Limited Delegation for Client-Side SSL*

Nicholas Santos
nicholas.j.santos@gmail.com

Sean W. Smith
sws@cs.dartmouth.edu

Department of Computer Science
Dartmouth College

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. In the real world, end-users find this feature rather useful. However, security officers find it infuriating: by sharing her password, Alice gives *all* of her privileges to Bob, who then becomes indistinguishable from her. As enterprises move to PKI for client authentication, such secret sharing becomes impractical. Although security officers appreciate this, end-users may likely be frustrated, because this more secure approach to authentication and authorization prevents their ad hoc but reasonable delegation. In this paper, we present a solution that satisfies users as well as security officers: using X.509 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-based applications. We validate this design with an SSL-based prototype: an extension for the Mozilla Firefox Web browser and a module for the Apache Web server that allow them to handle multiple chains of these certificates.

1 Introduction

In real-world situations, users often want to temporarily delegate some of their privileges to others, for reasons that are often rather legitimate. Most legacy computer systems implicitly tie a set of privileges to a password and thus make delegation surprisingly easy. 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. Like it or not, users are accustomed to this paradigm.

For many reasons, security experts promote PKI as a replacement for passwords. Our own university has rolled out an X.509 identity PKI to over 75% of the user population, and has migrated its Web-based applications from legacy passwords to client-side SSL for user identification and authentication. However, we fear that if PKI does not offer a way for users to delegate permissions for the scenarios they feel are reasonable, 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.[1] Hence, in order to be usable in many real-world enterprises, client-side PKI authentication needs *a secure, generalizable way to allow for delegation.* This delegation mechanism should be *decentralized,* to avoid the cost and hassle of enterprise-wide or even application-specific databases that need to keep track of every user and every privilege. (That would negate many of the reasons for PKI in the first place.)

In this paper, we develop a system—Web-based delegated authentication via proxy certificates—that empowers 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. 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. We equip Web servers with the ability to pass the credentials encoded in these proxy certificates 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 applica-

*This work was supported in part by the NSF, under grant CNS-0448499. The views and conclusions do not necessarily represent those of the sponsors. A preliminary version of this work appeared as the technical report [13]. The first author is now affiliated with Google.

[1] Indeed, we've already seen this happen.

tions on top of this Web server, and take advantage of delegated authentication without implementing it themselves.

This Paper. Section 2 discusses the high-level goals of our system. Section 3 discusses the PKI framework we used. Section 4 discusses our design. Section 5 discusses our prototype. Section 6 discusses related work. Section 7 concludes with some directions for future work.

2 Goal

We're considering an enterprise with a standard X.509 identity PKI for its users. We consider two classes of entities: end users (like Alice and Bob), and service providers (Web sites that Alice visits). The service providers follow the PKI gospel and use client-side SSL to identify and authenticate users.

Alice has privileges on these Web sites, and may wish to delegate some of these privileges to Bob. To support this action, we need three things. For the PKI, we need a format for a *delegation certificate*, a digitally signed statement from Alice giving Bob some rights. For the end users, we need a Web browser plug-in to issue and manage delegation certificates. Finally, for the service providers, we need a server module to verify delegation certificates during client-side SSL, and interpret the delegation appropriately.

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 delegation certificates.

We imagine a typical end user scenario might work as follows. Alice asks Bob to check her Web-based e-mail account while she's out of the country. He agrees. Bob e-mails Alice his *public key certificate*. Alice inspects this certificate, then uses it as the basis for a new certificate for Bob: a delegation certificate signed by Alice's secret key. This new 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, nor to log into the university record system and view her grades. Alice e-mails this certificate to Bob, along with the certificate chain attesting to her own public key certificate. 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 delegation certificate issued by Alice. 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.

Rejected Options. 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 enterprise's CA or through the service provider. For example, the CA could provide a Web-based service; Alice submits authenticated delegation requests, and the CA then issues 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 other approaches have disadvantages. First, they both put a burden on a central server. Decentralization is a boon to all parties—the process is less complicated for Alice, and it relieves the server of the responsibility. 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, they may have privacy problems. 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 arrangement (relatively) secret. Lastly, these approaches may have scalability problems. For example, it would be inefficient for each service provider to keep its own list of the parties to whom Alice has delegated privileges. In the approach where the CA issued the delegation certificate, the CA delegation service would have to change to accommodate every new service

provider and every new privilege that service might provide. 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.

