2007

Secure distributed single sign-on with two-factor authentication

Kaleb Brasee
The University of Toledo

A Thesis

Entitled

Secure Distributed Single Sign-On with Two-Factor Authentication

By

Kaleb Brasee

Submitted as partial fulfillment of the requirements for

The Master of Science in Engineering

_____

Advisor: Dr. Kami Makki

_____

College of Graduate Studies

The University of Toledo

December 2007

The University of Toledo

College of Engineering

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY *Kaleb Brasee*

ENTITLED *Secure Distributed Single Sign-On with Two-Factor Authentication*

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF Master of Science in Engineering

_____

Thesis Advisor: *Dr. Kami Makki*

Recommendation concurred by

Committee

_____

Dr. Gerald Heuring

On

_____

Dr. Hilda Standley

Final Examination

_____

Dean, College of Engineering

An Abstract of

Secure Distributed Single Sign-On with Two-Factor Authentication

Kaleb Brasee

Submitted as partial fulfillment of the requirements for

The Master of Science in Engineering

The University of Toledo

December 2007

In this thesis we present the Secure Distributed Single Sign-On (SeDSSO) architecture. SeDSSO provides secure fault-tolerant authentication using threshold key encryption with a distributed authentication service. The authentication service consists of *n* total authentication servers utilizing a (*t, n*) threshold encryption scheme, where *t* distinct server-signed messages are required to generate a message signed by the service. Existing distributed SSO schemes such as CorSSO and ThresPassport are examined and the benefits of our system over these schemes are presented. SeDSSO establishes secure portable identities by defining a two-factor authentication scheme that uses both a username/password and a unique USB device. The combination of a distributed authentication service and two-factor identities allows SeDSSO to securely authenticate users in any environment.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1. Motivation

As the number of personal Internet-site accounts grows, organizing and remembering confidential identity information becomes more difficult for the individual. It is often impossible to use the same information on every site. Common usernames may already be taken and sites frequently impose unique requirements for passwords (e.g., the password must consist of both lowercase and uppercase letters or it must contain a digit). In an RSA Security survey, more than 30% of users reported needing between 6 to 12 different passwords for their business-related logins and almost 25% said that they needed to remember 13 or more passwords [8]. When people cannot remember all of their information and are forced to physically record it, the secrecy of their identity is jeopardized.

Single sign-on (SSO) allows users to verify their identity on a central system and gain access to many different resources that trust the central system. The act of proving an identity is known as authentication. A widely-used Internet SSO system could help people protect their identity secrets by replacing many site-specific logins with a single

SSO login. This would make it possible for the average user to choose secure identity information and remember it without writing it down. Correspondingly, this system would reduce the need for insecure transmission of logins through email when users forget their information.

Various SSO architectures have been proposed and implemented over the past decade, but none have been used significantly on large-scale public Internet domains. Microsoft Passport is one of the most well-known attempts at widespread SSO. Many web sites initially planned to trust Passport identities that belonged to their users. However, after numerous difficulties and vulnerabilities, Passport support was abandoned by every site except those belonging to Microsoft [26].

The motivation for this thesis is the design of a SSO system that offers improvements over existing SSO schemes. Because many users and sites will rely on the SSO central authentication system, it needs to offer fail-safe authentication that remains available and secure through partial hardware and software failures. A robust system must also provide a way for users to safely sign on from any location, including potentially insecure computers found in places such as Internet cafés and public libraries. Our system is called SeDSSO (Secure Distributed Single Sign-On) and it provides SSO services with a fail-safe distributed authentication system and secure two-factor authentication user identities.

1.2. Thesis Organization

This thesis consists of seven chapters. Chapter 1 introduces our motivation for designing SeDSSO. Chapter 2 presents an overview of the following related work topics:

two-factor authentication, threshold cryptography, existing distributed SSO systems, and distributed certification. Chapter 3 covers the basic components that make up a complete SeDSSO system. Chapter 4 details SeDSSO user identities, including the two-factor authentication scheme and the USB device used for securely transporting identities. Chapter 5 fully describes the processes executed by SeDSSO components. Chapter 6 discusses our SeDSSO prototype and the results of performed tests. Chapter 7 presents our conclusions and suggested future work.

Chapter 2

Related Work

This chapter begins with an overview of two-factor authentication. Next, a summary of threshold cryptography is presented. The chapter then discusses CorSSO and ThresPassport, two existing distributed SSO schemes. A distributed SSO system provides single sign-on to users and allocates the responsibility of authenticating these users to a network of individual authentication servers.

Finally, the topic of certification is presented. A certificate binds information about an entity to that entity's public key and includes the signature of a trusted authority to vouch for the authenticity of the information contained on the certificate. An existing distributed certification authority scheme known as COCA (Cornell Online Certification Authority) is examined.

2.1. Two-Factor Authentication

2.1.1. Overview

The username and password system was introduced in the early 1960s as the need emerged to secure identities on timesharing systems [21]. Computing has changed

dramatically since that time, expanding from government and research to business and personal use. However, username/password pairs have remained the standard proof of identity ownership. This method is now the weakest link in modern computer security. The Carnegie Mellon Computer Emergency Response Team (CERT) reports that 80% of all security breaches it examines are related to passwords [1]. Identities can be stolen through technological means such as keystroke logging and phishing schemes. They can also be stolen through social engineering methods ranging from the complex (posing as an administrative authority and coercing the user) to the simple (viewing a handwritten username/password lying on a desk).

In light of these weaknesses, systems have been developed which require additional identity proof. Identity proof mechanisms are divided into general categories known as the *identity factors*. A two-factor authentication system requires that valid credentials from two different factors be presented before a user is trusted. Many different methods can be used to prove an identity, but most fall into one of the following factors:

1. "Something you know" – memorized information (e.g., a password or answer to a secret question).
2. "Something you have" – possession of a unique item containing secret information (e.g., a smart card, bar code, or USB-interface device).
3. "Something you are" – a physical trait that can be converted to digital information using specialized hardware (e.g., a retina or fingerprint scan or voice recording analysis).

On the Internet the username/password is a generally-assumed first factor belonging to the "something you know" category. Two-factor authentication system designers must choose a second factor and decide how to implement it. The second factor is often a physical device that stores a key, generates passwords, or responds to challenges from the authentication server.

2.1.2. Two-Factor Authentication Example

An example of an existing two-factor authentication system is RSA's SecurID. SecurID identifies users with a two-factor authentication system consisting of a personal identification number (PIN) and numeric password that users know, and a device that users have [22]. This device (known as the token) features a processor and memory with a small numeric display, and it is configurable for individual users. It generates a six-digit code every minute and constantly displays the code. In order to login, a user must enter both their PIN and a concatenation of the numeric password with the current token. Authentication is successful if the PIN exists, the numeric password for that PIN is correct, and the six-digit token code matches the code expected by the server. Since the SecurID token codes are time-dependent, the server and the token must be initially synchronized and maintain the same time values in order for the codes to match.

SecurID is widely used and it significantly complicates identity theft. Authentication is not possible without both knowledge of the PIN/password and possession of the token. However, under the right circumstances it is possible to intercept communications within this system (as well as other systems using time or usage-dependent information such as one-time passwords) and perform a man-in-the-

middle attack to hijack the user's authentication request [23]. Methods to recover the secret token key have also been discussed [24]. Still, the system is far more secure than one-factor authentication, with no successful attacks reported in SecurID's 15-year lifecycle [22].

2.1.3. Advantages and Disadvantages

The most obvious advantage of two-factor authentication is the increased difficulty for a malicious party to acquire both authentication factors. Standalone keystroke-logging attacks are usually insufficient because the captured data is not enough to gain authentication, will not work for subsequent logins, or will only work for a very short time. Additionally, if a malicious user just obtains a token-generating device it is useless because the login information is not known. The difficulty of obtaining both factors is why two-factor authentication is often referred to as *strong authentication*.

Even though two-factor authentication makes electronic identity theft more difficult, it is not perfect. In systems using time-based passwords there is a small window of opportunity in which a real-time attack can occur [20]. In SecurID the window of opportunity is at most 60 seconds but an attack could theoretically take place in this time frame (although as RSA stated, such an attack has yet to be reported). A challenge and response two-factor system eliminates this threat because each new session requires a response to a different random challenge.

On a more basic level, the argument has been made that two-factor authentication is inadequate to protect users against identity theft and phishing and that it "doesn't solve anything" [25]. Man-in-the-middle attacks allow a web site to pose as the service

provider's site to the user, while actually passing communications back and forth between the user and service provider. Once the man-in-the-middle system has captured the necessary information from this real connection, it can perform any action as the user with that service provider.

Trojan attacks work by installing inconspicuous software directly on the computer that the user is operating. Once this software detects that a secure connection has been established, the Trojan software uses this connection to perform its own malicious activities in the background.

Many previous two-factor authentication schemes have been vulnerable to one or both of these attacks. Section 6.7 of this thesis discusses how our system operates in regards to these risks.

2.2. Threshold Cryptography

Shamir and Blakley independently proposed the threshold scheme in 1979 [4, 9]. As the title of Shamir's paper ("How to Share a Secret") indicates, a threshold scheme is used to safely share a secret between distinct parties so that no individual party possesses the secret. A threshold scheme divides the secret data into n data pieces and performs the division so that t data pieces, $t \leq n$, are required to recreate the secret data. Each data piece is unrelated to all of the other pieces and acquiring less than t provides no information about the original data. Such a scheme is known as a (t, n) threshold scheme.

In cryptography, threshold schemes can be used to divide a private key into a number of partial keys. Partial keys can be used to encrypt and decrypt a message like a full key. When a message is encrypted with t different partial keys, the resulting t

messages can be combined into one encrypted message that is identical to the message encrypted with the private key. This act of combining partially signed messages can be done without knowledge of any of the keys. If a private key is split using a (*t, n*) threshold scheme then *n* servers will possess a partial key and it will take the signature of *t* servers to create a message signed with the private key. Therefore, an attacker will need to make *t* successful intrusions on different authentication servers to gain control of the authentication service private key.

