thesis develops a system—web-based delegated authentication via proxy certificates—that meets these criteria. We equip web browsers with the ability to issue proxy certificates carrying security policies, and the ability to pass proxy certificates to a web server via client-side SSL. The credentials encoded in these proxy certificates can be passed to server-side scripts, which can then make their own security decisions.

Pushing the delegation process below the application layer makes this solution generalizable. If Alice wants to delegate privileges to Bob, she does not have to visit each one of her web applications and explicitly delegate to Bob. She can issue one proxy certificate encoded with the policies for all these applications. Also, developers can build secure web applications on top of this web server, and take advantage of delegated authentication without implementing it themselves.

That last point is essential, because implementing delegation is not simply a matter of modifying some certificate verification code and calling it a day. Delegation also changes how we think about how permissions are propagated, which is explored in the next section.

1.4 Horizontal Considerations of Delegation

In traditional applications, one user has one identity. Delegation allows us to drop that paradigm. Instead of considering how one user can delegate to several others, we consider how several users can delegate to one. This is what I refer to as "horizontal" displacement of delegation. Imagine three real-world scenarios:

1. Five friends and I are hungry for hamburgers. Actually, I usually prefer pizza, but for the purposes of discussing delegation, hamburgers are better. There's no sense in all of us going out to get them, so each friend gives me some cash. I go to the nearest fast-food place to order the hamburgers.

   At most fast-food restaurants, the cashier would get very frustrated very quickly if I insisted on making six distinct orders, packaged in six distinct bags, with six distinct piles of change. Instead, I will have to sort out for myself who gets which burger and how much change each person gets. And she'll expect me to act as one user engaging in one transaction on behalf of six people. One user has a **collective identity**.

2. To get a room on-campus at Dartmouth College, we have a protocol called "Room Draw." This involves 1,000 students who gather in a gym at once. Each is assigned a unique number. These numbers are called over a loudspeaker with all of the eloquence of a greased-pig relay. When your number is called, you and any roommates you wish to live with approach one of a dozen desks, identify yourselves, and choose an unoccupied room.

   Earlier, I lamented a Dartmouth web application that forced users to share their passwords to delegate access. But in this case, the authorities did provide a mechanism for delegation. A student can sign a statement with her semi-secret student ID that delegates a friend to draw a room on her behalf, and the friend can present this statement at the desk.

   Two friends once asked me to pick a double room for the two of them. So both delegated room draw rights to me. And I had to assert my own identity as well. In this case, I gained the role of one user acting as three users simultaneously. One user has **multiple identities**.

The differences between the first and the second examples are subtle, and not entirely exclusive. In both models, multiple users grant capabilities to a single target. In the collective identity case, those capabilities are interpreted independent of who granted them, but in the multiple identity case, those capabilities are explicitly tied to an identity.

Perhaps examples will help to clarify. In the first case, the cashier only cares about the "amount" of rights (i.e., money) I had. This idea is also seen in democracies. Consider a Republican delegate sent to the United States Electoral College. Or consider a stock broker who buys stocks on-line on behalf of a group of investors. There's an implicit understanding that even though there's only one entity involved in the transaction, that individual acts on behalf of a large constituency. Separately delegated identities are collected into a single privilege.

In the second case, the two students have two distinct sets of rights. Consider a lawyer who speaks on behalf of multiple clients, as a representative of each client. Unlike the first case, the number of identities is irrelevant. One person is asserting multiple exclusive identities. Separately delegated identities maintain their individuality, and are held simultaneously.

There's also a third case:

3. Suppose Alice and Bob, who are both traveling abroad, each give me power of attorney. While away, Alice calls me on the phone and asks me to sue Bob on her behalf. There's clearly a conflict of interest. I should definitely not show up in Court representing both the defendant and the plaintiff—I may be able to represent one, or the other, but not both.

   This scenario can be characterized as an instance of the ever-popular "Chinese Wall" model (see Chapter 15, [KPS02]). This is the **exclusive identity** scenario. Multiple identities should never be asserted at once, contrary to the first two examples.

These three situations could not come up in traditional authentication schemes, where one user maps to one identity. Delegation changes our assumptions about roles and identities, and introduces new possibilities that have practical applications, both in the analog world and in the digital one. Any implementation of delegated authentication should take advantage of these possibilities.

1.5 Limitations

Before we move from an abstract discussion of delegation to an actual implementation, we should clarify what problems this thesis is *not* trying to solve.

There are some tricky semantics involved in how applications should deal with a user with multiple identities. We recognize that these semantics are difficult. And different applications will have different ways of handling such users. But the scope of this thesis does not extend beyond the lowest level of multiple-identity authentication. We will simply provide a framework for application developers to deal with multiple identities.

The goal of this thesis is not to explore how we can specify delegation in policy statements. There are many standardized languages, such as XACML, that allow security professionals to precisely specify authentication and access control rules. They are a useful tool for security administrators. But we assume that most users will not care for such a fine level of access control when they are determining which privileges to delegate in a proxy certificate. We need a simple mechanism for *users* to describe this delegation. This simplicity should be reflected in the server-side directives as well. A set of rudimentary directives—based loosely on the directives defined for access control in Apache—will be enough to demonstrate the possibilities of delegation. For the purposes of this thesis, that's what we care for. There will be an opportunity for policy languages to contribute to this system, but they will be discussed much later and only in the abstract.

Although there will be some discussion of user interface, we are not particularly concerned with user interface design. That's not to say user interface is unimportant—an intuitive interface is a user-trusted interface, and thus crucial to this system. But I plan to steer away from some of the more tricky interface problems (like "How does Bob give Alice his public key?", "How does a web application re-organize its layout for a multiple-identity user?", etc.). I believe that time and feedback yield the best insights into interface design, yet I have neither the time nor the manpower to conduct this sort of survey research. The reader should expect some hand waving in that direction.

The objective of this thesis is primarily to demonstrate that this security feature—delegated authentication in the web server–web browser interaction—is a feasible add-on to current systems.

1.6 One Last Admission

There is a non-technical motive behind our strategy for delegation, and that motive should be explicit as well. An implicit motivation in any security system is politics. For example, many browsers base their certificate trust decisions on the oligarchy model, meaning that the vendor designates a group of trusted authorities that ships with the browser (Chapter 15, [KPS02]). Look at the list of trusted CA certificates in Mozilla Firefox, or in Microsoft Internet Explorer. Why are those root certificates trusted? Who decides what the trusted root certificates are? They are trusted by the vendors, not by the users.

> "[S]uch are the evils of oligarchy; and there may be many other evils... Then oligarchy, or the form of government in which the rulers are elected for their wealth, may now be dismissed"
>
> —Book VIII, *The Republic* [Pla00].

There is a lot of political theory behind the design of X.509 and how it propagates authority. Engineers have politics too. Indeed, certificate theory raises questions about authority and trust that have been wrestled with since the Greeks. We have no hope of setting those issues to rest here. But the reader should be aware that one motivation behind this thesis is the political ideal that Alice and Bob should have the authority to delegate their own privileges. This thesis seeks to empower them with that authority.

Chapter 2

The Strategy

This chapter presents a high-level “human-computer” protocol specification for limited delegation with web applications.

2.1 Deployment

Suppose an institution (a university, government, or corporation) wishes to deploy such a system for delegated authentication. In such a system, there are three types of entities involved.

There is an authority which controls the CA. There are end-users (like Alice and Bob). There are also two types of end-users: end-users within the system, who have a certificate signed by the CA, and end-users outside the system, who may be strangers to the CA. And finally, there are service providers—web sites that Alice visits. Alice has privileges on these web sites, and may wish to delegate these privileges to Bob.

Notice that these parties may or may not be exclusive. The institution may control the CA, or the service provider, or both, or neither.

To introduce delegated authentication into this system, we require two pieces of software.

- End-users will need a web browser plug-in to issue and manage delegation certificates.
- Service providers will need a module to verify proxy certificates during client-side SSL, and interpret the proxy policy.

The web browser plug-in is easily distributed. Most modern browsers have a system for installing such a plug-in automatically by clicking a link, and users are accustomed to installing such add-ons. Mozilla Firefox, for example, has “extensions,” and a similar plug-in could be written for Internet Explorer.

The module for the service provider will be more cumbersome to install, and will vary depending on the particular web server software. Apache servers provide support for configurable modules that are dynamically loaded on start-up. The SSL-handling code is one such Apache module. So a service provider with Apache would have to replace the SSL-handling module with one equipped to handle proxy certificates, and restart Apache.

2.2 From the Point of View of Alice & Bob

The following list is a step-by-step tour of Alice and Bob’s role in the protocol, though it does not exhaustively explore all the possible features of delegation discussed in the previous chapter.

1. Alice asks Bob to check her web-based e-mail account while she’s out of the country. He agrees.
2. Bob e-mails Alice his **public key certificate**.
3. Alice inspects this certificate, then uses it as the basis for a new certificate for Bob: a **proxy certificate** signed by Alice’s secret key. The proxy certificate contains Bob’s name and public key, and explicitly authorizes Bob to log into Alice’s e-mail account and read e-mail. It contains no statement that authorizes him to send e-mail.
4. Alice e-mails the proxy certificate to Bob, along with the trusted certificate chain attesting to her own public key certificate.
5. Bob installs this certificate in his web browser. When he logs in to the web-based e-mail account via an SSL session, he presents the proxy certificate issued by Alice.
6. The web server logs the fact that Bob logged in with Alice’s identity. The server’s environment variables indicate to the web application that Bob has permission to read but not send Alice’s e-mail. If it is a well-designed application, it will check these permissions and act accordingly.

2.3 Notes on the Protocol

There are a few parts of this protocol that we have deliberately left vague. They evoke broad questions in authentication and delegation. Future technologies may be able to answer these questions better than current technologies can. Here, we explore our solutions, as well as other possible solutions.