Orthogonal Issues. Before we move on, we should clarify what problems we're *not* trying to solve.

Once we give Alice the ability to delegate her privileges to Bob, Bob may want the ability to act on behalf of two people at once. He may want to read both his mail and Alice's mail at the same time—in other words, he may want to assert multiple identities. There are some tricky semantics involved in how applications should deal with a user with multiple delegated identities. We recognize that these semantics are difficult. And different applications will have different ways of handling such users. But the scope of this project does not extend beyond the lowest level of multiple-identity authentication. We will, however, provide a framework for application developers to deal with multiple identities.

Additionally, the goal of this project 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 project, that's what we care for.

3 Delegation Certificates

As Section 2 described, we need a format for "delegation certificates." We chose X.509 proxy certificates.

SDSI-SPKI is attractive because it provides a much more straightforward and simple syntax for the delegation of credentials [3]. However, it's an X.509 world, and that's what the standard infrastructure supports. Prior experience in our lab (e.g., [4]) suggested that swimming against the current is not productive.

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 [6]. 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. But part of its goal was 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 a 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 [15]. 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 for the length of a "temporary" period of time [14]. This flexibility is intentional, because simplicity is of the essence. On the Grid, these proxy certificates can be issued to dynamically cre-

ated processes without requiring the approval of a CA.

The X.509 proxy certificate offers numerous advantages for our scheme. Because it contains so much auxiliary information, the server can keep comprehensive server logs on who Bob is and which identities he's assuming. It's also explicitly intended for delegation, as opposed to X.509 attribute certificates, which can handle more general attributes. But most importantly, current tools actually contain support for X.509 proxy certificates. The OpenSSL libraries can issue and verify them [8]. 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 technical differences are that proxy certificates can be signed by end entities, and a proxy certificate must define a critical `ProxyCertInfo` extension [14].

4 Our Design

To achieve the vision of Section 2, we need several things. Alice needs a way to issue proxy certificates. 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. Bob's Web browser needs to be able to send multiple chains of proxy certificates in an SSL session. 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.") The Web server must understand proxy certificates, and be equipped to deal with multiple chains of them.

This section explains the design of these tools for a suite of Web applications and Web standards: X.509 proxy certificates (Section 4.1), Mozilla Firefox (Section 4.2), SSL/TLS (Section 4.3), and the Apache Web server (Section 4.4).

4.1 Non-standard Proxy Certificates

For our project, we decided to depart from the standard that a proxy certificate must testify to the public key of a *temporary* keypair generated exclusively for that certificate [14]. Instead, in our system, proxy certificates will testify to an existing public key, and for which previous certificates—an identity certificate, and perhaps other proxy certificates—exist.

This departure from the standard gave us several advantages. It 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 our application scenarios, having lots of temporary keypairs will not be appealing for users. 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 could be useful in scenarios when Bob needs to represent more than one party in a service request authenticated via client-side SSL—which, by design, allows Bob to prove knowledge of only one private key. (The first author actually has done this in a process that has not yet made it to the Web: Dartmouth's on-campus housing auction. Two friends wished to share a room with each other. Neither could make it to the event, so they both delegated to the author, who then made a selection representing both of them.)

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 early 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. For this project, we take this approach.

Privilege 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 attributes "`www.mail.gov`: read" and "`www.aliceblog.com`: edit post." In other words, the attributes will consist of a list of permis-

sions, 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 [14].

This list of attributes constitutes a list of privileges granted to Bob. 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. We tie privileges to service providers to avoid the trouble 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. In effect, we solve the name collision problem by leveraging somebody else's infrastructure that has *already* solved the problem.

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 orthogonal. 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.

We have not yet answered the question: How do service providers notify Alice of their set of privilege names? We will do that in Section 4.4 below.

4.2 The Browser

The Mozilla Framework is an open source software development framework. The most famous (current) application to come out of it is the Firefox Web browser. The framework strives to be cross-platform, programming-language-independent, and locality-independent. The framework also has useful properties that make it easier to modify—most notably the availability of the source code.

