An Implementation of a Secure Web Client Using SPKI/SDSI Certificates

by

Andrew J. Maywah

S.B. Massachusetts Institute of Technology (1999)

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

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2000
[June 2000]

Author .....
Department of Electrical Engineering and Computer Science
May 22, 2000

Certified by .....
Ronald L. Rivest
Edwin Sibley Webster Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

An Implementation of a Secure Web Client Using SPKI/SDSI Certificates

by

Andrew J. Maywah

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 2000, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

On the Internet today there exists a multitude of documents and other electronic information resources. As the Internet grows in popularity, a growing concern for individuals is how to publish these documents on the World Wide Web and limit access to these documents according to their desired security policy. In this thesis, I designed and implemented a client that uses the SPKI/SDSI public key infrastructure to provide access control on the World Wide Web. In particular this Web access control client has the following features: security, portability, transparent operation, functional and user friendly interface, and extensibility. The primary goal of the access control system, named "Project Geronimo", is to provide a secure and reliable mechanism for Web publishers to restrict access to Web resources. This implementation of a Web client using the SPKI/SDSI Public Key Infrastructure (PKI) also serves to illustrate the effectiveness of SPKI/SDSI in practical use as an everyday network authentication and authorization tool. This is an implementation in C for Unix, and consists of a dynamic relocateable shared-object Netscape Communicator plug-in which integrates with the Netscape Navigator Web browser to securely retrieve HTML content over the Internet. The plug-in was tested extensively and proved to be a stable and reliable implementation. The findings of this project may serve to motivate others to integrate the SPKI/SDSI PKI in other Internet client/server applications.

Thesis Supervisor: Ronald L. Rivest
Title: Edwin Sibley Webster Professor of Computer Science and Engineering

Acknowledgments

First and foremost, I'd like to thank my entire family for their love and support throughout my entire MIT career. Thanks to my brother Phil, sister Nicole, Dad, and especially Mom for helping me keep my strength and sanity.

I'd also like to thank my research advisor, Professor Ronald Rivest, for his guidance and never ending support in my research and related professional activities. I am grateful to him for giving me the chance to work under his direction and expert tutelage. He is truly a legend in the world of crypto and security, but more than that, he's a great guy!

I don't know where I'd be without the help of the other members of the SPKI/SDSI research team and the CIS group. I'd like to thank Dwaine Clarke for being there with me at all hours of the night whether it was working on research, psets, or just chatting about life. Thanks also go to Matthew Fredette and Jean-Emile Elien for working so hard, often on their free time, to help us get this project off the ground. Matt Fredette was our saviour – writing and testing the `acl_response` code and helping us get things working for the demo. Many thanks to Dr. Kazuo Ohta for his help with and advice about research in Japan.

Last, but certainly, not least, I'd like to thank my best friends: George Lin, Heidi Li, Theodore Weatherly, Naveen Yalamanchi, Gong Ke Shen, May Tse, and Damon Lewis. They've always been there for me when I needed them and I will always be grateful to them.

This research was supported by a grant from NASA.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

On the Internet today there exist a multitude of documents and other electronic information resources. The majority of these documents are available for download via the HTTP protocol on The World Wide Web. Almost all businesses, government agencies, and private individuals now have a Web site, and the number is growing every day.

As this trend continues, a prominent concern is how to publish these documents on the World Wide Web and limit access to these documents according to desired security policies. Publishing documents on the WWW tends to be popular because it is an easy way to distribute information to people via the Internet. But why limit access? For example, a company may like to make important and confidential documents available only to its executive board easily and securely. Table 1.1 contains some reasons that organizations or individuals may want to limit access to their websites [9].

The World Wide Web is one mechanism for delivering a variety of Internet content easily, but what about security? Unfortunately, today's Web is extremely vulnerable to security compromises of varying degrees. How can one limit access over the Web

1. The information on your webserver may only be intended for company employees or members of your organization.
2. You may have an electronic publication or service that contains material that should only be available for customers who have paid to read the publication or access the service.
3. You may have confidential corporate strategy or proprietary information that should only be accessed by individuals bound by a non-disclosure agreement.
4. You may have a Web-based order-entry system on your website for your sales agents.

Table 1.1: Reasons for controlling access to a webserver.

reliably and easily?

Protecting confidential documents has long been a concern to publishers on any medium. In most situations, especially those involving the transfer of electronic documents over a network, security involves the three separate steps of identification (identifying who you are), authentication (establishing who you are), and authorization (determining what you are allowed to access). The process of executing these steps is known as access control.

But how does access control work when considering communication over the World Wide Web? The basic steps of authentication and authorization are the same. But, the details of the processes of authentication and authorization will differ between applications. Modern day authentication techniques use public key cryptography to provide authentication over communications networks such as the WWW. Public Key cryptography is a tool that can be used to gain some level of trust for authentication. Public Key cryptography, which is based on the difficulty of solving some hard mathematical functions, was invented in the 1970's. It facilitates encryption and decryption, and enables the creation of digital signatures and certificates.

A public key infrastructure (PKI) is a system of services and protocols for managing public keys. In this implementation, the SPKI/SDSI PKI was chosen to manage the use and distribution of public keys. A parallel research goal of this project is to explore the usability, efficiency, and overall effectiveness of the SPKI/SDSI PKI within an actual application.

1.2 Why SPKI/SDSI?

The Simple Public Key Infrastructure (SPKI) (pronounced "S-P-K-I") and Simple Distributed Security Infrastructure (SDSI) (pronounced "Sudsy") are public key infrastructures (PKI) that manage the use and distribution of public keys. These very similar PKIs were developed separately and then were later merged into a common specification (hereafter known as SPKI/SDSI) [5, 20].

SPKI/SDSI originated from the work of Professor Ronald Rivest, Professor Butler Lampson, and Intel researcher Carl Ellison. The research was motivated by the belief that traditional PKIs were incomplete and unnecessarily complicated. The researchers aimed to provide a public key architecture that is efficient, complete, natural, and intuitive. Their primary focus is to create a PKI that can be readily used to create secure distributed systems according to the administrator's desired security policy. To that end, authorization was build into SPKI/SDSI as a central component [20].

The SPKI/SDSI describes a simpler, more applicable PKI than those previously implemented. SPKI/SDSI makes extensive use of certificates which easily give names or identifiers to public keys in the system. These names are arbitrary and are not intended to carry any official or legal ownership binding or association. SPKI/SDSI certificates can also specify authorization given to public keys. To make propagation of authority simpler to determine, one certificate cannot explicitly perform both tasks of naming and authorization. Therefore, SPKI/SDSI has two distinct kids of certificates: name certificates and authorization certificates [20].

SPKI/SDSI certificates are designed to be displayed in human readable form. We believe that the idea of trust is better conveyed when human interaction is involved. Thus the certificate can have some free form text (and perhaps a photograph) describing the individual. For this reason the S-expression format is the chosen encoding for SPKI/SDSI objects [20].

The SPKI/SDSI presents a new model of key management and trust hierarchy that departs from traditional models. Its use as the security component in distributed computing systems will, hopefully, serve to demonstrate it effectiveness and benefits over traditional PKI models.

1.3 Organization

This thesis project is intended to research the design and implementation of a SPKI/SDSI client for providing access control on a World Wide Web HTTP server. In particular this client module will have the following features: security, portability, transparent operation, functional and user friendly interface, and extensibility. The primary goal of the client (and corresponding server) is to provide a secure and reliable mechanism for Web publishers to restrict access to certain documents. Given that SPKI/SDSI already emphasizes security, this design will focus on the important design criteria for a client using SPKI/SDSI as the security component. This thesis discusses the client implementation that was developed, and also focuses on the overall effectiveness of using SPKI/SDSI for the public key infrastructure.

This access control system has been named "SPKI/SDSI Project Geronimo". Geronimo, a Bedonkohe Apache leader of the Chiricahua Apache Native American Tribe, led his people's defense of their homeland against the U.S. military after the death of Cochise, the previous chief [10]. Similarly, the Apache webserver, used in this access control system implementation as the server component, protects websites from unauthorized access. Thus, it is appropriate to honor Geronimo by naming this project after him.

The following chapters of this document are organized as follows. Chapter 2 gives the reader some basic background on access control, public key cryptography, and public key infrastructures. Chapter 3 covers the building blocks of the SPKI/SDSI PKI and its implementation as well as information on the parts of the software library pertinent to this project. Chapter 4 presents the functional design specification, design criteria, and design considerations for the client. Chapter 5 presents the implementation of the protocol, server, and client. Chapter 6 shows an example of the Geronimo access control system in operation. Chapter 7 discusses future work and improvements that should be addressed. Finally, Chapter 8 gives a summary of the results.

Chapter 2

Background

2.1 Access Control

Access Control is the process of limiting who can get access to certain resources. Resources can constitute any number of things such as: a particular function, data, or location. The process of access control involves the three separate processes of identification (recognizing or indicating your identity), authentication (reliably establishing your identity), and authorization (determining what you are allowed to access) [21]. This section gives some basic background on these topics.

2.1.1 Identification

In general, Identification is the first step in the access control process. For access control, it is important to first identify the entity that is going to be granted or denied access. For example, entering a username for an application's "login" prompt provides the identification for a particular user in the system.

2.1.2 Authentication

The second step in access control is authentication. Authentication is closely related to identification in that authentication provides the proof of identity. In general, the process of authenticating an entity involves using one or more unique characteristics

that reliably identify qualities. Proof of identity can be shown by what one knows or what one has. For example, a security guard may inspect the photograph on your company badge to make sure that you are indeed a company employee [18].

2.1.3 Authorization

The third step in access control is authorization. Usually, the process of authorization takes place when a person controlling some resource determines who is allowed to access it. Usually this person has a list of users and what they are allowed to access. Then all the owner needs to do is verify the requestor's credentials and then check the list to see what they are allowed to access. This list is commonly known as an access control list (ACL). Using ACLs, one can implement a security policy specific to one's needs. One can implement access restrictions based on time of day, IP Address, etc. [18].

2.2 Public Key Cryptography

The concept of public key cryptography was invented by Ralph Merkle, Whitfield Diffie, and Martin Hellman in the late 1970's. The basic premise that they presented was that cryptographic keys should come in pairs – a private key that is kept secret and a public key that is distributed widely to many people. Essentially, the public key would be used by anyone wishing to encrypt a message for the owner of the key pair and only that owner can decrypt that message using his/her private key. Furthermore, it is computationally difficult to determine the private key from the corresponding public key [22].

An important development stemming from public key cryptography is the ability to digitally sign documents. This process allows the sender to provide a small "checksum" of a message that the recipient can use to easily verify that the message came from that sender and was not modified during transmission. This checksum, otherwise known as a digital signature (or simply signature) is computed using the

sender's private key and then verified using the sender's public key [22].

How does one find out who a particular public key belongs to with confidence? A data structure known as a certificate is traditionally used to bind a name (or identity) to a public key[1]. The certificate is digitally signed by a trusted third party whose signature can be verified by every principal in the system. This trusted third party is known as a certification authority (CA) [18].

How does authentication work over a communications network? Over networks, the proof of authentication has usually taken the form of a cryptographic authentication protocol. Cryptographic authentication protocols are considered to be more secure than password-based protocols which may be susceptible to security leaks via eavesdropping. Cryptographic authentication, also known as strong authentication, provides a computationally secure way to prove possession of the private key half of a public/private key pair. This proof traditionally takes the form of a digital signature [18]. A digital signature gives assurance that the document in question was signed by a particular private key, held in secret.

To perform authentication, one can use a public key infrastructure (PKI), to set up public and private keys for cryptographic authentication. What does a PKI do? A PKI provides services, protocols, and data structures supporting the implementation of applications using public key cryptography. Public Key infrastructures, such as X.509 [13] and PGP [22], have data structures for issuing and distributing public keys used in encryption.

2.3 Traditional PKIs

This section will briefly discuss some traditional PKIs in use today.

[1]Traditional certificates also contain other information about the issuer and subject such as a validity period and contact information. For simplicity, this discussion will not include these, but, will only focus on the relevant parts.

2.3.1 X.509

The ITU-T presented the X.509 recommendation as part of the X.500 series of recommendations that define a directory service. In this context, the directory is a server (or set of servers) that maintain a database of users and particular user information. X.509 was proposed in 1988 as a public key infrastructure using certificates and digital signatures to manage the distribution of public keys [23].