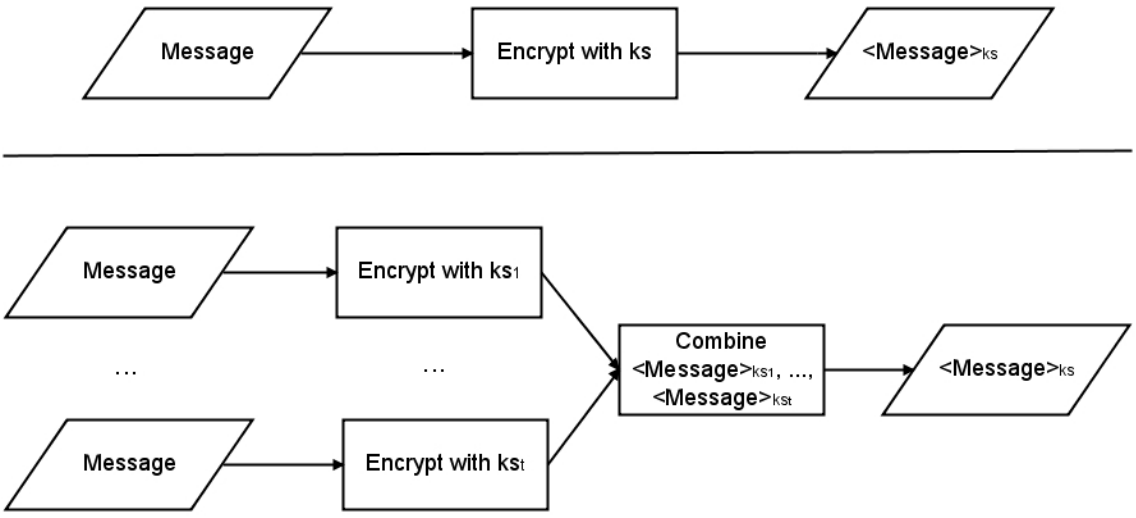

Figure 2.1: Encrypting a message with *t* partial keys and combining the partially encrypted messages produces the same output as a simple encryption with k_s. However, with threshold encryption no party is required to possess the entire private key.

When choosing the numbers *t* and *n*, *n* is simply the total number of servers available. This number can be changed without affecting any of the partial keys or the original key, so long as *n* remains greater than *t*. The number *t* cannot be modified without changing either the partial keys or the original key. Shamir suggested the formula $n = 2t - 1$ as a robust way for determining the total number of partial keys and

the number required to perform threshold operations [4]. When this formula is applied to a group of authentication servers, authentication is still possible even if $\lfloor n / 2 \rfloor$ (or $t - 1$) servers are inaccessible. Similarly, an attacker can steal up to $\lfloor n / 2 \rfloor$ (or $t - 1$) partial keys without learning the group's private key.

Modifications to the original Shamir threshold scheme were proposed in [10]. These modifications fix a vulnerability that allows a malicious user to cheat other parties in the system and acquire the partial keys necessary to reconstruct a full key. More current schemes for threshold signatures using the RSA encryption algorithm [16] have been proposed; a popular design is Shoup's scheme [15]. Additionally, numerous papers have been written that discuss the application of threshold cryptography in distributed system operations [13, 14].

2.3. CorSSO

Two distributed threshold SSO systems have recently been proposed. The first system to be created was CorSSO (Cornell Single Sign-on), a SSO system that provides distributed peer-to-peer network authentication [2]. This design moves authentication services that are commonly provided by application servers (or service providers) onto a set of dedicated authentication servers. A threshold scheme is used to split an authentication system's private key into a set of partial keys, so that user authentication requires the work of several authentication servers instead of one. In addition to allowing users to create one identity and use it on all of the application servers, this system improves scalability, distributes trust, and provides fault tolerance in the authentication process.

2.3.1. CorSSO Identity Setup Protocols

CorSSO defines a client/user as a principal C that creates both a public key K_C and private key k_C for itself. Likewise, an application server S creates its private key k_S and public key K_S. S becomes accessible to principals by registering its information with a set N_i of authentication servers (known as a sub-policy set). The authentication servers in N_i create a private key k_i and public key K_i for this particular set of authentication servers. K_i is sent to S for decrypting authentication system messages in the client authentication process. k_i is split using threshold encryption and a unique partial key is given to each authentication server. No authentication server stores the full k_i.

2.3.2. CorSSO Client Authentication Protocol

To access an application server, a client must first successfully authenticate with t authentication servers in the application server's namespace set. The client C requests an authentication policy (a set of chosen authentication servers) from application server S and S responds by sending back a policy set P with which C must authenticate. C selects a sub-policy set N_i with which it has registered, containing only elements that are also in the set P. C requests a certificate vouching for its identity from each authentication server. If C's identity verification is successful then each authentication server creates the same certificate and signs it with a different partial key of k_i. The authentication servers send these partially signed certificates back to C. When C has received t partial certificates from the authentication servers in N_i, it uses threshold cryptography to combine them into a single certificate signed with the private key k_i.

2.3.3. CorSSO Client-to-Application Server Access Protocol

When *C* has generated the certificate signed with k_i it contacts *S* again and requests an authentication challenge. This challenge is a pseudo-random generated message that *S* encrypts with its private key k_S and sends to *C*. *C* uses K_S to decrypt the message, encrypts the same message with its private key k_C and sends both the encrypted message and the certificate signed with k_i back to *S*. *S* grants *C* access to its services only if it can verify that the challenge message was signed with *C*'s private key and that the authentication servers have vouched for *C*'s identity.

2.3.4. CorSSO Disadvantages

CorSSO lacks a mechanism for transferring a user's private key so that the user can gain authentication on different computers. Copying this key without protection would allow anyone who steals the key to steal the identity of the user. CorSSO's use of the private key to identify users is similar to identification in the Kerberos authentication system which has been noted for its mobility limitations and lack of security in untrusted environments [17].

2.4. ThresPassport

ThresPassport is a distributed SSO system that uses threshold-based key sharing to split a service provider's secret key into a set of partial keys [3]. It was developed to address some shortcomings of the existing CorSSO system. In order for a service provider to trust a user's identity, a set of authentication servers must be able to construct a voucher message for the user that is signed with the secret key. In contrast to CorSSO,

ThresPassport does not rely on a trusted authority to operate a public key infrastructure (PKI) for its service providers, clients, and authentication servers. ThresPassport also replaces CorSSO's randomly generated private and public client keys with one-way hash[1] keys that can be generated using only a username and password.

2.4.1. ThresPassport Identity Setup Protocols

The ThresPassport protocol begins by establishing the identity of service providers and users with a set of *n* authentication servers. A service provider *S* acquires a unique identifier number *SID*. It then creates a secret key K_S and calculates the inverse key K_S^{-1} such that $K_S^{-1} = (1 \bmod (p - 1)) / K_S$, where *p* is a randomly generated prime number. A (*t*, *n*) scheme is used to split K_S into *n* partial keys, where signatures from *t* of these partial keys are required to act as the entire key K_S. *S* then sends its unique identifier *SID* along with partial keys K^1_S through K^n_S to authentication servers A_1 to A_n respectively, with each server receiving a different partial key. Each authentication server stores the partial key and *SID* and sends a success message to *S*.

Users are identified by a unique *UID* created by hashing their username, and a password is associated with the *UID*. For each authentication server, the username, password and authentication server identifier A_i are combined into strings and a one-way hash is executed on this combination to create a key (denoted K^i_U). This process is performed for each server to create keys K^1_U through K^n_U. *U* sends the *UID* and correct

[1] A hash function, also known as a one-way hash, creates a reproducible signature or fingerprint of some input data. The function operates in such a way that it is very unlikely to generate the same signature output from different input data. It is trivial to calculate a hash, but practically impossible to calculate the original data from the hash (hence the term one-way).

K^i_U to each authentication server. Upon successful storage of these values, the servers return a success response to *U*.

2.4.2. ThresPassport User Authentication Session Protocol

User authentication with a single authentication server is a straightforward process. The client software uses the entered username/password and the authentication server identifier to generate *UID* and K^i_U. User *U* then requests authentication from authentication server *A*. *A* generates a nonce n_A and sends it to *U*. *U* generates its own nonce n_U as well as a random number r_U and encrypts the message $< r_U, n_U, n_A >$ with the key K^i_U. *U* sends both the *UID* and this encrypted message to *A*. If *A* can decrypt this message correctly using the key stored for *UID* and can verify that *U* received and decrypted the nonce n_A, then *A* generates its own random number r_A and sends the message $< r_A, n_A, n_U >$ encrypted with K^i_U to *U*. Now that both *A* and *U* have the numbers r_A and r_U, they each create a temporary session key $SK_{U,A}$ by hashing a combination of r_A and r_U. This session key is used until the session is ended manually or expires.

This process is not executed in isolation, but occurs between the user and each authentication server as a part of the single sign-on protocol described in the next section.

2.4.3. ThresPassport Single Sign-On Protocol

When a user attempts to access a service provider, the following protocol is used to verify the user's identity and grant or deny access. The user *U* begins by requesting access to a service provider *S*. *S* responds with its *SID*, a nonce n_S, and possibly a list of

authentication servers if *U* does not already possess such a list. *U* chooses *t* authentication servers from the optional list or from a previously-used list and establishes session key connections with each as described in the previous section. *U* then sends the *SID*, *UID*, and nonce n_S to each authentication server. Each server constructs the same message < *UID*, *U*, n_S > and signs it with the partial key received when *S* registered on the authentication network. The result is *t* distinct messages that contain the same information but are signed with *t* different partial keys. The authentication servers send the messages back to *U*, and *U* uses threshold cryptography to combine them to create the message < *UID*, *U*, n_S > signed with K_S. *U* sends this message along with its *UID* to *S*. If the message encrypted with *S*'s public key contains the original nonce and correct *UID* then the user is granted access to the service.

2.4.4. ThresPassport Disadvantages

ThresPassport does not require a public key interface (PKI), and in [3] the authors claim that this is an advantage over systems that rely on a PKI. PKI algorithms require more computational power, and distributing a public and private key to each entity in the system increases the account management overhead. However, it is still arguable that the positives of a PKI outweigh these negatives. In ThresPassport, there is no way to verify that a contacted authentication server is genuine. All that is known about an authentication server is its IP address and *AID*, as the servers do not use cryptographic keys of any kind. Without a private key to verify the authentication server, it would be possible to execute an interception attack or DNS lookup table modification and allow

another system to pose as an authentication server without needing to possess any credentials.

ThresPassport's username/password identity allows users to login with any computer. However, a ThresPassport identity is no safer than any other keystroke-based identity. Capturing a user's username and password is simple, as the software to perform this capture could easily be installed on a public machine by an identity thief or on a home machine by a virus. If a ThresPassport user's login information is captured, the malicious entity gains control of the user's account.

The ramifications of identity theft in SSO are far worse than theft in today's one-login-per-site system. Instead of gaining access to one area of a user's identity the thief gains complete access, from the trivial (websites and forums) to the critical (bank accounts and credit cards). SSO needs a security framework that allows it to be easily used in multiple locations but also protects identities with something stronger than a username and password.