2.3.1 Proxy Certificates

This use of proxy certificates is non-standard. Recall from section 1.1 that according to the standard, a temporary keypair must be generated at the same time as the proxy certificate, and the certificate must testify to the public key [TWE$^+$04]. In our method, proxy certificates will always testify to a public key previously generated. There are many advantages to this strategy, and the system sketched out in this chapter would likely be infeasible without this departure from the standard.

For one, this strategy for generating proxy certificates does not require Alice to send a new temporary private key to Bob. In fact, no secret information is exchanged between them. Only their public key certificates are transmitted. As long as Alice can verify Bob's certificate, this will be secure.

Secondly, in a world with proxy certificates, the arguments for lots of keypairs will not be appealing for users. If Bob has two keypairs with different sets of privileges, then he could make things easier for himself by delegating all his private keys onto one, and just keep that one. In a human-usable delegation system, simplicity should be a major goal, and a single keypair for each keystore is much more simple.

Lastly, notice that we can repeat the delegation process with other users each delegating their own privileges to Bob's public key. This allows Bob to obtain a grab bag of certificates, all with the same name and public key, but corresponding to different delegated identities. This is intentional, and the design decision will become obvious when we must handle multiple certificate chains in the same session to validate multiple identities.

2.3.2 Distributing Certificates

In the example above, Alice issues a certificate to Bob, and sends that certificate to Bob via e-mail. This means that she has to export the certificate to a file and attach it to an e-mail. Then Bob has to download the e-mail, and import the certificate into his browser.

Alice and Bob could transmit certificates to each other more easily if we had some additional infrastructure. Alice could retrieve Bob's certificate from some public X.500-style or LDAP-style directory system. Or she could post Bob's new proxy certificate to a third-party server, for him to pick up later. Or, if Alice and Bob wanted to keep it secret that she had delegated access to him, we might want to define some protocol for Alice and Bob to share certificates without a third party.

An institution may define its own certificate distribution mechanism, but we will leave these mechanisms for others to design and implement. E-mail is good enough for our purposes.

2.3.3 The Rejection of Authority

In our protocol, Alice issues the certificate herself. She then transmits her certificate to Bob on her own, or via a disinterested third party like a public certificate database.

But this isn't the only way to solve the same problem. The delegation of privileges could also be handled through the CA, or through the service provider. The CA could provide a web-based service wherein Alice can submit signed delegation requests, and the CA would be the one that actually issued the certificate. Or, alternatively, Alice could log into the service provider's web application, and tell that application explicitly that Bob has permission to speak on her behalf. This second method bypasses certificates completely.

These solutions have disadvantages.

First, they both put a burden on a central server. As we discussed in Section 1.2, decentralization is a boon to all parties—the process is less complicated for Alice, and it relieves the responsibility of the server. Consider banks that charge ridiculously large "service fees" for printing an on-line bank statements, in the hopes that customers will just print the bank statement out at home. They want users to take care of business on their own.

Second, there may or may not be privacy concerns. Suppose that Alice delegates access to

Bob for emergencies—she may not want anyone to know about this delegation until he steps forward. Direct correspondence between Alice and Bob allows them to keep this (relatively) secret.

Lastly, there may be scalability issues. For example, there may be multiple service providers (web applications). It would be inefficient for each service provider to keep its own list of who Alice has delegated her privileges. And in the approach where the CA issued the delegation certificate, the CA delegation service would have to change to accommodate every new service provider. But if Alice and Bob issue their certificates to each other directly, then the scalability issues aren't so bad—Alice only has to keep track of which delegation-enabled service-providers she has visited.

These other solutions might also be problematic for political reasons. For example, the CA delegation-signing service may try to charge users a fee. Our solution empowers Alice to keep control of her own identity, and encourages good security practices by end users.

Chapter 3

The Tools

After the high-level overview in Chapter 2, some implementation questions tumble out. How does Alice encode a limited set of privileges in the proxy certificate, for multiple applications that may not be aware of one another? Furthermore, how does the web application define the privileges that it may require, and tell Alice about them? And if Bob has multiple delegated identities, how does he send multiple certificate chains to the server?

These questions will be gradually answered as we discuss the necessary collection of software tools, and precisely what these tools need to accomplish.

1. Alice needs a way to issue proxy certificates.
2. The web application needs a way to tell Alice what permissions she can delegate, so that Alice can select which of these permissions to encode in her proxy certificate.
3. Bob's web browser needs to be able to send multiple chains of proxy certificates in an SSL session.
4. Bob must be able to choose which identities he would like to assert. (In this example, he has a choice between his own identity "Bob" and his delegated identity "Alice.")
5. The web server must understand proxy certificates, and be equipped to deal with multiple chains of them.

This chapter outlines a blueprint for building these tools into a standard suite of web applications and web standards—namely Mozilla Firefox and the Apache web server—by leveraging X.509 proxy certificates and the SSL/TLS protocol.

3.1 Proxy Certificates

X.509 is the reigning certificate standard. And better yet, the X.509 proxy certificate is standardized well.

Some may ask why I didn't choose a different certificate format, like SDSI/SPKI (discussed in Section 5.1.1) or X.509 attribute certificates.

The X.509 proxy certificate has some minor structural advantages. Because it contains so much auxiliary information, this makes for more comprehensive server logs on who Bob is and which identities he's assuming. It's also explicitly intended for delegation, as opposed to attribute certificates, which can handle more general attributes.

But most importantly, there is moderate support for X.509 proxy certificates. The OpenSSL libraries can issue and verify them by simply making the right API calls[Lev05]. The same support is not behind X.509 attribute certificates. And it certainly cannot be said for SDSI/SPKI, for which there is little support in major applications, with the exception of some closed environments. X.509 proxy certificates piggy-back off standard X.509 certificates. The major difference is that proxy certificates can be signed by end entities, and a proxy certificate must define a critical `ProxyCertInfo` extension [TWE+04].

3.1.1 X.509 Certificates and Revocation

Because proxy certificates are usually short-lived, researchers often wave their hands at the problem of revocation, as the potential for damage is reduced greatly by the certificate's expiration date. In some projects, delegation is used to make the revocation process obsolete—the proxy certificates expire more quickly than a certificate revocation list (CRL) could be issued.

But even though it's infeasible for every end entity to issue its own CRL, CRL-based revocation is possible. We could give the CAs the responsibility to build CRLs for both CA-signed certificates, and proxy certificates signed by children of the CA. If Alice created a proxy certificate for Bob, then needed it revoked, she could tell her CA about the bad certificate. The CA would revoke it on her behalf on its next CRL. Also, the CA will not know about the proxy certificate until Alice tells her about it, so the CA's power to revoke its children's proxy certificates is unlikely to be abused.

This revocation scheme consolidates all the CRLs with the CAs. The certificate verifier would have to retrieve the CA's CRL to verify Alice's certificate anyway, so this would not require any modifications to the CRL distribution process. For this to work, Alice needs to keep a copy of every delegation certificate she signs until it expires, so that she can tell the CA to revoke it if needed.

In this implementation, I do not handle revocation. But Alice does keep a delegation history of all certificates she issues, so that the software could be modified to implement this feature in the future.

3.1.2 X.509 Certificates and Access Control Attributes

In order to give a user the ability to pick and choose which applications the delegate can use on their behalf, we allow them to define attributes in terms of a service (the URL of a service provider) and an ability (an arbitrary string expression).

This will become clearer with an example. Suppose Alice wants to delegate to Bob the ability to read her mail from her web-based "www.mail.gov" account, and to edit and post on her blog "www.aliceblog.com." So she gives him the following attributes:

www.mail.gov: read
www.aliceblog.com: edit post

In other words, the attributes will consist of a list of permissions, and each list will be tied to a service provider. Proxy certificates have a space allotted to specify such a list of permissions as a `policy OCTET-STRING` in the `ProxyCertInfo` extension [TWE$^+$04].

This list of attributes constitutes a list of what Bob can access. Alice must explicitly name each service provider that Bob can interact with on her behalf. This allows each service provider to define its own set of privileges at any granularity.

This scheme is devised to avoid the security hole that would arise if two service providers use the same privilege name. For example, this prevents Alice from confusing "edit" privileges on "mail.gov" with "edit" privileges on "aliceblog.com." Because the privileges are tied to the URL of the service provider, they remain unique. Observe that we are solving the name collision problem by leveraging somebody else's infrastructure that has *already* solved the problem. A URL address is unambiguous, so we can use it to name the service provider and bind that name to the privilege set it uses.

At least, that's how we'll think about the situation. URL addresses are not unambiguous. First, some web pages use server farms, with one address mapping to multiple servers. Secondly, an adversary can spoof a URL. But for our purposes, these nuances are irrelevant. When we use the URL in this context, we are not assuming a trusted relationship with the service provider at that address. The URL simply allows us to differentiate between different service providers and the privilege sets that they offer. As long as all the servers in a farm interpret the same set of privileges, it doesn't matter that the URL points to more than one machine. And an adversary gains no advantage by impersonating the server. So we can say that the URL is unambiguous—and understand that the exceptions to this statement are inconsequential.

There is one question that I have still not addressed. How do service providers notify Alice of their set of privilege names? I answer this when I discuss the modifications to the web server (see Section 3.4).

3.1.3 Exclusive Identity

It would also be useful to define a policy statement that obligates Bob to assert Alice's identity singularly. This would cover the "exclusive identity" model. Alice would specify in the certificate that as long as Bob is using this proxy certificate, he *cannot* attempt to take on multiple identities. Since web servers, by default, expect the user to assert only a single identity, this "normal behavior" should not be difficult to enforce. For this application, we do not implement it.

3.2 The Mozilla Framework

The Mozilla Framework is an open source software development framework. The most famous application to come out of it is the Firefox web browser. It strives to be cross-platform, programming-language-independent, and locality-independent. Mozilla has useful properties that make it easier to modify—most notably the availability of the source code. I will explain its high-level code architecture here so that when I need to discuss the mechanics of the implementation, it will be easier for the reader to understand how the pieces fit together. Readers who are already familiar with the Mozilla Framework and component object models may skip this section.