X.509 relies heavily on the use of central third-party certification authorities to provide "trust" when distributing public keys to principals. Each CA, in this case, creates the user certificates and signs them. Each principal has a CAs (self-signed) public key certificate and can use that to verify if another certificate is valid. The principal can thus believe that the public key contained within the certificate belongs to the principal named therein. Thus the CA is at the top of the trust hierarchy of the X.500 global directory naming structure [23].

X.509 attempts to identify all users in the system using information such as: legal name, email address, locality, etc. The X.509 subject field is meant to convey the exact identity of a key owner to a public key user. The X.500 directory expresses this idea in the form of a *distinguished name*. The distinguished name is intended to be a single, globally unique name that everyone could use when referring to an entity. The purpose of this is to bind an entity to a particular key pair.

2.3.2 PGP

PGP, the invention of Phil Zimmermann, has become widely popular as a tool for sending and receiving email confidentially and reliably. Authentication is provided by providing digital signatures of the email messages sent. Each PGP principal has a key-ring data structure which stores the public and private keys of the owner as well as the IDs and public keys of any other principals in the system. Each principal collects the public keys of other principals and PGP assigns a value to the "key legitimacy field" indicating the extent to which PGP will trust that it is a valid key for that

user. Keys are then propagated throughout the system and the values of the "key legitimacy field" are updated according to a formula that PGP has devised. The result is a trust hierarchy known as "a web of trust" [23].

2.3.3 Deficiencies of PKIs

In classic PKIs the intuition is that one is authenticating actual people. That is, the cryptographic signing key is expected to be bound securely to an identified person, and the authentication step is understood as verifying that the message or request received is from that person. However, recent SPKI/SDSI PKI research emphasizes that the authentication step merely verifies the signature on the request, and so verifies that the request is signed by a particular private key. There is no understanding that the request is from any particular identified owner – identification of people is totally out of the picture. In its place is the simpler notion of identifying a public key. The process of trying to identify people is often difficult and cumbersome. It is simpler to focus on identifying keys.

Some PKIs try to ascribe a trust hierarchy based on certificates. This means that they artificially equate a signed certificate with trust that the subject is indeed the owner of the key pair. Usually, as is in the case of X.509, a central certification authority (CA) is relied upon to digitally sign certificates to provide a third party's assurance that a particular public key belongs to a particular individual. This view of "trust" tends not follow the intuitive notion of trust since the notion of trust is not transitive. For example, if A trusts B, and B trusts C, does A necessarily trust C? In the physical world, this may not necessarily be the case. Newer PKI research prefer to discuss delegation instead of trust. Delegation basically asserts that if authority is given from one principal to another, the second principal is given authority to act on the first principal's behalf with respect to the given permission.

Some traditional PKIs have an added complexity that stems from their dependence on global name spaces. Both X.509 and PGP issues each user a globally unique name.

How does one find another name in the system? This question cannot be answered easily since the need to keep a global set of names unique necessitates using an identification scheme that is difficult to use. A globally unique name, likely, would be long, cumbersome, and difficult to remember. Classic PKIs tend to try to bind an individual's "real" name to their public key. The global naming system tries to identify a person right down to the city that they live in. This becomes problematic because large organizations tend to shy away from publishing membership information and organizational hierarchy. For example, consider the CIA publishing an X.500 directory entry of all their employees. The issue is that traditional PKIs try to use keys to identify people. They lose the notion of the key and say that the owner "signed the document". The owner does not need to be in the picture at all.

Chapter 3

SPKI/SDSI

This chapter describes the data structures and objects within the SPKI/SDSI PKI, as well as the software library implementation. Section 3.1 gives a brief description of the objects used. Section 3.2 presents the SPKI/SDSI software library and its components.

3.1 Theory

This section is intended to give a background on the relevant theory of SPKI/SDSI version 2.0 and its role in this thesis project as the Public Key Infrastructure. The concepts covered are: public keys, naming and certificates, groups, tags, access control lists, and finally certificate chain discovery. Much of this material was derived from the papers "SDSI - A Simple Distributed Security Infrastructure" [20], "SPKI Requirements" [7], and "SPKI Certificate Theory" [6]. Several other sources of documentation are available to the reader on the following Websites:

`http://theory.lcs.mit.edu/~cis/sdsi.html`
`http://world.std.com/~cme/html/spki.html`
`http://www.ietf.org/html.charters/spki-charter.html`

3.1.1 Public Keys

The crucial component to a public key infrastructure is the key pair. The key pair, consisting of the public and private keys is the main structure used by individuals. As such, in SPKI/SDSI, the public and private keys are central. The private key is kept secret while the public key is distributed freely. In contrast to classic PKI theory, SPKI/SDSI emphasizes that the public key is the principal and is the entity that is identified, rather than a person. Of course, the notion of an owner (e.g. person, corporation, computer) of a public key is allowed, but is not necessary. The owner controls the public keys and thus the key can be viewed as a "proxy" for the individual. But, there need not be an owner in SPKI/SDSI. Moreover, SPKI/SDSI stresses identification of public keys rather than actual people. Identifying people is often difficult and problematic because of the involvement with legal concerns. SPKI/SDSI emphasizes identification of keys because it is a simpler and more direct process. Figure 3-1 shows an example of a public key in SPKI/SDSI. In this example, on the second line, the string "rsa-pkcs1-md5" identifies the public key algorithm (RSA), encryption format (PKCS1), and hash algorithm (MD5) used for the key. On the third line, the string "(e #21#)" gives the hexadecimal value for the exponent of the key. Finally, the string "(n |AN2R2H..." specifies the modulus of the key.

```
(public-key
  (rsa-pkcs1-md5
    (e #21#)
      (n
      |AN2R2HmHSOL6hC9zuYwCHroklzwWqOB8hMYaqhmSyryIiHeIRfoIpUOO
      WrnKC7XkAPMDmNfehNy9q8ponk26HBaQvNOcdkD1H4wNbGz2zSwgm4sKc
      HXOd1OLb5n6BPM85kSMhYmB6EdBb3jTbZLhscNk/FmcL8yUypVGnEYJmZ
      J9|)))
```

Figure 3-1: A SPKI/SDSI Public Key.

Each principal is its own Certification Authority (CA). This means that every public key can sign statements and issue certificates with any other principal as the subject.

The owner of the public key can devise his own policy when issuing certificates, and does not have to adhere to any industry standard or worldwide policy. This allows for a simpler and more egalitarian design.

3.1.2 Naming and Certificates

Each public key has its own local name space with which it can refer to other principals or sets of principals. The names can be any kind of identifier such as nickname, account number, email address, etc. They are arbitrarily chosen and do not necessarily fit into any "global" *name space*[1]. For example, the principal I call Alice may not be the same as the principal you call Alice. This eliminates the need for a global namespace such as those described in some other PKIs.

One important contribution of SPKI/SDSI is the idea of linked name spaces. This concept allows principals to refer to names in other name spaces. Each principal can export his/her name/value associations by issuing name certificates. For example, if a user's local name space contains the name Alice, which refers to a particular public key, then that user can refer to the principal that Alice names Bob as:

```
(name Alice Bob)
```

or, using syntactic sugar,

```
Alice's Bob
```

In SPKI/SDSI there are two types of certificates: Name Certificates (which bind a name to a key) and Authorization Certificates (which assert authorization for a key for a particular resource). SPKI/SDSI keeps these two notions separate in order to allow for a more accurate view of delegation. An example of a name certificate that binds the name Alice to a particular public key is shown in Figure 3-2. In this

[1]Otherwise known as namespace

certificate, the issuer is the public key with exponent #25# (hexadecimal value). This key is issuing a name certificate that binds the key with exponent #21# to the name "Alice".

```
(cert
  (issuer
    (name
      (public-key
        (rsa-pkcs1-md5
          (e #25#)
          (n
            |ANhdEvNWm/nsFvPZcscRI1zkd/4/Svgf52WXGlebx8FFyOKZNqnqb
            nTXQkI1aGeMT+n/g7n/F3OKjHwNvBAa2T8EEuytz7+XyW89gucPWi3
            Syn7r2Rj+k4dnlFR2OIUYotGqDamwHqKgbdtIxJnCNSN7DrrzZ8S+j
            xYLbJ3SICkR|)))
    Alice))
  (subject
    (public-key
      (rsa-pkcs1-md5
        (e #21#)
        (n
          |AN2R2HmHSOL6hC9zuYwCHroklzwWqOB8hMYaqhmSyryIiHeIRfoIpU
          OOWrnKC7XkAPMDmNfehNy9q8ponk26HBaQvNOcdkD1H4wNbGz2zSwgm
          4sKcHXOd1OLb5n6BPM85kSMhYmB6EdBb3jTbZLhscNk/FmcL8yUypVG
          nEYJmZJ9|)))))
```

Figure 3-2: A SPKI/SDSI certificate.

As you can see, the certificate in Figure 3-2 is displayed in a LISP-like syntax known as an S-expression. S-expressions are structures, surrounded by parentheses, that contain other S-expressions or character strings. The S-expression format allows for human readability.

3.1.3 Groups

Another important feature of SPKI/SDSI is its use of groups to identify sets of principals within a name space. Groups are convenient when one wants to refer collectively to a set of principals, instead of having to refer to each separately. Each

SPKI/SDSI principal can define groups by simply using a name certificate to bind the group name to each of the principals in the group. To define a group in SPKI/SDSI, all a user has to do is issue a name certificate for each principal in the group, binding that principal to the name of the group. For example, if a user wanted to add the names Alice and Bob (for which already have been defined) to the `project_team_2000` group, the user would first create a name certificate binding the local name Alice to the name `project_team_2000` and then create another name certificate binding the local name Bob to `project_team_2000`.

3.1.4 Tags

The SPKI/SDSI access control list format contains a field called the tag which specifies the permission given to the subject listed in the ACL. The resource permission that the tag specifies is application specific. So for example, a tag for telnet access control might specify: the server, port, allowable remote host IP addresses, valid times, etc. Figure 3-3 shows an example of a SPKI/SDSI tag data structure. In this figure the tag gives the Subject permission to telnet into host rooster.lcs.mit.edu on port 23 as user alice. But, since each tag is application specific, the designer could have specified the username, hostname, and port in any order. The designer chooses to specify what data are important to specify permissions for the application's resource(s).

```
(tag
  (telnet
     rooster.lcs.mit.edu
     23
     alice))
```

Figure 3-3: An example of a SPKI/SDSI tag for telnet.

3.1.5 Access Control List

One of the important components of SPKI/SDSI is support for access control lists (ACL). The designers of SPKI/SDSI realized that the ACL data structure is an integral part of building secure distributed systems and specifying a security policy in general. Hence, ACLs (and similarly authorization certificates) allow a principal to specify permissions for principals to access resources. Figure 3-4 shows an example of a SPKI/SDSI ACL:

```
(acl
  (entry
    (name
      (hash md5 |YeqtrNZYOt94CneXDnxtDQ==|)
      project_2000_exec)
    (tag
      (http
        (* set GET POST)
        (* prefix
           http://rooster.lcs.mit.edu:8081/project2000/budget/)))))
```

Figure 3-4: A SPKI/SDSI ACL.

In the above figure the ACL contains only one entry. The entry specifies some permission given to the group `project_2000_exec`. This group is a name in the name space of the public key with hash (hash md5 |Yeqtr...). The actual permission granted is specified in the tag data structure discussed in section 3.1.4. In this example, the tag contains 3 elements: the protocol (http), the http method (either GET or POST), and the urls that are given access to (anything in the /project2000/budget/ directory on the rooster.lcs.mit.edu webserver running on port 8081). These permissions are only relevant in the context of this webserver. For a different application, such as discuss[2], the tag will have different components. Furthermore, the ACL creator can allow the subjects on the entries to further delegate the specified authority to other

[2]Discuss is a networked, electronic conferencing system similar to electronic bulletin-boards. Please see http://web.mit.edu/olh/Discuss/index.html for more information.

principals as they see fit. This is called *propagation* and can be specified in the entry section of the ACL. For example, if the owner of the acl in the above figure allowed propagation, then all the members of the group `project_2000_exec` would be able to delegate authority to others to access the /project/budget/ directory on the webserver. Figure 3-5 shows an ACL entry with propagation.

```
(entry
  (name
    (hash md5 |YeqtrNZYOt94CneXDnxtDQ==|)
    project_2000_exec)
  propagate)
```

Figure 3-5: An example of an ACL entry with propagation.