2.5. Certification and COCA

Certification is a method of providing trust in a PKI system. Without certification, an entity's key is vouched for by that entity only. When an uncertified system claims to belong to a certain individual or company, there is no guarantee that this is true. Certification uses certificate authorities (or CAs), trusted third parties that everyone in the system can rely upon, to securely and correctly vouch for the identity of the entities. The CA generates a certificate during the account creation process that binds personal information (name, address, phone number, and other identifying

characteristics) with the entity's public key, and signs a portion of the certificate with its own private key. To verify that a certificate is genuine, the CA's public key can be used to decrypt the certificate's signature and see that the CA signed the certificate with its private key.

The trust in certification is usually built in chains, with the set of working CAs all receiving their individual certificate from a highly-secure root CA. The root CA is never connected to any network, it is constantly protected in a restricted-access setting, and very few people can access the system. For these reasons, if a working CA server possesses a voucher certificate signed by the root CA its authenticity can be trusted more than the identity of an authentication server in a non-PKI environment.

As with authentication systems, a centralized CA can also act as a central point of failure. To solve this problem, the distributed certification system known as COCA (Cornell Online Certification Authority) has been proposed [5]. COCA uses threshold cryptography for distributed certificate operations and a Byzantine quorum system for fault-tolerance [11]. The threshold scheme employed by COCA is a $(t + 1, n)$ scheme where $n \geq 3t + 1$. With these constraints, COCA will maintain correct operations with up to t compromised certification servers. The threshold keys are periodically updated with a "proactive secret-sharing protocol". In order to control the system, a malicious party must steal $t + 1$ partial keys in a relatively short amount of time. Otherwise, the keys will expire and the attack will fail.

Each COCA certification server possesses a partial key of the entire system's private authentication key. A message must be signed by $t + 1$ partial keys to create a

threshold-encrypted message signed by the private key. Additionally, each server possesses an individual public and private key for communication within the certification network. These individual intranet keys can be changed frequently without the need to propagate this change to users and service providers. This operation adds security within the certification service but is also simple and efficient to perform.

Unlike the threshold protocols in CorSSO and ThresPassport where the user receives all of the partially encrypted messages and combines them, COCA users only need to contact one of the certificate servers. The contacted server forwards the user's request to $t + 1$ other certificate servers. When enough partial messages have been returned the contacted server combines them into one message signed with the whole system's private key. This approach makes it possible for COCA users to access the system without needing to possess individual server public keys, and prevents against a possible attack where a user could be sent many false partial messages and would have to determine which ones were real.

Chapter 3

SeDSSO Components

This chapter provides a detailed description of the individual components that make up the entire SeDSSO architecture. SeDSSO consists of three different components: service providers, users, and authentication servers. These components are shown in figure 3.1. Service providers are Internet sites that offer a service to users, such as email, forums, shopping, banking, etc.[2] Users are individuals who access service providers to perform desired tasks. Each user possesses an account that allows service providers and authentication servers to identify them. Authentication servers store information about all users and service providers that have registered with the SeDSSO system. Multiple authentication servers form the authentication service which is responsible for authenticating SeDSSO users.

[2] SeDSSO service providers are not to be confused with Internet service providers (ISPs), which are transparent to SeDSSO.

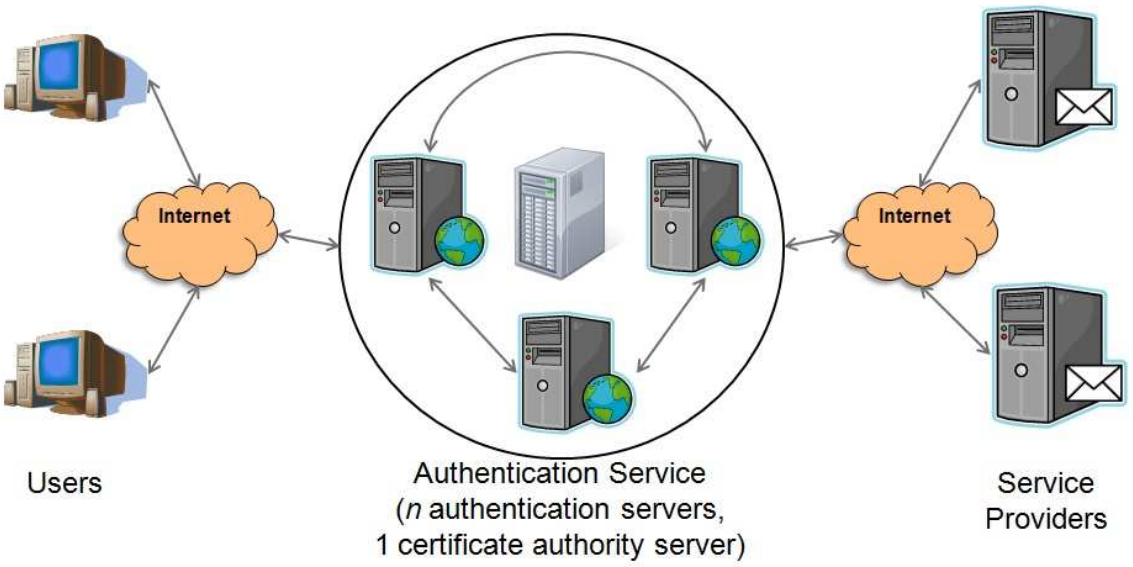

Figure 3.1: Users, Service Providers, and the Authentication Service are the three basic components of the SeDSSO system.

3.1. Authentication Service

Authentication servers are individual systems that work together to vouch for the identity of users. Because they authenticate users and store information about each SeDSSO user and service provider, these servers must be high-performance high-availability systems that can perform many intensive data storage, computation, and network I/O tasks simultaneously. Collectively, the group of authentication servers is referred to as the authentication service.

SeDSSO implements threshold encryption by deploying *n* authentication servers and generating one public and private key for the entire authentication service. This key generation takes place on the certificate authority (CA) server. The authentication service key generation is its only task, it is never connected to a network, and it is physically guarded. These steps are required to ensure the security of the authentication service's public and private keys and thereby maximize trust in the service. The CA

server splits the private key into *n* partial keys with a (*t*, *n*) threshold scheme, and one partial key is given to each authentication server. Since no authentication server possesses the entire private key, at least *t* servers must sign an identical message in order to act as the authentication service. Every user and service provider in the SeDSSO system is given access to the authentication service public key, making it possible to verify messages signed by the authentication service private key.

Individual authentication servers each possess a self-generated public and private key to use for server-to-server communications, similar to the intranet keys found in COCA. It is only necessary that authentication servers know these keys, and they do not need to be distributed to users and service providers. The presence of these keys facilitates secure communication within the authentication service.

3.2. Service Providers

Service providers offer some type of service to users through the provider's web site. The service provider can be a business web site or a personally-owned site and can offer any combination of free or payment-based services. The only requirement is that the service provider has the need to identify individual users. Joining SeDSSO allows this provider to offer personalized services to users without having to invest in standalone authentication software and hardware, because the authentication service performs this function for all service providers.

When a service provider account is created, it is given a unique service provider ID generated by the authentication service. The service provider creates its own public and private key pair and sends the public key to the authentication service. Each

authentication server associates the public key with the service provider ID. Once the service provider has been added, it can start accepting logins from SeDSSO users as described by the sign-on protocol.

Although the authentication service centralizes authentication for the entire SeDSSO system, its functionality does not extend into specific service provider requirements. Service providers must store all site-specific user data on their own servers, and can do this in any way they choose. As long as the stored user data is related to SeDSSO user identifiers then the service provider will be able to recall the data for that user as soon as the sign-on procedure is completed.

3.3. Users

The user account is an individual's representation on the SeDSSO service. A user's identity is represented by a username and password as well as a public and private key. The user creates all of these values, but the username must be verified by the authentication service to ensure that it has not been previously chosen. The username (and the corresponding username hash) is the information by which service providers, authentication servers, and other users identify an individual. It is possible for a person to separate their identity by possessing multiple accounts, although the need to remember too many usernames and passwords negates one of the major benefits that a SSO identity provides.

An email address may be entered at the time of user account creation. This address can be supplied by any email provider, even if they are not part of the SeDSSO system. When the creation process is complete, an email containing the new user's

username and password is sent to this address. Entering an email address when creating an account is not required, but it can be done to provide an additional way to recall the account password.

A user can sign on to any service provider with his or her existing SeDSSO user identity. Upon a user's first login, the service provider adds a new record to its own user database. Without sending any additional data to the service provider, a user should be able to perform tasks that do not require personal verification (such as browsing a store's items or posting comments on a forum). In situations where a SeDSSO identity must be tied to a real-life identity (such as money management and store purchases) the user will need to provide additional information to the service provider. This information will be associated with the user's account on the service provider system.

Chapter 4

SeDSSO User Identity System

SeDSSO represents users with a two-factor identity consisting of their username/password as well as information stored on a specialized USB device. The username and password is the factor that they know and the information on the USB device is the factor that they have. Possession of both factors is required for a user to successfully authenticate with the SeDSSO system. The advantage of this system is that a coordinated effort is required to steal a user's identity, and classic one-factor attacks are insufficient. Keystroke logging software cannot access the USB device information, and the theft and examination of the USB device does not reveal the corresponding username and password.

4.1 USB Identity Device (USBID)

The SeDSSO USB identity device (USBID) is a specialized device that combines a built-in processor with flash memory and communicates with a computer through the USB interface. All of the hardware is housed in a casing the size of a normal USB flash drive. The USBID is responsible for storing the public and private keys for one or more

users, as well as the secret counter values that allow users to gain authorization with service providers. This device must be accessible by the client software every time SeDSSO account creation or authentication is requested. A similar USB-interface computation device with specialized hardware was proposed in [12], but was designed for electronic payment instead of SSO identity proof.

The USBID architecture is shown in figure 4.1. The processor is powered by the USB port connection. The USBID processor generates the user's private and public key when the account is created and is responsible for performing all operations that require the use of identity factors, such as signing a message with the private key. This makes it unnecessary to pass the user's private key to the computer where it could be observed by a program designed to retrieve this information. The public key is passed to the user's system and sent to the authentication service for storage, but the private key remains exclusively in the USBID and is encrypted with the user's password.