The Framework. First, we quickly review Mozilla's high-level code architecture to help the reader to understand how we modified Firefox.

Mozilla's organizes its code via the *XPCOM (the Cross-Platform Component Object Model).* 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* (longwise, 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. One 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 [1] 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.

This structure is relevant to our project. 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 treat it exactly like the native component.

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.

We create new XPCOM components that handle proxy certificates. We then develop a GUI for this application to interact with the local XPCOM components, and by extension, the proxy certificate library. In this way, our extension can be divided into three pieces.

First, we need 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 4.4), 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.

Secondly, we need 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.

Finally, we need 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.

4.3 SSL/TLS

SSL/TLS is the ubiquitous protocol for secure communication on the Internet[2]. It consists of three essential pieces. In the "Hello" phase, the client and server initiate communication. In the "Handshake" phase, 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. Finally, in the "Application" phase, we can now send data across the network encrypted and MAC'd with the session secret via our favorite symmetric-key algorithm [2]. To incorporate 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 [2]. But for delegation, the client might need to transmit several certificate chains, with one chain corresponding to each delegated identity. To make this work, we have the client 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 1 and 2 demonstrate the change in the protocol when we add multiple certificate chains[3]. 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 bottom of a new chain. As an additional bonus, this ensures that legacy certificate-validation code (applications that don't know about multiple-identity 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 [3].

4.4 The Web Server

For the Web server, we focused on Apache, which is both open-source and market-dominant. For Apache, 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.

Code Additions. 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, the cryptographic library underlying Apache. The validation of multiple chains

[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.

[3] TLS protocol extensions (RFC 4366) could make this change more graceful by allowing the client to explicitly ask the server to accept multiple certificate chains. These standards are recent, and were not available at the implementation phase of this project.

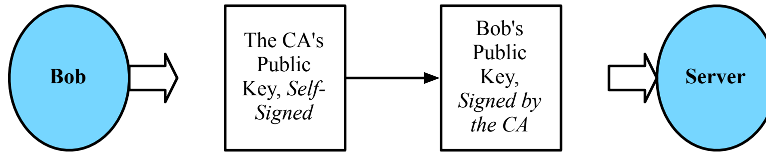

Figure 1: Passing client certificates to the server by the TLS 1.0 standard. Notice that Bob's public key certificate is sent first, while the CA certificate that testifies to it comes afterwards. (The self-signed CA certificate is optional. We include it here to enhance the illustration.)

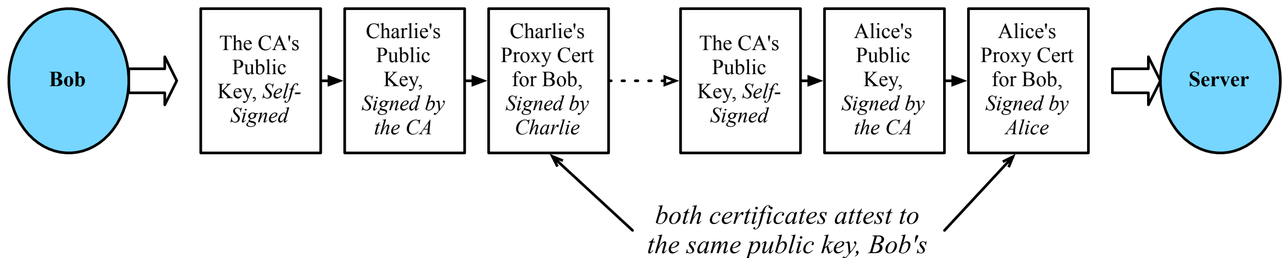

Figure 2: Passing multiple certificate chains to the server. As before, each certificate in the same chain testifies to the one sent before it. The dashed arrow represents the point where a traditional TLS server would register an error, because Bob did not sign the CA's public key certificate.

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.

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 a special directive in the configuration files.

These new directives are as follows.

The `SSLMultipleIdentities` directive tells the server to allow a user to assert multiple identities at once. The server will accept multiple certificate chains, one for each identity. And for each user, it will keep track of the list of privileges delegated by that user.[4] 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.

The `SSLExclusiveIdentity` 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` directive in a parent directory.