The ACL is central to the authorization mechanism mentioned in section 2.1.3. This is where the authenticated requester is compared with the entries on the ACL to determine whether to permit the user to access the requested resource.

3.1.6 Certificate Chain Discovery

How does the system verify authorization? In other words, how does the system determine, given that it can authenticate a principal, that the principal is indeed on the ACL? Well, if the key is directly on the ACL, the process is trivial. But what about the case where the key itself is not directly on the ACL, but instead groups or other keys, with propagation allowed, are on the ACL? How does the system determine that the key requesting access is a member of the specified group (or groups) on the ACL? The process of producing such a proof is known as *certificate chain discovery.*

The process of certificate chain discovery is performed by a function that takes an ACL, tag, and all of the certificates that are stored locally, to find a valid authorization chain from an entry on the ACL to the user's public key. In his Master's thesis, Jean-Emile Elien describes a polynomial running time algorithm for generating a valid

authorization chain if one exists [4]. This section gives a basic overview of that algorithm. Much of this material was derived from his thesis as well as "Certificate Chain Discovery in SPKI/SDSI", by Ronald Rivest, et. al. [19].

A name certificate defines a local name in the issuer's local name space. Only key K may issue certificates for names in its local name space. A name certificate can be represented as a signed four-tuple (K, A, S, V) where K is the issuer's public key, 'K A' is the local name being defined, S is the certificate's subject, that is, the new name being bound to the local name 'K A', and V is the validity specification. The validity specification provides additional requirements, beyond the verification of the signature, that a certificate needs to satisfy to be valid. Examples of validity specifications include validity time periods and online checks. As name certificates only define local names, A is exactly one word long. This certificate can also be represented as the rewrite rule:

$K_A \rightarrow S$

Authorization certificates can be represented as a signed five-tuple (K, S, D, T, V). Again, K is the issuer's public key, S is the name or key receiving the grant of authorization, D represents the delegation bit, T is the tag which specifies the specific permission being granted, and V is the validity specification. If the delegation bit is set to true, the subject may further delegate the permission specified in the tag to other keys and names. This certificate can be represented as the following rewrite rule[3]:

K □ → S □ if delegation is allowed.

K □ → S ■ if delegation is not allowed.

□ and ■ are referred to as "tickets".

[3]A rewrite rule operates on strings of symbols. The rule shows how one can replace a given sequence of symbols with another

Name certificates can be composed to derive new names, authorization certificates can be reduced to derive new authorizations, and authorization certificates and name certificates can be combined together to give new authorizations to new names. For example, the following rules:

K_A □ → K_B Carol friends ■

K_B Carol → K_C

K_C friends → K_C David

K_C David → K_D

can be composed to give the authorization rule:

K_A □ → K_D ■

A certificate sequence consisting of the above signed certificates in a chain would be sufficient to prove to a verification procedure on a server that 'K A' authorizes K_D. The use of tickets to represent delegation of authority in authorization certificates prevents them from being inappropriately composed with name certificates, as the purpose of authorization certificates is to grant permissions and not rewrite names. They also prevent an authorization certificate which does not have the delegation bit set from illegally reducing with other authorization certificates. Because of the ability to chain SPKI/SDSI certificates to produce derived names and authorizations, certificate chain discovery is non-trivial.

Reducing certificates are name certificates where the subject is a key. Thus, reducing certificates can be represented as rewrite rules of the form: K A -> K', where K' is a key. The finite closure over a set of certificate tuples is the set containing the original tuples and all the tuples that can be derived from the original tuples. Calculating the finite closure is central to the certificate chain discovery algorithm.

A fundamental procedure of the certificate chain discovery algorithm involves reducing certificates with non-reducing authorization certificates and name certificates to provably generate all possible derivable tuples in finite time.

Thus, the certificate chain discovery algorithm first transforms an acl to a set of authorization tuples, each with the issuer "Self". These tuples are combined with the tuples derived from the certificates that the user has. Invalid tuples and auth tuples whose tags do not intersect with the permission being requested (in this case, the tag formed from the url request) are removed from this set. The finite closure over the remaining tuples is then generated. Original and derived auth tuples in the closure are then used to build a directed graph; if there exists a path from "Self" to the user's public key in this graph, there exists an authorization chain from an entry on the acl to key. Simple bookkeeping is performed to help reproduce a chain of signed certificates that can prove to a verifier that authorization is granted. This certificate chain is a sequence of certificates where each certificate in the sequence is a link in the certification chain. In SPKI/SDSI this proof is known as a Sequence. Figure 3-6 is an example of a certificate sequence.

The worst case running time of the algorithm is $O(n^3L)$, where n is the size of the input set of certificates and L is the length of the longest subject in any input certificate. At most $O(n^3L)$ tuples are produced when the closure is generated. In practice, we believe the algorithm will run much faster, and will be very practical.

3.2 Library Implementation

This section discusses the SPKI/SDSI software library and tools[4] used in the client implementation. This library was designed and written in C for Unix by Matthew Fredette [15]. Alexander Morcos also implemented a software implementation of SPKI/SDSI in Java [17]. This implementation followed the same general format as

[4]Please see http://theory.lcs.mit.edu/~cis/sdsi.html to download the distribution.

```
(sequence
 (def
  rule12
  (sequence
   (cert
    (issuer
     (name
      (public-key
       (rsa-pkcs1-md5
        (e #25#)
        (n
         |AL3DELPZ5HfIIBIJR3v7A7qQhg9gKdi29nckrOBvaTeyPu2u3O
         W4U5AUFBK1yEknWLINWXzxh+lXYUhX15cXCe/cVydY61P8votBB
         PE92jXPge1b7JvV8Hgnzvzy8XvsE4rczktbbImxTB5eZO1LyyNQ
         46YYiKK+kV8OnpvBibkT|)))
      friends))
    (subject (name Andrew)))
   (signature
    (hash md5 |t9nwczT1CCCFdsq9Q4gQwg==|)
    (public-key
     (rsa-pkcs1-md5
      (e #25#)
      (n
       |AL3DELPZ5HfIIBIJR3v7A7qQhg9gKdi29nckrOBvaTeyPu2u3OW4
       U5AUFBK1yEknWLINWXzxh+lXYUhX15cXCe/cVydY61P8votBBPE92
       jXPge1b7JvV8Hgnzvzy8XvsE4rczktbbImxTB5eZO1LyyNQ46YYiK
       K+kV8OnpvBibkT|)))
    (rsa-pkcs1-md5
     |AK6lMFcne9oBOPvIdqtHXIyFxdHN/KOOhnlO6f/y/XYc4xoavbFDUP
     lnDRTz57TW81Gmm+cNK+omWXGTJCaoZzqYOkeN2RwOD+k/nzJbQRm/F
     mXfLYztNGmwgKQdA/jpmOAb2zQIwNaDk1gKsbh2/t7Mwk/VLvbSjrYD
     8OqQxHVZ|))))
 (def rule22 (sequence (ref rule12) (ref rule11)))
 (ref rule22))
```

Figure 3-6: Example of a certificate sequence.

that of Matthew Fredette, though it was felt that Fredette's implementation was more up to date.

3.2.1 Objects

The software library was implemented based on the SPKI/SDSI 2.0 specification. It contains a C language interface to the data structures and methods necessary to create the above objects. The following objects in table 3.2.1 were used:

Object Name
sdsi2Key
sdsi2Hash
sdsi2Signature
sdsi2Acl
sdsi2Sequence
sdsi2Cert
sdsi2Key
sdsi2Name
sdsi2Tag

Table 3.1: SPKI/SDSI Library Objects

The `sdsi2Sexp` object (shown in Figure 3-7) is a union type that can represent each of the sdsi2 objects. The elements are all C `struct` data structures that define each of the objects types. The `type` field is used to identify which object you are using.

3.2.2 Certificate-cache

The library relies on the user keeping a certificate-cache. The certificate cache is designed to store certificate issued in the user's public key name space as well as certificates issued to the user's public key. The certificate cache can also store any other name certificates issued by (or for) any principal in SPKI/SDSI. The certificate-cache is a file stored on the user's filesystem in their .sdsi directory ($HOME/.sdsi/certificate-cache.sdsi2).

```
/* sdsi2Sexp */
typedef union _sdsi2Sexp {
  int type;
  sdsi2BufferSet buffer_set;
  sdsi2Canonical canon;
  sdsi2Sparts sparts;
  sdsi2Hash hash;
  sdsi2Signature signature;
  sdsi2Cert cert;
  sdsi2Name name;
  sdsi2Key key;
  sdsi2Tag tag;
  sdsi2Sequence sequence;
  sdsi2Acl acl;
#ifdef SDSI2_FEATURE_ONLINE_TESTS
  sdsi2ValidityList vlist;
#endif /* SDSI2_FEATURE_ONLINE_TESTS */
#ifdef SDSI2_FEATURE_K_OF_N
  sdsi2Threshold threshold;
#endif /* SDSI2_FEATURE_K_OF_N */
} sdsi2Sexp;
```

Figure 3-7: The sdsi2Sexp data structure.

3.2.3 Acl-response

For the access control system that was designed, the most important feature of the software library that was used is the certificate discovery functionality (discussed in section 3.1.6). To perform the certificate reduction, the SPKI/SDSI library provides the `sdsi2_acl_response` function that takes as arguments an ACL, the prover's public key, a tag, a timestamp, and flags. Certificates are collected from the user's certificate-cache (see section 3.2.2) as needed. The certificates along with the arguments are fed into a separate Perl process running Jean-Emile Elien's `certgen.pl` certificate discovery implementation [4]. The output of that Perl script is fed into another Perl process which formats the output and returns it to the calling function. Jean-Emile Elien also implemented his certificate discovery algorithm in the C programming language, called `fclose.c`. The Perl script is favored over the C implementation because Perl has a faster running time.

3.2.4 Key Generation

The key pairs that the SDSI library uses are RSA keypairs. The software relies on ssh and pgp key-generation utilities to create the key pairs. Then the user can convert those keys into sdsi2keys using the `pgp-ssh-to-sdsi2` program available with the software distribution.

During key generation the user is prompted to select a password that will be used as a symmetric encryption key to encrypt the private key stored on disk. This is done so that someone who has access to the user's account cannot masquerade as them by using their private key. Every time the user wishes to use the private key, they must provided this password to recover the decrypted key.

3.2.5 User Interfaces

The software distribution also contains user tools for creating/editing name and authorization certificates. The tools provided are the `sdsi2sh` and the `sdsi2ui`. The

sdsi2sh provides a command line shell interface to all of the library functionality. For example, the shell allows you to create certificates, sign certificates, and cache certificates. The shell supports local environment variables and automatically assigns values created in the shell to consecutive variables called dollar values (e.g. $0, $1, $2, etc.). This allows easy scripting for implementing simple re-usable functionality. Figure 3-8 shows the commands available in the shell.

```
sdsi2sh> help

command summary:

help or ?         - displays help
define            - defines a new S-expression variable
define-string     - defines a new bytestring variable
load              - loads an S-expression from a file
print             - prints a variable
width             - sets the display width
dollar            - sets the next $ temporary number
save              - saves a variable to a file
verify            - verifies a signature
check-acl         - checks an access against an ACL
sequence          - adds to or gets from a sequence
hash              - hashes an object
sign              - signs an object
reduce            - reduces a sequence
cache             - adds an object to the cache
search            - compose a cache search
acl-response      - compose a response to an ACL
warranty          - shows the nonexistant warranty

sdsi2sh>
```

Figure 3-8: The list of sdsish commands.

The sdsi2ui gives the user a graphical user interface (GUI) to create, sign, and cache certificates. The sdsi2ui basically uses a Netscape Navigator web browser with a CGI[5] display driven by a small Perl webserver running on the local machine. Some

[5]Common Gateway Interface

of the design ideas for the `sdsi2ui` were gathered from Gillian Elcock's research into a Web-based user interface for SDSI [3].