Mozilla's view of the world consists of two core ideas that make it easy to modify: XPCOM (the Cross-Platform Component Object Model) and the XUL GUI[1].

In XUL, the web page is part of the GUI—or maybe the GUI is a part of the web page. Either way, both the GUI and the web page are expressed as one large XML-based document. The entire GUI is parsed as an XML tree of nodes that can be manipulated by the standard functions of the document object model. Javascript code can be included in the document to respond to user events. All in all, the GUI is essentially a dynamic HTML page that's interpreted into native GUI calls. Developers can dynamically add elements to this GUI in the same way that they would add such elements to an HTML page—with Javascript that adds nodes to the XML tree at runtime [Dea06].

XPCOM is the system for organizing all of the software libraries underlying Mozilla. In XPCOM, a central component manager keeps track of a number of exclusive, encapsulated components that each implement a well-modularized set of functions. These components can be written in any language for which XPCOM language bindings are defined, including Python and Java—but are typically written in C++ or Javascript. The methods and attributes of an XPCOM component can only be accessed by defining a public interface through a second language called XPIDL (long-wise, that's the Cross-Platform Interface Description Language). This interface then allows the component to be used as an object in any XPCOM-supported language. You can define an interface in XPIDL, then implement it with several different components (possibly in different programming languages), that satisfy the interface in different ways. For example, the `nsISocketProvider` interface is implemented by one component that handles SSL sockets, and another component that handles TLS sockets. See Chapter 8 of [BKO$^+$02] for more information.

The components are managed by a component manager, which keeps a hash table with entries for each component. The entries of the hash table are indexed by human-readable URIs called contract IDs. Each entry also contains a universally unique identifier (UUID) as a sequence of integers, and the memory location of a constructor for this component. When one component wants to use another, it gives the component manager the URI, and the component manager gives it back an object.

So why do we care? If we could overwrite an entry in the hash table, we could replace any native Firefox component with our own component. As long as the custom component implements all the XPIDL-defined functions, the rest of the Mozilla Framework will

[1]The acronym XUL isn't long-wise significant; what's important is that it's pronounced "Zool" and is an homage to Ghostbusters.

treat it exactly like the native component. If this thesis were a novel, we would call this foreshadowing.

3.2.1 The Extensions

The Mozilla Framework provides a simple mechanism for installing "extensions" from over a network. The modification for proxy certificates should be packaged as such an extension. After all, few users would be willing to download a custom browser with modified source code to use delegated authentication. Modifying the source of the Mozilla Framework is not an option.

We create new XPCOM components that handle proxy certificates. We can then develop a GUI for this application as simply an (XUL) web page that the browser visits locally. The Javascript in this web page is trusted because it is loaded locally, so it can interact with the local XPCOM components, and by extension, the proxy certificate library.

For this project, our extension can be divided into three pieces:

An interface to allow Alice to **issue proxy certificates**.

> At first glance, there's no reason for this to be built into the browser—it could easily be a stand-alone application. The reason it's in the web browser is not for Alice's benefit, but for the service provider's benefit. As we will soon see (Section 3.4), each service provider will propagate the set of privilege names that it defines by talking to this extension. Then the user interface can show Alice a list of delegation-enabled service providers she's visited, as well as the privileges she can delegate for them.

A back-end database to **manage proxy certificates** issued to Bob.

> Network Security Services (NSS), the cryptographic library underlying the Mozilla Framework, does not behave properly around proxy certificates. At best, it's schizophrenic. NSS will often accept them at first—but as soon as it realizes that the proxy certificates have been signed by an end entity, it may immediately trash them. We simply need a database that can handle proxy certificates properly, and will safely store them outside of NSS.

A way for Bob to use his proxy certificate(s) in **client-side SSL authentication**.

> This part of the extension will be responsible for getting the proxy certificates to the server during an SSL session. This is more difficult than it sounds, because this will

require slight changes to the SSL protocol.

3.3 SSL/TLS

SSL/TLS is the ubiquitous protocol for secure communication on the Internet[2]. It consists of three essential pieces:

1. The "Hello" phase, wherein the client and server initiate communication.
2. The "Handshake" phase, wherein the server and client exchange information using a chosen asymmetric-key algorithm, with the goal of establishing a session secret. This information may optionally include a certificate exchange, and the server and client may optionally verify each other's certificates before they agree to connect.
3. The "Application" phase, wherein we can now send data across the network encrypted and MAC'd with the session secret via your favorite symmetric-key algorithm [DA99].

To allow delegation into this protocol, we only need change a narrow segment of the client behavior during the Handshake phase. We can leave the rest of the protocol alone.

The Handshake phase changes because SSL/TLS expects the client to transmit no more than one chain of certificates. In this chain, each certificate testifies to the public key of the keypair that signed the certificate that came before it [DA99]. But for delegation, the client might need to transmit several certificate chains, with one chain corresponding to each delegated identity. To make it work, we transmit the certificate chains for each delegated identity in serial, and assert that the first proxy certificate in each chain must testify to the same public key. Figures 3.1 and 3.2 demonstrate the change in the protocol when we add multiple certificate chains. We do not permit Bob to use two different keypairs in the same session.

The "one public key" rule gives us an easy way to distinguish between certificate chains—when the validator sees a certificate in the chain that contains the same public key as the first certificate, this is the beginning of a new chain. As an additional bonus, this ensures that legacy certificate-validation code (applications that don't know about multiple-identity

[2]SSL/TLS is a suite of several different standardized protocols. All these protocols are just variations on the same high-level ideas, though. For our purposes, they're interchangeable.

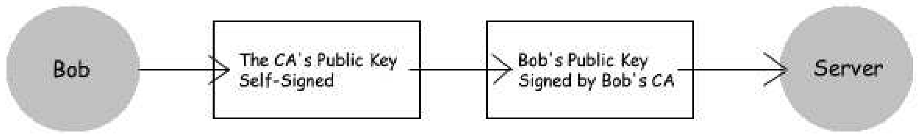

Bob
The CA's Public Key Self-Signed
Bob's Public Key Signed by Bob's CA
Server

Figure 3.1: Passing client certificates to the server by the TLS 1.0 standard. Notice that Bob's public key certificate is sent first, while the CA certificate that testifies to it comes afterwards. (The self-signed CA certificate is optional. We include it here to enhance the illustration.)

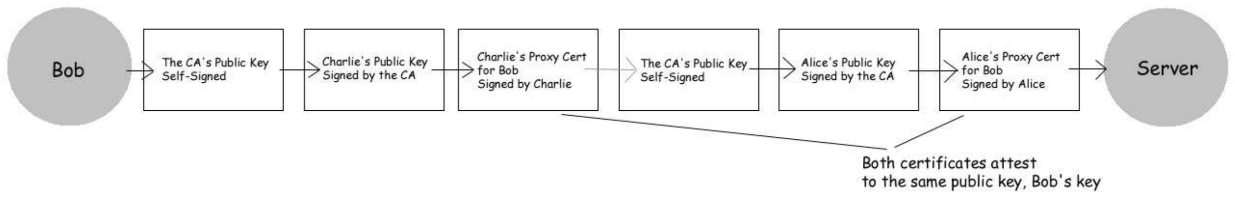

Bob
The CA's Public Key Self-Signed
Charlie's Public Key Signed by the CA
Charlie's Proxy Cert for Bob Signed by Charlie
The CA's Public Key Self-Signed
Alice's Public Key Signed by the CA
Alice's Proxy Cert for Bob Signed by Alice
Server
Both certificates attest to the same public key, Bob's key

Figure 3.2: Passing multiple certificate chains to the server. As before, each certificate in the same chain testifies to the one sent before it. The gray arrow represents the point where a traditional TLS server would register an error, because Bob did not sign the CA's public key certificate.

delegation) will reject any user that tries to assert multiple delegated identities. A certificate chain, after all, should not have a cycle.

From a theory standpoint, the idea that "public key" is a unique identifier of the user is also a cleaner way to think about PKI. The SDSI/SPKI certificate model makes this observation elegantly. The security of public-key crypto-systems implicitly depends on the assumption that public keys are unique. If two users had the same public key, then their cryptographic operations would be indistinguishable [EFL+99].

Bob may assert multiple delegated identities in a single SSL/TLS session, but only one keypair. This isn't as restrictive as it seems. Because Bob can issue proxy certificates on the fly, and one proxy certificate can be used to issue another, he can delegate all his identities to the same keypair[3].

[3]The `ProxyCertInfo` extension has a path length constraint which may interfere with this solution by restricting the length of the certificate chain. If Bob had multiple private keys, he may not be able to delegate all his privileges to one because it would break path length constraints. We wave our hands at this

3.4 The Apache Web Server

I won't repeat the advantages of open source software, especially because in the case of Apache, we will just modify the source code and recompile. Fortunately, we only have to modify mod_ssl, which can hook into the Apache server from a dynamically loaded library. We can easily distribute this library to server administrators to enable delegation.

There are exactly two additions we need to make. First, we need to modify the certificate validation code to accept proxy certificates and to be able to recognize when the client is sending multiple chains of proxy certificates.

Recognizing the proxy certificates is simple—that functionality comes standard with OpenSSL, and OpenSSL is the cryptographic library underlying Apache. The validation of multiple chains of certificates is not so easy to implement, because it requires changes to both Apache and OpenSSL. It has also some tricky semantics. What happens if the client sends two certificate chains, and only one of them is valid? On one hand, the SSL protocols imply that if the server sees an invalid client certificate, it should notify the client by sending an error message, and should cut short the SSL session. But it seems more natural for the server to accept the valid chain, and quietly fail to grant the privileges specified in the invalid chain. In this implementation, the tie goes to the specification—if one certificate chain is invalid, the whole authentication should fail. This will make it clearer from a user interface perspective that something has gone wrong.

3.4.1 Access Control Directives

We also define some additional directives in the Apache configuration files. These directives will tell Apache *how* to respond to proxy certificates.

A legacy server will only accept a single certificate chain. So the modified server will, by default, only accept a single identity. It will consider multiple identities only when it sees special directives in the configuration files.