A server connection environment variable

```
_SERVER[ ''SSL_DELEGATED_IDENTITIES'' ]
```

will be set to `STRING`, 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. Application-level scripts can read this environment variable, and thus find out easily which identities Bob has.

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

[4] This is the simple-minded way to enumerate sets of privileges delegated by a set of users. There are more specialized ways to model multiple simultaneous identities. We've explored this a bit elsewhere [13].

The SSLRequirePrivilege directive takes a *name* and a *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 SSLMultipleIdentities 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 we will discuss shortly.

The SSLRequestPrivilege directive also takes a *name* and a *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 these 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 _SERVER[''SSL_DELEGATED_PRIVILEGES''] to be STRING, a Lisp s-expression. Bob's privileges will be stored as a list of lists. Each sub-list will start with the identity name (the user delegating to Bob), followed by the privileges granted by that end entity.

Privileges in the Grand Scheme. To determine its set of privileges, the server parses all of its access files for the SSLRequirePrivilege 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 can use 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.

5 Our Prototype

To illustrate the functionality of our prototype, we'll consider an example. 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 five days. (He's trying to get back a jeweled falcon, and he's asked Spade to negotiate on his behalf.) The next five figures illustrate the process he goes through to do this.

We will also discuss how to manipulate and issue proxy certificates with NSS, the Mozilla Framework's cryptography library, and to understand them with Apache. The point of this discussion is to examine the particular successes and pitfalls of our implementation, so that the reader has an idea of what it would take to reproduce such a system.

5.1 The Mozilla Extension

Making Room for Proxy Certificates. The ProxyCertInfo X.509 extension must be attached to all proxy certificates and marked critical [14]. By specification, cryptographic libraries must trash any certificate with unrecognized critical extensions [6]. 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

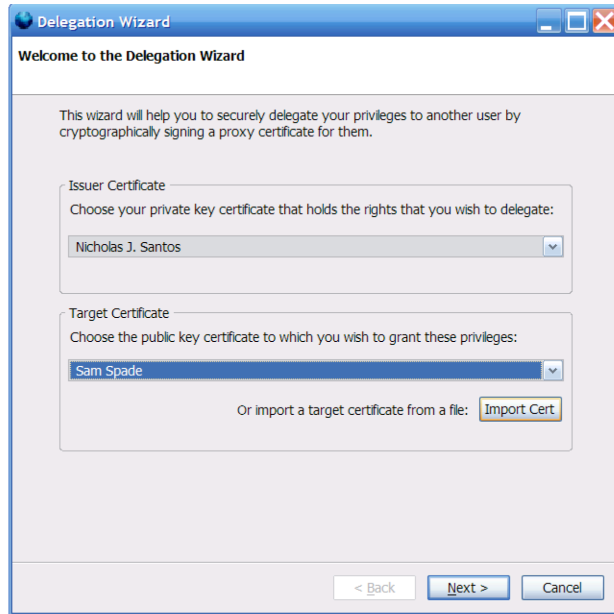

Figure 3: Using our Delegation Wizard (part 1): Choosing a user certificate (any certificate for which we have the private key) and a target certificate to which its identity is delegated.

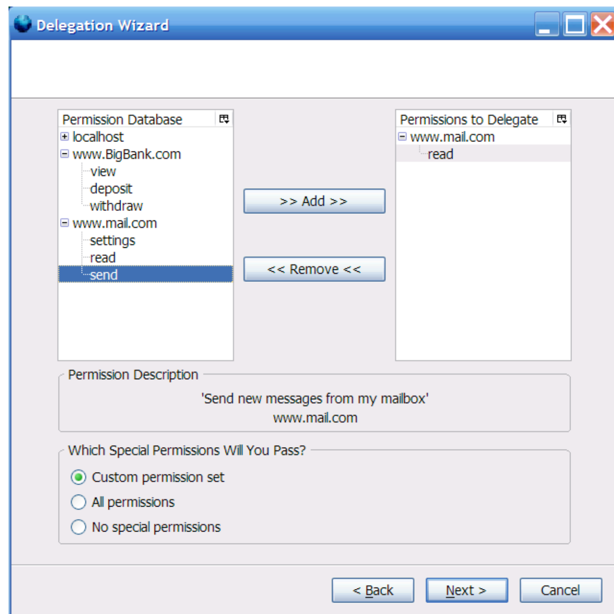

Figure 4: Using our Delegation Wizard (part 2): Choosing permissions to delegate. Each tree organizes the permissions by service provider. The tree on the left is a list of all available permissions; the tree on the right is a list of the permissions that will be delegated. The tree of available permissions is built by iterating through the cookies, and parsing them for the permissions.

