UNIVERSITY OF CALIFORNIA,
IRVINE

SCURL Authentication
A decentralized approach to entity authentication

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Computer Science

by

Michael Scott Wolfe

Thesis Committee:
Professor David Kay, Chair
Professor Richard Taylor
Professor Michael Goodrich

2011

© 2011 Michael Scott Wolfe

DEDICATION

to

Mom and Dad

Your love, compassion, and support are the reasons for my success.
Words cannot express my gratitude to you both for what you have instilled in me.

TABLE OF CONTENTS

Page

LIST OF FIGURES **v**

LIST OF TABLES **vi**

ACKNOWLEDGMENTS **vii**

ABSTRACT OF THE THESIS **viii**

1 Introduction **1**

2 Background **5**
2.1 CREST 6
2.2 TLS/SSL 7
2.3 Self-certifying pathnames 21
2.3.1 Self-certifying URLs 30
2.4 Revocation 33
2.5 User Authentication 41
2.5.1 OpenID 42
2.5.2 Shibboleth 53

3 SCURL Authentication **65**
3.1 Design 66
3.1.1 SCURL Authentication Protocol 69
3.1.2 Revocation 75
3.2 Implementation 80
3.2.1 SCURL Authentication Protocol 82
3.2.2 Revocation Service 85

4 OpenID **90**
4.1 OpenID Protocol 90
4.2 libopkele 97
4.3 CREST OpenID Implementation 99

5 Evaluation **105**
5.1 Benchmark 106
5.1.1 OpenID 110

6 Conclusion **113**

Bibliography **115**

Appendices **118**
A Appendix A 118

LIST OF FIGURES

Page

2.1 Beginning of the TLS handshake . . . 16
2.2 TLS handshake without client authentication . . . 17
2.3 TLS handshake with client authentication . . . 19
2.4 A SFS Self-certifying pathname . . . 22
2.5 SFS key negotiation protocol . . . 27
2.6 SFS server authentication failure . . . 30
2.7 An SFS-HTTP self-certifying URL . . . 31
2.8 The SFS revocation system . . . 38
2.9 A non-applicable revocation program . . . 39
2.10 A revocation program example . . . 40
2.11 OpenID protocol (Initialization) . . . 47
2.12 OpenID protocol (Authentication) . . . 50
2.13 OpenID protocol (Verification) . . . 52
2.14 Shibboleth discovery and authentication . . . 58
2.15 Shibboleth HTTP POST binding . . . 61
2.16 Shibboleth HTTP Artifact binding . . . 62

3.1 A CREST SCURL . . . 67
3.2 SCURL authentication protocol . . . 70
3.3 CREST revocation service timing . . . 79
3.4 CREST revocation configuration . . . 89

4.1 OpenID association storage example . . . 103

LIST OF TABLES

Page

2.1 Five CREST Axioms . . . 6
2.2 TLS Characteristics . . . 20
2.3 SFS characteristics . . . 30
2.4 OpenID protocol characteristics . . . 53
2.5 Shibboleth characteristics . . . 64

3.1 SCURL authentication protocol characteristics . . . 75

5.1 SCURL authentication timing (same host) . . . 107
5.2 SCURL authentication CPU (same host) . . . 108
5.3 SCURL authentication timing (wired) . . . 109
5.4 SCURL authentication timing (wireless) . . . 109
5.5 SCURL authentication CPU (wired) . . . 110
5.6 SCURL authentication CPU (wireless) . . . 110
5.7 OpenID timing with the Racket FFI . . . 111

A.1 Client Hello Message Definition . . . 118
A.2 Server Hello Message Definition . . . 118
A.3 Client Authenticate Message Definition . . . 119
A.4 Server Authenticate Message Definition . . . 119
A.5 Client Done Message Definition . . . 119
A.6 Key Revocation Response Message Definition . . . 119

ACKNOWLEDGMENTS

I would like to thank Jane for lending me her strength and unending supply of optimism. I would like to thank Cynthia for the endless years of torture that helped to prepare me for this adventure. I would like to thank Jean Halliday for showing me how fun a challenge can be. I would like to thank Donna Pompei for making a difficult subject easy to understand. I would like to thank Michael Gorlick for never letting me take the easy path. Finally, I would like to thank Dr. Richard Taylor for giving me the opportunity to learn from a group of very intelligent people.

ABSTRACT OF THE THESIS

SCURL Authentication
A decentralized approach to entity authentication

By

Michael Scott Wolfe

Master of Science in Computer Science

University of California, Irvine, 2011

Professor David Kay, Chair

Self-certifying URLs can be used to authenticate parties in a peer-to-peer environment and CREST is a feasible framework for existing user authentication protocols. Existing entity authentication protocols do not provide decentralized, explicit authentication for both parties. The SCURL authentication protocol uses Self-certifying URL's to authenticate parties and was developed through examination of the TLS/SSL protocol and the Self-certifying File System. The SCURL authentication protocol is implemented in the Scheme programming language and is a feasible authentication protocol for the CREST framework. Additionally, the CREST developer must be able to utilize existing user authentication protocols. OpenID and Shibboleth are dissected to understand the current state of user authentication and the existing C++ OpenID library libopkele was used to develop a Racket OpenID module. The Racket OpenID module proves that it is possible to leverage existing user authentication libraries in the CREST framework.

Chapter 1

Introduction

Entity authentication has been heavily researched through the years. The TLS/SSL protocol and the SFS key negotiation protocol are two great examples of an entity authentication protocol, but both protocols suffer from different deficiencies. The TLS/SSL protocol suffers from a hole in revocation, misplaced trust, and an insecure business model. The SFS key negotiation protocol suffers from one sided, implicit authentication. These deficiencies are exacerbated when the protocols are used in a peer-to-peer architecture.

The TLS/SSL protocol uses digital certificates and certificate authorities to authenticate an entity. The digital certificate is issued by a certificate authority and signed using public key cryptography. The root certificate of a certificate authority is used to verify the authenticity of all digital certificates issued by that certificate authority. A certificate authority has no way to revoke a root certificate when the private key is compromised. This is a large security flaw in the TLS/SSL protocol and will eventually be the cause of major damage.

The second issue with certificate authorities is misplaced trust. Certificate authorities do background checks on a customer purchasing a digital certificate. The level of detail of the

background check varies among certificate authorities and is usually reflected in the cost of the digital certificate. Certificate authorities declare that a digital certificate conveys trust as to the intent of the party. This is simply not true and the most a digital certificate can provide proof of is that an entity owns the correct private key. Trust cannot be provided by an entity authentication protocol and is better suited to be defined by a decentralized web of trust approach.

Certificate authorities created a business model to increase profits. This is a standard approach to creating, maintaining, and growing a business. The problem lies in the fact that some decisions will increase profits at the cost of security. The life span of a digital certificate is abused to increase profits through extension fees when it should be determined by the strength of the entity's public key. This is a terrible practice for a company that sells security.

The SFS key negotiation protocol uses public key cryptography and self-certifying pathnames to authenticate a server to a client. This protocol provides a decentralized approach to entity authentication with no need for a certificate authority or central node that can become compromised. The SFS key negotiation protocol only authenticates the server to the client. This protocol was built to verify the authenticity of an entity that is running a remote file system. In this design, the server does not care to authenticate the client and was not a major concern, but it would be a major concern if this protocol is used in a peer-to-peer architecture.

The SFS key negotiation protocol's main goal is to establish a secure communications channel. The authenticity of the server is implicitly verified by creating the secure communications channel. This means that the authenticity is only verified after generating and trading secret information. It is important to establish the authenticity of a party as early

as possible to decrease the amount of resources expended when being attacked. The implicit authentication found in the SFS key negotiation protocol does not follow this guideline and must be addressed.

The SCURL authentication protocol and the CREST revocation system are the solutions to entity authentication in a peer-to-peer architecture. This protocol is a decentralized approach to entity authentication using public key cryptography and CREST SCURL's. The SCURL authentication protocol explicitly authenticates both parties and allows for revocation of all compromised public keys. The SCURL authentication protocol does not provide any form of trust about the intent of the remote entity, but the CREST revocation system was created to help developers leverage any web of trust that they wish to join.

The CREST framework uses the SCURL authentication protocol to authenticate at the entity level, but the CREST developer will need a way to assign credentials or capabilities at the user level. The leading user authentication protocols, OpenID and Shibboleth, are solutions to the single sign-on problem that emerged with the many web services being offered today. A CREST developer must be able to use these protocols for the CREST framework to be considered a feasible environment for future development efforts.

The OpenID and Shibboleth solutions separate the concerns of the party authenticating a user identity from the concerns of the party providing a service to the owner of the identity. The user is required to create a single identity at a trusted identity provider and can use this identity at multiple service providers. The goal of both OpenID and Shibboleth focuses on how the identity provider can securely vouch for the authenticity of the user to the service provider.

The OpenID protocol and Shibboleth allow a user to prove ownership of their identifier to a service provider through their identity provider. The OpenID protocol does not place restrictions on how an identity provider or service provider is trusted and allows any identifier in an abstract global namespace. Shibboleth establishes trust between an identity provider and a service provider by defining a federation. Shibboleth requires the use of federations where each federation is a whitelist of approved identity providers. These protocols are discussed to provide a deeper insight into the current state of single sign-on user authentication protocols.

An OpenID implementation was built for use in the CREST framework to prove that the CREST framework is a feasible environment for future development. The C++ library libopkele provides basic OpenID functionality and was used to create a Racket OpenID module. The Racket OpenID module is a fully functioning OpenID implementation for the service provider and is usable by CREST developers.

In the following sections, we will present information about the CREST framework, the TLS/SSL protocol, the SFS key negotiation protocol, the origins of the Self-certifying URL, the OpenID protocol, and Shibboleth. The SCURL authentication protocol is presented as a decentralized, entity authentication solution for the CREST framework. A discussion of the OpenID solution for the single sign-on problem is presented followed by the details of the Racket based OpenID implementation built for the CREST framework. Finally, the performance of the SCURL authentication protocol and the Racket OpenID implementation is presented to prove that they are both feasible solutions for use in the CREST framework.

Chapter 2

Background

It is important to understand both the platform for which the authentication protocol is meant and the current state of existing authentication protocols. By understanding the current state of existing authentication protocols we will be able to identify the problems that arise when putting them to use both in and out of a peer-to-peer environment. The CREST framework provides the landscape of where the SCURL authentication protocol is meant to be leveraged. The TLS/SSL protocol and the SFS key negotiation protocol will provide the understanding of current authentication protocols. By discussing these subjects, we hope to provide the motivation for the SCURL authentication protocol presented in Section 3.

Following that, the details of the OpenID and Shibboleth user authentication protocols are presented to examine the state of existing single sign-on protocols. A single sign-on protocol will surely become the standard for performing user authentication in a widely distributed world where users already suffer from the onerous task of memorizing identities and passwords. A system where a single user name and password can be used with any service provider will surely become the norm as time progresses. The CREST framework

must allow a developer to choose their own user authentication protocol and as such must provide an environment where these protocols can be successfully leveraged.