These new directives are as follows:

- `SSLCollectiveIdentity`:

problem—if it happens, then we blame it on Bob for using his keys incorrectly.

This directive tells the server to use the collective identity model. The server will accept multiple certificate chains, one for each identity. Privileges will be counted and totaled.

To ensure the same privilege is not counted twice, we ensure that if there are multiple certificate chains, no two chains derive authority from the same end entity[4].

- `SSLMultipleIdentities`:

  This directive tells the server to use the multiple identity model. Again, the server will accept multiple certificate chains. The server will additionally keep track of who has delegated which of their privileges.

  Notice that this directive simply records more information than `SSLCollectiveIdentity`. This information may be necessary for user-level scripts to make security decisions.

- `SSLExclusiveIdentity`:

  This directive tells the server to accept only the first identity. In truth, it is the default case. But because these directives are interpreted on a per-directory basis, this directive is useful for overriding the `SSLMultipleIdentities` or `SSLCollectiveIdentity` directives in a parent directory.

A server connection environment variable will be set.

```
_SERVER[ ``SSL_DELEGATED_IDENTITIES'' ] := STRING
```

`STRING` is an ASCII character string encoded as a Lisp s-expression. It will contain the common name of each certificate in an identity asserted by Bob. Obviously, this encoding doesn't work if common names have parentheses. So if the server sees a common name with a parenthesis, it simply refuses to grant this identity.

((*name1*) (*name2*) (*name3*))

Application-level scripts can read this environment variable, and thus find out easily what identities Bob has.

[4]How do we ensure that Alice doesn't have two keypairs, with two different names, and use them to both to assert two identities, thus giving her two votes? We can alleviate the problem by checking both the public key and distinguished name fields of each identity for duplication. This is not nearly an adequate solution, and is a potential security leak if not taken into account. Hopefully, for most applications, this issue will not be a problem.

We will also have directives for interpreting the `policy` field of the `ProxyCertInfo` extension.

- `SSLRequirePrivilege` ***name ``Description''***:

  The *name* must be an alphabetic character string (with no white space), and must not conflict with any other privilege names. It will specifying the name of a privilege as it appears in the proxy certificate policy. If this directive is used, then Bob will be allowed access in the current directory subtree iff he has this privilege. If either `SSLMultipleIdentities` or `SSLCollectiveIdentity` is given as well, Bob will need to have this privilege for every identity that he tries to assert, or he will be rejected.

  The *Description* portion of the directive will be used for the propagation of the privilege set, which I will discuss shortly.

- `SSLRequestPrivilege` ***name ``Description''***:

  This directive is similar to the `SSLRequirePrivilege` directive. But in this case, Bob can access the directory whether or not he has the described privilege. The directive is used to specify privileges that are not used to restrict access at the server level, but are exposed via the environment variables anyway. (An application-level script might use this privilege as part of an XACML-based decision request.)

Notice that with the last two directives, every server can define a set of privileges. After the server verifies Bob's proxy certificate chain(s), it will check the `policy` field of each `ProxyCertInfo` extension in the chain, and grant Bob the privileges encoded in it. It will then set a connection environment variable like so:

```
_SERVER[ ``SSL_DELEGATED_PRIVILEGES'' ] := STRING
```

Again, `STRING` will be a Lisp s-expression. For every privilege that Bob is granted, it will add that privilege to this list. How this list is formatted will depend on how the server is configured to deal with multiple identities.

In the default case, suppose Bob is granted privileges `priv1`, `priv2`, and `priv3`. Then the list will look like:

(*priv1 priv2 priv3*)

If `SSLCollectiveIdentity` is enabled, it will look like:

((*priv1 num1*) (*priv2 num2*) (*priv3 num3*))

where *numX* is the number of identities that granted Bob privilege *privX*.

If `SSLMultipleIdentities` is enabled, each sub-list will start with the identity name, followed by the privileges granted by that end entity.

(((*name1*) *priv1 priv2*) ((*name2*) *priv3*))

3.5 Privileges in the Grand Scheme

To determine its set of privileges, the server parses all of its access files for the `SSLRequire-Privilege` and `SSLRequestPrivilege` directives described above, then builds a set of `( name , description )` ordered pairs of privileges. They are keyed by the name, so if the same name appears twice, one pair is rejected. All servers implicitly define the reserved pair `( all , ‘‘All privileges’’ )`. When Alice visits the server, the server gives her a cookie containing the privileges defined as Lisp s-expressions. The Firefox extension can then read these cookies, and add them to its list of (service provider, privilege) pairs. When Alice wants to sign a proxy certificate for Bob, the user interface will provide her with a list of all the servers that she's ever visited that support this form of delegation.

When Bob wishes to assert the privileges granted by Alice, he authenticates with this certificate in a client-side SSL session. He will pass the server the entire certificate chain, so that the server will see both Alice's end-entity certificate, and the proxy certificate she signed for Bob. The server then interprets the policy in the proxy certificate, and records in an environment variable that Alice granted Bob those privileges.

The reader should take note that this still leaves the web application with a lot of responsibility to make authorization decisions. Our system simply records what Alice has granted in the proxy certificate policy—it makes no claim that Alice had the authority to grant Bob this privilege. An application simply uses the identity and privilege information to make authorization decisions; our directives are part of a server-level system that make it a easier to process and manage this information.

The reader should also be aware that although we've rigorously defined a privilege-propagation system, this system is a sideshow to the main attraction. We are primarily interested in access control with limited delegation. This privilege system is intentionally bare-boned. It is not designed to stand on its own. It is designed with just enough functionality to exercise the power of limited delegation.

Chapter 4

Trials and Tribulations

This chapter could be alternatively be titled "Mis-Adventures in Software Engineering." Here, I describe the implementation of the delegation system.

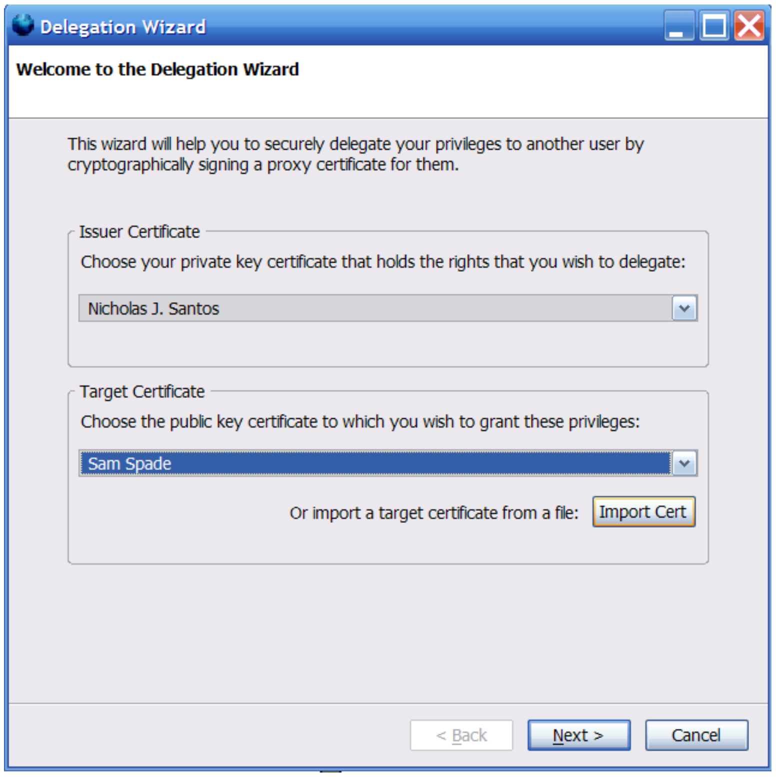
Any description of software implementation has to be especially careful, because it must negotiate two competing interests. It must be thorough enough that the reader could recreate the system, but it must be fleeting enough that the same reader does not feel the slog of months of software design.

This chapter tries to walk that tightrope.

4.1 The Proxy Certificate Firefox Extension

Nicholas Santos has hired Detective Sam Spade to represent him. He would like to sign a proxy certificate for Spade that will delegate all his permissions to Spade for a period of 5 days. (He's trying to get back a jeweled falcon that dates back from the Crusades, you see, and he's asked Spade to reclaim it.) The next five figures illustrate the process he goes through to do this. In between figures, there will be a discussion of how to manipulate and issue proxy certificates with NSS, the Mozilla Framework's cryptography library.

The text does not follow the figures; the figures should be self-explanatory.


Delegation Wizard

Welcome to the Delegation Wizard

This wizard will help you to securely delegate your privileges to another user by cryptographically signing a proxy certificate for them.

Issuer Certificate

Choose your private key certificate that holds the rights that you wish to delegate:

Nicholas J. Santos

Target Certificate

Choose the public key certificate to which you wish to grant these privileges:

Sam Spade

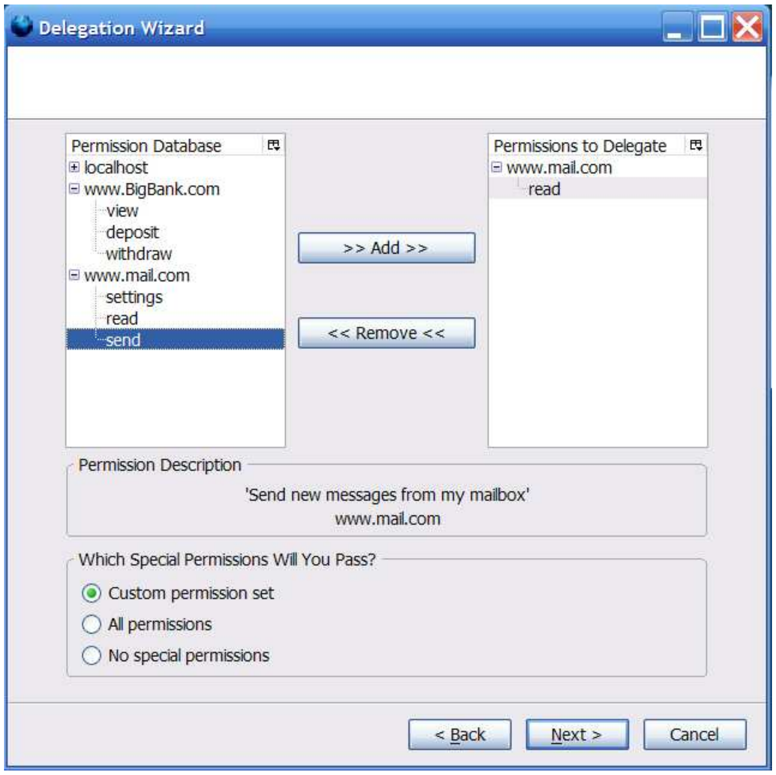
Or import a target certificate from a file:

Figure 4.1: The Delegation Wizard (part 1): Choosing a user certificate (any certificate for which we have the private key) and a target certificate to which its identity is delegated. Perceptive readers may notice that this screenshot does not look quite like Firefox—it is a build of Bon Echo, the alpha version of Firefox 2.0.