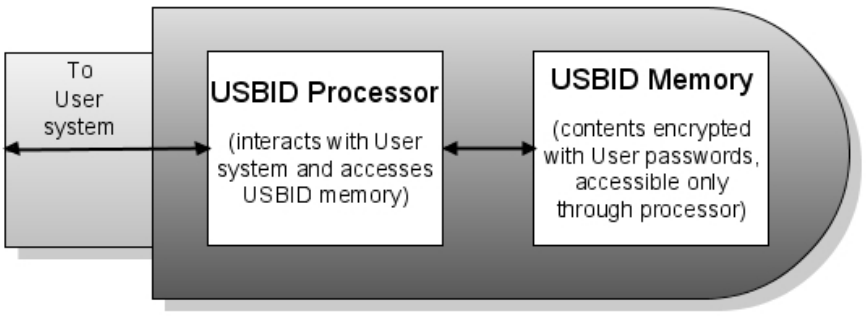

Figure 4.1: The USBID consists of both a processor and memory. The memory cannot be accessed directly by the user.

When a communication message needs to be signed during the authentication process, the client software passes the message to the USBID processor. The processor

retrieves the encrypted private key from memory and decrypts it with the password. Once the private key has been decrypted it is used to sign the message, and the signed message is returned to the client software on the user's system.

The USBID memory is standard flash memory. However, unlike common flash drives, the USBID does not allow access to the memory through a computer file system. Only the USBID processor can access this memory. The client software translates actions in the SeDSSO client software user interface into low-level device driver commands, and these commands indicate to the USBID processor what information must be retrieved during processing. The processor acts as a black box, providing the necessary output but keeping memory retrieval, storage and modifications transparent to the user system.

4.2 Counter System

The counter system is part of the "have" factor in SeDSSO's two-factor authentication scheme. To make authentication impossible without the USBID, a pseudo-random number generator seed is created and stored on the service provider's system and the user's USBID. The USBID uses the seed to generate a number during the authentication process and this number is sent to the service provider. If the service provider generates the same number then the user's possession of the seed (and therefore possession of the USBID) has been proven.

Although the authentication service implements two-factor authentication on its own by requiring the user's private key from the USBID, the counter system provides an effective additional layer of security. Even if *t* authentication servers are hacked so that a

malicious party can gain authentication as a user without the USBID, the service provider requires the counter value on the USBID independently. Possession of the correct counter value is still required to gain access to any service provider. An attacker who gains control of a user's identity without possessing the USBID cannot access any service providers that the user has contacted in the past, because once the first authentication has been performed then a working counter is established.

4.3. Counter Value Operation

When a user attempts an authenticated connection to a service provider for the first time, the counter value between these two parties does not yet exist. In this case, the voucher for the user's identity generated by the authentication service is sufficient for authentication. The service provider creates a seed that will be used for generating the counter, and its successful authentication response to the user includes this seed. In the future, the service provider will require that the next counter value be sent by the user in order to gain authentication.

Three variables describe the state of the counter: *seed*, *depth*, and *maxDepth*. *Seed* is the number originally generated by the service provider and is used to seed the pseudo-random number generator responsible for creating the counter value that is sent. *Depth* is the number of times that the seeded generator is executed to produce the next counter. When a new *seed* is generated *depth* starts at 1, and each time a connection is successful the user and service provider increment *depth* by 1. *MaxDepth* is the maximum value that *depth* can attain. This number changes whenever a new *seed* is created, and is set to the last 2 digits of the newly created *seed* + 1. If *maxDepth* did not

limit the number of times that *seed* is used then the authentication process would become prohibitively processor intensive as the user's number of authentications increased.

Once *depth* has reached *maxDepth* both the user and service provider are required to independently calculate new *seed*, *depth*, and *maxDepth* values. The final counter value (produced by iterating *maxDepth* times on a *seed*-seeded pseudo-random number generator) is used as the new *seed*. *Depth* is reset to a value of 1, and *maxDepth* is set to the final 2 digits of the new *seed* + 1. Both parties use this process to create the next counter value and expect the other party to do the same.

Chapter 5

SeDSSO Processes

This chapter describes the operation and communication processes necessary for SeDSSO to function. The first section covers the setup processes which are responsible for initializing secure connections as well as adding new components. The authentication processes are presented next and deal with both user-to-authentication service communication as well as user-to-service provider sign-on. Finally, the processes for managing existing SeDSSO identities are discussed.

5.1. Setup Processes

The first process in this section describes the steps necessary to generate a session key and set up a secure symmetric-encryption connection. This session key is generated at the beginning of every communication process between two existing SeDSSO parties. Additionally, this section details the processes for adding new authentication servers, service providers, and users.

5.1.1. Session Key Generation

Because public/private key pairs place strict length limitations on the encrypted payload and require far more CPU effort than symmetric keys, they are only used at the beginning of a session. Once it has been verified that both communicating parties know the private key corresponding to their claimed identity, a symmetric session key is created and used for the remainder of the communication. In the following protocol, *C* is the connecting system and *R* is the receiving system.

Note that in step 1, the connecting system can send its public key as an optional parameter in situations where the receiving system does not yet have this key stored. This is necessary in some situations such as user account creation where the user account does not exist.

1. $C \rightarrow R$: $< nonce_C, [K_C] >_{KR}$
2. $R \rightarrow C$: $< nonce_C$ XOR 00…0001, $nonce_R, SK >_{KC}$
3. *C* verifies that the first parameter in the above message is its generated nonce with the last bit flipped. If so, *SK* is stored as the symmetric key for this session.
4. $C \rightarrow R$: $< nonce_R$ XOR 00…0001 $>_{SK}$
5. *R* verifies that the parameter in the above message is its generated nonce with the last bit flipped.
6. $R \rightarrow C$: < "success" $>_{SK}$

5.1.2. Adding an Authentication Server

Because this process occurs very infrequently, must be highly secure and consists of extra-network steps, it is not implemented using a communication protocol. The SeDSSO simulation is provided all of the authentication server information before execution. In a real SSO system the security risk of adding a new authentication server is high enough to warrant an addition consisting of exclusively extra-network communication. When the new authentication server identity is established, it is necessary to synchronize the server's data with the data stored by the other authentication servers.

The authentication server parameters are as follows: *AID* is the authentication server ID, K_A is the authentication server's individual public key, k_A is the authentication server's individual private key, IP_A is the authentication server's receiving IP address, and P_A is the authentication server's receiving port. In addition, the authentication service has a single public key K_{AS} and a corresponding private key k_{AS}. No server has possession of the entire service private key, but each possesses a distinct partial private key k_{pAS}. When t distinct k_{pAS} keys are used to create t encryptions of the same message, the encryptions can be combined to form one message encrypted with the private key k_{AS}.

5.1.3. Adding a Service Provider

The service provider uses extra-network communication to add itself to a single authentication server A_c. A_c then uses the following protocol to add the service provider to every other authentication server.

The service provider data is referred to as follows: *SID* is the service provider ID, K_S is the service provider's public key, IP_S is the service provider's receiving IP address, and P_S is the service provider's receiving port.

1. A_c establishes a secure session key connection with each authentication servers $A_1 \ldots A_n$.
2. $A_c \rightarrow A_1 \ldots A_n$: "ADD_SP", 1, < *SID*, K_S, IP_S, P_S > $_{SK}$
3. $A_1 \ldots A_n$: add this service provider to the service provider database
4. $A_1 \ldots A_n \rightarrow A_c$: "ADD_SP", 2, < "success" or "failure" > $_{SK}$

5.1.4. Adding a User

User data collection and generation takes place in the initialization functions when the user software is executed. This inputs and generates all data necessary to begin the user addition process.

The user data is referred to as follows: *UID* is the unique user ID, *UP* is a hash of the username and password combined, K_U/k_U is the user's public/private key combination, and *INV* is the account invalidation code.

The addition process begins after data collection has taken place on the user's computer. The user enters the username and password. *UID* is calculated by hashing the username and *UP* is calculated by hashing the username and password combination. A secure pseudorandom number and computing environment data is used to seed the generator for K_U, k_U and *INV*.

1. User *U* establishes a secure session key connection with a random available authentication server *A*. *U* sends its newly-created public key as the optional argument.
2. $U \rightarrow A$: "CREATE_USER", 1, < *UID*, *UP*, K_U, hash(*INV*) > $_{SK}$
3. *A* verifies that the *UID* is not already claimed by another user account. If so, *A* returns a failure and ends this process. If not, *A* continues.
4. *A* establishes a secure session key connection with all other authentication servers $A_1 \ldots A_n$. Each connection uses an independent *SK*.
5. $A \rightarrow A_1 \ldots A_n$: "ADD_USER", 1, < *UID*, *UP*, K_U, hash(*INV*) > $_{SK}$
6. $A_1 \ldots A_n$ decrypt and analyze the message and return one of the following messages to *A*.
    a. If the message cannot be decrypted or data is in an improper format, send: "ADD_USER", 2, < "general_failure" > $_{SK}$.
    b. If the *UID* has already been taken, send: "ADD_USER", 2, < "uid_failure" > $_{SK}$.
    c. If the data passes validation, save the user data to a temporary variable (without yet adding the user) and send: "ADD_USER", 2, < "success" > $_{SK}$ to *A*.
7. *A* receives messages from $A_1 \ldots A_n$ and tallies their responses.
    a. If *A* received *t* or more "success" messages and no "uid_failure" messages, add the user and send: "ADD_USER", 3, < "add" > $_{SK}$ to $A_1 \ldots A_n$.
    b. If *A* received less than *t* "success" messages or 1 or more "uid_failure" messages, send: "ADD_USER", 3, < "discard" > $_{SK}$.

8. $A_1 \ldots A_n$ receive the "ADD_USER", 3 message from *A* and either add the user or discard the user's information without adding.
9. *A* sends a message to *U* describing the results of the user creation process.
    a. If *A* received *t* or more "success" messages and no "uid_failure" messages, send: "CREATE_USER", 2, < "success", $UID >_{SK}$.
    b. If one or more "uid_failure" messages are received the user account is not created. Send: "CREATE_USER", 2, < "uid_failure", $UID >_{SK}$.
    c. If *A* received less than *t* "success" messages after a specified time limit, send: "CREATE_USER", 2, < "general_failure", $UID >_{SK}$.
10. *U* receives the message from *A* and reports the status to the user accordingly.
    a. If *U* received "success", report that the user account has been successfully created and is ready for use. The client software stores *UID*, K_U and k_U on the USBID for use in future logins. The invalidation code *INV* is stored on the hard drive, not the USBID, for reasons that are discussed in invalidation protocol section 5.3.1.
    b. If *U* received "uid_failure", report that the desired username is not available and the user should choose a new name.
    c. If *U* received "general_failure", report that the authentication system is not available at this time and the user should try again later.