The `sdsi2ui` gives the user a graphical view of the names and keys in their local namespace. However, it has limited functionality. The `sdsi2ui` cannot list the names in your cache that are not in your namespace. Moreover, it cannot perform the `acl-response` functionality, cannot verify a sequence (a certificate discovery proof) against an acl, and cannot display the certificates in the cache. These limitations make it a more suitable tool for just managing local names. Figure 3-9, displays a screenshot of the `sdsi2ui`.

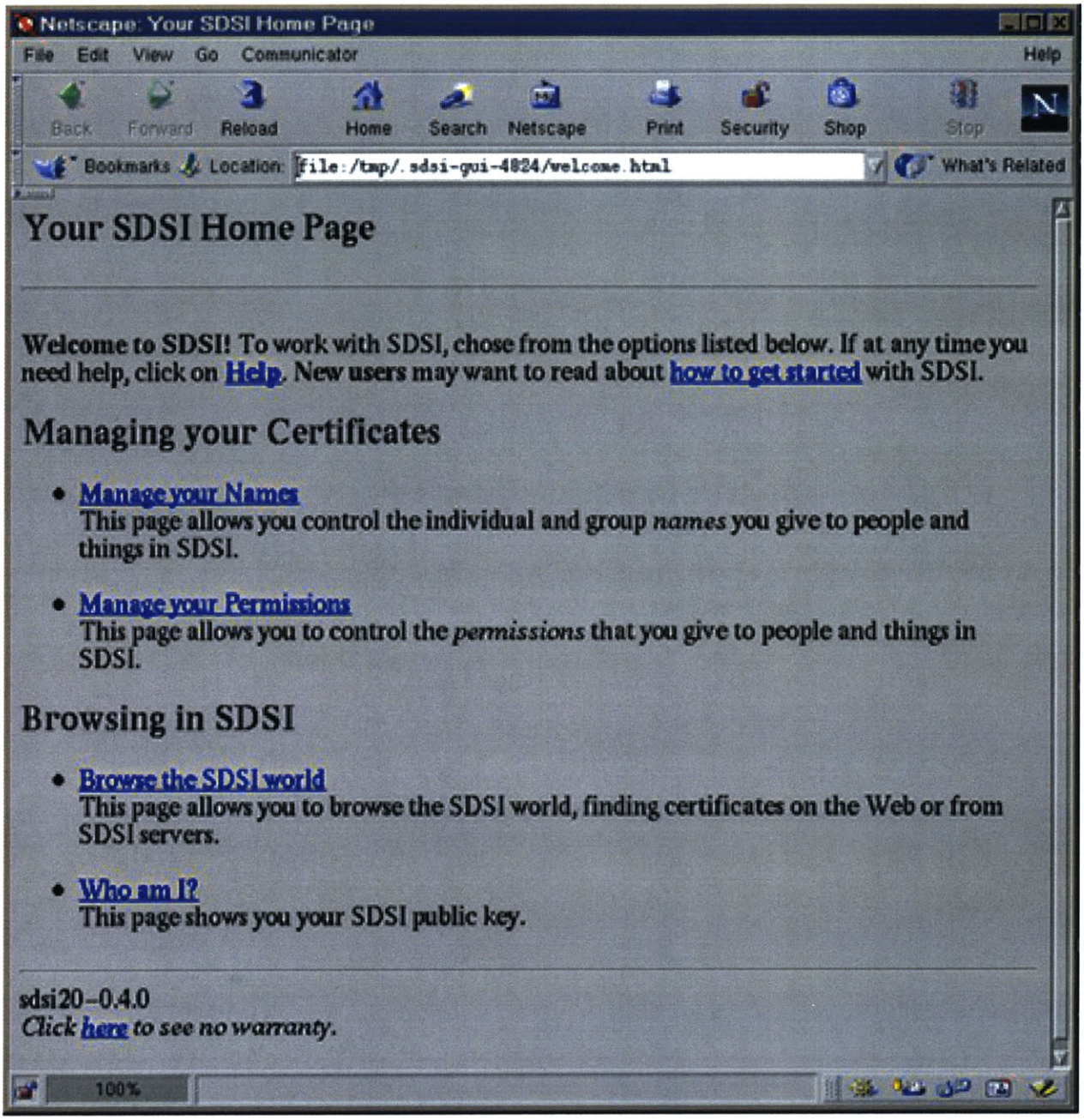

Figure 3-9: A Screenshot of the `sdsi2ui`.

Chapter 4

Design

4.1 Functional Specification

In this section, the basic desired functionality of the system is presented along with an overview of the basic black box requirements of the client.

The primary goal of the system is to allow users that present the necessary and valid credentials to access resources on a Web server implementing the Hypertext Transfer Protocol (HTTP). The user should be able to use a Web browser in the usual fashion, by clicking on links to access the resources at the specified URL. The system should support all HTTP methods, e.g. ("GET", "POST", "HEAD", etc.) and have adequate support for displaying MIME[1] types other than "text/html". The system should be built upon the Distributed Client/Server Model and allow the client to request appropriate services from the Web server in the traditional fashion.

The system must also support user retrieval or creation of the necessary credentials for authentication. Each principal in the system must have the functionality necessary to create/edit/store the credentials necessary for proving authorization.

[1]Multipurpose Internet Mail Extensions

4.2 Design Criteria

This section describes the important criteria necessary in a World Wide Web access control system. The following criteria are mentioned in order of most important to least critical.

4.2.1 Security

The most important design component is that the access control system must correctly enforce the security policy designated by the website administrator. The system needs to protect the resources stored on the website from unauthorized users. The system must provide a means for guaranteeing that user's possesing the private key of only authorized public keys have access to the data within the authorization specification.

4.2.2 Portability

One important consideration for an access control system is that many different users from locations all around the world would need easy access to the client software. It is presumed that the client will need to be installed on a variety of platforms. This necessitates a portable client software distribution. The client should be relatively small in size (e.g. < 2 MB) and easy to install.

4.2.3 Transparency

Often, complex systems require a lot of user intervention. The lower layers of abstraction are revealed to the user in an effort to allow the user to do more with the system. This needlessly confuses the user and adds complexity to the system. Limited user intervention and the correct level of abstraction are paramount in a simple to operate system. Thus transparency is an important design criterion.

4.2.4 Functional and User Friendly Interface

The user should be provided with sufficient support from the client software to gain access to SPKI/SDSI ACL protected documents. The client software should be easy to use and provide the correct level of abstraction to the user. The interface should not be significantly more complex than a graphical Web browser. Thus a functional user interface will be able to exploit all the features of the SPKI/SDSI PKI.

4.2.5 Extensibility

As new requirements for World Wide Web use emerge, it is important that the system be amenable to changes and extensions. It is a given that new versions will be needed in the future and modifications will need to be made. There must be sufficient room for updating the client software to implement changes in the World Wide Web, Internet protocols, and public key infrastructure.

4.3 Design Issues and Considerations

This section presents a discussion on the many possible implementation choices that can fulfill the design requirements. It also presents a discussion on the benefits and drawbacks of each.

4.3.1 Security

The basis for security is a major piece of the foundation when designing a secure system. The first design decision made about this system was to use the SPKI/SDSI Public Key Infrastructure as the tool for implementing reliable access control. The intention of this decision was to use the SPKI/SDSI software library to implement a secure system and thus analyze the effectiveness of SPKI/SDSI as a practical and effective public key infrastructure [15]. The design presented herein will base its protection on the computational security of public key cryptography. For Web document

transfers over the Internet, public key cryptography has proven to be one of the most used tools for providing authentication.

4.3.2 Portability

The client software consists basically of a web browser that can provide the user's credentials to a SPKI/SDSI protected web server and retrieve the requested document. The goal for distribution of the client software is to provide multiple versions that users can download from a website or group of websites. Each version would be built for a specific platform.

The proposition of implementing a Web browser entirely from scratch, complete with SPKI/SDSI handlers, was totally out of the question. Developing a web browser is a large and difficult project in and of itself. Therefore the only options explored were extensions/add-ons to existing web browsers.

The source code for the Netscape Communicator web browser is freely available so one can extend the functionality of the web browser. One could modify this source code by adding a few modules, add handlers for the SPKI/SDSI software, and maybe modify some of the existing code. This option allows one to build and package the entire client system so all the client needs to do is basically run the binary executable. The biggest drawback is that one would have to compile the entire web browser distribution (over 13 MB of source code) for every platform that one wanted to support. The time and effort that this would take in porting the software would be tremendous. There may also be legal issues related to the actual re-distribution of the modified Netscape Communicator software.

Netscape Communicator provides a mechanism, known as "plug-ins", to easily extend the capabilities of their web browser. Each browser plug-in has an associated MIME type for the type of data it handles. MIME types are used on the client side to send the incoming data to an appropriate application that can handle the data. Web

servers send this information to the client as the Content-Type header in HTTP. So for example, a data stream with MIME type "text/html" will be interpreted as html code and will be displayed by the Web browser. A data stream with MIME type of "application/pdf" will be interpreted as an Adobe Acrobat PDF file and will be sent to the apropriate application, whether external or a plug-in. This choice affords the benefits of only having to port a small set of source code to all desired platforms. Since plug-ins are small shared objects, they are easy to download and distribute over the World Wide Web.

Microsoft Internet Explorer also offers a similar extension interface through Active-X controls. This option had to be abandoned because of the developer's limited experience with Windows 32-bit development. Furthermore, a client was desired that would be available across multiple platforms. Due to the lack of UNIX support for Internet Explorer, using this option would hamper porting efforts.

4.3.3 Transparency

Using the Netscape Communicator source code would offer the opportunity to get a perfectly transparent interface. Since we have access to the lower level source code, we can modify the source code as necessary to provide functionality for our system at the right level of abstraction. Though, the full integration of SPKI/SDSI within the Netscape Communicator software, may prove to be overly time consuming and complex.

The option to use a Netscape plug-in gives a limited level of control over the transparency of your system. The plug-in applications programming interface (API) provides a specific level of functionality that your application has access to. Although the plug-in API has many features and moderate accessibility to the browser itself, one may have to overcome shortcomings with unconventional code use (otherwise known as "hacks") which expose some of the implementation details of the system. For example, there was an issue on what the proper behavior of the system should

be if a user did not have the plug-in installed. Correctness would dictate that the browser display an appropriate error message to the user. Since it is not possible to find out if a client has a plug-in or not, the server can only prepare for the positive case (the user has the plug-in). This exposes too much of the implementation to the user. Though, in time, modifications to the Web browser as well as the plug-in API may allow for more transparent operation.

4.3.4 Functional and User Friendly Interface

The most important consideration for the user interface was what the overall "look and feel" should be like. The available options ranged from a highly specialized graphical user interface (GUI) to a web browser interface. A tty client interface was abandoned due to the need for a more flexible client that could support display of Mime types other than text/html.

The client software should have an interface that only communicates with the user in exceptional circumstances. Small boxes would pop up on the user's screen informing the user of errors such as the inability to find a needed Java classfile. These boxes would also prompt the user for necessary information such as the user's private key password.

The user should also be able to manage his/her certificates and groups in a way that is compatible with the system. A graphical user interface and software library were already developed for the SPKI/SDSI PKI by graduate student Matthew Fredette [14]. This library already provides a functional user interface to managing public keys, names, and groups. It creates a certificate cache on the user's filesystem. The access control system client should be able to use this certificate cache with accessibility to all its features and state.

4.3.5 Extensibility

One important concern that was addressed was the extensibility of the distribution. If the system were to use the full Netscape Communicator source code distribution, clients would either have to download and compile the entire source for their particular platform/architecture or they would have to download a modified Netscape binary which may lack other functionality that they may wish to add. It would be time consuming and arduous for users to download and compile the entire Netscape source distribution (with the added SPKI/SDSI functionlity). Furthermore, the source distribution would have to be updated appropriately for each new version of Netscape to keep up with its changes, whereas modifying a small plug-in would entail much less effort in comparison. Similar arguments apply to the decision of whether to develop the server as an Apache module or to modify the freely available source code for Apache. We decided to implement the server as an Apache module.

Chapter 5

Implementation

The Geronimo system consists of 2 components: the client and the server. The client and server communicate via a protocol which provides the authentication and authorization of the client and thus allows the client to retrieve the protected document. This chapter discusses the design of the system protocol, a brief overview of the server implementation, and finally an extensive discussion of the client which was implemented for this thesis.