4.1.1 Making Room for Proxy Certificates

The `ProxyCertInfo` X.509 extension must be attached to all proxy certificates and marked critical [TWE+04]. By specification, cryptographic libraries must trash any certificate with unrecognized critical extensions [HPFS02]. So before we load any components that handle proxy certificates, the Object Identifier (OID) for the `ProxyCertInfo` extension must be dynamically registered with NSS. RFC 3820 additionally lists a number of OIDs for proxy policy languages that must be understood by any proxy certificate implementation. Those OIDs must be registered as well.

The `ProxyCertInfo` extension contains three fields. The optional `pCPathLenConstraint` describes the depth of the cert chain below this one. The required `policyLanguage` is an OID for a policy language. The optional `policy` is, as noted earlier, the designated place for issuers to record policy info on what permissions they're delegating.



Permission Description

'Send new messages from my mailbox'
www.mail.com

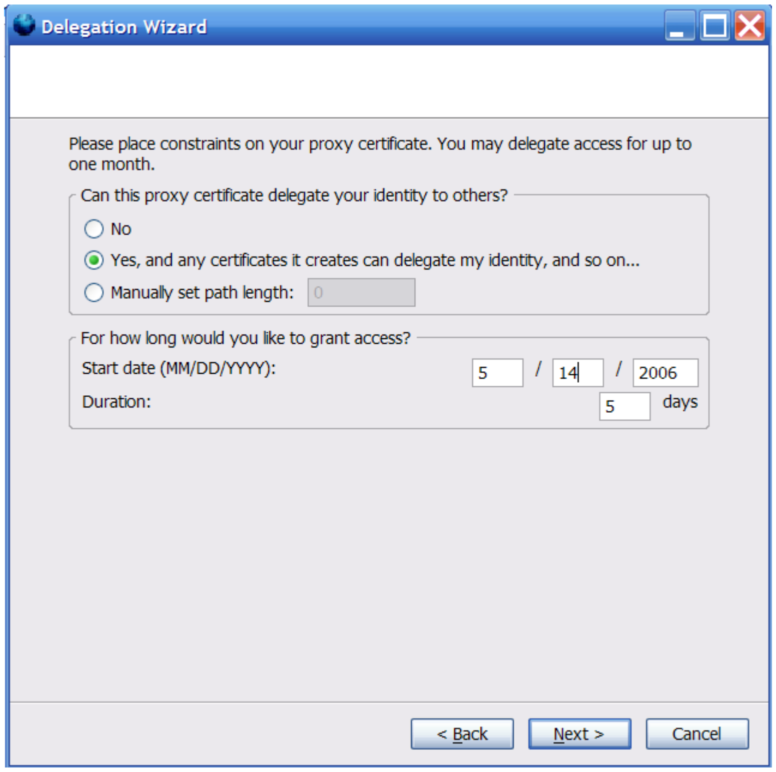
Which Special Permissions Will You Pass?

[✓ Custom permission set]
All permissions
No special permissions

Figure 4.2: The Delegation Wizard (part 2): Choosing permissions to delegate. Each tree organizes the permissions by service provider. The tree on the left is a list of all available permissions; the tree on the right is a list of the permissions that will be delegated. The tree of available permissions is built by iterating through the cookies, and parsing them for the permissions. A routine parser will do.

NSS allows developers to write templates—arrays of constants—that tell the ASN.1 encoder how to encode and decode types. A few wrapper functions and a `ProxyCertInfo` template are required to encode and decode that extension. The Mozilla Framework also has a separate ASN.1 handler that pretty prints ASN.1 sequences for GUIs. A few more functions on top of that object will make the `ProxyCertInfo` extension readable from a dialog box, as shown in figure 4.5.

The reader should be aware that in order to handle proxy certificates adequately, an application needs a lot more infrastructure than this. It needs wrapper objects for the certificates, for their validity periods, and for some certificate-based GUI objects. But these needs can be satisfied by simply copying the existing certificate-handling code, modifying it slightly for proxy certificates (and for publicly exported APIs), and re-compiling it into the extension. It's not academically interesting, and we will say no more of it.



Please place constraints on your proxy certificate. You may delegate access for up to one month.

Can this proxy certificate delegate your identity to others?

No

[✓ Yes, and any certificates it creates can delegate my identity, and so on...]

Manually set path length: 0

For how long would you like to grant access?

Start date (MM/DD/YYYY): 5 / 14 / 2006

Duration: 5 days

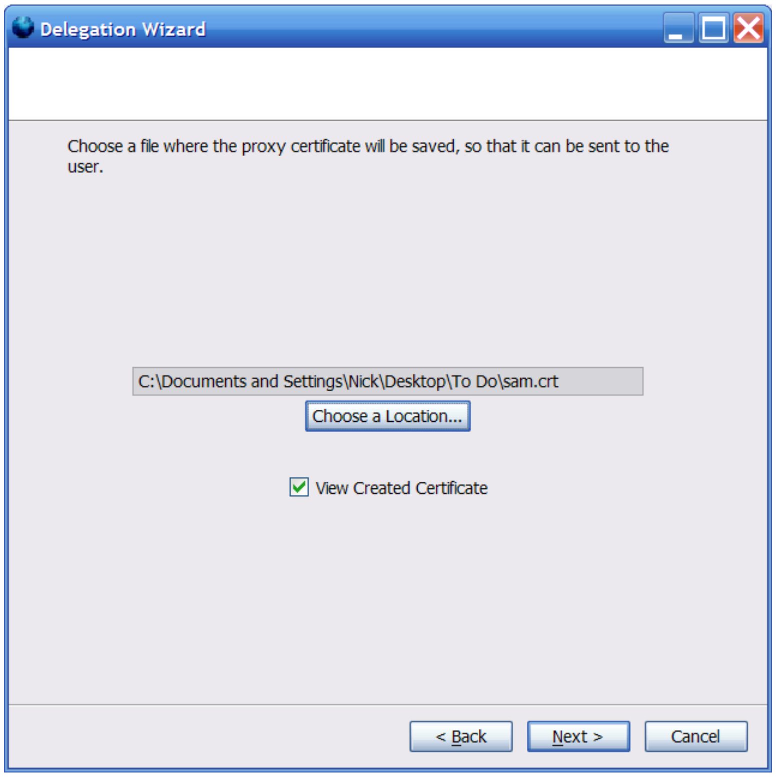
Figure 4.3: The Delegation Wizard (part 3): The constraints page, which allows setting a path length constraint and a validity period.

4.1.2 Issuing Proxy Certificates

To issue a proxy certificate, we need an issuer and a target. The issuer testifies to the private key that will sign the new certificate. The target testifies to the name and public key that will be the subject of the new certificate (see figure 4.1). This information is used to construct a certificate request internally. We call an NSS function to transform this certificate request into an unsigned certificate, at the same time adding an issuer and a validity period.

When creating any certificate—not just proxy certificates—there are standards and there are styles. The first is mandated in writing, the second is mandated by strong social pressure and convention. For standards, it's best to work straight from the source, RFCs 3280 and 3820. For style, I recommend Peter Gutmann's "X.509 Style Guide." A full listing of both these sources is in the bibliography.

For every proxy certificate we issue, we copy the name and public key directly from the target certificate. Per Gutmann's recommendation, we use the time in seconds since the



Choose a file where the proxy certificate will be saved, so that it can be sent to the user.

C:\Documents and Settings\Nick\Desktop\To Do\sam.crt

☑ View Created Certificate

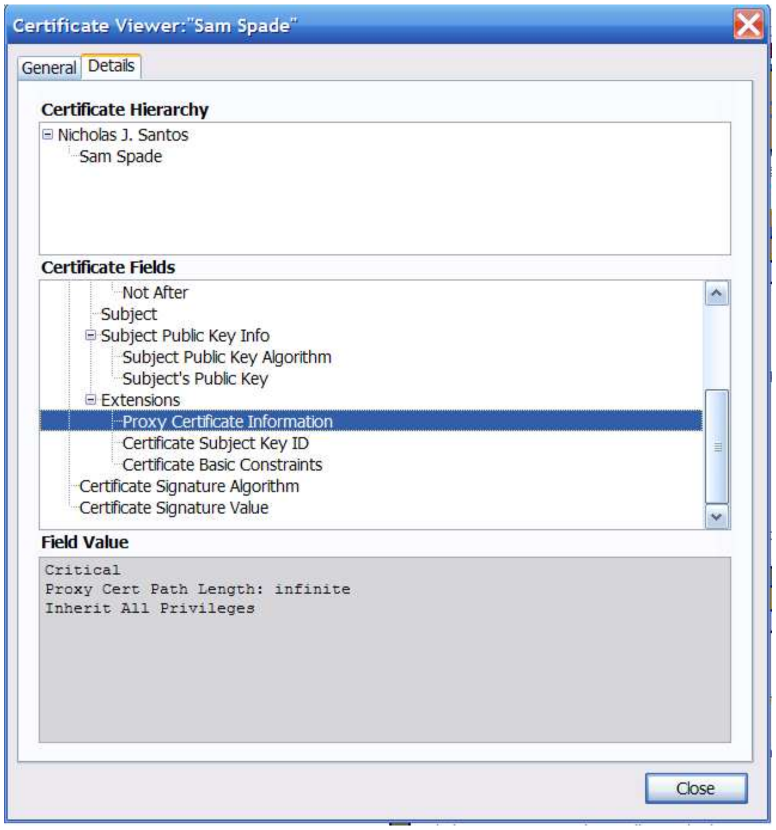
Figure 4.4: The Delegation Wizard (part 4): Saving the certificate to a file, so it can be e-mailed to Detective Spade.