5.2. Authentication Processes

The processes for user authentication are defined in this section. Authentication requires the user to communicate with the authentication service to obtain an identity

voucher. This voucher must contain a fresh nonce that the service provider sent to the user and must be signed with the authentication service private key. Every authentication process between a user and service provider relies on this voucher.

5.2.1. User Authentication Voucher Generation

Before a user can access a service provider, that user must receive a message signed by the authentication system that vouches for their identity. This message contains the user ID and service provider ID, the username/password hash, and a nonce created by the service provider to eliminate the possibility of replay attacks.

1. User *U* establishes a secure session key connection with a random available authentication server *A*.
2. $U \rightarrow A$: "AUTHENTICATE_USER", 1, $< UID, UP, nonce >_{SK}$.
3. *A* randomly selects a set *AuthSet* of *t*-1 authentications servers which it intends to contact. *A* adds both itself and these servers to a set *ContactedSet*.
4. *A* creates 2 response sets, one to collect the successful authentication responses and the other to collect the failed authentication responses.
5. *A* establishes a secure session key connection with each authentication server in the *AuthSet*. Each connection uses an independent *SK*.
6. $A \rightarrow \forall A_x \in AuthSet$: "AUTHENTICATION_CHECK", 1, $< UID, UP, nonce >_{SK}$.
7. *A* examines the information it received from *U*.

a. If the *UID* exists and the *UP* corresponds to this *UID*, add *A*'s response to the success set.

b. If the *UID* does not exist or the *UP* does not correspond to this *UID*, add *A*'s response to the failure set.

8. The servers in *AuthSet* decrypt and analyze the message and return one of the following messages to *A*.

   a. If there is an error decrypting the message, the *UID* is not found, or the *UP* for this *UID* is incorrect, send a failure message: "AUTHENTICATION_CHECK", 2, < "failure" > $_{SK}$.

   b. If the *UID* exists and the *UP* corresponds to this ID, send a success message: "AUTHENTICATION_CHECK", 2, < < *UID, nonce* > $_{kpAS}$ > $_{SK}$.

9. *A* receives all responses from the *AuthSet* servers and adds each to the appropriate response set.

10. If any responses are present in the failure set:

    a. *A* randomly selects an authentication server which is not present in *ContactedSet*. It adds this random server to *ContactedSet*.

    b. *A* sends the message from step 6 to the random server, examines the received response, and adds the response to the success or failure set accordingly.

    c. Step 10 is repeated until *t* successes have been counted, time runs out, or there are no more authentication servers to contact.

11. When *A* has received *t* successful responses, *n* total responses, or has timed out, it performs one of the two actions:

a. If *t* or more authentication servers responded with a successful authentication message, *A* threshold combines *t* of the partial authentication messages into one message and sends the following to *U*: "AUTHENTICATE_USER", 2, < "success", < *UID*, *nonce* $>_{kAS}>_{SK}$.

b. If less than *t* authentication servers responded with a successful authentication message, *A* sends the following to *U*: "AUTHENTICATE_USER", 2, < "failure" $>_{SK}$.

5.2.2. Initial User Sign-on to a Service Provider

The sign-on procedure describes the steps necessary for a SeDSSO user with an existing account to gain access to a service provider. The following process describes a user's first access to a service provider.

1. User *U* wants to access a service provider *S* for the first time. The client software provides an interface for *U* to contact *S*, enter the account username and password, and begin the authentication process.
2. *U* establishes a secure session key connection with *S*.
3. *U* → *S*: "USER_SIGN_ON", 1, < *UID* $>_{SK}$.
4. *S* → *U*: "USER_SIGN_ON", 2, < $nonce_S >_{SK}$.
5. *U* performs the authentication message request procedure (from section 5.2.1) using *UID* and $nonce_S$.
    a. If the authentication is successful, *U* receives the message < "success", < *UID*, $nonce_S >_{kAS}>_{SK}$ and continues the sign-on procedure.

b. If the authentication is not successful, *U* receives the message < "failure" > $_{SK}$, aborts the sign-on procedure and instructs *S* to do the same by sending "USER_SIGN_ON", 3, < "failure" > $_{SK}$.

6. *U* → *S*: "USER_SIGN_ON", 3, < "success", < *UID*, $nonce_S$ > $_{kAS}$ > $_{SK}$.
7. *S* decrypts the message with its session key and then with the public authentication system key K_{AS}.
    a. If the message cannot be decrypted, if *UID* is incorrect, if $nonce_S$ does not match the nonce originally generated by *S*, or if the user has signed on to *S* previously then the sign-on to *S* is denied and *S* sends "USER_SIGN_ON", 4, < "failure" > $_{SK}$ to *U*.
    b. If the *UID* correctly matches *U*, if $nonce_S$ is equal to the nonce generated by *S* in step 2, and if *U* has never signed on to *S*, then the authentication procedure continues.
8. *S* generates a pseudo-random long number $seed_{US}$ to use as a common seed for the counter values when *U* signs on to *S*. *S* stores $seed_{US}$ as well as an integer $depth_{US}$ (initialized to 1), which tracks the number of repetitions necessary to generate the next counter value. *S* also calculates the maximum depth max_depth_{US} by observing the two least significant of $seed_{US}$ and setting max_depth_{US} to a number consisting of these two digits plus 1.
9. *S* → *U*: "USER_SIGN_ON", 4, < "success", $seed_{US}$ > $_{SK}$. *S* grants an access session to *U*.
10. *U* stores $seed_{US}$, initializes its own stored $depth_{US}$ to 1, sets max_depth_{US} to the two least significant digits in $seed_{US}$ + 1, and associates these values with *S* to use

for subsequent sign-on attempts. The user is now granted a session to the service provider.

5.2.3. Subsequent User Sign-on to a Service Provider

The following procedure is performed when a user attempts to sign on to a service provider that they have already successfully logged on in the past. The $seed_{US}$, $depth_{US}$ and max_depth_{US} fields are stored by both *U* and *S* and must remain synchronized for successful authorization.

1. User *U* wants to access a service provider *S* that it has accessed before. The client software provides an interface for the user to choose *S*, enter their account username and password, and begin the authentication process.
2. *U* establishes a secure session key connection with *S*.
3. $U \rightarrow S$: "USER_SIGN_ON", 1, $< UID >_{SK}$.
4. $S \rightarrow U$: "USER_SIGN_ON", 2, $< nonce_S >_{SK}$.
5. *U* performs the authentication message request procedure (from section 5.2.1) using *UID* and $nonce_S$.
    a. If the authentication is successful, *U* receives the message < "success", $< UID, nonce_S >_{kAS}$ > and continues the sign-on procedure.
    b. If the authentication is not successful, *U* receives the message < "failure" $>_{SK}$, aborts the sign-on procedure and instructs *S* to do the same by sending "USER_SIGN_ON", 3, < "failure" $>_{SK}$.

6. *U* generates the next counter value to send by retrieving the stored $seed_{US}$ value, using it to seed a new generator, iterating through the generator $depth_{US}$ times and saving that generated number as $counter_{US}$.
7. $U \rightarrow S$: "USER_SIGN_ON", 3, < "success", < *UID*, $nonce_S >_{kAS}$, $counter_{US} >_{SK}$.
8. *S* decrypts the message with its session key and then with the public authentication system key K_{AS}.
    a. If the message cannot be decrypted, if *UID* is incorrect, if $nonce_S$ does not match the nonce originally generated by *S*, or if the user has never signed on to *S* before, then the sign-on to *S* is denied and *S* sends "USER_SIGN_ON", 4, < "failure" $>_{SK}$ to *U*.
    b. If the *UID* correctly matches *U*, if $nonce_S$ is equal to the nonce generated by *S* in step 2, and if *U* has signed on to *S* before, then the authentication procedure continues.
9. *S* uses the same process that *U* used in step 6 to calculate $counter_{US}$.
    a. If the counter generated by *S* matches the counter sent by *U*, send "USER_SIGN_ON", 4, < "success" $>_{SK}$ to *U*. *S* grants an access session to *U* and increments $depth_{US}$ by 1.
    b. If the counter generated by *S* does not match the counter sent by *U*, send "USER_SIGN_ON", 4, < "failure" $>_{SK}$ to *U*. *S* does not grant access to *U* and does not increment $depth_{US}$.
10. *U* receives the message from *S* and decrypts the contents with the session key.
    a. If the message is "success", *U* increments $depth_{US}$ by 1. The user is now granted a session to the service provider.

b. If the message is “failure”, the client software reports an authorization error to the user. The $depth_{US}$ is not incremented.

When max_depth_{US} successful logins have been performed, $depth_{US}$ equals max_depth_{US} and both *U* and *S* must generate new $seed_{US}$, $depth_{US}$ and max_depth_{US} values. This is done by using the last used counter value as the new $seed_{US}$, setting $depth_{US}$ back to 1, and calculating a new max_depth_{US} by creating a number from the last 2 digits of $seed_{US}$ and adding 1. *U* and *S* perform this counter update without indicating in a message that the change is being performed.

5.3. Identity Management Processes

Identity management involves modifying an existing SeDSSO account on the authentication service. The following protocol allows a user account to be invalidated, so that any subsequent attempts to sign on are unsuccessful.

5.3.1. User Account Invalidation

The user’s system generates an invalidation number when a user account is created. The secure hash of this value is distributed to each authentication server for storage, and the actual value is stored on the user’s system (not the USBID). Access to the invalidation code is the only information necessary to invalidate the account because a thief may change the password and user information immediately after theft. In the event of a USBID theft, a computer system possessing the invalidation file can prevent the stolen account from being used.

1. User *U* begins the invalidation process by initiating invalidation in the client software. If the software can locate the file containing the invalidation number *INV* then the process continues.
2. $U \rightarrow A_1 \ldots A_n$: "INVALIDATE_USER", 1, *UID*, *INV*.
3. Each authentication server hashes the received *INV* value.
    a. If the calculated hash is equivalent to the stored hash(*INV*) for *U*, the authentication server removes the user's account from the system and sends "INVALIDATE_USER", 2, "success" to *U*.
    b. If the calculated hash is not equivalent to the stored hash(*INV)* for *U*, the authentication server does not remove the user's account from the system and sends "INVALIDATE_USER", 2, "failure" to *U*.
4. *U*'s client software tallies the responses received from all authentication servers.
    a. If more than $n - t$ invalidation attempts succeeded, user authentication is no longer possible and a successful invalidation is reported.
    b. If $n - t$ or fewer invalidation attempts succeeded, user authentication is still possible and a failed invalidation is reported.