2.1 CREST

"CREST is a new architectural style for highly dynamic distributed applications." [23] The CREST architectural style is based on the premise of computation exchange among entities, or CREST islands. The CREST architectural style focuses on computational exchange and is a generalization of the REST architectural style, which focuses on state exchange. The creators of the CREST architectural style took lessons from the early days of the World Wide Web before the definition of the REST principles and sought to provide guidance for development in the CREST architectural style by defining five CREST axioms, presented in Table 2.1.

CA1. A resource is a locus of computations, named by an URL.
CA2. The representation of a computation is an expression plus metadata to describe the expression.
CA3. All computations are context-free.
CA4. Only a few primitive operations are always available, but additional per-resource and per-computation operations are also encouraged.
CA5. The presence of intermediaries is promoted.

Table 2.1: The five CREST axioms (Taken from page 3 of [23].)

The CREST framework embodies the CREST axioms and was developed to encourage the use of CREST as an architectural style and guide developers in the creation of CREST based applications. For example, a CREST based word-counting application would involve two CREST islands. Island A interfaces with the user and island B has access to a library that provides word counting functionality. The user will enter an URL at island A and submit

a request for the number or words. Island A will form a computation that fetches the user indicated resource, determines the number of words using the word counting functionality located on island B and return the result. Island A sends the thus formed computation to island B in the form of a query. Finally, island B evaluates the computation and returns the result to island A.

The authentication between island A and island B is crucial and must be done as one of the first steps in the communication process to provide the first layer of trust in this complex relationship. The authentication must be mutual where both islands can authenticate each other before moving forward with communication. The CREST framework is a peer-to-peer environment where computations are evaluated across multiple islands. The SCURL authentication protocol must provide strong authentication among CREST islands and the developer will require a user authentication protocol that is used to assign and limit the capabilities provided to a user.

2.2 TLS/SSL

The Transport Layer Security protocol (TLS) and the Secure Sockets Layer protocol (SSL), its predecessor, protects sensitive information that must be transmitted through dangerous channels. The protocol provides protection in three distinct areas: authentication, privacy, and data integrity. [15] Authentication validates the identity of both the client and the server. Privacy prevents a third party from understanding the sensitive information. Data integrity guarantees the correctness of the information across the channel.

Identity is authenticated with the use of public key cryptography, digital signatures, digital certificates, and certificate revocation. Public key cryptography is a cryptographic system

that uses two different keys and an asymmetric key algorithm to perform encryption and decryption. An asymmetric key algorithm is distinct because the key used to perform encryption is not the same key used to perform decryption. One key, the public key, is widely distributed and the other key, the private key, is kept as a closely guarded secret. The two keys are mathematically related, but it is extremely computationally expensive to determine the private key from the public key. The distribution of the keys forms a one to many relationship that is leveraged for both secrecy and authenticity.

The public key ensures secrecy when used to encrypt plaintext. The holder of the private key is the only entity that can decrypt the ciphertext and understand the information. Secrecy is ensured because the private key is a closely guarded secret and should only be known to the owner of the public key. If the private key is widely known then the secrecy provided by the public key is forfeited and the private key is considered compromised.

The private key proves the authenticity of the plaintext when used to encrypt. To prove the authenticity of the plaintext both the plaintext and ciphertext must be received through the communication channel. The ciphertext can be decrypted with the public key and compared against the known plaintext. The plaintext came from the holder of the private key when the decrypted ciphertext matches the known plaintext.

Digital signatures leverage the properties of encryption using a private key to ensure the origins of plaintext data. To produce a digital signature the plaintext data is hashed and encrypted using a private key. The digital signature will be attached to the plaintext data and transferred to another entity. Anyone with access to the public key can authenticate the origins of the data by comparing a fresh hash of the plaintext data against the decrypted digital signature. Digital certificates are the way in which digital signatures and public key cryptography are used to authenticate entities in the TLS/SSL protocol.

Authentication of an identity is verified with an X.509 digital certificate. [8] A digital certificate associates a public key with a subject and a digital signature authenticates the association. A digital certificate will contain the issuer, the subject, and the subject's public key. A digital signature is attached to the digital certificate using the private key of the issuer. A self-signed, or root, digital certificate is a certificate in which the issuer is the same as the subject. A self-signed certificate is authenticated using the public key listed in the certificate and proves that the subject has access to the correct private key. A "voucher" digital certificate is a certificate in which the issuer and the subject are different. The "voucher" digital certificate is authenticated using the public key of the issuer found in a different self-signed digital certificate. A "voucher" certificate proves that the issuer believes the subject has access to the correct private key.

Certificate revocation stops the use of compromised private keys. The digital certificate is revoked and the public key specified in the certificate should not be used when the private key of an X.509 certificate is compromised. An X.509 certificate can be revoked by using either a Certificate Revocation List (CRL) [10] or the Online Certificate Status Protocol (OCSP) [25]. A CRL is a list of certificates that have been revoked and should no longer be trusted. This method of revocation can be costly as access to the entire list must be provided for every certificate authentication and the CRL must be up to date. The OCSP is a protocol for querying the revocation status of a certificate from an external entity. The OCSP is an alternative to using a CRL and provides a better means of transferring information in a distributive fashion. The OCSP transfers less information than an entire CRL when a request is serviced which means that it can handle more requests at a faster rate.

Privacy in TLS/SSL is provided by encryption of the data as it is passed between the two parties. The encryption is done using a shared secret key and a symmetric key algorithm. In a symmetric key algorithm the same key is used for both encryption and decryption purposes.

The shared secret key is generated using exchanged random numbers and a *PreMasterSecret* that is always sent from the client to the server and encrypted with the server's public key. Public key cryptography is used to establish the shared secret between the two parties, but is not used for the duration of the communication because it is computationally slower than using a symmetric key algorithm.

Data integrity guarantees the correctness of the data across the channel. Privacy does not protect against data modification while in transit. It is important for parties to discard any messages that have been tampered with. It is possible for an attacker to tamper with the data and cause irreparable damage if the modification is not detected. To ensure data integrity the data sent across the channel is appended with a message authentication code. The message authentication code uses a keyed hash function to produce a value that can be used to verify the integrity of the data. The rest of this section will detail the historical background of the protocol, the issues which arise from using digital certificates for authentication and the details of the protocol.

History

The TLS/SSL protocol has a long history and is one of the most used forms of authentication in the world today. Effort on the original SSL protocol was started in 1993 by Netscape. They began by using the Secure Network Programming API [37] that provides security for traditional client-server style applications in the form of a secure sockets API similar to the BSD socket API. The intent was that current applications could back their way into secure communication by swapping the BSD socket API for the Secure Network Programming API. Eventually, this effort grew into the SSL 1.0 protocol, which was never publicly released by Netscape.

SSL 2.0 [21] was the first publicly available version of the specification released in 1995. It contained several security flaws including:

- The same cryptographic key being used to generate message authentication codes and to perform encryption, which increases the information available to cryptanalysis.
- The message authentication code algorithm is vulnerable to a length extension attack, which allows an attacker to concatenate information onto the data and generate a correct message authentication code for the modified data.
- A man in the middle downgrade attack is possible because there is no protection during the handshake. During a downgrade attack, the attacker modifies the handshake messages causing the client and server to establish a connection using a weaker form of encryption. The attacker can then break the weaker form of encryption based upon known vulnerabilities.
- The TCP connection close is used to indicate the end of a session, which an attacker can easily forge.

These issues led to a quick release of SSL 3.0 [6] in 1996.

In 1999, TLS 1.0 [13] was released which provided upgrades and security fixes to the SSL 3.0 specification. The cipher suites in TLS 1.0 are stronger than the cipher suites in SSL 3.0. The cipher suites in SSL 3.0 have a weaker method for generating keys and are dependent upon the MD5 hash function that has proven collisions and is no longer secure. The MAC generation was changed to increase security. The SSL 3.0 MACs are hash-based whereas the TLS 1.0 MACs are generated using both the MD5 and SHA-1 hash functions and enhanced with the use of a key for encryption. The use of two hash functions was intended to ensure that protection remained even if one of those algorithms is found to have a vulnerability. The

use of a key for encrypting the MAC ensures that only someone who knows the decryption key can validate the MAC.

In 2006, TLS 1.1 [14] was released which fixed a cipher block chaining attack. Protection against this attack was added by explicitly determining the initialization vector and changing the way padding errors were handled. The current version of the protocol is TLS 1.2 [15] and was released in 2008. The change from TLS 1.1 to TLS 1.2 changed many uses of the MD5/SHA-1 hash functions to use the SHA-256 hash function as the MD5 hash function has been proven to produce collisions and the SHA-1 hash function has a proven mathematical weakness in its algorithm. Although, the SHA-256 hash function is similar to the SHA-1 hash function it is considered more secure than SHA-1.

Issues

The TLS 1.2 specification has one known vulnerability. It is vulnerable to a man in the middle attack during the renegotiation procedure. An attacker can start two TLS connections, one with a server and one with a client. The attacker will then try to merge the two connections together by tricking the server into establishing a renegotiation connection. The merged connection will use the information selected by the attacker during the initial connection, but will use the client's information to complete the connection. This vulnerability allows the attacker to place them self between the client and server. A fix for this vulnerability is specified in RFC 5746 [17], but is not fixed in all implementations. A simple way to defend against this attack is for a server to disallow renegotiations. The next section details the criticisms of the TLS/SSL protocol.

The criticism of certificate authorities centers on the revocation methods, the business model supporting the certificate authority and the optional client authentication. A Certifi-

cate Revocation List can be large which causes bandwidth and distribution problems when they are used. Alternatively, the use of the Online Certificate Status Protocol suffers from a lack of historical status. The Online Certificate Status Protocol usually does not provide a status for older certificates. The largest flaw is that there is no way to revoke a root certificate. A root certificate is a self-signed certificate generated by a certificate authority and is used to verify the plethora of digital certificates that the certificate authority has generated. The private key used to sign these root certificates is a closely guarded secret and the certificate authority cannot revoke the certificate when the private key is compromised. The root certificates for the largest certificate authorities usually come pre-installed on most web browsers and the front door will be wide open when a root certificate is compromised. It is foolishness to think that these root certificates will never be compromised, as a server will never be perfectly secure.

Comodo, one of the largest certificate authorities, was successfully attacked on March 15, 2011. [9] The attacker was able to generate nine fraudulent certificates for domains such as mail.google.com, login.yahoo.com, login.skype.com and others. Comodo claims that the attacker did not gain access to the private key that signs the root certificate, but the incident proves that certificate authorities will never be completely secure against attackers.

Another large certificate authority, DigiNotar, was successfully attacked on June 17, 2011. Fox-IT performed a security audit of the compromised servers and published a report detailing how the attack was possible. [22] All of the certificate servers were members of the same domain and there was no separation between critical components. This allowed the attacker to use a single administrator account to compromise all of the servers. The administrator password was weak and susceptible to a brute force attack. The software on the servers was out of date and no anti-malware or anti-virus software was in use.