UNIX epoch as the serial number, to ensure unique serial numbers[1].

We additionally attach three or four extensions.

- We have already covered the `ProxyCertInfo` extension, we will say no more of it.
- A `BasicConstraints` extension that indicates that this certificate may *not* be a CA. We include it on the advice of OpenSSL's proxy certificate guide [Lev05].
- A `SubjectKeyIdentifier` extension that contains a SHA-1 hash of the target certificate's DER-encoded public key data.
- The `AuthorityKeyIdentifier` is attached if and only if the issuer certificate has the `SubjectKeyIdentifier` extension. If it does, the key identifier from the issuer is simply copied into the key identifier field of this `AuthorityKeyIdentifier`.

[1]A malicious user could certainly issue multiple certificates with the same serial number by changing the system clock, but this would do no actual damage. It would merely spite the standard.


Certificate Viewer:"Sam Spade"
General
Details
Certificate Hierarchy
Nicholas J. Santos
Sam Spade
Certificate Fields
Not After
Subject
Subject Public Key Info
Subject Public Key Algorithm
Subject's Public Key
Extensions
Proxy Certificate Information
Certificate Subject Key ID
Certificate Basic Constraints
Certificate Signature Algorithm
Certificate Signature Value
Field Value
Close

```
Critical
Proxy Cert Path Length: infinite
Inherit All Privileges
```

Figure 4.5: The Delegation Wizard (part 5): The ASN.1 structure of the new proxy certificate.

The `SubjectKeyIdentifier` and `AuthorityKeyIdentifier` extensions are not really necessary, and may just be another example of redundancy in the X.509 standard. However, they do reinforce the idea that the public key identifier is a better indicator of identity than a X.500 distinguished name, because an identity is only as unique as its keypair. We mainly include this extension for ideological reasons. But we should mention that these extensions made the certificate validation code easier to debug, because the crypto library code that compared two key identifiers was usually much simpler than the crypto library code that compared X.500 names.

NSS did not encode the `AuthorityKeyIdentifier` correctly, so the encoding code was copied, fixed with only minor modification, and recompiled into our library.

Once the extensions are added, the certificate is ready to be signed. A Mozilla XPCOM component provides functionality to log into any PKCS #11 interfaces (cryptographic tokens), asking the user for a password if we need one. Other publicly-exposed NSS functions allow us to get a handle on the private key, and use it to sign the data in a DER encoding.

To my knowledge, this is the first implementation of a proxy certificate issuer with NSS.

4.1.3 Storing Proxy Certificates

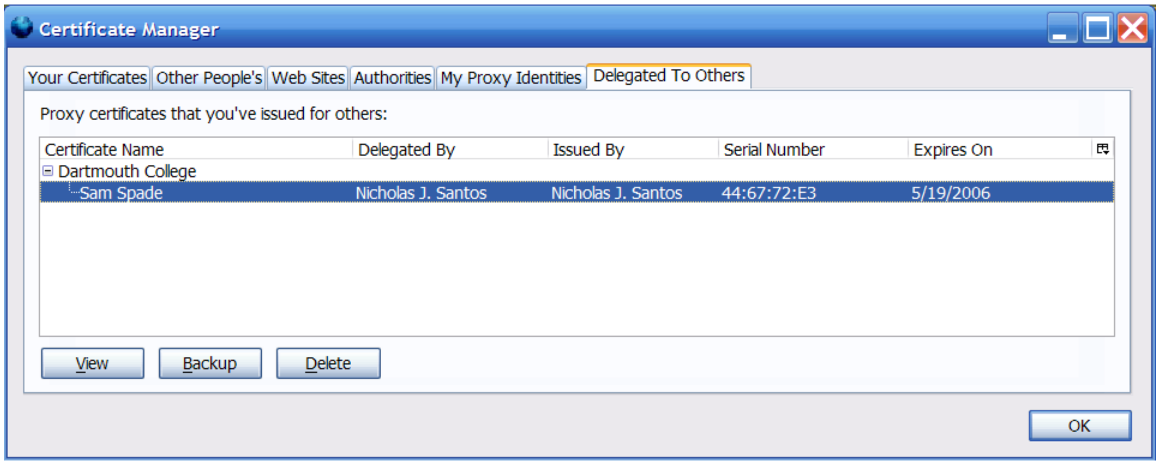
Once we have a DER encoding of the proxy certificate we need a place to store it. NSS is no good, for numerous reasons. It is not aware that end-entities can sign certificates. Furthermore, this Firefox extension may be uninstalled, and if it is, it shouldn't be abandoning proxy certificates in the regular NSS database.

The key insight to storing proxy certificates is that the storage medium does not need to store any secrets, like private keys. It can leave the private keys in the NSS secure storage, and only needs to remember where those private keys are located. The only disadvantage of this method is that the user could delete his private key from the NSS database, thus rendering his proxy certificates useless. Our implementation does not protect against this case; it just blames the user for the problem.

Because proxy certificates do not need to be stored securely, it would have sufficed to keep them in any non-volatile storage mechanism. In our extension, they are simply stored in an SQLite database[2]. The key for each proxy certificate in the database is derived from the serial number and the issuer name. Because issuers *should* issue certificates with unique serial numbers, this key should be unique. Each entry in the database also contains a DER encoding of the certificate, as well as keys to access the issuer certificate, "delegator" certificate, and private key in the regular database. The "delegator" certificate is the last end-entity certificate in the chain ending in this proxy certificate. It names the end entity from which this proxy certificate derives its identity. (In chains with only one proxy certificate, the issuer is the delegator.)

The database can be described in two sections: delegated certificates and the user's proxy certificates. The sections must not be assumed to be mutually exclusive, although they likely will be for most users. The portion for delegated certificates is merely a log file. It tells Alice which proxy certificates she has issued, and allows her to re-export them if she needs to send them again (figure 4.6). The other section of the database holds certificates delegated to the user, certificates for which the user has the private key (figure 4.7). These are the identities that can be used in an SSL session.

[2]SQLite is a open source file-based database engine with a C API that accepts and executes queries in a subset of the SQL language. More information can be found at `http://www.sqlite.org/`.



Proxy certificates that you've issued for others:

Certificate Name	Delegated By	Issued By	Serial Number	Expires On
⊟ Dartmouth College				
Sam Spade	Nicholas J. Santos	Nicholas J. Santos	44:67:72:E3	5/19/2006

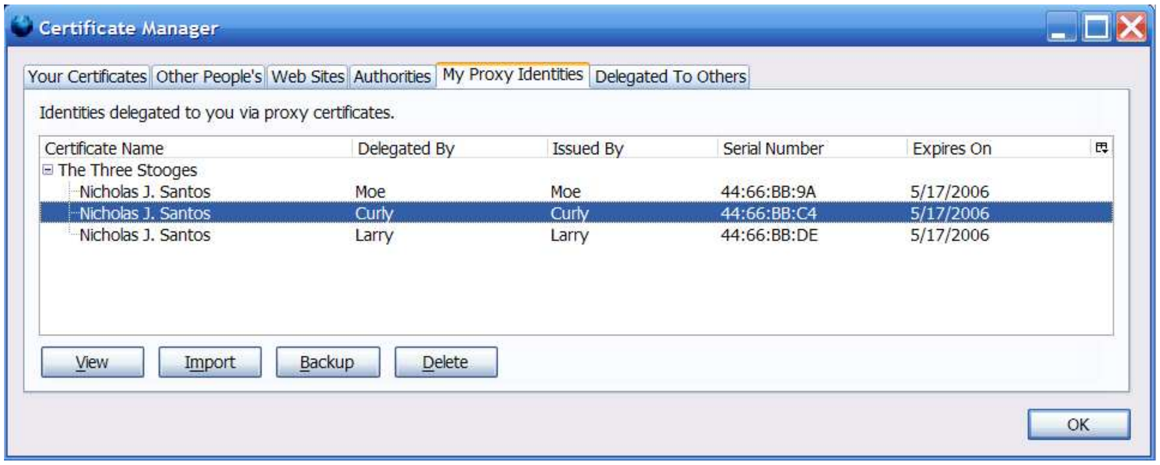
Figure 4.6: Viewing the certificates in the database that have been issued by *this user.*

4.1.4 Using Proxy Certificates

The code to inject proxy certificates into an SSL session performs an interesting acrobatic stunt.

Recall from the original discussion of the Cross-Platform Component Object Model (XPCOM) that the Mozilla Framework keeps all its components in a hash table, hashed by a human-readable contract ID. If we ask the component registrar to load a component with the same contract ID, the registrar will, by default, simply overwrite that entry of the hash table. From reading the source code, this feature seems to be intentional, although it is not documented. But this also means that there is no documentation assuring us that this is a safe mechanism for extension development. We use it cautiously.

This feature allows us to play man-in-the-middle with Firefox's SSL/TLS handling code. First, the XPCOM objects that handle SSL and TLS sessions are registered at a second contract ID. Then, custom SSL/TLS handlers are registered at the first contract IDs, overwriting the entries there. These custom handlers intercept method calls intended for the original handlers, change the passed arguments, and then pass the altered arguments along to the traditional handlers by looking them up at the second contract ID. The same man-in-the-middle game can be played with return values. Thus, we can change the method arguments and return values at will to produce the desired effect. That's the theory—but it's not so simple in practice.