Chapter 6

SeDSSO Implementation and Results

A project simulating the operation of each SeDSSO component has been created to test the performance and correct operation of a working SeDSSO system. The project is programmed in Java and compiled with the Java SE 6 platform. Java security libraries are used for public-key and symmetric-key encryption, Java network libraries are used for communication between components, and the ThreshSig library [27] created by Stephen Weis is used for threshold cryptography.

The certificate authority (CA) server, authentication server, service provider, and user are implemented as separate classes within the program and each is executed on a different virtual machine. SeDSSO processes describing the communication between these components have been implemented in the simulation project according to the specifications in chapter 5. Routines were developed to test performance by measuring the operation time of selected processes and test correctness by verifying the output against expected results. This chapter discusses the data collected from these tests.

6.1. Certificate Authority Server Program

The certificate authority (CA) server program is implemented as a set of 4 major Java classes.

1. RootCA.java:
    - initializes itself as a working instance of a certificate authority server
    - generates the public key for the authentication service, and a set of partial keys to distribute to individual authentication servers
    - instantiates a RootCAReceiver object that waits for messages from SeDSSO authentication servers
2. RootCAReceiver.java:
    - binds to a specific port on the authentication server's IP address, receiving initial incoming messages and creating new RootCAConnection objects to handle the connections
3. RootCAConnection.java:
    - manages a connection with another component from start to finish
    - sends and receives messages to and from the other component
    - uses a RootCAProtocol object to track the state of the connection, process incoming messages, and create outgoing messages
4. RootCAProtocol.java:
    - contains code to distribute the public and partial authentication service keys to authentication servers
    - stores the current process and step number, and processes an incoming message only if it is the expected message

- creates messages and sends them to the connected SeDSSO component

The CA server program generates the threshold keys necessary for the distributed authentication service. RSA-based threshold cryptography is implemented using ThreshSig, a Java implementation of Shoup's threshold signature scheme created by Stephen Weis. The CA server uses a ThreshSig Dealer object to create a 512-bit RSA public/private key pair and split the private key into a set of partial keys. Once the keys have been created, the CA server accepts connections from authentication servers and distributes these keys.

While this over-the-network distribution of the partial keys conflicts with the manual distribution described in the SeDSSO protocol, the simulation operates this way for ease of setup and testing. In a real threshold cryptography system, a root CA would not be accessible by other systems.

6.2. Authentication Server Program

The authentication server program is implemented as a set of 4 major Java classes.

1. AuthServer.java:
    - initializes itself as a working instance of an authentication server
    - stores all the information that an authentication server must retain about itself, other authentication servers, service providers, and users
    - instantiates an AuthServerReceiver object that waits for messages from other SeDSSO components
2. AuthServerReceiver.java:

- binds to a specific port on the authentication server's IP address, receiving initial incoming messages and creating new AuthServerConnection objects to handle the connections

3. AuthServerConnection.java:
    - manages a connection with another component from start to finish
    - sends and receives messages to and from the other component
    - uses an AuthServerProtocol object to track the state of the connection, process incoming messages, and create outgoing messages
4. AuthServerProtocol.java:
    - contains all SeDSSO authentication server protocol code (the implementation of the chapter 5 processes)
    - stores the current process and step number, and processes an incoming message only if it is the expected message
    - creates messages and sends them to the connected SeDSSO component

The authentication server implementation creates a working server instance and using this instance to perform all authentication server operations. SeDSSO requires a set of authentication servers to form an authentication service, so each server class retrieves the predefined addresses, ports, and public keys of the other servers at runtime. Constants in the AuthServer class make it possible to change both the total number of authentication servers (the *n* value) and the required number of successful authentication servers (the *t* value) from one execution to the next. This was used to easily perform the same tests using authentication services of different sizes.

Each authentication server possesses a ThreshSig KeyShare object which encapsulates the server's partial key. These KeyShares are used to create signatures of the user identity voucher described in section 5.2.1. ThreshSig enables *t* signatures to be combined into one voucher by an authentication server, and this voucher is sent to the user who forwards it to the service provider.

6.3. Service Provider Program

The service provider program is implemented as a set of 4 major Java classes.

1. ServiceProvider.java:
    - initializes itself as a working instance of a service provider system
    - stores all the information that a service provider must retain about itself, authentication servers and users
    - accepts command line input to begin the process for creating a service provider record on the authentication service
    - instantiates a ServiceProviderReceiver object that waits for messages from SeDSSO users
2. ServiceProviderReceiver.java:
    - binds to a specific port on the service provider's IP address, receiving initial incoming messages and creating new ServiceProviderConnection objects to handle the connections
3. ServiceProviderConnection.java:
    - manages a connection with another component from start to finish
    - sends and receives messages to and from the other component

- uses a ServiceProviderProtocol object to track the state of the connection, process incoming messages, and create outgoing messages

4. ServiceProviderProtocol.java:
    - contains all SeDSSO service provider protocol code (the implementation of the chapter 5 processes)
    - stores the current process and step number, and processes an incoming messages only if it is the expected message
    - creates messages and sends them to the connected SeDSSO component

The service provider implementation does not actually provide a service, but it performs all functions necessary to create a service provider identity and add it to the authentication service. It also allows new and returning users to connect to the service and performs all the steps necessary to trust a user. The counter system is fully implemented, with the code necessary to generate a new counter value for users and modify the counters each time a successful login takes place.

When the service provider receives a user identity voucher, it uses ThreshSig code to verify that the original voucher message was properly signed by the authentication service.

6.4. User Program

The user program is implemented as a set of 3 major Java classes.

1. User.java:
    - initializes itself as a working instance of the client software

- stores all the information that a client program must retain about the user/users, authentication servers and service providers
- accepts input to create a user account on the authentication service, retrieve the list of service providers, connect to a service provider with an existing user account, and invalidate the user account on the authentication service

2. UserConnection.java:
    - manages a connection with another component from start to finish
    - sends and receives messages to and from the other component
    - uses a UserProtocol object to track the state of the connection, process incoming messages, and create outgoing messages
3. UserProtocol.java:
    - contains all SeDSSO user protocol code (the implementation of the chapter 5 processes)
    - stores the current process and step number, and processes an incoming message only if it is the expected message
    - creates messages and sends them to the connected SeDSSO component

The user program performs all actions that would be initiated through the SeDSSO client-side software. This program is responsible for beginning the user account creation process with the authentication service. Once an account has been established the program retrieves the list of service providers from the authentication service and allows the user to perform initial and subsequent logins to these service providers.

Additionally, the program enables the user to contact the authentication service and invalidate their account.

Unlike the service provider and authentication server programs, the client software does not require a connection receiver running in the background because users are responsible for sending the first message in all user-related processes. The authentication server addresses are set in the client software, and a random authentication server is chosen from this list to begin communications with the authentication service. If a server is unavailable then another server contact is attempted. This process repeats until a working connection is established or all the authentication servers are found to be unavailable.

USBID devices have not been implemented due to the extensive development and monetary investment that this would require. At this time the USBID functionality is simulated in the client software. The user's private key and counter values are stored as User class attributes instead of directly on the USBID, and the USBID is always assumed to be present in processes where it is required.

6.5. Implementation Tests

6.5.1. Test Specifications

SeDSSO implementation tests use high-resolution system timer measurements and command line output provided by a set of User class functions. In addition to reporting the success or failure of a test, the completion time of the test is measured from the time the user program begins the process to the time it receives the final result message for that process.

The three SeDSSO functions that compose the majority of a working system's operations are used for testing. The first test is user account creation described in section 5.1.4. In an ideal situation *n* authentication servers are available and the user account is created successfully on every server. However, user account creation should succeed even when some authentication servers are unavailable (provided that at least *t* servers are working). The user-contacted authentication server must record the unavailable servers and inform them of the new user when those servers become available. In the event that less than *t* authentication servers are working the user should receive a message reporting that account creation failed and no authentication server should store the user's information.

The second test is user sign-on to a service provider. Signing on consists of several different processes described in section 5.2. The user contacts the service provider and requests access, and the service provider returns a random nonce value. The user then requests an identity voucher from the authentication service and sends the nonce to be included in the voucher. If *t* or more authentication servers are available and if those servers can authenticate the user, a voucher message signed with the authentication private key is sent back to the user. Once the service provider examines and verifies a successful voucher, the counter value operation is performed. If the counter value is created successfully (for a new counter) or verified successfully (for an existing counter) then the service provider trusts the user and reports a successful sign on.

The final test is user invalidation with the authentication service. This process is presented in section 5.3.1. The user program sends an invalidation message to each authentication server individually, and the process is successful if more than $n - t$ servers

are invalidated. In this case, less than *t* authentication servers will trust the user and authentication is no longer possible. If any invalidation attempts fail, the user system will log them and periodically attempt to invalidate its account on these authentication servers.

6.5.2. Test Environment

Individual components of the SeDSSO simulation were executed on separate Sun Blade workstations running the Solaris 10 operating system. Each workstation contains a 1 GHz Ultra SPARC III 64-bit processor and 1024 MB RAM, and they all connect to the same 100 Mbps network.

Tests were run first with 3 authentication servers and then with 9 authentication servers, allowing SeDSSO performance to be analyzed as the size of the distributed authentication service increases. In addition, the tests were performed with all of the authentication servers working and then with varying numbers of servers working. This enables the performance effect of unavailable servers to be measured. The number of components running simultaneously in our tests ranged from a minimum of 5 (1 CA server, 2 authentication servers, 1 user, and 1 service provider) to a maximum of 12 (1 CA server, 9 authentication servers, 1 user and 1 service provider).

The time necessary to detect an unavailable system varies in different operating system environments. Most UNIX and Linux operating systems do not retry the connection after the first failure and instead return a socket error within several milliseconds, while Windows retries the connection 5 times with increasing wait times as described by [18, 19]. In initial SeDSSO tests (run on Windows systems) the delay

averaged around 1 second per unavailable system, resulting in poor performance and skewed time measurements. Although a partial workaround for the Windows delay was found, the test environment was moved to the Solaris systems in order to achieve more realistic test results.

6.6. Test Results

The following test results were calculated by averaging the results of 50 individual tests. Prior to the measurements, the tested operation was run once to make sure that the Java virtual machine had performed all of the necessary compilations.

6.6.1. User Account Creation

Figure 6.1 presents the time that it takes to create a SeDSSO user account with an authentication service composed of 3 servers ($n = 3$). The minimum number of servers required to use the authentication system private key was set to 2 ($t = 2$). When all authentication servers are available the average account creation time is .6842 seconds and with only two servers working that time decreased to .6173 seconds. If less than 2 servers are available the user program correctly reports an inability to achieve account creation.