The DigiNotar attacker generated 531 fraudulent certificates using common names such as *.google.com, *.microsoft.com, *.thawte.com, and *.mozilla.org. The initial attack occurred on June 17, 2011 and the Fox-IT investigation was started about two and a half months later on August 30, 2011. Some of the generated certificates had serial numbers that were not captured in the certificate authorities system records and the DigiNotar OSCP-responder was set to indicate 'good' when presented with a certificate whose serial number was unknown. These two issues combined to allow the certificates with unknown serial numbers to be validated by the DigiNotar OSCP-responder. The DigiNotar OSCP-responder now indicates 'revoked' when a certificate with an unknown serial number is presented.

These two attacks have shown that it is impossible to create a completely secure system. The certificate authorities cannot provide the kind of trust that they advertise and no recourse exists for anyone who places their trust in these certificates. Both of these attacks were handled by the certificate authority through revocation of the fake digital certificates. The response time in both situations was not quick enough to stop all damage from occurring and probably never will be. There will be a time when the root certificate is compromised and there will be no clear way to address this issue.

Certificate authorities also suffer from problems found in their business model because of their commercial nature. They abuse the expiration date of certificates to charge an extension fee as opposed to limiting the expiration based upon key strength. They deny almost all warranties or guarantees to the user when what they sell is a guarantee of authenticity. Certificate authorities decide how a certificate is issued and each certificate authority uses a different set of requirements.

A certificate authority wants to assign both authenticity and trust when a certificate is issued to a paying customer. The certificate authority would have people believe that

a certificate is only issued once the subject has proven their intent and dominion of the server. The certificate authority requires the owner to fulfill several requirements to establish the intent of the subject. The amount and kind of requirements fulfilled before issuing a certificate depends on the certificate authority and ranges from only validating the name on the domain registration to validating the official tax status of a company. None of these requirements can actually determine the intent of the subject and merely act as a deterrent to the undetermined. Coupled with this faulty trust, an attacker only needs a certificate from the weakest certificate authority installed in a web browser to gain the user's trust.

It is a misconception that digital certificates convey a notion of trust. Digital certificates authenticate that the server has proven ownership of the listed public key to the certificate authority. Authenticating that the claimed public key is correct and that the associated private key is known is the best that can be achieved by an entity authentication protocol. Trust must be provided outside of the authentication protocol and handled at a separate level because of its inherent complexity.

Verifying the authenticity of the client party is an optional part of the TLS protocol. The client must send their digital certificate to the server when it is requested. When the client's digital certificate is a "voucher" certificate the server will need to have a root certificate that can be used to verify the authenticity of the client's digital certificate. The client's digital certificate is generated before this protocol is attempted and in the case of a "voucher" certificate would be generated by the system administrator of the organization that is being accessed. This process requires setup and distribution of digital certificates by a central authority. In a global environment, like the World Wide Web, it is uncommon for a service to require client authentication in the case of a web browser client.

Protocol Details

This section will detail the TLS/SSL protocol as it is an excellent example of performing authentication and is a good protocol to review in hopes of gaining insight into what should and what should not be done during authentication. Before beginning the protocol, it is assumed that the server has generated a public key and has been issued a digital certificate from a certificate authority. The TLS protocol handshake is the core of the TLS protocol. The diagram in Figure 2.2 shows the complete TLS handshake.

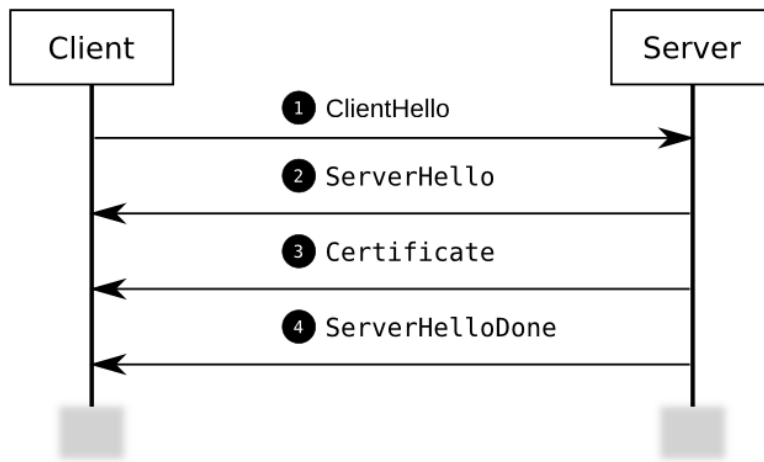

Figure 2.1: The start of the TLS handshake.

Figure 2.1 presents the beginning of the TLS handshake where the client and server negotiate the handshake parameters. The client starts the handshake after establishing an insecure connection to the server by sending the ClientHello message in step 1. The ClientHello message includes a list of supported ciphers and hash functions that will be used to establish the secure channel. The server responds with a ServerHello message and a Certificate message in steps 2 and 3. The ServerHello message includes its choice of the strongest supported cipher and hash function. The Certificate message contains the server's digital certificate. The server sends a ServerHelloDone message in step 4 to indicate that the handshake negotiation

is complete.

The client is responsible for authenticating the digital certificate received from the server. A root certificate is used to verify the signature attached to the digital certificate. If the signature is verified then the certificate's subject will be compared against the information used to establish the insecure connection. The client will continue the TLS handshake when the signature is verified and the subject matches the connection information.

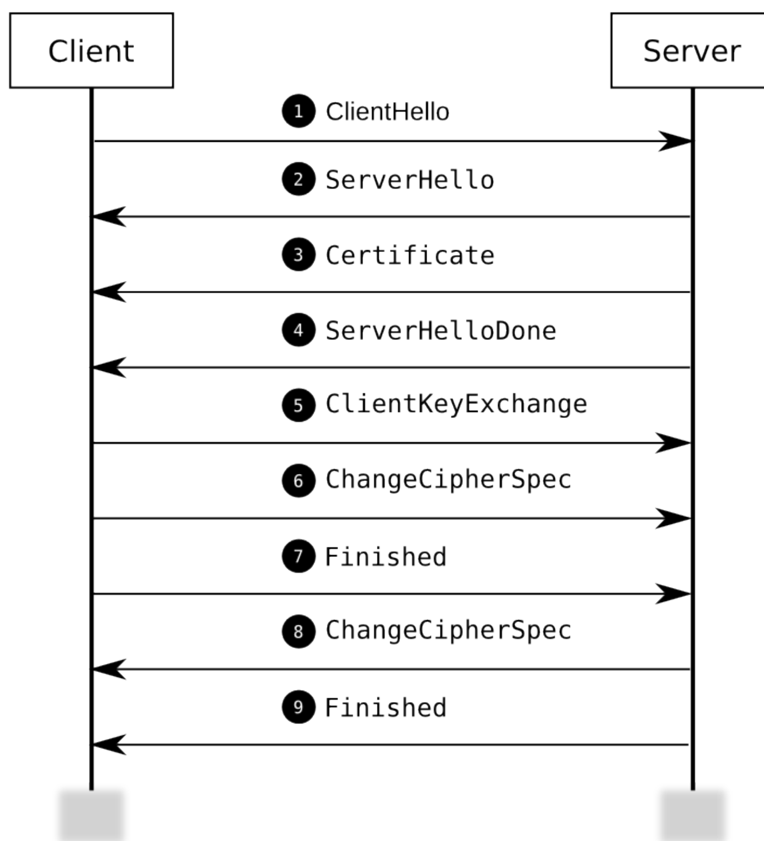

Figure 2.2: The TLS handshake without client authentication.

The remainder of the protocol is depicted in Figure 2.2. The client sends the ClientKeyExchange message, the ChangeCipherSpec message, and the Finished message in steps 5 through 7. The ClientKeyExchange message contains the client generated cryptographic

information determined by the agreed upon cipher and hash function. This information is used to form the shared secret that is the basis of the secure channel. The ChangeCipherSpec message indicates that all messages after this message will be encrypted. Finally, the Finished message indicates the end of the client side of the protocol and is encrypted and tagged with a MAC.

The server must decrypt the client's Finished message and verify the MAC that is generated using all of the previous handshake messages. Decrypting the message correctly proves that the client has generated the same secret key and a correct MAC proves that none of the information was modified during the protocol. The server responds with a ChangeCipherSpec message and a Finished message in steps 8 and 9. The ChangeCipherSpec message indicates that all messages after this message will be encrypted. The Finished message indicates the end of the server side of the protocol and is encrypted and tagged with a MAC. The client must decrypt and verify the MAC before completing the handshake. Any further communication is encrypted for secrecy and tagged with a MAC for data integrity.

Additionally, part of the TLS handshake allows the server to authenticate the client. This addition to the TLS handshake is optional and the authenticity of the client is established in a similar manner as the server authenticity. The client authenticated TLS handshake is depicted in Figure 2.3. The server sends a CertificateRequest message to request a digital certificate from the client in step 3B. The client must respond with a Certificate message and a CertificateVerify message in steps 4B and 5B. The Certificate message contains the client's digital certificate and the CertificateVerify message contains a digital signature over all previous handshake messages. The CertificateVerify message is used to prove that the client has access to the private key associated with the public key specified in the Certificate message.

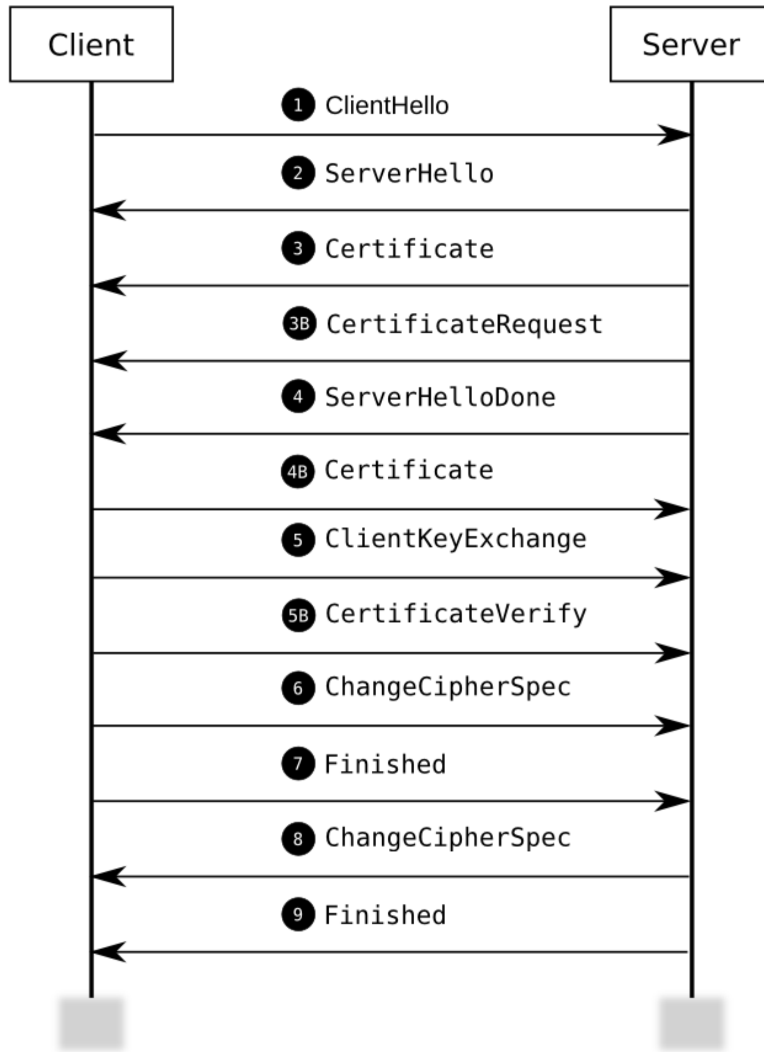

Figure 2.3: The TLS handshake with client authentication. Steps with a 'B' represent an optional step that is not present in the simple TLS handshake depicted in Figure 2.2

The protocol deals with packet tampering, eavesdropping and replay attacks while the handshake is occurring. Information passed from the server to the client is public, but is digitally signed with the server's private key to protect against packet tampering. The sensitive information passed from the client to the server is encrypted with the server's public key to protect against eavesdropping. A nonce, or number used once, is sent with the ClientHello and ServerHello messages to protect against replay attacks. The nonce values will be different for each occurrence of the protocol, which ensures that the MAC value in a Finished message will also be different.