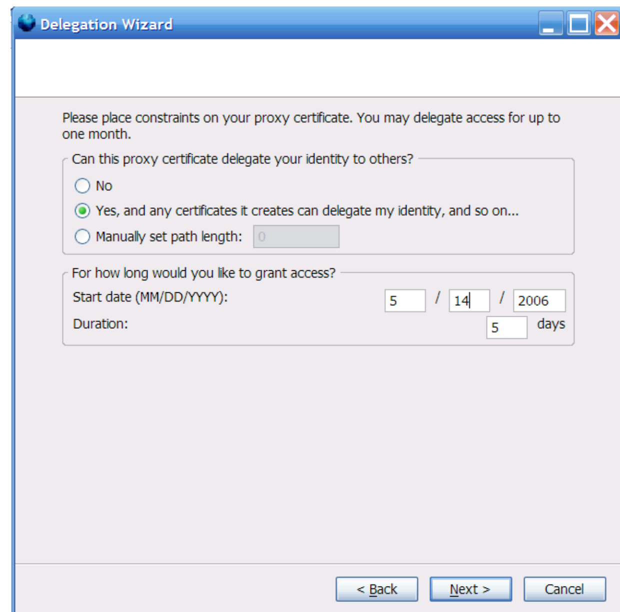

Figure 5: Using our Delegation Wizard (part 3): The constraints page, which allows setting a path length constraint and a validity period.

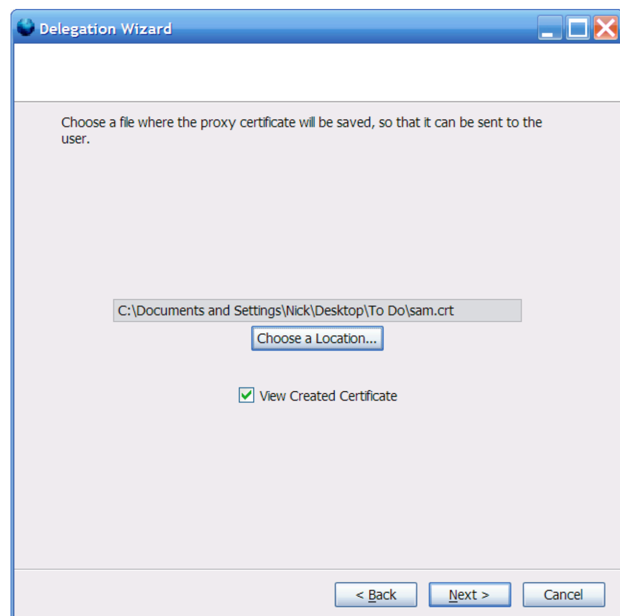

Figure 6: Using our Delegation Wizard (part 4): Saving the certificate to a file, so it can be e-mailed to Detective Spade.

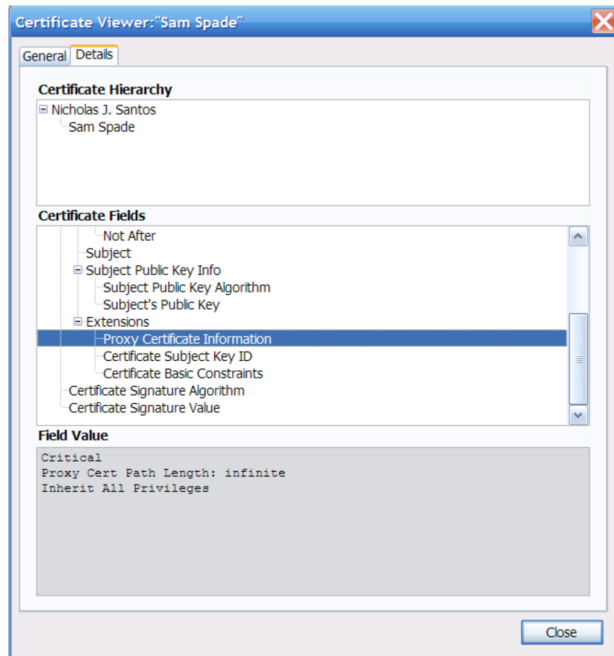

Figure 7: Using our Delegation Wizard (part 5): The ASN.1 structure of the new proxy certificate.