Figure 6.1: User account creation times for $n = 3$ and $t = 2$.

User account creation tests were also run on a SeDSSO system with $n = 9$ and $t = 5$, and authentication service sets of 9, 7, and 5 working servers were tested. This data is shown in figure 6.2. With all servers working the average completion time is .7139 seconds, decreasing to .6815 seconds when only 7 servers are available and further decreasing to .6766 seconds with only 5 servers functioning. If any less than 5 servers are available then an account creation error occurs.

Figure 6.2: User account creation times for $n = 9$ and $t = 5$.

If less than n authentication servers are running then unavailability is encountered at two points in the account creation process. When the user randomly selects an initial authentication server to contact (as described in step 1 of the protocol in section 5.1.4) there is a chance that an unavailable server will be contacted. One or more additional random attempts will be necessary to find an authentication server that is available. Once a connection with a working authentication server has been established, that authentication server will encounter the unavailable server or servers as it attempts to connect to all other authentication servers.

The account creation time decreases as the number of unavailable servers increases because detecting unavailability is faster than the account creation process. Unavailability is detected in several milliseconds, but the communication between two working systems can take several tenths of a second (although the multi-threaded authentication server implementation minimizes the delay by allowing multiple

connections to progress simultaneously). In the n = 3 test, having only 2 available servers results in a 9.8% decrease in account creation time. The n = 9 test shows a decrease of 4.5% when only 7 servers are available, and that time is decreased by an additional .72% when moving to 5 available servers.

While the account creation times appear better when fewer authentication servers are working, a realistic SeDSSO authentication service would need to pass the newly-created user to the unavailable servers when they resume availability (this process was not implemented in our simulation). In that case, the additional overhead would make the total performance requirement of unavailable servers more costly than when all authentication servers are working.

6.6.2. User Sign-On

The times measured for user sign-on tests with n = 3 and t = 2 are shown in figure 6.3. Three available authentication servers yield an average sign-on time of 1.7438 seconds. If one of the servers is disabled the time rises to 2.0891 seconds, a 19.8% increase in sign-on time.

Figure 6.3: User sign-on times for $n = 3$ and $t = 2$.

User sign-on was also tested with $n = 9$, $t = 5$, and 9, 7, and 5 authentication servers available. This data is shown in figure 6.4. When all servers are available the average sign-on time is 1.8328 seconds. With only 7 servers available the time increases to 2.3167 seconds (a 26.4% sign-on time penalty), and 5 servers functioning raises the sign-on time to 2.4883 seconds (an additional 7.4% increase in time).

Figure 6.4: User sign-on times for $n = 9$ and $t = 5$.

Unlike account creation, the sign-on process does not need to attempt a connection with all n authentication servers. Once the first server is contacted, that server only needs to receive signatures from t-1 different servers in order to sign the user identity voucher with the authentication service private key. The contacted authentication server chooses the set of t-1 servers at random and attempts to create connections with all of them simultaneously. The process is designed this way to minimize the load on the authentication service and improve sign-on times.

When the entire authentication service is available, all of the initial random server connections are successful and the voucher is created in the fastest time possible. This is verified by the times for 3 and 9 servers available in figure 6.3 and figure 6.4 respectively. As the number of available servers declines, the more likely it becomes that the user needs to contact multiple authentication servers until it discovers a working server. Additionally, the contacted server may encounter connection errors with other

servers and thus need to attempt new connections to collect t signatures. While communication with a newly-contacted server consumes the same amount of processing time as the initial connections, the new connections begin at a delayed time and subsequently increase the total length of the single sign-on process.

6.6.3. User Account Invalidation

User account invalidation is a straightforward process. The user program contacts each authentication server individually and presents the invalidation number. The hash of this number must match the hash that was presented at the user account creation in order for an authentication server to remove the user's account. If more than $n - t$ servers invalidate the user account then future authentication attempts are impossible and invalidation is a success. Figure 6.5 and 6.6 show the times for account invalidation.

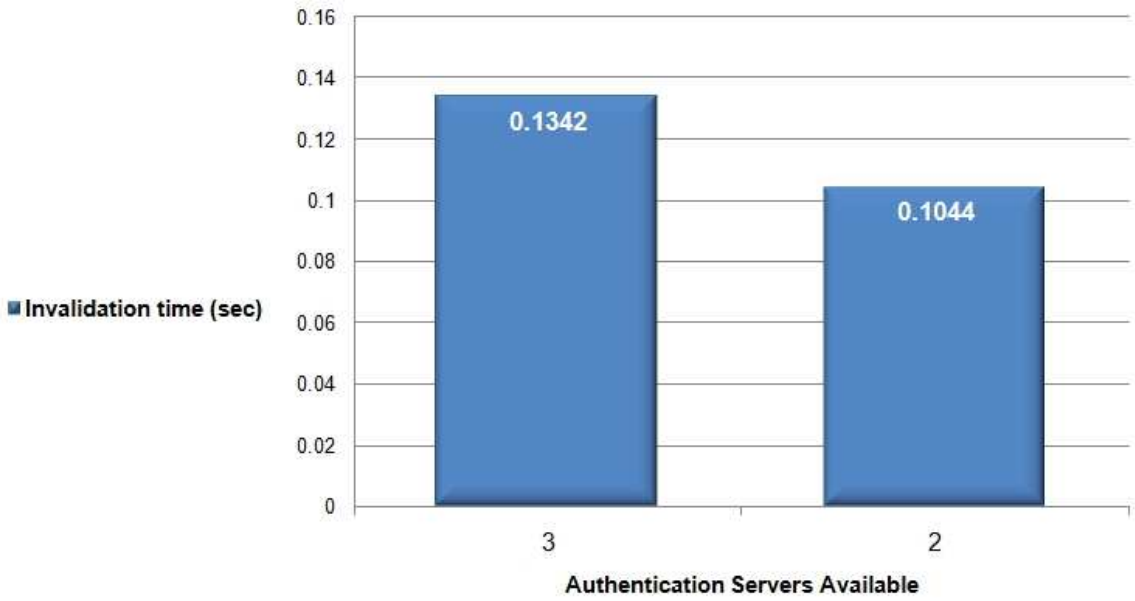

Figure 6.5: User account invalidation times for $n = 3$ and $t = 2$.

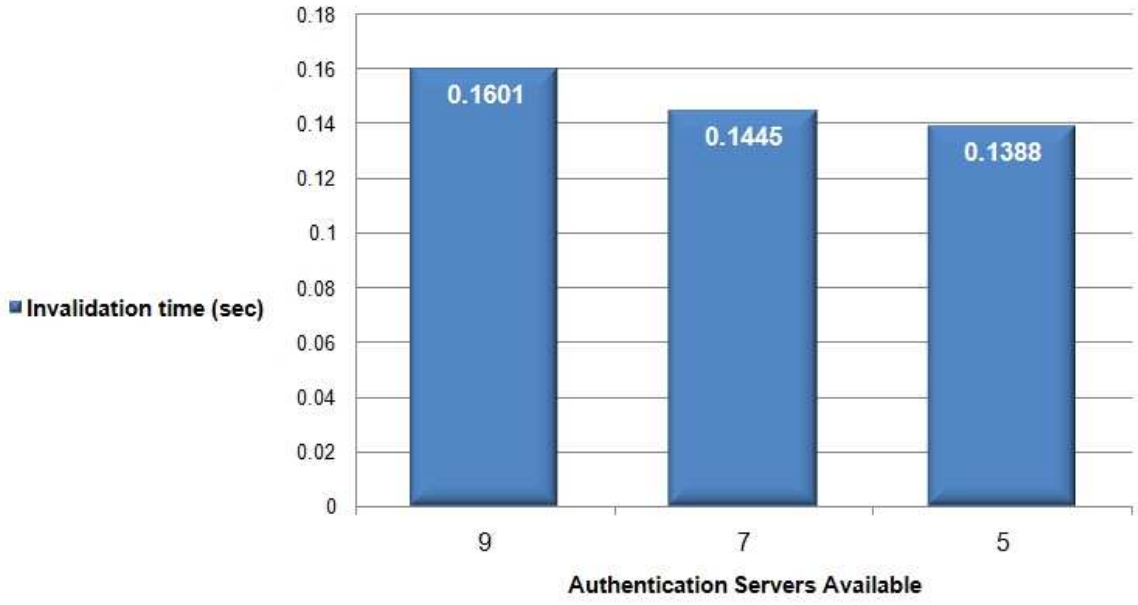

Figure 6.6: User account invalidation times for $n = 9$ and $t = 5$.

For the system where n=3 and t=2, invalidation with all 3 authentication servers available takes .1342 seconds on average. If only 2 servers are available, the time is reduced to .1044 seconds. Likewise, the invalidation times for n=9, t=5 with 9, 7, and 5 servers available are .1601 seconds, .1445 seconds, and .1388 seconds respectively.

The user simulation performs invalidation sequentially with each authentication server. If a server is available then the invalidation process is executed completely, and that server responds with either a success or failure. When a server is unavailable, the user considers the invalidation to have failed for that server. The act of invalidation requires more time than the detection of an unavailable server, resulting in data that is similar to the measurements from user account creation. As the number of available servers decreases, invalidation time decreases.

Even though user sign-on is impossible after $n - t$ authentication servers have performed invalidation, it is beneficial for user security and server performance and

storage to invalidate the user on all authentication servers, including those that might have been unavailable at the time of invalidation. This requirement could be implemented in the user software, but a more reliable method would involve the authentication service creating an invalidation queue for unavailable servers. When a server came back online, it would need to perform all actions on the queue. Despite the slightly faster invalidation times when some servers are unavailable, the overhead of these servers would cause more work than if all authentication servers had been available.

6.7. Security Analysis

In the past, SSO systems have experienced vulnerabilities to two major security attacks. A man-in-the-middle attack occurs when an eavesdropper intercepts messages between two parties to change them without either party knowing that such an attack has taken place. Given the distributed flow of internet traffic, it is possible for an eavesdropper with access to a routing device to observe raw communication messages in any protocol. These attacks have taken place on systems with various security protocols, including some that rely on public-key cryptography.

SeDSSO is immune to man-in-the-middle attacks. In order for a man-in-the-middle attack to work against SeDSSO's public-key authentication system, the eavesdropper needs to replace the real key pairs with counterfeit key pairs and assume that the communicating systems will still operate given these replacements. The public keys belonging to individual SeDSSO authentication servers and the public key for the entire authentication service are widely distributed, and a root certificate authority vouches for their authenticity. The public key for the certificate authority can be

embedded directly in the USBID as well as service provider software so that user and service provider systems can make sure that an authentication public key is correct.