Table 2.2 provides a quick look at the characteristics of the TLS protocol. We will be using these characteristics to provide a level comparison across the different protocols discussed in this paper. The major issues in the TLS protocol are the use of certificate authorities, the lack of revocation for root certificates, and implicitly proving the authenticity of a party. Certificate authorities introduce a centralized dependency that can be costly in a peer-to-peer environment and their abuse of the implementation hinders valuable security measures. The lack of revocation for root certificates is a large hole in the revocation and invalidates the entire use of digital certificates. Finally, the authenticity of the server is not explicitly verified and relies upon fully establishing a secure communication channel prior to knowing whether the entity is fully authenticated.

Public Key based Authentication	Yes
Server Authentication	Yes
Explicit Server Authentication	No
Client Authentication	Optional
Explicit Client Authentication	Yes
Certificate Revocation	Yes
Root Certificate Revocation	No
Decentralized	No
Privacy	Yes
Data Integrity	Yes

Table 2.2: The different characteristics of the TLS protocol.

2.3 Self-certifying pathnames

The Self-certifying File System, SFS, provides secure access to files stored on a remote machine. The security provided by SFS stops an attacker from gaining access to sensitive information being transferred between two parties. The protection is provided by authentication, privacy, and data integrity. Authentication validates the identity of the server to the client. Privacy prevents an external party from understanding the sensitive information. Data integrity guarantees the correctness of the information across the channel.

The identity of the SFS server is authenticated with a host id and a self-certifying pathname. The host id is a unique identifier which each SFS server generates using their public key. A self-certifying pathname provides a decentralized means to fully authenticate an SFS server to an SFS client. Each SFS server generates a public and private key pair. The public key combined with other attributes is used to generate the SFS server's host id. The network address, the remote path of the file and the host id are combined to form the self-certifying pathname. Finally, the SFS key negotiation protocol uses a self-certifying pathname to connect to and verify the authenticity of an SFS server. This ensures that the SFS client is speaking to the correct SFS server.

The host id is a cryptographic encoding of the SFS server's network address and public key. The network address and public key are used as input to a hash function. The output of the hash function is the host id, which is a unique value that can only be replicated when the same input is used. Any party with access to the public key of an SFS server can form the SFS server's host id and validate that it is the same as a known host id. The host id provides a decentralized method of proving the association of a public key with an SFS server.

```
/sfs/ics.uci.edu:p46jb493dkyfewepp6ykr7qq3i8hcusz/crest.txt
     Location            Host ID                  Remote Path
```

Figure 2.4: A Self-certifying pathname in SFS (Adapted from Figure 3-1, page 21 of [12].)

A self-certifying pathname, as depicted in Figure 2.4, is composed of a location, host id, and a path to the remote file. The location is a network address or DNS hostname that is used to create an insecure connection to the SFS server. The host id is used to authenticate the claimed identity of the SFS server. The path to the remote file indicates the file that should be fetched from the SFS server.

During the SFS key negotiation protocol, the SFS client forms an insecure connection to the SFS server and requests the SFS server's public key. The returned public key is used to generate the SFS server's host id, which is compared against the known host id in the self-certifying pathname. The SFS server has returned the correct public key if the host id's match. The SFS server proves ownership of the correlating private key by completing the SFS key negotiation protocol and forming a secure communication channel.

As with digital certificates, it is possible for the private key of an SFS server to be compromised. A revocation certificate is issued and distributed when an SFS server's private key is compromised. The SFS revocation system provides a way to block access to compromised SFS servers with valid revocation certificates. The SFS revocation system is the last piece of a complete identity authentication system and is discussed in Section 2.4.

Privacy in SFS is guaranteed by the encryption of sensitive information. Public key cryptography is used to encrypt sensitive information transferred during the SFS key negotiation protocol. A shared secret key and a symmetric key algorithm are used to encrypt sensitive

information after the SFS key negotiation protocol has completed. Encryption ensures that a third party will not be able to understand the SFS client's requests or the contents of the file that is returned by the SFS server.

Data integrity is guaranteed by a message authentication code. The message authentication code is generated for each message sent between the SFS client and SFS server. If information is modified between the SFS client and the SFS server, the message authentication code will be incorrect and the entire message will be ignored. For example, an attacker can modify a message to cause the SFS server to delete a file, but verification of the message authentication code will ensure that the SFS server detects the modification and does not perform the requested operation. The rest of this section will discuss the history of SFS, the issues with respect to authentication, the details of the self-certifying pathname, and the SFS key negotiation protocol.

History

SFS version 0.5 was released alongside of David Maziéres' Ph.D. dissertation in June of 2000. [12] SFS version 0.7.2 was released in July of 2002 and was the only other release that we were able to find. There are minor differences between these two versions with respect to the host id algorithm. These changes were an effort to increase the strength of SFS against cryptanalysis and not because of a security flaw.

The SFS documentation does not list any security concerns with respect to host ids, self-certifying pathnames or the SFS key negotiation protocol. The SFS documentation lists many security concerns including vulnerabilities with a global file system, vulnerabilities relying upon NFS, and vulnerabilities with the implementation itself. These issues will not be discussed because they do not effect self-certifying pathname authentication, which is our

focus. The single concern that has been found is that SFS and self-certifying pathnames suffer from the use of out-dated security functions. SFS uses the SHA-1 hash function that is known to have security weaknesses.

SFS is a remote file system and design decisions were made which reflect that domain. These decisions translate to concerns when attempting to extract SFS technology into a new domain. The first concern is that the syntax of a self-certifying pathname is not viable in the web domain and must be modified. Secondly, fully authenticating the identity of an SFS server with the SFS key negotiation protocol is a consequence of forming a secure communication channel and is not explicitly checked. Lastly, the authentication is one sided where only the SFS server is authenticated to the SFS client. The following section will present the details of a self-certifying pathname and the SFS key negotiation protocol.

Protocol Details

Self-certifying pathnames and the SFS key negotiation protocol provide an excellent example of how to perform authentication without reliance upon an external third party. In the SFS system, both the client and server generate a public and private key pair that is used for authentication and secrecy. The SFS server's public key is a long term key and is only changed by the administrator of the server. The SFS client's public key is only used for privacy and is renewed by default every hour. Renewing the SFS client's public key every hour increases the security of the system.

The SFS server's public key is combined with the location to form the host id. The host id is a unique identifier that can only be generated by using the correct public key and location associated with the SFS server. When an SFS client knows an SFS server's self-certifying pathname it can fully authenticate that the SFS server it connected to is the correct SFS

server by using the host id included in the self-certifying pathname.

The host id specified in the self-certifying pathname is easily formed by any party with access to a cryptography library. In SFS version 0.5, host ids were produced with the formula:

$$HostID = \text{SHA-1}\,(\,\textit{``HostID'', Location, PublicKey, ``HostID'', Location, PublicKey})$$

The location and public key of the SFS server was appended to the "HostID" symbol, duplicated and passed through a SHA-1 hash function. The crucial component of forming the host id is the public key owned by the server. If two SFS servers owned the same public key then the difference in location will cause a different host id to be generated, but all security will be lost because either server could masquerade as the other. Doubling the input to the SHA-1 hash function does not change the collision resistance of the host id value and was done to increase the strength against cryptanalysis. The SHA-1 hash function was the best hash function available when SFS was released, but should no longer be used because it is known to produce collisions.

In SFS version 0.7.2, the host id algorithm was changed to:

$$HostID = \text{SHA-1}\,(\text{SHA-1}\,(\,\textit{``HostID'', PublicKey})\,,\,\textit{``HostID'', PublicKey})$$

The location was removed from the formula and the input is no longer doubled. The public key is appended to the "HostID" symbol and passed through a SHA-1 hash function. Then the input is appended to the first SHA-1 hash output and passed through the SHA-1 hash function again. This version increased the computational complexity by using two SHA-1 hash functions in the formula. The "HostID" symbol and location values from version 0.5 act as constants and do not increase security. In version 0.7.2 only the "HostID" symbol

was used and the location value was removed. The change from doubling the SHA-1 input to hashing the input before doubling is another attempt to increase the strength against cryptanalysis. The value produced by both of these versions yields the host id in a byte format and must be converted to its string format before it can be used in the self-certifying pathname.

The compact byte format of the host id is produced by the host id algorithm. The compact byte format is first converted to the expanded byte format and then converted to the string format as seen in Figure 2.4. The compact byte format of the host id is 20 bytes in length while both the expanded byte format and the string format are 32 bytes in length. To convert from the compact byte format to the expanded byte format every five bits are expanded into an eight bit byte where the upper three bits are padded with zeros. In the expanded byte format each byte will represent one of 32 possible values ranging from 0 to 31. The expanded byte format is converted to the string format by mapping the 32 possible values to correlating ASCII values defined in the set {2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, G, H, I, J, K, M, N, P, Q, R, S, T, U, V, W, X, Y, Z}. The ASCII values '0', 'O', '1', and 'L' are discarded to minimize confusion in the ASCII representation of the host id. The string format of the host id can be converted back into the compact byte format by performing the above steps in reverse.

The SFS key negotiation protocol is initiated when a user attempts to access a file on an SFS server as depicted in Figure 2.5. In step 1, the client establishes an insecure connection to the server specified by the location in the self-certifying pathname and requests the server's public key. At this point in the protocol the server is completely untrusted by the client. The client will receive the server's public key in step 2 and validate that the public key produces the same host id that is specified in the self-certifying pathname. The client will generate two random key halves if the public key is validated. In step 3, the client's random

key halves are encrypted using the server's public key and sent to the server along with the client's public key. The server will generate its own random key halves and encrypt them using the client's public key. In step 4, the server's encrypted random key halves are sent to the client. Finally, both the client and server decrypt the information that was exchanged using their respective private keys and generate the same shared secret. The shared secret is used to ensure privacy and security for the remainder of the connection.