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.

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 7.

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

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 3). 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, we recommend Peter Gutmann's "X.509 Style Guide."[5]

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 UNIX epoch as the serial number, to ensure unique serial numbers[5].

We additionally attach several extensions. We have already covered the `ProxyCertInfo` extension. On the advice of OpenSSL's proxy certificate guide [8], we include a `BasicConstraints` extension that indicates that this certificate may *not* be a CA. We include a `SubjectKeyIdentifier` extension that contains a SHA-1 hash of the target certificate's DER-encoded public key data. Finally, we attach an `AuthorityKeyIdentifier` 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`.

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

[5]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.

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 we copied the encoding function, fixed it, and compiled it 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 our knowledge, this is the first implementation of a proxy certificate issuer with NSS.

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 (such as 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 protected for confidentiality, it would have sufficed to keep them in any non-volatile storage mechanism. In our extension, they are simply stored in an SQLite database[6]. The database key for each proxy certificate is derived from the serial number and the issuer name. Because issuers *should* issue certificates with unique serial numbers, this database key should be unique. Each entry in the database also contains a DER encoding of the certificate, as well as database keys to access the issuer certificate, "delegator" certificate, and private key in the regular database. The "delegator" certificate is the 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.)

[6]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/.

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. It tells Alice which proxy certificates she has issued, and allows her to re-export them if she needs to send them again (Figure 8). The other section of the database holds certificates delegated to the user, certificates for which the user has the private key (Figure 9). These are the identities that can be used in an SSL session.

Using Proxy Certificates, in Theory. 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 otherwise documented. But this also means that there is no documentation assuring us that this is a safe mechanism for extension development. Thus, 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.

Using Proxy Certificates, in Practice. The actual implementation is far more complicated and involves several levels of indirection. The flow can be

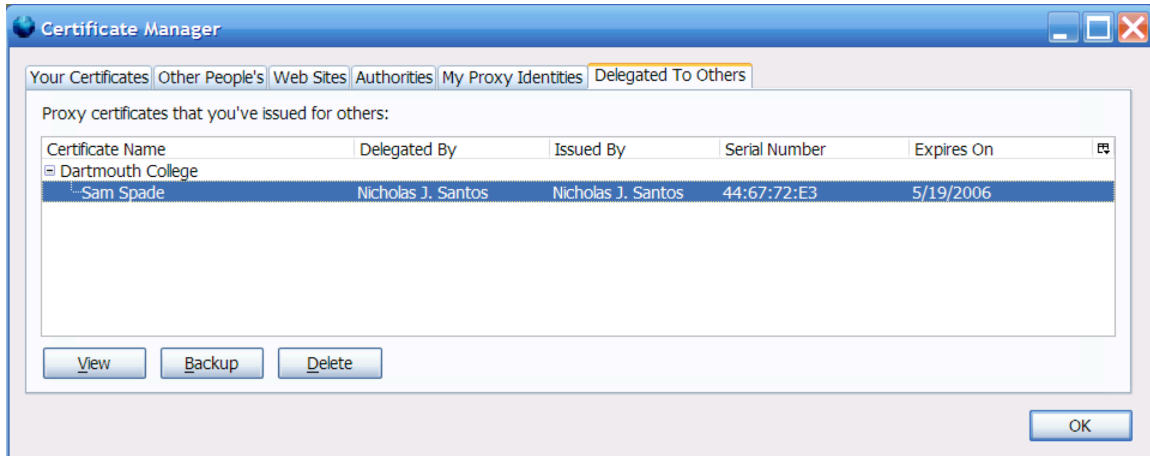

Figure 8: Viewing the certificates in the database that have been issued *by* this user.