Identities delegated to you via proxy certificates.

Certificate Name	Delegated By	Issued By	Serial Number	Expires On
⊟ The Three Stooges				
Nicholas J. Santos	Moe	Moe	44:66:BB:9A	5/17/2006
Nicholas J. Santos	Curly	Curly	44:66:BB:C4	5/17/2006
Nicholas J. Santos	Larry	Larry	44:66:BB:DE	5/17/2006

Figure 4.7: Viewing the certificates in the database that have been issued to this user. Observe that only fools delegate their privileges to this particular user.

4.1.5 Using Proxy Certificates, The Long Version

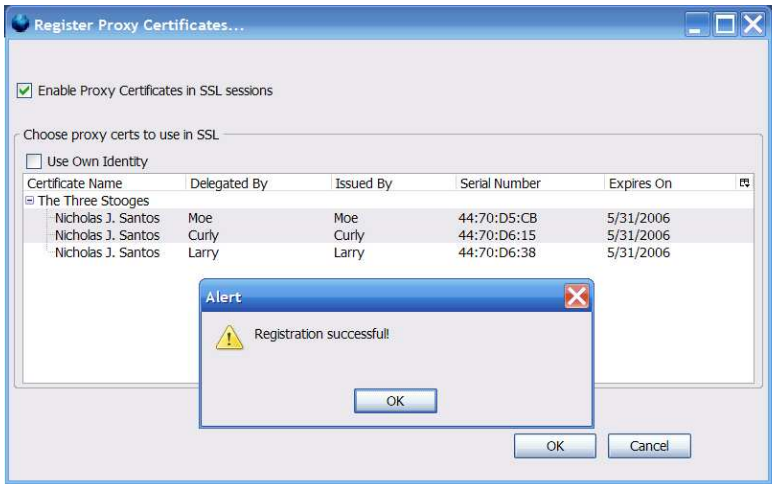
The actual implementation is far more complicated and involves several levels of indirection. The flow can be confusing.

There are custom SSL/TLS handlers that intercept calls to the traditional SSL/TLS handlers in XPCOM. But those handlers turn around and use the pure C NSS libraries to handle the SSL handshake. Our method only allows us to intercept method calls between the object-oriented XPCOM components. Pure C method calls cannot be intercepted by the same trick. So we need to use a different trick.

There is a certain method call to the SSL/TLS handler objects, and an argument of that call contains a function pointer. (More correctly, it provides the first in a long path of pointers that eventually leads to a function pointer deep in the NSS code.) This function pointer refers to a callback function whose responsibility is to retrieve the client certificate for the SSL handshake. (In the NSS documentation, this callback is known as the `ClientAuthDataHook` [NSS05].) We can apply a second man-in-the-middle strategy to *this* function, by changing the function pointer to point to a function of our choosing. NSS calls the custom function when it needs a certificate.

Unfortunately, that's not the end of it. The custom certificate-retrieval callback only returns a single certificate—NSS builds the rest of the chain. Fortunately, there is a way to fool NSS into building an arbitrary chain. NSS stores its certificates in data structures with a lot of redundant information. These data structures contain a DER encoding of the certificate, as well as pre-computed fields so that it doesn't have to decode and re-encode the certificate repeatedly. But there's the rub: it constructs chains based on the values of the pre-computed fields, but the data actually sent across the network is the DER encoding. And it assumes these fields are in-sync. By feeding it out-of-sync data, we can fool NSS to build a certificate chain based on the mock-up certificate fields, and NSS will end up sending arbitrary DER data across the SSL session. This DER data will, by more than coincidence, be the proxy certificates that we intend to transmit.

And that's how an extension can add limited delegation with proxy certificates to Firefox without modifying any existing code.


Register Proxy Certificates...

☑ Enable Proxy Certificates in SSL sessions

Choose proxy certs to use in SSL

☐ Use Own Identity

Certificate Name	Delegated By	Issued By	Serial Number	Expires On
⊟ The Three Stooges				
Nicholas J. Santos	Moe	Moe	44:70:D5:CB	5/31/2006
Nicholas J. Santos	Curly	Curly	44:70:D6:15	5/31/2006
Nicholas J. Santos	Larry	Larry	44:70:D6:38	5/31/2006

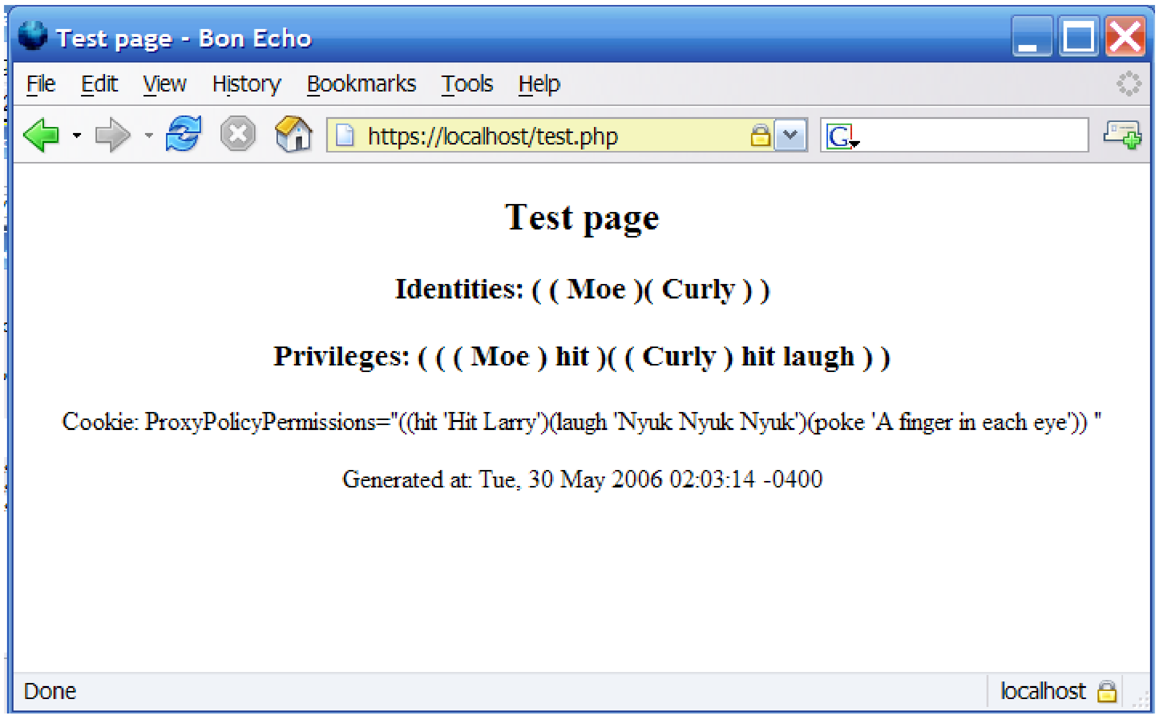
Alert

Registration successful!

Figure 4.8: Registering two proxy certificates to use in an SSL session. The chains for both will be sent in client-side SSL/TLS authentication.

4.2 Apache with Delegation

The Apache codebase is much smaller than the Mozilla codebase, and our application allows its source code to be modified. The changes made to Apache are thus much more lightweight.


Test page - Bon Echo
File Edit View History Bookmarks Tools Help
https://localhost/test.php
Done
localhost

Test page

Identities: ((Moe)(Curly))

Privileges: (((Moe) hit)((Curly) hit laugh))

```
Cookie: ProxyPolicyPermissions="((hit 'Hit Larry')(laugh 'Nyuk Nyuk Nyuk')(poke 'A finger in each eye')) "
```

Generated at: Tue, 30 May 2006 02:03:14 -0400

Figure 4.9: This test page shows the values of environment variables SSL_DELEGATED_IDENTITIES and SSL_DELEGATED_PRIVILEGES.

4.2.1 New Directives

The Apache build system leans heavily on an automated parser generator. A single macro can add a new configuration directive, and define the callback function that will process the arguments to that directive. The new proxy certificate-handling directives defined in section 3.4.1 were designed to take advantage of this existing infrastructure. Adding them is not complicated. The `SSLMultipleIdentities` and `SSLCollectiveIdentity` directives are processed by changing global context variables. The two privilege directives are processed by adding them to a global privilege table, sorted by unique privilege name. For each directory-specific `SSLRequirePrivilege` directive, we take the corresponding directory context structure and give it a pointer to the required entry in the global privilege table.

4.2.2 Privilege Propagation

Apache comes packaged with a module, mod_usertrack, that enables tracking cookies. This module supplies the basis for the code needed to pass the supported privilege set to the client via a cookie. We register a hook function with the main Apache module to get called every time a client connects. When this function gets called, we can iterate through the permission table, encode it in a cookie, and add that cookie to the HTTP reply headers. (Actually, since the same cookie is transmitted each time, and no permissions can be added after the server starts, we just compute this cookie once.)

4.2.3 Certificate Verification

Because there may be multiple chains of proxy certificates in an SSL session, OpenSSL needs to be modified to accept a) proxy certificates, and accept b) multiple chains of them.

Proxy certificate support in OpenSSL is contingent upon the state of a particular environment variable. But the Windows code for reading this environment variable did not appear to be working correctly, so OpenSSL was modified to accept proxy certificates all the time.

Apache lets OpenSSL take care of the standard certificate verification, but sets a callback function to go through the certificates after OpenSSL has verified them, and do any custom verification. The OpenSSL verification function normally stops immediately as soon as it can't find the issuer for a certificate in the chain, then looks in the local store for a trusted chain of issuers. The verification succeeds iff it finds such a chain. If there are multiple chains, OpenSSL accepts the first chain and says that it is satisfied. This appears to be non-standard. We modify the OpenSSL verify function so that after it verifies the first chain, it looks at the first certificate in this chain and the first certificate in the chain of the "unverified" certs. If these two certs match, and the global flag for multiple chains is set, it calls itself recursively on the "unverified" cert chain to verify the other chains.