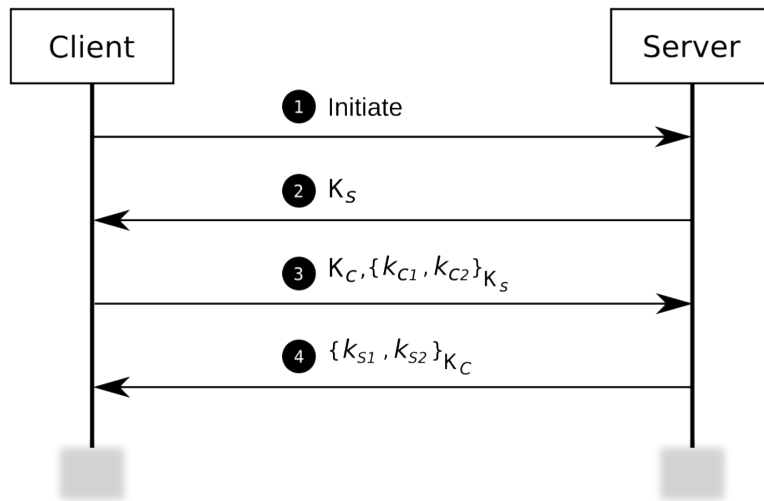

Figure 2.5: The flow of events during the SFS key negotiation protocol (Adapted from Figure 4-1, page 34 of [12].)

The shared secret is generated once the SFS key negotiation protocol has completed and the SFS client and SFS server have traded random key halves. The shared secret is actually composed of two different keys where each key handles encrypting or decrypting a certain direction for the channel. The key termed "kcs" handles encryption for the client and decryption for the server and the key termed "ksc" handles encryption for the server and decryption for the client. The following formulas are used to compute the shared key after

the random key halves have been decrypted.

$$kcs = \text{SHA-1}\,(\,\text{“}KCS\text{”}, K_s, K_{s1}, K_c, K_{c1})$$

$$ksc = \text{SHA-1}\,(\,\text{“}KSC\text{”}, K_s, K_{s2}, K_c, K_{c2})$$

The “kcs” key is generated by performing a SHA-1 hash function on the “KCS” symbol, the SFS server’s public key, the SFS server’s first key half, the SFS client’s public key, and the SFS client’s first key half. These shared secrets are used to encrypt and decrypt all information passed between the SFS client and SFS server after the SFS key negotiation protocol is complete. These keys are guaranteed to be known to only the SFS client and SFS server because the key halves were encrypted by public keys prior to transmission. The only way these shared secrets would be known by another party is if a private key was compromised and if that is the case then the authenticity of the SFS server cannot be trusted either.

The SFS key negotiation protocol serves two purposes: to validate the authenticity of the SFS server to the SFS client and to establish a secure communication channel between the client and server. The full authenticity of the SFS server is determined by two actions. The first action compares the host id generated with the broadcasted public key against the host id specified in the self-certifying pathname. This proves that the SFS server contacted claims to own the expected public key. The second action is to have the SFS server prove ownership of the private key that is correlated to the public key. The ownership of the private key is not explicitly checked and is a side effect of establishing the secure communication channel.

The full authenticity of the SFS server relies upon the SFS server proving that it has ownership over the private key associated with the broadcasted public key. In the SFS key negotiation protocol the SFS server must decrypt the random key halves and form a shared

secret. The random key halves generated by the client are encrypted using the SFS server's public key and can only be decrypted correctly with the associated private key. The shared secret will not be formed correctly if the SFS server does not hold the private key associated with the broadcasted public key. The shared secret will be different for both parties if the SFS server does not use the correct private key and any message sent to the client will be indecipherable. The SFS client will know that the SFS server does not hold the correct private key if the messages are indecipherable and will drop the connection to the SFS server because it could not prove its authenticity.

The SFS server failing to prove ownership of the correct private key is depicted in Figure 2.6. Step 1 encompasses the beginning of the SFS key negotiation protocol as depicted in Figure 2.5 and includes trading encrypted key halves and generating the shared secret key from the traded key halves. In step 2, the SFS server will send a message in response to a request from the SFS client. The message will be encrypted using the SFS server's shared secret. In step 3, the SFS client will decrypt the received message using its shared secret. In this example, the SFS server does not own the correct private key and the shared secrets generated by either side are different. The message decrypted by the SFS client is indecipherable and in step 4 the SFS client will break the connection and stop communication with the SFS server.

Table 2.3 provides the characteristics of self-certifying pathnames and the SFS key negotiation protocol. Self-certifying pathnames and the SFS key negotiation protocol provides a means for the SFS client to fully authenticate the identity of the SFS server that it is connecting to. The authenticity of the SFS server is verified without contacting an external third party or the use of digital certificates. The authenticity of the SFS client is not verified to the SFS server. The SFS server only cares about which SFS user it is communicating with and not which SFS client the user is communicating from. The authentication is not

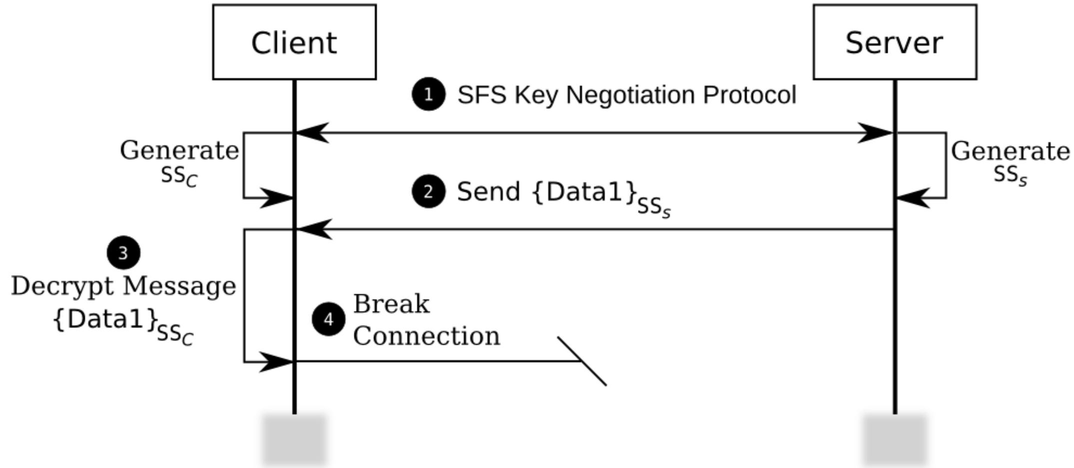

Figure 2.6: The events that transpire when the SFS server is not authenticated.

explicitly performed and is a side effect of creating the secure communication channel. This protocol has a small, compact footprint and is meant to authenticate the SFS server to the SFS client and establish a secure communication channel in a very efficient manner.

Public Key based Authentication	Yes
Server Authentication	Yes
Explicit Server Authentication	No
Client Authentication	No
Explicit Client Authentication	N/A
Certificate Revocation	Yes*
Root Certificate Revocation	N/A
Decentralized	Yes
Privacy	Yes
Data Integrity	Yes

Table 2.3: The different characteristics of Self-certifying pathnames and the SFS key negotiation protocol. *Discussed in Section 2.4

2.3.1 Self-certifying URLs

Self-certifying pathnames are used to authenticate SFS servers without reliance upon external information in the file system domain. The SFS-HTTP system [24] was the first

http://www.uci.edu:p46jb493dkyfewepp6ykr7qq3i8hcusz/index.html

Location Host ID Remote Path

Figure 2.7: A self-certifying URL in the SFS-HTTP system (Adapted from Figure 1, page 2 of [24].)

attempt to translate the SFS system and self-certifying pathnames into the HTTP domain. The SFS-HTTP system provides security and authentication for the web by leveraging the SFS libraries and self-certifying pathnames. The SFS-HTTP system allowed users to access HTTP resources from a web server securely by tunneling the communication through modified versions of the SFS client and the SFS server. The SFS client interfaced with the web browser by acting as a web proxy while the SFS server interfaced with the web server by acting as a web client. This section will explain the differences between a self-certifying pathname and a self-certifying URL, briefly explain the SFS-HTTP system, and address the issues with the self-certifying URL syntax.

The SFS-HTTP system was possible because self-certifying pathnames were easily converted to self-certifying URLs. A self-certifying URL, depicted in Figure 2.7, provides the same core mechanics as a self-certifying pathname but with a syntactical representation that allows them to be used in the HTTP domain. The location translated directly into the domain element of a URL and is still either a network address or DNS hostname. The host id is placed in the port element position of the URL and is still a cryptographic hash of the server's public key and location.

The SFS-HTTP system was designed to securely serve HTTP resources in a similar manner to how SFS securely served remote files. The use case of the SFS-HTTP system would start with the user's agent forwarding a request for a self-certifying URL to the SFS-HTTP client. The SFS-HTTP server is contacted based upon the location element of the given self-

certifying URL. The SFS key negotiation protocol is used to authenticate the SFS-HTTP server and create a secure channel. The request for the HTTP resource is forwarded to the SFS-HTTP server. The SFS-HTTP server will forward the request to the web server that it is configured to service. The HTTP response is returned to the SFS-HTTP server. The SFS-HTTP server will return the response to the SFS-HTTP client over the secure connection. Finally, the HTTP response will be returned to the user's agent.

There are issues with the syntactical representation of self-certifying URLs seen in the SFS-HTTP system. The host id is placed in direct conflict with the port element of the URL syntax. This is a bad position for the host id as there is no way to specify a different port in the self-certifying URL. This was not an issue for the SFS-HTTP system as the port used to connect to an SFS-HTTP server is explicitly defined in the SFS-HTTP client's configuration.

The SFS-HTTP system succesfully converted self-certifying pathnames into self-certifying URLs. The self-certifying URL presented does not fully conform to the URL syntax definition, but it was successfully used as a resource locater in the HTTP domain. Additionally, the SFS-HTTP system leveraged the libraries built for the SFS system and made use of the SFS key negotiation protocol to establish a secure connection and authenticate the SFS-HTTP server to the SFS-HTTP client. From this system we know that self-certifying URL's are a feasible way of identifying resources, authenticating entities, and building a secure communication channel using the SFS key negotiation protocol in the web domain.

2.4 Revocation

The SFS revocation system blocks access to an SFS server when it is deemed unsafe. It provides a highly configurable blacklist that allows the SFS client to determine which self-certifying pathnames the user should be blocked from accessing. How an SFS server is deemed unsafe is not always obvious because of the new and unforeseen ways in which attackers attempt to gain access to sensitive information. The two ways to block an SFS server are host id blocking and key revocation. Host id blocking is used to block an SFS server for any reason at any time. Key revocation is used to block an SFS server because the SFS server's private key has been compromised.

Host id blocking is used to block access to an SFS server. It is possible that a malicious SFS server has a valid host id that is used to authenticate and successfully perform the SFS key negotiation protocol. This server should not be accessed even though the self-certifying pathname is usable. How the SFS server is deemed malicious is out of scope for the SFS revocation system, but it is important that there is a way to block access to an SFS server for any reason. When host id blocking is used, it will only apply to the user who blocked it and will not be applied to all users on the same system.