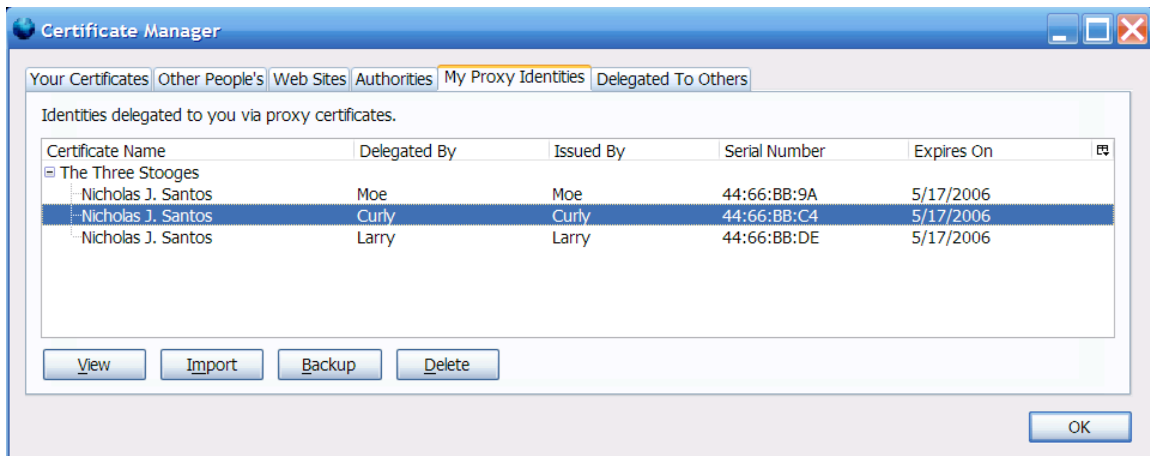

Figure 9: Viewing the certificates in the database that have been issued *to* this user. Observe that only fools delegate their privileges to this particular user.

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.

The SSL/TLS handler objects allow clients to set a callback function that retrieves the client certificate for the SSL handshake. (In the NSS documentation, this callback is known as the `ClientAuthDataHook` [12].) 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 callback 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 uses the values of the pre-computed fields to decide which certificates to push onto the certificate chain, but the it uses the DER encodings to construct the bits of data actually sent across the network. 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

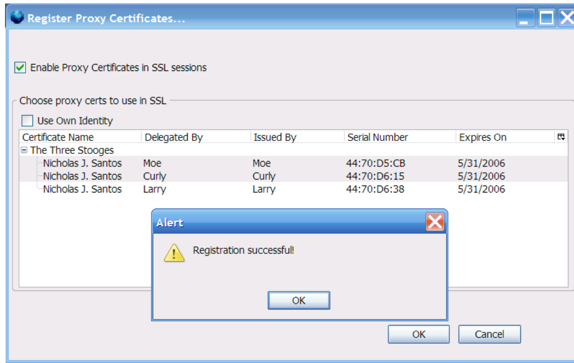

Figure 10: Registering two proxy certificates to use in an SSL session. The chains for both will be sent in client-side SSL/TLS authentication.

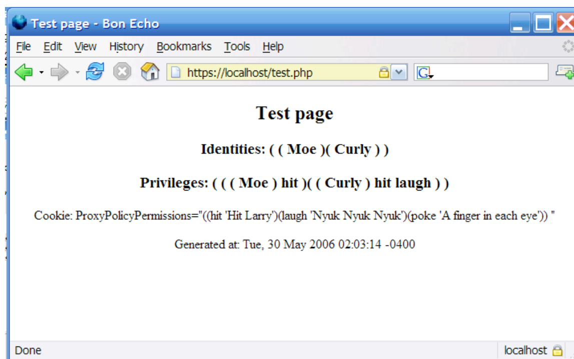

Figure 11: This test page shows the values of environment variables `SSL_DELEGATED_IDENTITIES` and `SSL_DELEGATED_PRIVILEGES`.

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.[7]

5.2 The Apache Codebase

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.

[7] A lot of our reviewers were surprised by this result—in particular, that a Firefox extension could throw so much weight around and behave so intrusively. It's worth pointing out that the term "extension" is slightly misleading. "Extension" suggests that the software is sandboxed; that it can only "extend" the browser. But in reality, installing an extension is just as dangerous as executing a binary.