The SeDSSO public and private keys are generated using the RSA public-key cryptography algorithm. Every communication session begins by encrypting messages with public keys until a secure symmetric session key can be created (as described in section 5.1.1). In order to read or modify communications the attacker needs to know a private key or the symmetric session key generated during public/private key communications. Given a secure RSA key pair (such as 2048-bit size) and a secure symmetric AES key (such as 256-bit size), the probability of calculating one of the keys within a reasonable timeframe is virtually zero. The National Institute of Standards and Technology (NIST) estimates that based on projected computer system speed increases, 2048-bit RSA keys and 256-bit AES keys should remain secure until at least 2030 [6].

Trojan horse attacks are more subversive because they take direct control of the user's system. The Trojan program runs in the background and waits until a connection has been established. It then sends requests over this connection to perform malicious activities with the user's identity. The communication protocol and server architecture of an authentication system would be unable to prevent this, no matter how secure it is. Protection must be implemented directly in the client-side software or hardware. Although a simulation of SeDSSO has been programmed, the full client software is not yet developed. Consequently, testing to gauge SeDSSO's Trojan attack resistance cannot be performed at this time.

Several security approaches may allow SeDSSO and other two-factor authentication schemes to effectively resist Trojan attacks. Client software that makes it impossible for a SeDSSO connection to be established without forced user interaction could alert user to a Trojan operating in the background, but it is difficult to guarantee that this interaction cannot be bypassed in some way. The new initiative known as trusted computing may also be able to defend against these attacks by limiting the ability of other programs to interact with the user's session. However, at this time the future of trusted computing is unclear and the potential advantages and disadvantages are still being discussed [7].

Chapter 7

Conclusion and Future Work

7.1. Conclusion

In this thesis we presented SeDSSO, a secure and fail-safe Internet authentication SSO architecture. Threshold encryption and a distributed authentication service allow SeDSSO to eliminate authentication as a central point of failure. Although the existing single sign-on systems CorSSO and ThresPassport rely on distributed authentication with threshold encryption, SeDSSO improves on their security and usability by implementing a two-factor authentication scheme consisting of a username/password combination and the USBID.

A protocol describing the interaction between SeDSSO users, service providers, and the authentication service has been developed. Our simulation implements every function of this protocol and yields consistently correct operations with favorable performance measurements. The simulation also demonstrates the advantages of distributed authentication. Even with $t-1$ authentication servers disabled (almost half of the authentication service), all functions are still available and in most cases the system suffers only a minor performance penalty.

As more people use more Internet sites, they need a way to replace many identities with one easy-to-use highly secure entity that can be used anywhere without fear of identity theft. SeDSSO was designed to fulfill this need, and initial tests show the potential of our solution. However, more work must be done to test SeDSSO in an environment that realistically simulates the stress that a high-volume Internet authentication service would need to endure.

7.2. Future Work

7.2.1. Complete Implementation

Now that the SeDSSO protocol has been developed and a simulation has been programmed, the next step in extending this project is the development of a complete realistic implementation. Each authentication server should run on its own high-performance system and they should be arranged in a separate authentication service network. Threshold encryption should be implemented on a longer RSA key, with tests to measure and compare the performance of 1024-bit, 2048-bit, and possibly larger keys. A network-isolated CA server should be used to generate the authentication public and private key and corresponding partial keys.

Many test operations should be performed at once with sign-on attempts, account creations, and account invalidations occurring simultaneously. This would allow for more realistic measurements than the ones presented in chapter 6, which were performed in isolation.

A more realistic SeDSSO prototype requires the creation of a physical USBID device. This USB device must consist of a specialized microcontroller, flash memory,

and the architecture necessary to connect them. A low-level communication protocol between the USBID and the user's system would need to be designed. Both the microcontroller and an operating system driver must implement an end of this protocol to allow communication between the client system and the authentication service. Once a working driver is written, it would be possible to program the client software to use the USBID as defined in chapters 4 and 5 of this thesis.

A realistic implementation would make it possible to analyze SeDSSO's response to security attacks of different types. Investigation of the implementation's response to simple Trojan virus programs could identify potential vulnerabilities. If any vulnerabilities are discovered, client-side modifications could be proposed and programmed in an attempt to secure the system.

7.2.2. Unavailable Authentication Server Detection

It may be possible to reduce unavailable authentication server delays in the sign-on process by creating a way to monitor the status of these servers in real time. There are two different methods by which this could be achieved, and each has a set of potential issues that would need to be researched and resolved.

The first method for detecting unavailable servers would involve the addition of new systems to the authentication service known as the availability servers. An availability server sends a small message to each authentication server at a set time interval, and a return message from each server is required to verify availability. If an authentication server does not respond, the availability server marks the non-responding server as unavailable for the duration of the interval.

When a user or authentication server needs to contact authentication servers at random, the contacting party asks an availability server for an updated availability list (or reuses a recently-acquired list that has not expired). Using this list allows the contacting system to choose only those servers which were recently available.

While this method would allow for propagation of the server status list to all authentication servers and users, the bandwidth load imposed by a large number of users would require a set of high-performance availability servers possibly rivaling the authentication servers themselves. Implementation of this type of scheme in a SeDSSO system would allow the true performance requirement to be assessed.

The second method would move the creation of this server status list from a set of availability servers to the authentication servers themselves, with each authentication server maintaining its own list. If one server discovers that another server is unavailable, the server that made the discovery adds a message to its list indicating this unavailability. When a random authentication server selection must be made, this list prevents servers which were recently unavailable from being contacted. Consequently, unavailable servers will be avoided in the random selection process (following the unavailability discovery) and delays will be minimized for authentication servers. Servers can be considered available again either after a specified period of time or whenever they notify other servers that they are back online.

This method has the advantage of not needing the addition of high-performance availability servers. However, in order to minimize load on the authentication servers, it may be necessary to limit the availability data to authentication servers themselves instead of distributing it to every user on a regular basis. If this is the case, some of the

possible delay (when the user randomly selects an initial contact authentication server) would remain. As with the first method, implementation would be necessary to judge the performance cost and benefit of this change.

References

[1] DigitalPersona, Inc., "Solving the Weakest Link: Password Security," http://www.digitalpersona.com/resources/downloads/Weakest_Link_wp_0205.pdf

[2] W. Josephson, E. Sirer, and F. Schneider, "Peer-to-Peer Authentication with a Distributed Single Sign-On Service," *3rd Int. Workshop on Peer-to-Peer Systems (IPTPS'04)*, San Diego, USA, February 2004.

[3] T. Chen, B. Zhu, S. Li, X. Cheng, "ThresPassport - A Distributed Single Sign-On Service," *International Conference on Intelligent Computing (ICIC) 2005*, Hefei, China, August 2005.

[4] A. Shamir, "How to Share a Secret," *Communications of the ACM Vol. 22 No. 11*, November 1979.

[5] L. Zhou, F. Schneider, and R. van Renesse, "COCA: A Secure Distributed On-line Certification Authority," *ACM Transactions on Computer Systems 20, 4*, pp. 329—368, November 2002.

[6] NIST, "Recommendation for Key Management – Part 1: General (Revised)," *NIST Special Publication 800-57*, May 2006.

[7] S. Schoen, "Trusted Computing – Promise and Risk," Electronic Frontier Foundation (EFF) web site, www.eff.org/Infrastructure/trusted_computing/20031001_tc.pdf.

[8] RSA Security, "The 2nd Annual RSA Security Password Management Survey," August 2006.

[9] G. R. Blakley, "Safeguarding Cryptographic Keys," *AFIPS Conference Proceedings, vol.48. 1979 National Computer Conference*, pp. 313-317, 1979.

[10] M. Tompa and H. Woll, "How to Share a Secret with Cheaters," *Proceedings on Advances in Cryptology -- CRYPTO '86*, pp. 261-265, 1986.

[11] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," *ACM Transactions on Programming Languages and Systems*, Volume 4 Issue 3, July 1982, pp. 382-401, 1982.

[12] M. Ghosh and S. Makki, "A Secure Framework for Electronic Payment System," *Proceedings of the International Conference on Internet Computing*, Las Vegas, Nevada, USA, June 21-24, 2004.

[13] P. Fouque and J. Stern, "Fully Distributed Threshold RSA under Standard Assumptions," *ASIACRYPT 2001*, pp. 310-330, 2001.

[14] I. Damg˚ard and M. Koprowski, "Practical Threshold RSA Signatures without a Trusted Dealer," *Eurocrypt '01*, pp. 152-165, 2001.

[15] V. Shoup, "Practical Threshold Signatures," *Eurocrypt '00*, Vol. 1807, pp. 207-220, 2000.

[16] R. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *Communications of the ACM*, pp. 120-126, 1978.

[17] A. Pashalidis and C. Mitchell, “A Taxonomy of Single Sign-On Systems,” *Information Security and Privacy, 8th Australasian Conference, ACISP 2003,* 2003.

[18] Sun Developer Network, Java Bug Database, “Slow socket unavailability detection on Windows,” bug ID: 4424770, http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4424770, March 2001.

[19] Microsoft TechNet, description of the TcpMaxDataRetransmissions registry setting, http://www.microsoft.com/technet/prodtechnol/windows2000serv/reskit/regentry/58805.mspx?mfr=true, April 2007.

[20] J. Tuomy, “Addressing High-Risk Remote Access Applications with Challenge / Response User Authentication,” *Telecommunications American Edition, March ‘95*, Vol. 29, p. 58, 1995.

[21] R. Smith, “Authentication: From Passwords to Public Keys,” 1st edition, Addison-Wesley, 2002.

[22] RSA Security, RSA SecurID product information, http://www.rsa.com/.

[23] P. Madsen, Y. Koga, and K. Takahashi, “Federated Identity Management for Protecting Users from ID Theft,” *Proceedings of the 2005 Workshop on Digital Identity Management*, Fairfax, VA, 2005.

[24] A. Biryukov, J. Lano, B. Preneel, “Cryptanalysis of the Alleged SecurID Hash Function,” *Lecture Notes in Computer Science, proceedings of SAC'2003*, 2003.

[25] B. Schneier, “Two-Factor Authentication: Too Little, Too Late,” *Communications of the ACM*, Vol. 48, No. 4, April 2005.

[26] D. Kormann and A. Rubin, “Risks of the Passport Single Signon Protocol,” *IEEE Computer Networks*, July 2000.

[27] ThreshSig: Java Threshold Signature Package, created by Stephen A. Weis, http://threshsig.sourceforge.net/.