5.1 The Geronimo Protocol

This section describes the protocol for the proposed access control system. It was designed by Terrance Harmon, a fellow member of Project Geronimo. and is specified in his Bachelor's thesis [16].

The initial draft of the protocol was based on that used by the Digest Authentication Scheme [8]. Digest uses structures within HTTP to transfer information between the client and the server. The structures used are the WWW-Authenticate header and the Authorization Request header in HTTP 1.1 [1]. Digest has its own specification for what goes in those headers and it was used as a model for the initial draft. However, the initial version of the protocol has been modified; Terrance wrote his thesis before much of the SPKI/SDSI client and server were developed.

The original specification of the protocol uses an augmented BNF grammar, and relies on non-terminals defined in that document and other aspects of the HTTP/1.1 specification. The protocol document establishes formally the scheme by which clients and servers using HTTP should request and authenticate using SPKI/SDSI [16].

The important interactions of the protocol are as follows:

- The client sends an HTTP[1] request (without credentials) to the Apache server.
- The web server sends a response to the user containing the acl in the body with Content-Type "application/x-spki-sdsi".
- The Netscape web browser then loads the plug-in into memory and creates a new instance. The plug-in will then retrieve the user's public and private keys.
- The plug-in generates a SPKI/SDSI tag specifying the permission that the client is requesting.
- Using the acl, tag, user's public key, and user's certificate cache, the plug-in generates a sequence of certificates using the SPKI/SDSI certificate chain discovery algorithm discussed in section 3.1.6.
- The plug-in signs the request tag, and sends the signature and certificate chain to the server to verify. A copy of the user's public-key is included in the signature.
- The server verifies the request by extracting the public-key from the signature, recreating the request's tag, verifying the signature using the key, and verifying that there is a valid authorization chain from an entry on the acl to the key via the certificate chain presented. Upon successful verification, the server returns the requested document to the client. The HTTP status code "200 OK" is used for this reply. If the verification fails, a server-side customizable error page is

[1]The method can be any of the methods supported by the Apache webserver (e.g. "GET", "POST", "HEAD", etc.)

returned to the user. The HTTP status code "403 Forbidden" is used for this reply.

Please see Figure 5-1 for a diagram of the protocol.

The protocol assumes that the communication between the client and the server is conducted over a secure channel. This means that the network is not subject to active attacks, such as replay attacks, or passive eavesdropping. Unfortunately, the very nature of the Internet is that of an insecure channel. There would exist any number of entry points for man-in-the-middle attacks or replay attacks. The team decided that the best way to handle this is to run the protocol over an Secure Sockets Layer (SSL) [18] connection. SSL is a protocol designed by Netscape Communications to enable encrypted, authenticated communications across the Internet. SSL is used mostly (but not exclusively) in communications between web browsers and web servers. SSL will help to maintain the integrity of the data transmitted in the protocol. SSL also provides confidentiality[2] of data. Passive eavesdroppers and attackers actively trying to subvert the system are a concern. For example, without a secure channel, the system is prone to replay attacks and man-in-the-middle attacks. Also, it would be easy for an eavesdropper to see the contents of a protected document as is it sent to an authorized user.

5.2 Server Implementation

This section describes the SPKI/SDSI server. It was designed and implemented by Dwaine Clarke, as part of his M.Eng. thesis. The Apache server is one of the most popular servers on the Internet. It is free, and its source is open; its core capabilities can be extended by implementing modules using the server's Application Programming Interface (API). Its 'modular' architecture facilitates the extending of its functionality by independent developers. Because of these reasons, we chose to

[2]Since all data is encrypted, even passive eavesdroppers will not be able to read the data

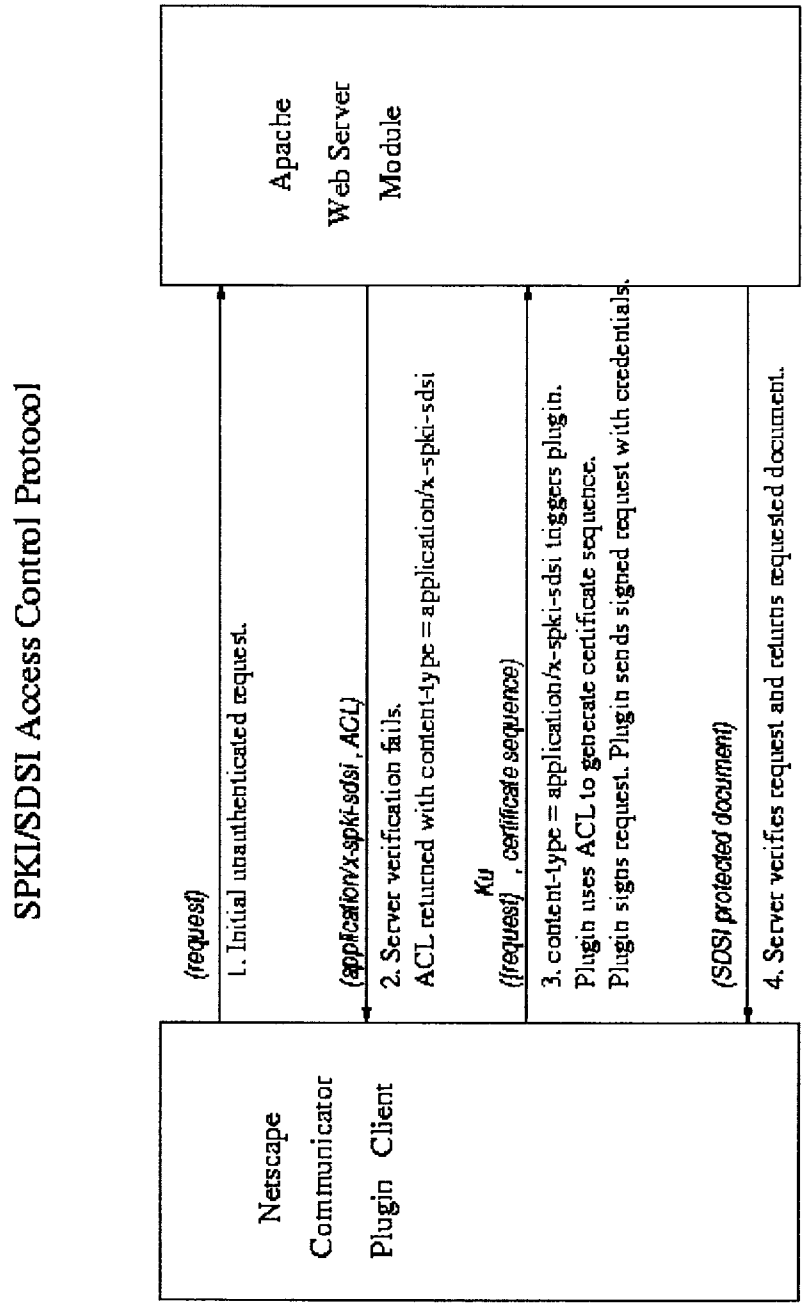

Figure 5-1: A diagram of the protocol used in the access control system.

implement a SPKI/SDSI Apache module to incorporate the infrastructure into the web server. Examples of other existing modules are those that provide basic and cookie-based authentication, rewrite urls dynamically, translate from one language to another, and facilitate server-side scripting.

The SPKI/SDSI Project Geronimo module is an Apache Auth Handler. It intercepts the processing of the Apache request during its authorization phase. Its principal functions are to protect web objects using SPKI/SDSI ACLs, and to verify signatures and certificate chains in requests coming from the plug-in. If the client's request is successfully authenticated and authorized, the module continues to pass the handling of the request to the other Apache functions; otherwise an HTTP error code, and a server-side customizable error page, are returned.

Web pages and other files are protected on a per-directory basis by creating a SPKI/SDSI .htaccess file (See Figure 5-2. Files are specified relative to the server's document root.) in the directory. The first directive in this file signals to Apache to use the SPKI/SDSI Auth module to handle the request. The "AclFile" directive specifies the file with the directory's ACL. "ErrorFile" points to the location of the directory's customizable error page. The pathnames for AclFile and ErrorFile can be specified fully, or relative to the server's document root.

```
SetHandler spki/sdsi-auth
AclFile demo/spki-sdsi-acls/ABC_financial_acl
ErrorFile demo/ABC/public/error.html
```

Figure 5-2: Sample .htaccess file.

To verify requests, the module first extracts the client's public-key from the signature on the request. The full tag for the request is signed by the client and the module recreates the tag using the HTTP request information. It then verifies the request's signature against this tag using the public-key from the signature. After authenticating the request, the module then checks if the request is authorized by checking

if the certificate sequence provided in the request creates a valid authorization chain from an entry on the acl to the public-key from the request's signature. It calls the SDSI2.0 library's `sdsi2_acl_check` procedure with the acl protecting the directory, the client's public-key, the tag it created, the request's certificate sequence, and the current time. The SDSI2.0 library procedure reduces the certificate chain and then compares the result to each entry on the acl. If there is a valid authorization chain form an entry on the acl to a public key,then the request is authorized. If the request is unauthenticated, or unauthorized, the module returns the appropriate HTTP error code. Figure 5-3 illustrates the authentication and authorization processes.

Server administrators can also design their own html error pages to be returned to clients when the authentication/authorization process fails. These pages are kept in the server's document root hierarchy, and are also specified on a per-directory basis using the "ErrorFile" directive in the .htaccess files. They are returned to the client with the HTTP error code. The idea is that the server administrator, or the maintainer of directory on the server, can put pertinant and useful information on this page to assist users who should be authorized, but are not able to gain access to the directory, because they do not have the necessary certificates, say.

Dwaine Clarke also designed and implemented a CGI-based interface to manage SPKI/SDSI acls on the web. Different CGI forms, used to create, view, and edit acls, are generated by a single script. This script is password protected using HTTP's Basic Auth and can only be run over a secure ssl connection.

5.3 Client

We decided to develop the Geronimo client as a Netscape Communicator plug-in. This decision was mainly based on the strong arguments presented by portability and ease of extensibility. The plug-in would be built for each supported platform and distributed on a website. This section discusses how plug-ins work, in general, and then describes the implementation details of the Geronimo client.