Key revocation is used to block access to an SFS server when a private key has been compromised. The authenticity provided by the host id can no longer be trusted and more than one party can successfully pass the SFS key negotiation protocol using the same public key when a private key is compromised. This is a breach in security and all trust for the public key should be removed. The SFS server must generate a new public and private key pair along with a new host id and self-certifying pathname. Key revocation will block access to the SFS server for all users on the same system and is only done when a revocation certificate is available.

A revocation certificate is a self-authenticating certificate that indicates that a private key is compromised. To revoke a public key used to form a self-certifying pathname the owner of the private key must generate a revocation certificate. A revocation certificate is composed of the following fields

$$\{\text{“PathRevoke”}, Location, K, \text{NULL}\}_{K^{-1}}$$

The “PathRevoke” symbol is a constant, the Location is the same location in the self-certifying pathname, and K is the public key used to form the self-certifying pathname that is being revoked. A digital signature is created with the private key and attached to the revocation certificate. The revocation certificate is authenticated without external input because digital signature can be verified using the public key contained within the revocation certificate. Verifying the digital signature proves that the revocation certificate was created by someone with access to the private key. The revocation certificate is a decentralized way to publish the fact that a self-certifying pathname has been compromised.

The SFS revocation system is a necessary part of the authenticity provided by self-certifying pathnames. The SFS key negotiation protocol authenticates an SFS server when a safe self-certifying pathname is used. The SFS revocation system ensures that the self-certifying pathname is safe. A safe self-certifying pathname is defined by the rules that the SFS revocation system is configured with. The rest of this section will present the history, issues, and details of the SFS revocation system.

There were no changes made to the SFS revocation system between SFS version 0.5 and SFS version 0.7.2. There is one security issue with the SFS revocation system that cannot be avoided. The SFS revocation system executes an external process to determine the revocation status of a self-certifying pathname. If the external process is malicious then the results of

the revocation system cannot be trusted. This is an extreme case where the attacker has compromised the integrity of the local system. This issue is an inherent risk that comes with the design of the SFS revocation system. The SFS client assumes that anyone with the correct user privileges may configure the SFS revocation system and this issue must be addressed at the operating system level by an administrator.

Details

The SFS revocation system is a configurable blacklist using revocation programs. A revocation program is an extension point in the SFS revocation system. The revocation program contains an external process that is executed to determine the revocation status of a self-certifying pathname. The external process is how the user grows the functionality of the revocation system and provides a large amount of flexibility. Key revocation is indicated when a revocation program returns an authenticated revocation certificate to the revocation system. Host id blocking is indicated when the block attribute of the revocation program is enabled.

A revocation program is defined as a tuple with four elements:

$$\{block, filter, exclude, program[args]\}$$

The block flag is a boolean value, the filter and exclude values are PERL style regular expressions, and the program is an externally executable process. The filter and exclude regular expressions are matched against the self-certifying pathname. The revocation program will be applied when the self-certifying pathname matches the filter value and fails to match the exclude value. The program element of a revocation program will be executed when a revocation program is applied to a self-certifying pathname. The self-certifying pathname

will be revoked when the program completes execution without exception and returns a revocation certificate. The block flag of the revocation program will be checked when the external process completes execution without exception, but does not return a revocation certificate. Host id blocking is indicated when the block flag is enabled for the revocation program.

The revocation program's external process provides an extension point into the SFS revocation system. For example, a simple revocation certificate distributor responds to HTTP requests with a revocation certificate embedded in the response. The revocation certificate distributor structures their URL's as such: "http://www.uci.org/revocationcertificates/{HostID}" where the "{HostID}" symbol is replaced with the string representation of the host id for the self-certifying pathname in question. An URL can be constructed using the host id of a self-certifying pathname and a request for the HTTP resource would return either a revocation certificate or an HTTP 404 response. It is a simple task to write a BASH script that constructs the URL for a given self-certifying pathname, requests the resource, extracts the revocation certificate, and returns it to the SFS revocation system. The SFS revocation system provides an extensible and powerful framework in which the administrator is responsible for harnessing the power of revocation.

The SFS revocation system is configured with revocation programs at any time. A common installation would be to setup the SFS client as a service on the local machine. The SFS client would be configured at startup with a common set of revocation programs. At any time a user of the SFS client may use the sfskey utility to register a new revocation program with the SFS client. The sfskey utility is a command line process and the use of a few simple flags will add a new revocation program to the system. For example, the following command

will register a new revocation program:

sfskey revokeprog -b -f “uci/ics/.*” -e “uci/ics/trusted/.*” /bin/query-revocation-distributor.sh

The ’b’ flag indicates that host id blocking will be applied if no revocation certificate is found. The ’f’ and ’e’ flags define the filter and exclude regular expressions respectively and the final argument is the executable program that will be run when a self-certifying pathname is applied to the revocation program.

The SFS revocation service is run every time the user attempts to access a file on a new SFS server. The revocation service will perform the steps depicted in Figure 2.8 for all registered revocation programs when the revocation service is run. First, the revocation service will determine if the self-certifying pathname applies to the revocation program. The filter regular expression must match the self-certifying pathname and the exclude regular expression must fail to match the self-certifying pathname. If the revocation program is not applicable then the next revocation program will be attempted. When the revocation program is applicable, the executable program contained in the revocation program will be run and the host id of the self-certifying pathname will be passed as the last argument to the executable program.

When executing an external process the revocation system is only concerned with the output and the exit code of the process. The executable process is considered to have run successfully when it completes execution and returns an exit code of zero. The revocation system will only accept a revocation certificate written to standard output as the returned result for the executable process. The authenticity of the revocation certificate will be verified if the executable program runs successfully and returns a revocation certificate as the result. If the revocation certificate is authenticated the revocation system will complete and return

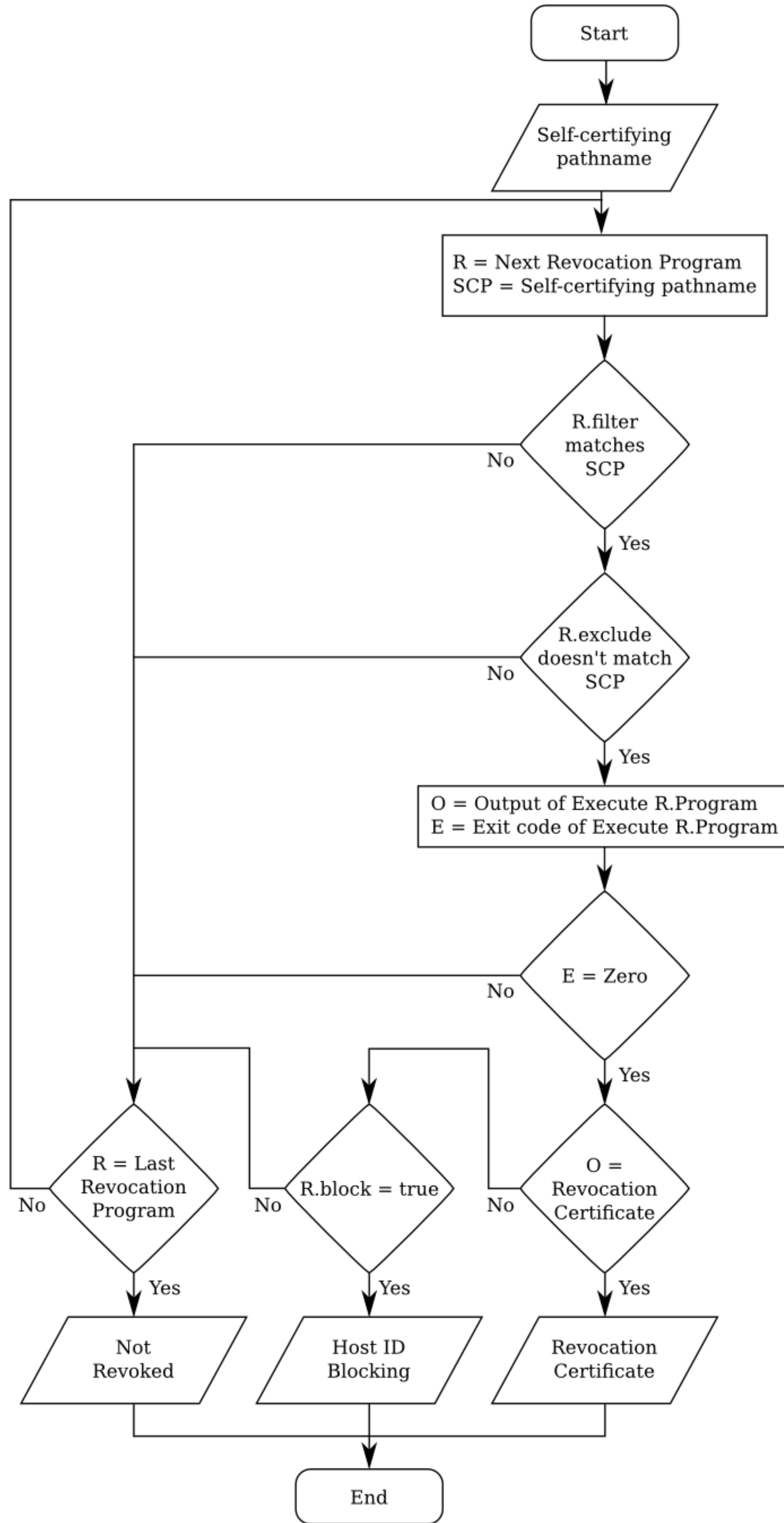

Figure 2.8: A flowchart depicting the SFS revocation system.

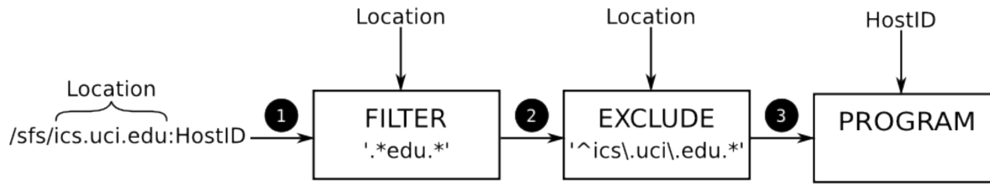

Figure 2.9: An example of a revocation program failing to be applied to a self-certifying pathname.

the revocation certificate to the SFS client.

The state of the block flag is checked when the executable program runs successfully and does not return a revocation certificate. The revocation system will complete and indicate that host id blocking is enabled when the block flag of the revocation program is enabled. If the block flag is not enabled then the next applicable revocation program will be attempted. The state of the blocked flag will not be checked when the revocation program fails to successfully run and the next applicable revocation program will be attempted. Finally, the revocation system will indicate that the self-certifying pathname is not revoked when all applicable revocation programs have been run and neither a revocation certificate nor host id blocking is found. The SFS client will begin the SFS key negotiation protocol with the SFS server indicated in the self-certifying pathname.