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 4.4 were designed to take advantage of this existing infrastructure. Adding them is not complicated. The `SSLMultipleIdentities` and `SSLExclusiveIdentity` 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.

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.)

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.

6 Related Work

SDSI/SPKI SDSI/SPKI is an alternative certificate standard that stresses simplicity over the complication of X.509.

The SDSI/SPKI group at MIT 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 [10]. 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.

Greenpass The Greenpass project grafted SDSI-SPKI delegation on top of X.509 identity certificates for EAP-TLS. In that project, 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 (the guest via a captive portal). 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 [4].

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.

Distributed Systems. 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 [15, 11]. Marchesini et al married Grid-style MyProxy to hardware trustworthiness levels [9]. 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 [7].

7 Conclusions

In the real world, users like to delegate privileges. If next-generation authentication systems (such as PKI) do not allow for this delegation, users will find a way to work around them—and undermine the security that drove adoption of the strong system in the first place. In this paper, we have presented both the design and prototype of a way to extend PKI (via standard client-side SSL) to permit this delegation.

When users can delegate rights to each other, we can end up with a user with a set of delegated rights from multiple sources. This raises the question: how can applications make sense of this heterogeneous set of rights? Consider some real-world examples. When a group of people elect a delegate to the U.S. Electoral College, they delegate a simple duty—to elect a presi-

dent. But when a number of persons sign their power of attorney over to a single lawyer, a complex set of rules governs how the lawyer can use these rights, to prevent a conflict of interest. Clearly then, some applications need fine-grained controls over how to enforce delegated rights, and some don't. Further exploration there is one area of future work. Another area is exploration of the user interfaces involved in delegation.

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 paper is the political ideal that Alice and Bob should have the authority to delegate their own privileges. This paper seeks to empower them with that authority.

Code Availability

We plan to make the code available for public download in 1Q2007.

References

[1] 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.

[2] T. Dierks and C. Allen. *TLS Protocol Version 1.0*. IETF RFC 2246, January 1999.

[3] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. *SPKI Certificate Theory*. IETF RFC 2693, September 1999.

[4] Nicholas C. Goffee, Sung Hoon Kim, Sean Smith, Punch Taylor, Meiyuan Zhao, and John Marchesini. Greenpass: Decentralized, PKI-based Authorization for Wireless LANs. In *3rd Annual PKI Research and Development Workshop Proceedings*, pages 26–41. NIST/NIH/Internet2, 2004.

[5] Peter Gutmann. X.509 style guide. October 200. Retrieved on March 9, 2006 from http://www.cs.auckland.ac.nz/~pgut001/pubs/x509guide.txt.

[6] R. Housley, W. Polk, W. Ford, and D. Solo. *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. IETF RFC 3280, April 2002.

[7] Jon Howell and David Kotz. An Access Control Calculus for Spanning Administrative Domains. Technical Report PCS-TR99-361, Dartmouth College, 1999.

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

[9] J. Marchesini and S. W. Smith. SHEMP: Secure Hardware Enhanced MyProxy. In *Proceedings of Third Annual Conference on Privacy, Security and Trust*, October 2005.

[10] Andrew J. Maywah. An Implementation of a Secure Web Client Using SPKI/SDSI Certificates. Master's thesis, Massachusetts Institute of Technology, May 2000. Retrieved on-line from http://theory.lcs.mit.edu/~cis/theses/maywah-masters.ps.

[11] J. Novotny, S. Tueke, and V. Welch. An Online Credential Repository for the Grid: MyProxy. In *Proceedings of the 10th International Symposium on High Performance Distributed Computing (HPDC-10)*, pages 104–111. IEEE, 2001.

[12] Bob Relyea, editor. *Network Security Services (NSS)*. The Mozilla Foundation, May 2005. Retrieved on March 11, 2006 from http://www.mozilla.org/projects/security/pki/nss/.

[13] Nicholas Santos. Limited Delegation (Without Sharing Secrets) for Web Applications. Technical Report TR2006-574, Dartmouth College, May 2006.

[14] S. Tuecke, V. Welch, D. Engert, L. Pearlman, and M. Thompson. *Internet X.509 Public Key Infrastructure (PKI) Proxy Certificate Profile*. IETF RFC 3820, June 2004.

[15] 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.