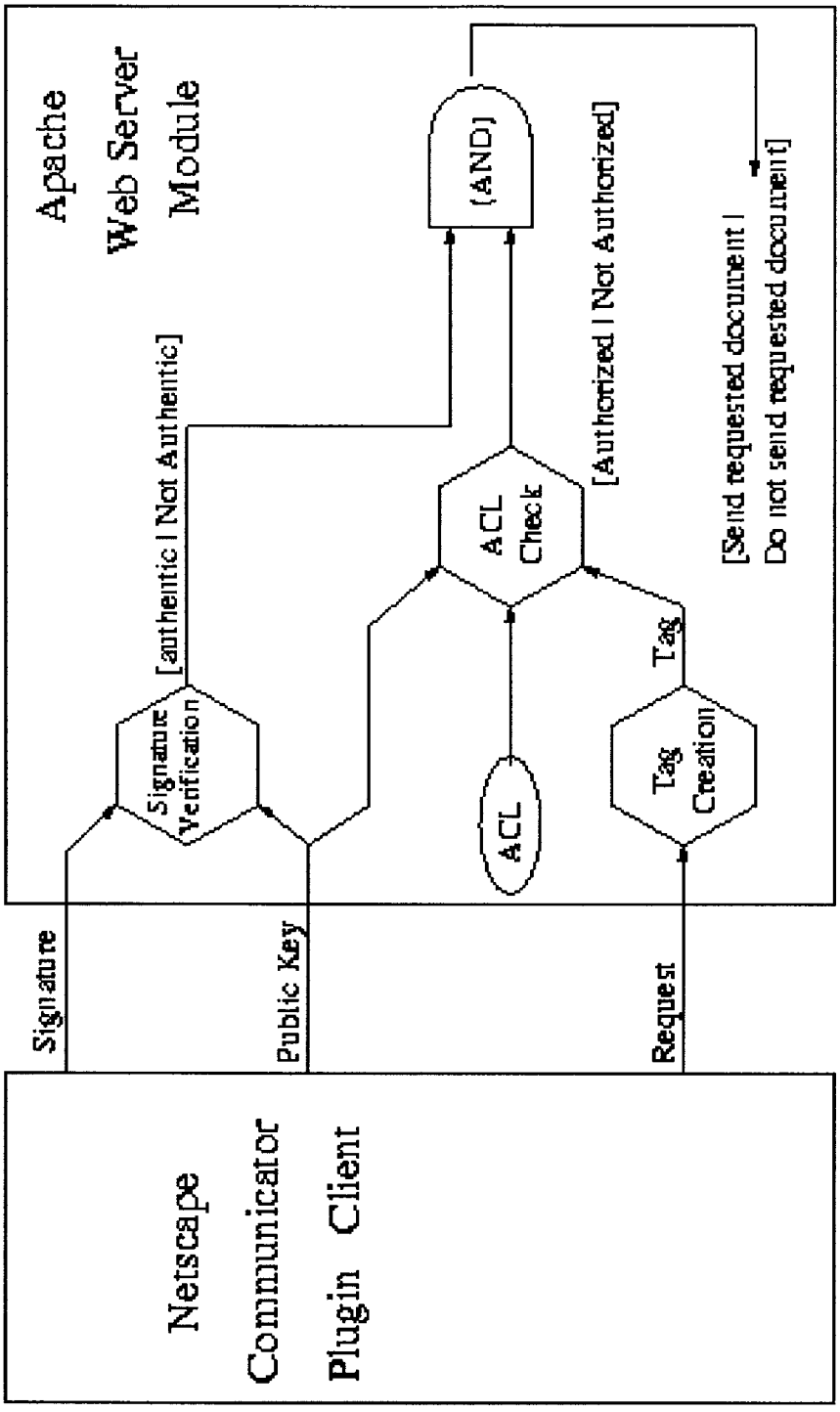

Figure 5-3: A diagram of the server-side access control check.

5.3.1 Development

Netscape Corporation provides a Plug-In Software Development Kit [2] that instructs users how to construct a plug-in. Since the SPKI/SDSI library was written in the C programming language, we felt that for compatibility reasons, it would be best to develop the plug-in in C as well. The plug-in development kit facilitated this decision as most of its source code was also written in C.

5.3.2 Plug-In Basics

Browser plug-ins get registered by the browser upon startup and get invoked by the browser when it receives data with the registered MIME type. For example, the Adobe Acrobat Reader plug-in registers itself as "application/pdf"; the Web browser initializes the plug-in when it receives data of this type. Netscape plug-ins can be invoked by one of two different ways: by a "full page" data response, or "embedded" as part of a larger HTML document.

When a Web server sends a data response to a browser, the browser looks at the Content-Type header (sent as part of the HTTP protocol) to determine the MIME type of the data. Communicator looks up this MIME type and loads the plug-in if it finds a plug-in registered to that type. This loads the plug-in into full-page mode since the entire full page data response is consumed and processed by the plug-in. Full-page plug-ins are commonly used for document viewers, such as Adobe Acrobat [2].

The other way a plug-in can be invoked is through an embed or object HTML tags. This embedded plug-in is a part of a larger HTML (with Content-Type text/html) document and is loaded at the time the document is displayed. Embedded plug-ins are commonly used for multimedia relating to the text content on the page. For example, on a web page about a particular music group, there may be an embedded music clip of type "audio/x-aiff". Depending the plug-in installed for that type, the music clip would be played while the client viewed the page. Please see Figure 5-4

to see an example of the embed HTML tag. the embed tag allows the Web page designer to control the width, height, visibility, etc. of the embedded object [2].

```
<EMBED SRC=''sound_clip.aiff'' TYPE=''audio/x-aiff'' HIDDEN>
```

Figure 5-4: An example of the use of the HTML embed tag.

5.3.3 Runtime Model

When Netscape starts up, it looks for all available plug-ins in either the `/usr/local/lib/netscape/plug-ins` or `$HOME/.netcape/plugins` directory. Then it queries each of the plug-in libraries to determine which MIME type(s) it can handle, and registers each plug-in library for those MIME types. Table 5.1 outlines the stages in the life of a plug-in after its MIME type(s) have been registered.

1. **Initialization**. When Netscape encounters a registered MIME type (either embedded in an HTML page or as a "full page" data stream), it dynamically loads the associated plug-in into memory (initialization), if it hasn't been loaded already. Netscape does this by calling the plug-in API function `NPP_Initialize` which the plug-in has defined.

2. **Instance Creation**. Netscape then calls the plug-in API function `NPP_New` to create a new instance of the plug-in. Multiple instances of the same plug-in can exist simultaneously if there are multiple embedded objects on a page, if more than one Communicator window is open with "full page" data of the associated MIME type, or a combination of these.

3. **Instance Deletion**. A plug-in instance is deleted when a user leaves the instance's page, or closes the window. Netscape informs the plug-in instance that it will be deleted by calling the `NPP_Destroy` function. The plug-in can then free up allocated memory and close open files before being deleted.

4. **Shutdown**. When the last plug-in instance is deleted during a particular Netscape session, the plug-in code is unloaded from memory. Netscape calls the function `NPP_Shutdown` to alert the last plug-in. The plug-in can then free global resources accordingly.

Table 5.1: The stages in the life of a plug-in.

The plug-in is implemented as a relocatable shared object in C for UNIX. This differs from archived objects[3] in that all the references to objects must be defined in that shared object. Another minor dilemma that was encountered was that the SPKI/SDSI library is implemented as a static archived library. This is not compatible to be linked with the plug-in shared object. The solution that was decided upon was to use a special shared library building tool, libtool [12], to link the plug-in with the SPKI/SDSI library.

5.3.4 Geronimo Plug-In Initialization

As the protocol describes (section 5.1), the user initially sends a "regular" HTTP request for a document on a website. Since the document is protected by the SPKI/SDSI Geronimo access control system, the webserver returns the "application/x-spki-sdsi" MIME type document containing the acl. A new Geronimo plug-in instance is created to handle the current request.

During initialization, the plug-in retrieves the user's public key by looking for the "my-principal.sdsi2" file in the user's .sdsi directory ($HOME/.sdsi). The Public key is then parsed and stored in memory. The plug-in then attempts to retrieve the user's private key from the user's identity file in their .ssh directory. Since the private key is encrypted with the password that the user selected during installation (see section 3.2.4), a small Java window[4] pops up to prompt the user for their password. Using this password, the plug-in retrieves the user's private key using the `sdsi2_ssh_parse_secret_identity` function.

5.3.5 Instance Creation

After initialization, Netscape calls the plug-in API function `NPP_New` to create a new instance of the plug-in. This function first checks to see if it was created in embedded mode or in full page mode (Please see section 5.3.2 for discussion of these

[3]In Unix, these static archived libraries are made using the 'ar' program.
[4]Please see Section 5.3.8 which discusses this password box

modes). If the instance was created in embed mode, then this instance is embedded in the session window[5].

After the instance is created, Netscape calls the plug-in API function `NPP_NewStream` to alert the plug-in instance that it is going to receive `stream`[6] data from Communicator. If the plug-in is called in embed mode, this data is simply the arguments passed to the plug-in in the <EMBED> HTML tag. If the plug-in was called in full page mode, this data is the entire data stream returned by the server in the body of the HTTP response. If the plug-in is the session window instance, it ignores this data and simply proceeds to display itself and closes the data stream it received from Communicator. If the plug-in is called in full page mode, it configures the data stream to be sent to a file. Netscape then calls the `NPP_StreamAsFile` function to notify the plug-in of the filename where the data is stored on the system.

5.3.6 Plug-In Operation

In the `NPP_StreamAsFile` function, the plug-in checks to see if the session window has been started by checking the `session_start` flag in the global session state. If it is set, then there already is a session window open and the current instance does not have to create one (since it is the responsibility of the very first plug-in instance to open the session window). If it is not set, then the current plug-in instance calls the `open_session_window` function that opens a custom javascript url which creates a small borwser window. This new browser window contains an embedded plug-in and is known as the session window. Please see Figure 5-5 to see the source code for `open_session_window`.

The plug-in instance then opens the file containing the data received from the server's initial response of MIME type ``application/x-spki-sdsi''. The plug-in then extracts the ACL contained within that file and parses it into canonical

[5]Please see section 5.3.7 for more discussion on the session window

[6]A `stream` is a data structure that represents a URL and the data it points to.

```
/* opens the small sdsi session window
 *
 */
int
open_session_window(NPP instance) {
  NPError  err;
  char *url;
  char url1[400] = "javascript: var win =
window.open('','displayWindow','toolbar=no,scrollbars=yes,
resizable=yes,width=250,height=250'); var d = win.document;
d.write('<HTML><HEAD><TITLE>SDSI Session Window</TITLE></HEAD>
<BODY><embed type=\"application/x-spki-sdsi\" hidden=\"TRUE\"
name=\"SDSI Session Window\"></embed><H1>SPKI/SDSI Authentication
Session Window</H1><BR><P>This window authenticates the user: ";
  char target[30] = "_self";
  char url2[250] = ". Remember to close this window to end your SDSI
session.</P><FORM><CENTER><INPUT TYPE=\"button\" VALUE=\"Close\"
onClick=\"window.close();\"></CENTER><BR></FORM><br></BODY></HTML>');
d.close(); history.go( -1);";

  url = (char *) NPN_MemAlloc(800 * sizeof(char));

  strcpy(url,url1);
  strncat(url,session.user,strlen(session.user));
  strcat(url,url2);

  err = NPN_GetURL(instance, url, target);

return err;

}
```

Figure 5-5: Source code for the open_session_window function.

form. Then, it creates a SPKI/SDSI tag[7] which represents the permission that the client is requesting, and signs the tag. The plug-in calls the `acl_response` function provided by the SPKI/SDSI library for implementing the certificate chain discovery and generating the certificate sequence proving membership on the acl. If no sequence is found the `acl_response` function returns a null sequence ``(sequence)''. Then plug-in finally sends a new request to the web server with the tag signature and sequence in the header of the request.

In the event that the credentials do not verify on the server end, the server sends back a customizable error page which may instruct the user how to get the necessary certificates to prove authentication if indeed he/she is authorized to view the page. If the credentials do verify, then a new browser window is created containing the protected document received from the server.

5.3.7 Session Window

After the first plug-in instance is created, the plug-in displays a small "session window" indicating that the user is currently in a SPKI/SDSI authentication session. This session window contains a small instance of the plug-in (using the <EMBED> HTML tag) that holds the user's key pair in the browser's memory. The initial plug-in instance destoys itself by "going back" in the browser's history. The instance opens a javascript URL, ``javascript:history(-1)'' to go to the previous page in the browser's history. This then makes sure that the only plug-in instance is contained within the session window. Thus, the user can close the session window at any time to destroy the last plug-in instance and hence shutdown the plug-in entirely. At this point, the plug-in session is ended. Basically the only purpose of the session window is to keep the plug-in code in Netscape's memory to keep the session state (e.g. the public and private keys) alive. This is so that the user does not have to re-enter their password every time they access a protected document within each Netscape session

[7]Please see section 5.3.10 for more discussion on the components of the tag

since when the last plug-in is deleted, the entire plug-in global state is shutdown with `NPP_Shutdown`. Please see Figure 5-6 for a screenshot of the session window.