Figure 2.9 shows the process of how the revocation service determines whether a revocation program is applicable to a self-certifying pathname. Step 1 indicates that the location successfully matched the filter regular expression. Step 2 indicates that the location successfully matched the exclude regular expression. The revocation program will not be applied to this self-certifying pathname, in step three, because the exclude regular expression successfully matched the location. The revocation system would continue to the next revocation program to determine if it is applicable to the self-certifying pathname.

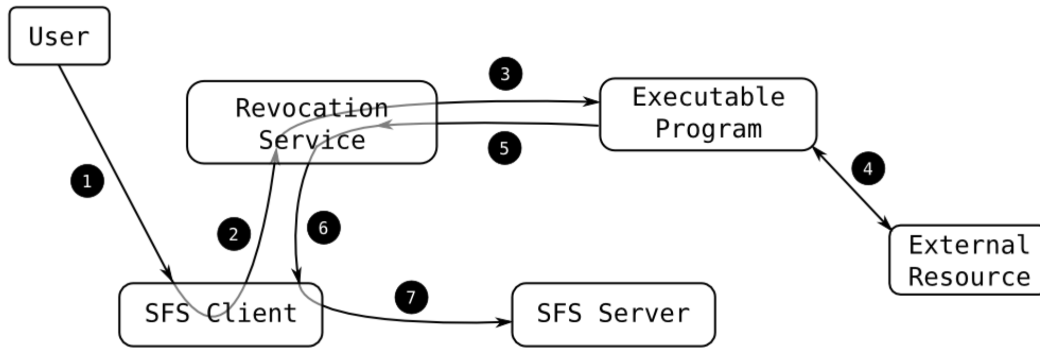

Figure 2.10: A revocation program accessing an external resource to perform revocation.

The control flow depicted in Figure 2.10 shows what happens when an SFS user successfully accesses a remote SFS fileserver. In step 1, the user attempts to access a self-certifying pathname under the local "/sfs" directory. In step 2, the SFS client intercepts the request and runs the revocation service on the self-certifying pathname. The revocation service runs each applicable revocation program for the self-certifying pathname. In step 3, the revocation service finds an applicable revocation program and runs the executable process. In step 4, the executable process communicates with an external resource in an attempt to determine if the self-certifying pathname has an associated revocation certificate. No revocation certificate is found and in step 5 the executable process completes successfully without writing a revocation certificate to standard output.

The block flag of the revocation program is checked because the executable process completed successfully without returning a revocation certificate. The block flag is not enabled and the revocation service will continue to find all other applicable revocation programs. In step 6, the revocation service tells the SFS client that the self-certifying pathname is not revoked because no revocation certificates or an enabled block flag was found. Finally, in step 7, the SFS client will initiate the SFS key negotiation protocol with the SFS server specified by the self-certifying pathname.

The SFS revocation system provides a highly configurable and highly extensible blacklist for SFS servers. It is a crucial component of the authentication provided by self-certifying pathnames and the SFS key negotiation protocol. The self-authenticating revocation certificate handles revocation of compromised keys in a decentralized manner and is an elegant solution for a problem that is present in any system based on public key cryptography. Revocation certificates, host id blocking, and the SFS revocation system provide a flexible, decentralized approach to assigning trust to SFS servers.

2.5 User Authentication

User authentication in a CREST island will be dealt with by the CREST developer. The CREST framework must provide user level capabilities such that the developers of a system can limit the scope of a user's influence and protect the resources that are being consumed. How a user is authenticated and assigned capabilities is determined by the CREST developer. Most developers will not write their own user authentication system and it is crucial for the growth of the CREST framework that CREST developers be capable of leveraging existing user authentication libraries.

The leading user authentication protocols focus on solving the single sign-on problem. In the web domain, each service has traditionally had each user establish an identifier owned by that service. This practice has led to user's having to maintain a user name and password with each service that they utilize. Many users have solved this problem by using the same or a similar user name at each service as well as using the same password with each service. This practice leads to problems with identity management and password reuse. Identity management is a thorn in the side of all web users and password reuse is a common problem that decreases the strength of the password. If the password is compromised at any service

then the user's identity is compromised at multiple services. It is up to the user to reset their password at each service in an attempt to re-establish the security of their identity.

A single sign-on protocol addresses these issues by providing a way for users to create a single identifier that can be used with multiple services. In a single sign-on protocol the user will be able to prove their ownership of an identifier to a service without having to provide credentials, passwords or other sensitive information to that service. The user is only required to create a single identifier with a trusted entity and may use that identifier with any service that supports the protocol and trusts the entity that owns the users identifier.

The following sections discuss two of the leading user authentication protocols that attempt to solve the single sign-on problem: OpenID and Shibboleth. It is important to gain an understanding of the current technologies used in user authentication because these will be the authentication libraries that developers will want to use in the future. Understanding these protocols was crucial to determining if the CREST framework is a feasible framework in which to utilize existing user authentication libraries.

2.5.1 OpenID

The OpenID protocol solves the single sign-on problem in the HTTP domain and is an open standard. OpenID defines how a user can authenticate their identity to a service in a decentralized manner. The identity is owned by the identity provider and the service is provided by the relying party. The relying party does not need to maintain the user's password at their site. The user will be able to use the same identifier across multiple relying parties. This is a boon for both the user and the relying party because the user will no longer need to remember an identifier and credentials for every site they wish to use and the relying party will no longer need to store sensitive information.

The OpenID specification solves the single sign-on problem by defining a set of messages that are passed between the Identity Provider and the Relying Party. These messages do not indicate how the user is authenticated to the Identity Provider, but how the Identity Provider can securely inform the Relying Party that the user is or is not authenticated. To begin the user will provide their OpenID identifier to the Relying Party. The Relying Party will ask the Identity Provider to authenticate the user by redirecting the user to the Identity Provider with an authentication assertion message. The Identity Provider will authenticate the user and return the user to the Relying Party with a positive assertion or a negative assertion. There is a large amount of effort expended to make this protocol seamless to the user and secure for the user, Identity Provider, and Relying Party.

The integrity of the information as it passes through the user's agent is guaranteed by an association. The OpenID protocol supports a shared association and a private association. The shared association is a shared secret that signs and verifies OpenID messages. A private association is known only to the Identity Provider. The Relying Party will send a check authentication request message over a direct connection to the Identity Provider. The Identity Provider will verify the authenticity of the parameters for the Relying Party and return a response. The rest of this section will discuss the historical background, the issues with the OpenID protocol, and the details of the OpenID protocol.

History

Brad Fitzpatrick developed the original OpenID protocol while working for SixApart in May of 2005. The original version of the protocol was called the Yadis, Yet Another Distributed Identity System, protocol and was implemented on the LiveJournal website for use with blog post comments. Around May of 2005 the openid.net domain name was given to SixApart for use with the Yadis protocol. The Yadis protocol was renamed to OpenID and the OpenID

1.0 [33] specification was released.

In May of 2006 the OpenID 1.1 specification [34] was released. The differences between the 1.0 and 1.1 specifications are minor. The 1.1 specification adds a recommendation that relying parties should append a nonce to the authentication assertion message parameters to defend against replay attacks. The change is only a recommendation and neither the 1.0 nor 1.1 specifications define how to generate or verify a nonce. The second change was the addition of an *is_valid* parameter to the check authentication response message. This message is sent from the identity provider to the relying party after the identity provider has authenticated a positive assertion for the relying party. In the 1.0 specification the indication of validation was implied through the lifetime parameter and in the 1.1 specification the validation is explicitly defined by the *is_valid* parameter.

In December 2007, the OpenID 2.0 specification [7] was released. There were significant changes made to the OpenID protocol since the 1.1 version. The 2.0 specification added the use of XRI identifiers [35] and the XRI resolution protocol [18]. XRI identifiers can be used to initiate the OpenID protocol instead of an URL and the XRI resolution protocol is used during discovery if an XRI identifier is being used. The 1.1 specification only used HTML based discovery while the 2.0 specification added the XRI resolution protocol and the Yadis protocol [26]. The Yadis name was reused for a different protocol after the first OpenID specification was published. Additionally, the 2.0 specification added a nonce value to defend against replay attacks.

The OpenID protocol grew between the 1.0 specification and the 2.0 specification with help from the Light-Weight Identity protocol developers and the XRI/i-names developers. OpenID Authentication 2.0 is the current specification of the OpenID protocol and is held by the OpenID Foundation, which is a non-profit organization. There are many members

of the OpenID community that will help shape the future of the specification including corporate members such as: David Recordon (Facebook), Eric Sachs (Google), Nataraj (Raj) Nagaratnam (IBM), Michael B. Jones (Microsoft), Andrew Nash (Paypal), Pamela Dingle (Ping Identity), Nico Popp (VeriSign), and Raj Mata (Yahoo!).

Issues

The known security issues in OpenID 2.0 are in the implementations of the specification. Nonce verification must be handled correctly by the implementation or there is the possibility that a replay attack can succeed. If an OpenID implementation does not use TLS/SSL and does not check the nonce then an OpenID assertion could be used multiple times in a replay attack against the Relying Party. This is not a hole in the protocol, but a danger to implementations not performing the protocol correctly.

Associations only prevent tampering of OpenID messages that pass through the user's agent. Implementations must be aware that the communication during discovery, establishing a shared association, and verifying an assertion with a private association are susceptible to man in the middle attacks. The specification recommends using TLS/SSL to establish connections during these unprotected phases of the protocol. The use of TLS/SSL provides a defense against man in the middle attacks, but provides no guarantee as to whether the entity should be trusted or not.

The OpenID protocol is not protected against phishing attacks. For example, a malicious Relying Party can redirect an OpenID user to a fake website that may look extremely similar to the login page that the user is expecting to be redirected to. If the user fails to notice a difference in the website or the difference in the URL that they were redirected to then they will continue to enter their user name and password as they normally do. This action

will supply the malicious Relying Party with the user's identifier and password. This is an inherent weakness in the OpenID protocol and is a real danger to the OpenID user.

Trust between any of the parties involved in the OpenID protocol is not guaranteed. It is up to the Relying Party implementation to decide which parties are trustworthy and which are not. The Relying Party must decide which Identity Provider their users may use to login and access resources. They must also decide if it is appropriate to allow users from a semi-trusted Identity Provider certain resources versus a fully trusted Identity Provider whose users have access to all of the resources. It is possible for anyone to setup an Identity Provider that does not perform authentication and always responds with a positive assertion. This type of authentication may not be acceptable to all Relying Parties.

Lastly, a major criticism of the adoption of OpenID is that many of the large organizations have become Identity Providers, but are not Relying Parties. For example, both Google and Yahoo are Identity Providers, but not Relying Parties. A user with a Google OpenID cannot login to Yahoo and access their resources. This type of situation hinders the growth of OpenID. For the user this is frustrating, as the largest organizations are still requiring that a user create and maintain multiple identifiers.