When the Apache callback function receives the verified certificate chains, it iterates through them again. For each chain, it looks for the end-entity (the first non-proxy certificate) in each chain, and pushes it onto an identity list. It also interprets the `ProxyCertInfo` extension's `policy` field, and determines which privileges are delegated all the way down the chain. The implementation of this part should be self-evident.

Chapter 5

Related Work & Future Work

No thesis is an island. The first part of this chapter will present other work that has influenced this research, and discuss how this research is different. Some of these projects have been mentioned in previous chapters. The second part of this chapter will discuss tangential research topics that we would like to see explored.

5.1 Related Work

5.1.1 SDSI/SPKI and Project Geronimo

SDSI/SPKI is an alternative certificate standard that stresses simplicity over the complication of X.509. It's most relevant to this research because it provides a much more straightforward and simple syntax for the delegation of credentials [EFL$^+$99].

The SDSI/SPKI group at MIT also developed an Apache module and Netscape Communicator plug-in that allowed users to authenticate with the server using SDSI/SPKI certificates. (Project Geronimo was the name of the Apache module.) To accomplish this goal, they developed an entire new protocol on top of HTTP that performed this authentication. In their system, the server notified the client of the permissions it supported by sending an access control list (ACL) to the client during the authentication handshake. (Thus, the protocol handshake was used for authorization as well as authentication.) The client could then use this ACL to determine which certificates to send back [May00]. This ACL solved the problem that we addressed by sending the permissions in cookies.

Our system differs in that it is also concerned with providing the user with an interface to delegate their credentials. We also depend more on the existing protocols and standards (X.509 and SSL/TLS) when we can, rather than creating new ones.

5.1.2 Greenpass

We discussed the Greenpass project in Section 1.2. In that problem, system administrators could maintain a secure wireless network without the hassle of verifying the identity of temporary guests. Regular users could delegate access to the network to their guests. This made the network more manageable and usable from both the administrative and end-user standpoints.

In order to sign these delegation certificates, both the regular user and her guest would visit a web site on a virtual private network. This site provided an interface wherein the regular user could verify the guest and create the certificate, and the guest could import the new certificate into her browser. Neither had to install new software; they only needed to run a trusted Java applet [GKS$^+$04].

Notice that the Greenpass project and our project ran into a similar problem: how does Alice transmit a delegation certificate to Bob? We could circumnavigate this problem by using public e-mail. Because the guests in Greenpass did not have access to the network, they accomplished the same task with the assistance of an internal web server.

5.1.3 Distributed Systems

As discussed in Section 1.2, delegation offers a decentralized way to propagate privileges. It can also be used to delegate a limited set of privileges for a very limited time to a less trustworthy key. For these reasons, people working in distributed systems love delegation. They use it as a lightweight mechanism for granting privileges to temporary processes. It's lightweight because it doesn't require a central authority, and no new identities need to be created.

The Grid created proxy certificates to take advantage of these features of delegation [WFK$^+$04].

And here at Dartmouth, Jon Howell specifically extended Lampson's access control calculus to include delegation, so that he could use formal semantics to analyze distributed systems that lacked a central authority [HK99].

We mention this work because it has driven a lot of the standards and formal models behind delegation, even though it applies delegation to different problems.

5.2 Future Work

5.2.1 A Delegated Server

In this research, we assumed that only clients could hold proxy certificates. But what about service providers?

There are plenty of applications where this could be useful. The most obvious example is that of web server farms. Each server in the farm can have its own identity and public/private keypair, and additionally hold a proxy certificate delegated from a chief server. Thus, they can all assert the same identity. These proxy certificates could even grant different privileges to different servers in the farm, depending on the abilities of the server, and the physical security of its location. This is similar to what the Grid currently does [WFK+04].

Delegation with multiple identities may offer advantages to servers as well. For example, suppose the financial aid office at Faber College decides that they would like to make the lives of students easier by creating a web application where students can take care of all their financial aid paperwork at once. They might even out-source this web application to a third-party web server.

Clearly, such a web site would require students to submit sensitive information. The students would like to be assured that the web server speaks on behalf of Faber College. But this server should also speak on behalf of the Federal Department of Education because it processes federal financial aid forms. There is no CA that can speak on behalf of both institutions, but if the server can transmit two chains of certificates rooted at both institutions' CAs, then the problem is solved.

5.2.2 Proxy Policy Languages

The language for encoding privileges in the `policy` field of the `ProxyCertInfo` extension was admittedly a hack of a language. As mentioned many times before, it was designed only to exercise the potential of our proxy certificates. But there's still the open problem of what *should* be in the proxy policy field.

Ideally, the Apache server should not interpret the proxy certificate policy at all. It should grab all the certificates—multiple chains of them if that's what Bob sent—and whisk them off to a centralized security server. This central server would be responsible for the security decision, and it would send a response to the Apache server that told it what privileges to grant Bob.

That seems to be the popular idea among security professionals. But there's still the client side. How does Alice's web browser encode the proxy policy? Should Alice's browser be responsible for encoding that policy in the cutting-edge policy language, or should she send it off to a central policy-encoder? How does a web server notify its clients of the privilege set it supports? Is there a better solution than our mis-use of cookies?

Scout Sinclair brought up the point that we can keep the information encoded in the proxy policy private, by encrypting it server-specific parts of it with the service provider's public key. So Alice doesn't have to disclose to Bob (or anyone else) exactly which privileges she gave him.

5.2.3 Deployment

In a password-based system, it is difficult to gauge how often people share passwords to delegate access. We would like to see a social experiment where our system of proxy certificates is actually deployed in a closed environment, to see how often people actually delegate access. It would be easy to monitor whether the system was being used, by having each service provider count how many times it authenticated a user who held a proxy certificate.

Chapter 6

Conclusions

Users are already accustomed to the idea of delegation. We are accustomed to speaking on behalf of others, and appointing others to speak on our behalf. On an everyday basis, we act on behalf of our organizations, and on behalf of our employers. We create living wills to delegate others to act on our behalf when we are unable to do so, and we delegate lawyers to speak for us in a legal setting.

Thus, Alice expects to be able to use delegation in her on-line interactions. Delegation will help Alice to do the work that she needs to do, in the way that she expects to do it. In this way, delegation promotes stronger security, better security practices, and enhanced usability.

This thesis has explored the rigorous particulars of how this delegation is used, and how this delegation could feasibly be implemented using current technologies, without breaking current protocols. My hope is that this implementation will serve as a model for future developers who wish to build delegation into browser and web server interaction, and will help to bring PKI-based delegation into mainstream security practices.

Bibliography

[BKO$^+$02] David Boswell, Brian King, Ian Oeschger, Pete Collins, and Eric Murphy. *Creating Applications with Mozilla.* O'Reilly, September 2002. Retrieved on-line from `http://books.mozdev.org/index.html`.

[DA99] T. Dierks and C. Allen. TLS protocol version 1.0. January 1999. RFC2246.

[Dea06] Neil Deakin. XUL tutorial. February 2006. Retrieved on March 11, 2006 from `http://xulplanet.com/tutorials/xultu/`.

[EFL$^+$99] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. SPKI certificate theory. September 1999. RFC 2693.

[GKS$^+$04] Nicholas C. Goffee, Sung Hoon Kim, Sean Smith, Punch Taylor, Meiyuan Zhao, and John Marchesini. Greenpass: Decentralized, pki-based authorization for wireless lans. *3rd Annual PKI Research and Development Workshop Proceedings*, pages 26–41, 2004. Retrieved from `http://www.cs.dartmouth.edu/~sws/pubs/greenpass04.pdf`.

[glo06] About the globus toolkit. 2006. Retrieved from `http://www.globus.org/toolkit/about.html`.

[HK99] Jon Howell and David Kotz. An access control calculus for spanning administrative domains. 1999. Technical Report PCS-TR99-361. Retrieved on-line from `ftp://ftp.cs.dartmouth.edu/TR/TR99-361.pdf`.

[HPFS02] R. Housley, W. Polk, W. Ford, and D. Solo. Internet X.509 Public Key Infrastructure certificate and certificate revocation list (CRL) profile. April 2002. RFC 3280.

[KPS02] Charlie Kaufman, Radia Perlman, and Mike Speciner. *Network Security: PRIVATE Communication in a PUBLIC World.* Prentice Hall PTR, second edition, 2002.

[Lev05] Richard Levitte. *HOWTO Proxy Certificates*, May 2005. Retrieved on March 11, 2006 from `http://www.openssl.org/docs/HOWTO/proxy_certificates.txt`.

[May00] Andrew J. Maywah. An implementation of a secure web client using spki/sdsi certificates. May 2000. Retrieved on-line from `http://theory.lcs.mit.edu/~cis/theses/maywah-masters.ps`.

[MS05] J. Marchesini and S. W. Smith. Shemp: Secure hardware enhanced myproxy. *Proceedings of Third Annual Conference on Privacy, Security and Trust*, October 2005. Retrieved from `http://www.cs.dartmouth.edu/~sws/pubs/ms05b.pdf`.

[NSS05] *Network Security Services (NSS)*, May 2005. Retrieved on March 11, 2006 from `http://www.mozilla.org/projects/security/pki/nss/`.

[Pla00] Plato. *The Republic.* The Internet Classics Archive, 2000. Translated by Benjamin Jowett. Retrieved on-line at `http://classics.mit.edu/Plato/republic.html`.

[TWE+04] S. Tuecke, V. Welch, D. Engert, L. Pearlman, and M. Thompson. Internet X.509 Public Key Infrastructure (PKI) proxy certificate profile. June 2004. RFC 3820.

[WFK+04] Von Welch, Ian Foster, Carl Kesselman, Olle Mulmo, Laura Pearlman, Steven Tuecke, Jarek Gawor, Sam Meder, and Frank Siebenlist. X.509 proxy certificates for dynamic delegation. In *3rd Annual PKI Research and Development Workshop*, 2004.