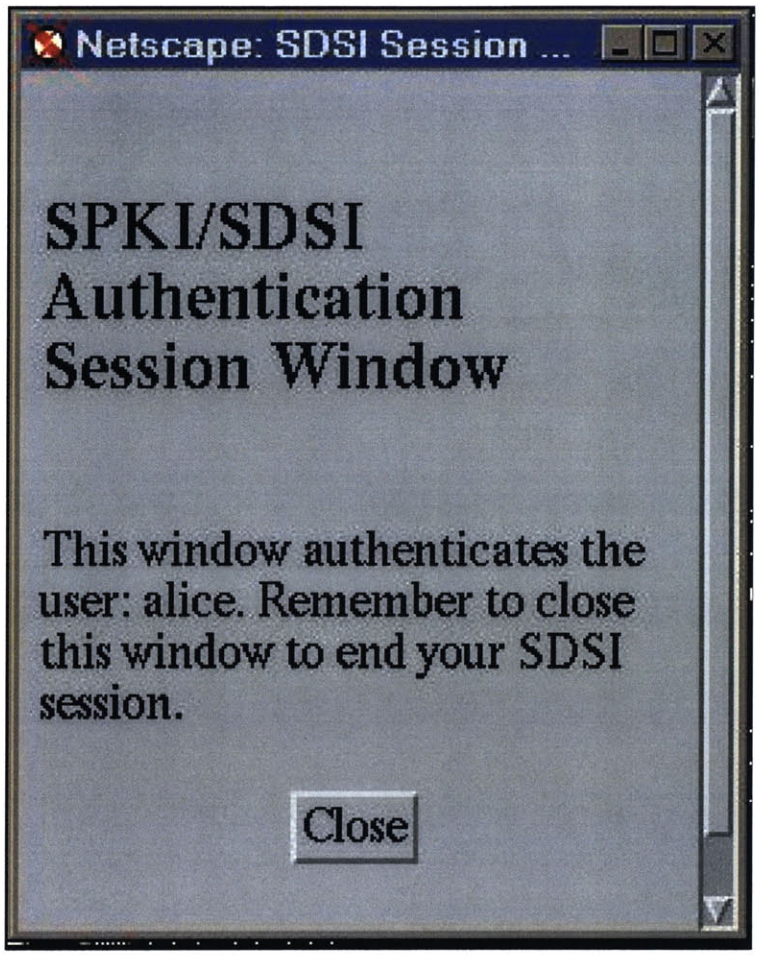

Figure 5-6: A screenshot of the session window.

5.3.8 Password Window

The plug-in pops up a small java window during the initialization phase to prompt the user for their private key password. This window is created using the AWT functionlity in the standard Java development package. This development package allows for quick and easy implementations of windows and frames. The java window is created by running the Java interpreter with the `PasswordFrame` Java class which is placed in the user's .sdsi directory during plug-in installation. Please see Figure 5-7 for the Java source code that implements this password window.

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class PasswordFrame extends Frame
    implements ActionListener
{
    PasswordFrame thisFrame;
    PasswordPanel myPasswordPanel = new PasswordPanel();
    public String myPassword = "";

    public PasswordFrame(String title)
    {
        super(title);
        this.setSize(430, 140);
        myPasswordPanel.setSize(420, 130);
        this.add("Center", myPasswordPanel);
        this.setResizable(true);
        this.setLocation((int)CENTER_ALIGNMENT,(int)CENTER_ALIGNMENT);
        this.setVisible(true);
        myPasswordPanel.myTextfield.addActionListener(this);
    }
    public void actionPerformed(ActionEvent e)
    {
        System.out.print(e.getActionCommand());
        myPassword = e.getActionCommand();
        this.dispose();
        System.exit(0);
    }
    public static void main(String[] args)
    {
        PasswordFrame myFrame = new PasswordFrame("Please enter your SPKI/SDSI password");
    }
}
class PasswordPanel extends Panel
{
    Label myLabel = new Label("Please enter your SPKI/SDSI password below and press return");
    TextField myTextfield = new TextField(10);

    public Insets insets() {
        return new Insets(10, 10, 10, 10);
    }
    public PasswordPanel()
    {
        myTextfield.setEchoChar('*');
        setLayout(new GridLayout(0,1));
        add(myLabel);
        add(myTextfield);
    }
}
```

Figure 5-7: `PasswordFrame.java` source code which implements the password pop up frame.

Netscape Communicator's plug-in API allows for plug-in developers to call Java native methods directly from plug-in C source code using a system called Liveconnect [2]. During the development phase, the plug-in initially used this functionality to open the Java password prompt window. This was abandoned for the current implementation because during the testing phase, it was discovered that the Netscape web browser main thread of execution only waited for the user to enter the password without displaying the password prompt frame at the same time. This problem was due to the fact that Netscape used the same execution thread to both display the frame and wait looping for the response at the same time. Therefore the password frame window was implemented in another process run from the Java interpreter.

5.3.9 Globals

The Geronimo plug-in keeps a global session structure in memory containing state about the current client session. This structure is global so that all plug-in instances can have access to this global state for each session. The data that is kept in the struct is: the users public key, private key, username, home direcory location, sdsi directory location, and a flag indicating if the session window was displayed. Please see Figure 5-8 for the C code describing the global session structure.

```
typedef struct _session_type
{
  sdsi2Key *pubkey;
  sdsi2Key *privkey;
  char *user;
  char *home;
  char *sdsi_dir;
  char *stash_file;
  int session_start;   //Used to check if session window was started.
} session_type;

static session_type session;
```

Figure 5-8: Specification of global plug-in session state.

5.3.10 Permissions Specification

For this access control system, one needs to carefully consider what information needs to be present in the tag. The specified tag format needs to be such that will allow the server administrator flexibility in implementing their desired security policy. The main considerations are to have the tag specify either just the HTTP [1] method and location, or to include other state information such as allowed client IP Address and allowed time range for access. These options would allow the server administrator to specify a stricter security policy for their website.

5.3.11 User Certificate Management

The user has both the `sdsi2sh` shell and `sdsi2ui` user interface at their disposal to create, cache, and sign certificates (please see section 3.2.5). The user can use the shell (through convenient shell scripts or otherwise) to enter certificates into their cache. The user would, possibly through a webserver, retrieve the certificate(s) issued to them by the server administrator and then enter these into their certificate cache.

5.4 Limitations

The Netscape plug-in API did not fit perfectly with the functionality that the client needed. Thus, there were limitations to the prototype client implementation that hindered ideal operation.

One of the more major limitations was the inability to control Netscape's caching scheme. After a user was authenticated and the protected document was retrieved by the plug-in, it fed the protected document to Communicator via the `NPN_NewStream`, `NPN_WriteStream`, and `NPN_DeleteStream` plug-in API functions. This is the main functionality that Netscape provides for representing URLs and the data they contain. It allows the plug-in to display documents of different MIME types through Communicator. Unfortunately, Netscape caches pages that are consumed and displayed by Communicator itself, Thus, protected documents may be stored to the filesystem as

a cached page. This is a security hole that has not been resolved. Possible solutions may be to insert a meta HTML tag into the document with a no-cacheing directive, such as <META HTTP-EQUIV="pragma" CONTENT="no-cache">, or ask Web page authors to write thier HTML pages with such an HTML tag included.

Netscape provides functionality through Liveconnect for plug-ins to access Java and Javascript functions. Initially, it was intended that the Geronimo client use this functionality to access Java AWT functions to display the password pop-up window. However, during the development phase, we realized that Netscape used the same thread of execution to run the Java code and wait looping for the user's private key password to be entered. This caused Netscape to hang indefinitely waiting for the user to enter their password in the Java window that wasn't drawn. As a result, the password box had to be displayed in a separate process with a pipe to the plug-in to feed the password to it. Therefore, the client depends on the local system having a Java interpreter installed in the user's execution path.

The plug-in uses a javascript URL to display the session window. It also uses a javascript url ("`javascript:history(-1)`") to go back to the previous page in the browser's history. If the user does not have javascript enabled on their browser, this functionality will not work at all. Currently Netscape does not provide functionality for plug-ins to control window sizes, so the session window must be implemented using javascript. Likewise, the plug-in cannot access the browser's history except through javascript.

One drawback is that if the user does not have the plug-in, he/she is unable to actually find that out since the first server response page is of a MIME type that won't be registered with the Netscape browser. The browser will simply alert the user that it does not know how to handle the ``application/x-spki-sdsi'' MIME type and will prompt the user to save the data as a file.The user would be unable to know that the reason they were not receiving the requested document is that they did not have the required plug-in. Many different schemes were discussed to solve this problem,

however no solution was optimal. The solution that is currently used is to put the responsibility on that of the website administrator to alert users of their websites that certain documents require the SPKI/SDSI Geronimo plug-in to work effectively. We will see how this works in practice and make modifications as necessary.

Chapter 6

Example

This chapter gives an example run-through of the Geronimo access control system operation detailing the messages actually passed between the client and server.

In this example, Bob is the system administrator of ABCD Corporation's main website. ABCD is a large multinational corporation with offices in over 100 diffreent countries. ABCD Corporation's executive committee would like to make the budget and corporate strategy information available to all the regional managers in each of its worldwide offices. Since this information is sensitive, Bob needs to make sure that only the regional managers are able to access the information on the website. Bob decides to use the SPKI/SDSI Project Geronimo Access Control system to protect the sensitive data on the website.

Bob has installed an Apache webserver with the `spkisdsi_auth` Apache module auth hander for the Geronimo project. He has also created a custom error page that tells users who should be allowed access but who are denied access to contact him by email and send their public key so that he can generate the necessary certificates to add them to the `ABCD_regional_managers` SPKI/SDSI group which is on the webserver's ACL for the protected pages. The location of the ACL and the error page is specified in a .htaccess file located in the financial directory. Here is the ACL:

```
(acl
 (entry
  (name
   (hash md5 |Gv9v77A2VRLRjckzhh+fiw==|)
   ABCD_executive_committee)
  (tag
   (http
    (* set GET POST)
    (*
     prefix
     http://www.corp.abcd.com/financial/)))
 )
 (entry
  (name
   (hash md5 |Gv9v77A2VRLRjckzhh+fiw==|)
   ABCD_regional_managers)
  (tag
   (http
    (* set GET POST)
    (*
     prefix
     http://www.corp.abcd.com/financial/)))
 ))
```

Alice has just been appointed regional manager for the Japan office of ABCD Corporation. The SPKI/SDSI Project Geronimo Netscape plug-in has already been installed in her account and she has already generated her key-pair and selected a password "fiddlesticks" for encrypting her private key.

In the memorandum that Alice received from corporate headquarters in Cambridge, MA, Alice sees the url of the corporate website that Bob has created containing the budget information that she needs to review for an upcoming meeting. She enters the url into her Netscape web browser and then sees the home page that Bob has created. Alice then clicks on the link labeled "ABCD Budget 2000". Her web browser sends the following request to the ABCD webserver:

```
GET /demo/ABC/financial/budget2000.html HTTP/1.1
Accept: image/gif, image/x-xbitmap, image/jpeg, */*
```

```
Accept-Language: en-us,ja;q=0.7,zh-cn;q=0.3
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0
Host: wmm.corp.abcd.com:8080
Connection: Keep-Alive
```

The webserver interprets this request and realizes that Alice is trying to access the budget page in the SPKI/SDSI Geronimo protected financial directory. It then sends this request to the spkisdsi_auth module auth handler for a determination if access is allowed. The module sees that the request does not have any credentials and therfore should be denied access. It sends back the directory's ACL[1] in the body with Content-Type ``application/x-spki-sdsi'':

```
HTTP/1.1 200 OK
Date: Sun, 14 May 2000 15:26:01 GMT
Server: Apache/1.3.9 Ben-SSL/1.37 (Unix)
Connection: close
Content-Type: application/x-spki-sdsi

HOST: rooster.lcs.mit.edu
PORT: 8081
URI: /demo/ABC/financial/budget2000.html
URL: http://rooster.lcs.mit.edu:8081/demo/ABC/financial/budget2000.html
ACL: (acl (entry (name (hash md5 |Gv9v77A2VRLRjckzhh+fiw==|)
     ABC_executive_committee) (tag (http (* set GET) (* prefix
     http://rooster.lcs.mit.edu:8081/demo/ABC/financial/)))) (entry
     (name (hash md5 |Gv9v77A2VRLRjckzhh+fiw==|) ABC_auditors) (tag
     (http (* set GET) (* prefix http://rooster.lcs.mit.edu:8081/demo/ABC/financial/)))))
```

Netscape then sees that the Content-Type is ``application/x-spki-sdsi'' and finds the Geronimo plug-in registered for that MIME type. It loads the plug-in into memory and calls `NPP_Initialize` to initialize the plug-in. The plug-in retrieves the user's public key and executes the `PasswordFrame` Java class from the Java interpreter in a separate process to prompt Alice for her private key password. A small frame pops up and Alice enters "fiddlesticks" into the text field and hits the enter/return

[1]Please note that each entry in the body is contained on one continuous line. There are no newline characters except at the end of each entry as a delimiter. The example shown here has newline breaks only for formatting purposes.

key. The plug-in decrypts her private key from .ssh/identity in Alice's home directory and stores it in memory.

After initialization, a new plug-in instance is created to handle Alice's current request. The instance realized that it was called with a full page request and thus knows it is not the session window. It also realizes that the session window has not been started, so it executes the `open_session_window` function which opens the SPKI/SDSI Geronimo session window.

The plug-in receives the data from the server in a file. The file contains the ACL that protects the financial directory. The plug-in opens this file, extracts the ACL, and parses it into canonical form. The plug-in then prepares a tag for the "http://www.corp.abcd.com/financial/budget2000.html" url that Alice is requesting. Here is the tag:

```
(tag (http GET http://www.corp.abcd.com/financial/budget2000.html))
```

The plug-in then signs the tag with Alice's private key that it has stored in memory. It then runs the `acl_response` function provided by the SPKI/SDSI library. Since Alice is a new regional manager, she does not yet have the necessary certificates to generate the certificate sequence. Thus the plug-in gets a null sequence ``(sequence)''. The plug-in finally sends back the tag signature and sequence to the server:

```
GET /demo/ABCD/financial/budget2000.html HTTP/1.0
User-Agent: SPKI/SDSI2 Geronimo Access Control Plug-in 1.0
Host: wmm.corp.abcd.com:8080
Connection: Close
Spki-Sdsi-Signature: (signature (hash md5 |20N8...
Spki-Sdsi-Sequence: (sequence)
```

Since the sequence is null, the server denies access to the requested document. It sends back Bob's custom error page which has contact information and instructions on how to get the certificates Alice needs to prove membership in the `ABCD_regional_managers` group that Bob created. The following is a portion of the error page that the server sends to Alice:

```
<HTML>
 <HEAD>
  <TITLE>ABCD Corporation: public/error.html</TITLE>
 </HEAD>

<BODY Text=#000000 bgcolor=#FFFFFF>

<center><b><font size=7 color="#000099">
<u>ABC Corp <br>SPKI/SDSI Error Page </u></font></b>
</center>
...
```

The plug-in receives this response from the server and feeds it into a Netscape Communicator stream using the plug-in API functions `NPN_NewStream`, `NPN_WriteStream`, and `NPN_DeleteStream`. The plug-in displays this page in a new Netscape browser window by specifying the display mode as "_blank". Alice then reads the error page and realizes that she needs to contact Bob to get the necessary certificates.

Alice sends Bob an email with her public key and gives Bob a call to make sure he got the email and knows that it's really her. Alice could have used more secure method of sending her public key to Bob, but one is not presented for the sake of clarity. Bob adds Alice's key to the `ABCD_regional_managers` group using the SPKI/SDSI `sdsi2sh` shell (section 3.2.5). He then emails the group membership certificate to Alice. Alice finally adds this certificate to her cache using the `sdsi2sh` shell.

Alice clicks on the budget page link in Netscape again. Since the session window is active, the plug-in is still loaded in Netscape's memory and she is not prompted for

her private key password again. The browser sends an identical request to the server as described above. The server sends back the ACL in the body with Content-Type ``application/x-spki-sdsi''. A new plug-in instance is created. This plug-in instance retrieves the ACL and runs the acl_response function that generated the valid certificate sequence proving that Alice is a member of the ABCD_regional_managers SPKI/SDSI group. The plug-in finally sends the certificate sequence along with the request tag signature:

```
GET /demo/ABCD/financial/budget2000.html HTTP/1.0
User-Agent: SPKI/SDSI2 Geronimo Access Control Plug-in 1.0
Host: wmm.corp.abcd.com:8080
Connection: Close
Spki-Sdsi-Signature: (signature (hash md5 |20N8...
Spki-Sdsi-Sequence: (sequence (def rule12 (sequence (cert (issuer...
```

The server then verifies the signature and certificate chain. Success! Alice's request is authentic and authorized. The server finally returns the requested document to the plug-in which displays it in a new browser window.

Chapter 7

Extensions/Future Work

7.1 Distributed Certificate Discovery

In the system described, a user obtains new certificates, adds them to his/her cache, then runs the certficate chain discovery algorithm to derive all the new names and new authorizations that pertain to him/her. How about applying the certificate chain discovery algorithm to a distributed setting, where each public-key maintains a server which can issue certificates on behalf of the key? There are several servers on a network and each maintains its own certificate cache. In such a system, how does a key determine if a certficate issued by another server now authorizes him to access a resource it previously wasn't able to? These questions were raised by Sameer Ajmani, a fellow graduate student in the Lab for Computer Science, who is also using SPKI/SDSI in his thesis. His research focuses on certificate chain discovery performed over a distributed network of certificate-caches rather than one local certificate-cache.

Our SPKI/SDSI team looked at this problem briefly. With each public key, K, maintaining its own server, S, one possible solution is for S to store and return the certificates signed by K, and a list of public keys that are in the set K N for any name N for which K has issued a name cert. For each public key in this set, the cert chain which reduces to it is also stored. Servers communicate with each other to propagate new certificates throughout the system.

Other opportunites for research in this area exists. How would a solution described in the previous paragraph work in practice? Do other practical solutions exist, and if so what are their advantages and disadvantages? How should the protocol for propagating a new certificate be specified? SPKI/SDSI is very flexible, and Sameer's thesis describes at least one application of the infrastructure to a distributed system.

7.2 Distribution

The Geronimo development team plans to distribute the plug-in source code as a software package available for free download from the project website[1]. This will allow other developers to use the software and give us helpful feedback on any bugs or improvements for the source code. It may also spur further developemtn with the SPKI/SDSI PKI and help to guide other developers in the development process.

Currently, the plug-in has only been built and tested for two different platforms: Sparc-solaris-2.6 and i686-Linux-2.2.5-15. There are future plans to port the plug-in to other Unix systems and architectures. Since the SPKI/SDSI library distribution currently only builds for Unix platforms, there are currently no plans to port the plug-in to Win32 or WinNT systems.

7.3 Other Browsers and Web Servers

The current Geronimo plug-in implementation is only compatible with Netscape Communicator. This is an unecessary limitation on the user since users tend to have specific preferences for using a particular web browser. However, there are no current plans to develope a 'plug-in' for other web browers, such as Microsoft Internet Explorer. Similarly, there are no current plans to develop a server module for webservers other than Apache.

[1]The URL is: `http://theory.lcs.mit.edu/~cis/sdsi-geronimo.html`

7.4 Security

The current system implementation only allows for the authentication of the client. Server authentication is needed to prevent man-in-the-middle attacks. Future implementations will need to support server authentication. There are no current plans to design a protocol that supports authentication of the server.

The only server verification that is available is provided through SSL, which is based on the X.509 PKI. A possible extension to the project would be to develop an implementation of SSL using SPKI/SDSI certificates. It is possible that such an implementation could be merged with the Geronimo protocol to provide an enhanced access control service.

There is some discussion of a possible combination of Project Geronimo with current research on a secure filesystem[2] called the Self-Certifying Filesystem.

7.5 Performance

The current implementation takes approximately 20 seconds to perform signing operations with the SPKI/SDSI software library. Ths is due to the use of an unoptimized multi-precision library for computing with large numbers. There are plans to replace the present multiprecision code with the GNU MP library [11]. GNU MP is a library for arbitrary precision arithmetic, operating on signed integers, rational numbers, and floating point numbers. Its functions are optimized for speed.

[2]Please see http://www.fs.net

Chapter 8

Conclusions

In conclusion, the prototype client implementation presented in this thesis was tested in the laboratory environment and produced satisfactory results. The implementation met all of the goals set forth in the functional specification (section 4.1) and most of the design goals.

8.1 Usability

The client interface provides the user with a transparent, functional, and user friendly way to access protected documents over the Web. The client is presented with few obstructions to the normal browsing behavior. The client merely has to perform a relatively small amount of certificate management to make sure they have all the necessary certificates available in their certificate cache. The system is relatively intuitive. Much of the functionality presented to the user is easy to use and understand.

The tag signing operations tend to take a long time. The user is left waiting for 20 seconds, on average, to receive the protected document after they enter their private key password into the Java pop-up frame. This problem will be easy to resolve. Using the GNU GMP multi-precision library will give a significant speedup and thereby decrease the wait time for the user.

8.2 Security

In practice, the system seems to operate within security specification. That is, only the user with the correct private key is able to retrieve a protected document.

8.3 Error Handling

In general, client side error handling is not ideal. In cases where the user has Javascript disabled or does not have a Java interpreter installed, the mechanism fails without explanation. Perhaps given more time and research, better solutions for handling these error conditions can be implemented.

Bibliography

[1] T. Berners-Lee, R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, and P. Leach. Hypertext transfer protocol – http/1.1. Internet Request for Comments, RFC 2616, June 1999.

[2] Netscape Communications Corporation. Plug-in guide. `http://devedge.netscape.com/docs/manuals/communicator/plugin/index.htm`, January 1998.

[3] Gilian Elcock. Web-based user interface for a simple distributed security infrastructure (sdsi). Master of engineering thesis, Massachusetts Institute of Technology, EECS, June 1997.

[4] Jean-Emile Elien. Certificate discovery using SPKI/SDSI 2.0 certificates. Master of engineering thesis, Massachusetts Institute of Technology, EECS, May 1998.

[5] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. Simple public key certificate. Internet Draft, `http://world.std.com/~cme/spki.txt`, July 1999.

[6] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. Spki certificate theory. Internet Request for Comments, RFC 2693, September 1999.

[7] Carl Ellison. Spki requirements. Internet Request for Comments, RFC 2692, September 1999.

[8] John Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. Http authentication: Basic and digest authentication. Internet Request for Comments, RFC 2617, June 1999.

[9] Simson Garfinkel and Gene Spafford. *Web Security and Commerce.* O'Reilly and Associates, Inc., Sebastopol, California, 1997.

[10] Digital West Media Inc. Chochise and geronimo: The chiricahua apaches (desertusa). `http://www.desertusa.com/magfeb98/feb_pap/du_apache.html`, January 2000.

[11] Free Software Foundation Inc. Gmp – gnu project – free software foundation (fsf). `http://www.gnu.org/software/gmp/gmp.html`, December 1999.

[12] Free Software Foundation Inc. Gnu libtool – gnu project – free software foundation (fsf). `http://www.gnu.org/software/libtool/manual.html`, January 2000.

[13] Stephen T. Kent. Internet privacy enhanced mail. *Communications of the ACM*, 36(8):48–60, August 1993.

[14] Matthew Fredette.
An implementation of SDSI - The Simple Distributed Security Infrastructure.
`http://theory.lcs.mit.edu/~cis/sdsi.html`, May 1997.

[15] Matthew Fredette. *SDSI Library Documentation.*
`http://theory.lcs.mit.edu/~cis/sdsi/library-documentation.html`,
May 1997.

[16] Terrance Harmon. *SPKI/SDSI: Secure Web Security.* MIT, Advanced Undergraduate Project, November 1998.

[17] Alexander Morcos. A java implementation of simple distributed security infrastructure. Master of engineering thesis, Massachusetts Institute of Technology, EECS, June 1997.

[18] Radia Perlman, Charlie Kaufman, and Mike Speciner. *Network Security: Private Communication in a Public World.* Prentice-Hall, Englewood-Cliffs, NJ, 1995.

[19] Ronald Rivest, Dwaine Clarke, Jean-Emile Elien, Carl Ellison, Matthew Fredette, and Alexander Morcos. Certificate chain discovery in spki/sdsi. `http://theory.lcs.mit.edu/~rivest/ClarkeElElFrMoRi-CertficateChainDiscoveryInSPKISDSI.ps`, November 1999.

[20] Ronald L. Rivest and Butler Lampson. *SDSI - a simple distributed security infrastructure.* `http://theory.lcs.mit.edu/~rivest/sdsi10.html`, *September 1996.*

[21] Aviel D. Rubin, Daniel Geer, and Marcus J. Ranum. Web Security Sourcebook. *John Wiley and Sons, Inc., New York, New York, 1997.*

[22] Bruce Schneier. Applied Cryptography. *John Wiley & Sons, 1996.*

[23] William Stallings. Cryptography and Network Security: Principles and Practice. *Prentice-Hall, Upper Saddle River, NJ, second edition, 1999.*