Protocol Details

The OpenID protocol involves three entities the user's agent, the relying party, and the OpenID provider. The user's agent is normally a web browser. The relying party is the entity providing a resource to the user. The OpenID provider holds the identity and information necessary to authenticate the user. The OpenID protocol is divided into three distinct phases: initialization, authentication, and verification. During the initialization phase the relying party normalizes the user-supplied identifier, discovers the user's information needed

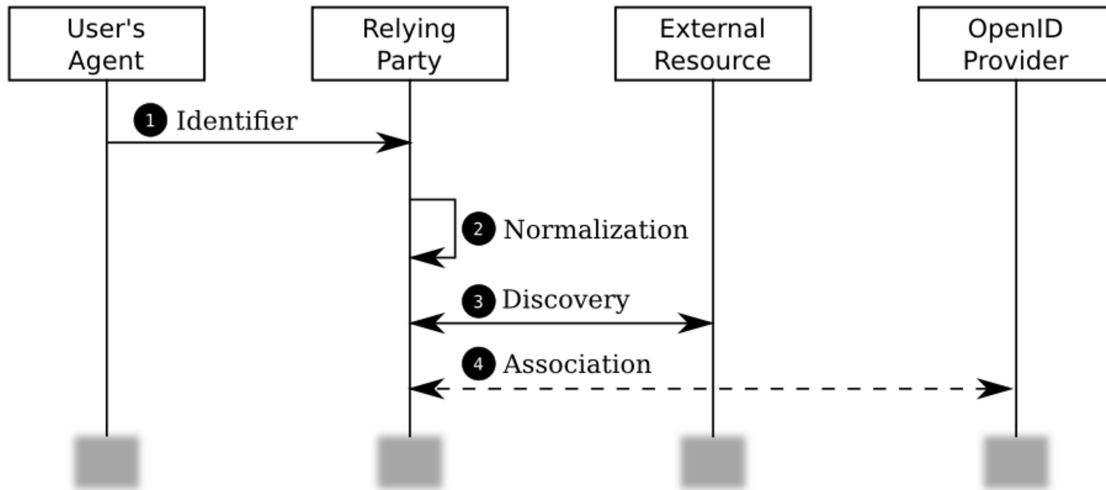

Figure 2.11: The initialization phase of the OpenID protocol (Dashed lines represent optional messages.)

to perform authentication and possibly creates an association with the OpenID provider. In the authentication phase the relying party will redirect the user to the OpenID provider with an authentication assertion where the user must authenticate with their OpenID provider. How this authentication is done is not specified by the OpenID specification. Finally, during the verification phase the user will be redirected back to the relying party with either a positive or a negative assertion that is verified.

The initialization phase determines an OpenID endpoint from the user supplied identifier and is depicted in Figure 2.11. This OpenID endpoint is sent the OpenID messages requesting that the user be authenticated. To begin the initialization phase the user must first provide the relying party with an identifier, step 1. OpenID allows users to provide relying parties with an identifier in the form of either an Extensible Resource Identifier (XRI) [35] or an URL. An XRI is an abstract identifier for a resource where the identifier has nothing in common with the physical location of the resource. The XRI specification provides the syntax definition of an XRI and defines how to locate a resource identified by an XRI. The XRI specification is a solution for unique, non-reusable identifiers in an abstract space.

OpenID has adopted the use of XRIs to help solve the issue of reusable identifiers involved with using URLs as identifiers.

The user-supplied identifier given to the Relying Party during initialization is either an OpenID provider identifier or the user's OpenID identifier. When the identifier is an OpenID provider identifier the OpenID provider will assist the user in choosing which OpenID identifier they want to share with the Relying Party. The identifier is normalized in step 2 to find the syntactically correct XRI or URL that can be used during discovery.

The user-supplied identifier is normalized by converting it into a syntactically correct XRI or URL format. If the user supplied identifier starts with the prefix "xri://" then the prefix is stripped off to obtain the XRI in its canonical form. The user-supplied identifier is treated as an XRI if the canonical form starts with an XRI Global Context Symbol. The symbols: "=", "@", and "+" are examples of an XRI Global Context Symbol defined in the XRI syntax specification. If the user-supplied identifier is not an XRI then it is treated as an URL. If the user-supplied identifier is missing the scheme element of the URL then it will be prefixed with "http://". All fragments and fragment delimiters are removed and all HTTP redirects are followed before the final URL identifier is found.

Discovery begins once the user-supplied identifier has been normalized successfully and is performed in step 3 in Figure 2.11. Discovery locates an OpenID endpoint using the normalized identifier. An OpenID endpoint is any entity that accepts and processes OpenID messages. The XRI resolution protocol [18] is used to perform discovery when the normalized identifier is an XRI. The Yadis protocol [26] is attempted first to perform discovery when the normalized identifier is an URL. HTML-Based discovery is the last method of discovery available and is attempted when the Yadis protocol is unsuccessful.

The XRI resolution protocol and the Yadis protocol yield an XRDS document. An XRDS document is an XML formatted document that contains meta-data about a resource. The XRDS document retrieved during discovery must contain an OpenID endpoint. HTML-Based discovery is attempted when an XRDS document with an OpenID endpoint is not found. HTML-Based discovery uses HTTP and HTML documents to find an OpenID endpoint. An HTTP GET request is sent to the normalized identifier. The HTTP response must yield an HTML document that contains an OpenID endpoint and the user's local identifier at the OpenID endpoint. The information must be defined in LINK elements inside of the HEAD element of the HTML document.

Creating a shared association is the last step in the initialization phase and is shown in step 4. The OpenID specification recommends that a shared association between the Relying Party and the OpenID Provider be created. If a shared association is not created than a private association will be created by the OpenID Provider during the authentication phase. An association is a secret key that is used to sign and verify OpenID messages. It is used to guarantee data integrity between the Relying Party and the OpenID Provider for messages passed through the user's agent. A shared association is established over a direct connection between the Relying Party and the OpenID Provider. The OpenID specification recommends that the secret key be exchanged over a secure channel established by using either the TLS/SSL protocol or the Diffie-Hellman key exchange. The secret key can be exchanged over an insecure channel, but this will severely weaken the data integrity guaranteed by the association.

Figure 2.11 shows the actions taken during the initialization phase of the OpenID protocol. Step 1 begins the initialization phase when the end user supplies an identifier to the relying party. The identifier is normalized during step 2 to form either an XRI or an URL. Discovery occurs in step 3 using the normalized identifier. The external resource used for discovery

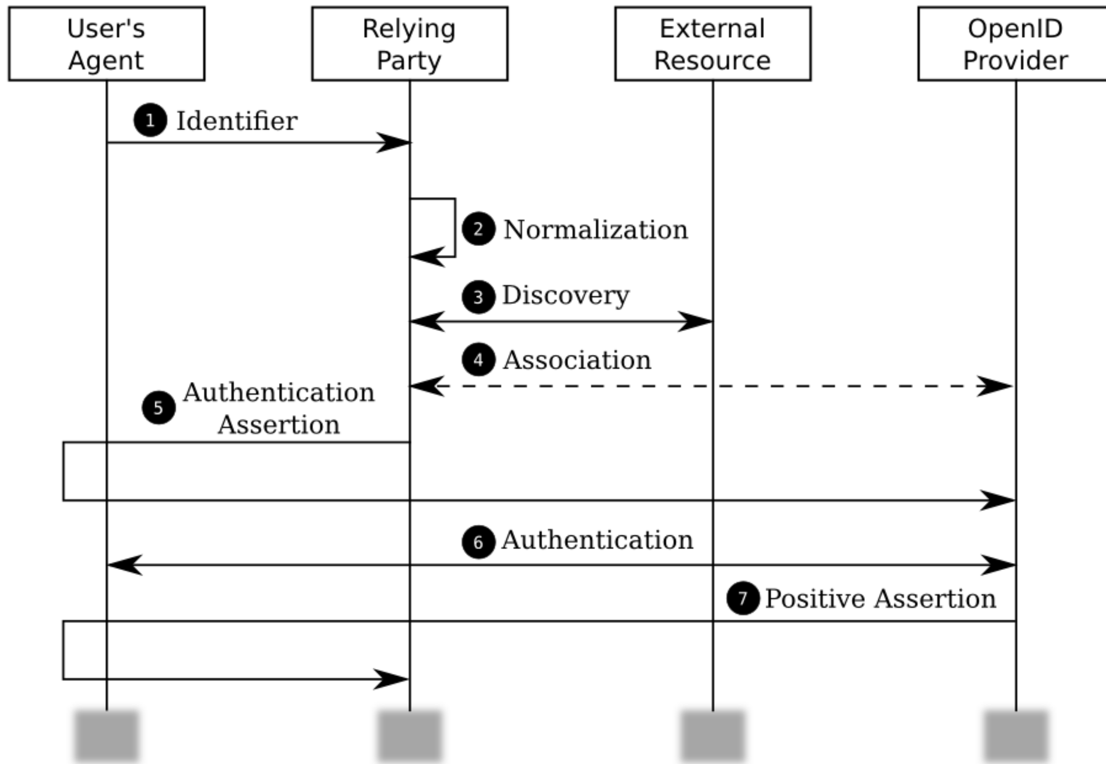

Figure 2.12: The authentication phase of the OpenID protocol (Dashed lines represent optional messages.)

could be an XRI resolution service, a third party server used to serve the XRDS document, or the OpenID Provider. A shared association could be created in step 4 and is depicted as dashed lines to indicate that it is an optional step. Finally, the initialization phase is completed and the authentication phase begins.

The authentication phase beings with an OpenID endpoint and optionally, a shared association. Using this information the Relying Party redirects the user to the OpenID endpoint with an authentication assertion message, step 5 in Figure 2.12. The user must then prove to the OpenID provider that they own the OpenID identifier. How the user is authenticated is out of scope for the OpenID specification and may include anything from inputting a user name and password to the use of a biometric fingerprint scanner. Step 6 shows the OpenID

provider authenticating the user. A private association will be created at this point if a shared association was not created during the initialization phase. In step 7, the user is redirected back to the relying party with a positive assertion to indicate that the user passed authentication or a negative assertion to indicate that the user failed authentication. The assertion is signed using the association.

Verification is the final phase of the OpenID protocol and begins once the Relying Party receives a positive assertion. The positive assertion is verified to ensure that the message came from the expected OpenID provider and that the information was not modified during the delivery. When a shared association is created during the initialization phase then it is used to verify the integrity of the data and the validity of the signature in the positive assertion. When a private association is created during the authentication phase the OpenID endpoint will be asked to validate the information sent through the user's agent.

In step 8 of Figure 2.13 the positive assertion is verified by the Relying Party. The Relying Party will not contact the OpenID endpoint when a shared association was created in step 4. The Relying Party will create a direct connection to the OpenID endpoint when a shared association was not created. A direct request is sent to the OpenID endpoint that includes all of the information in the positive assertion. The OpenID endpoint verifies the signature and returns the result to the Relying Party. Finally, the user has proven their ownership of the identifier to the OpenID Provider and the Relying Party has verified the integrity and validity of the positive assertion. The Relying Party can now allow the user access to the restricted content and provide a session id if needed for future transactions as shown in step 9.

Table 2.4 shows the different characteristics of the OpenID protocol. The OpenID protocol provides a decentralized approach to the single sign-on problem. Anyone can create an

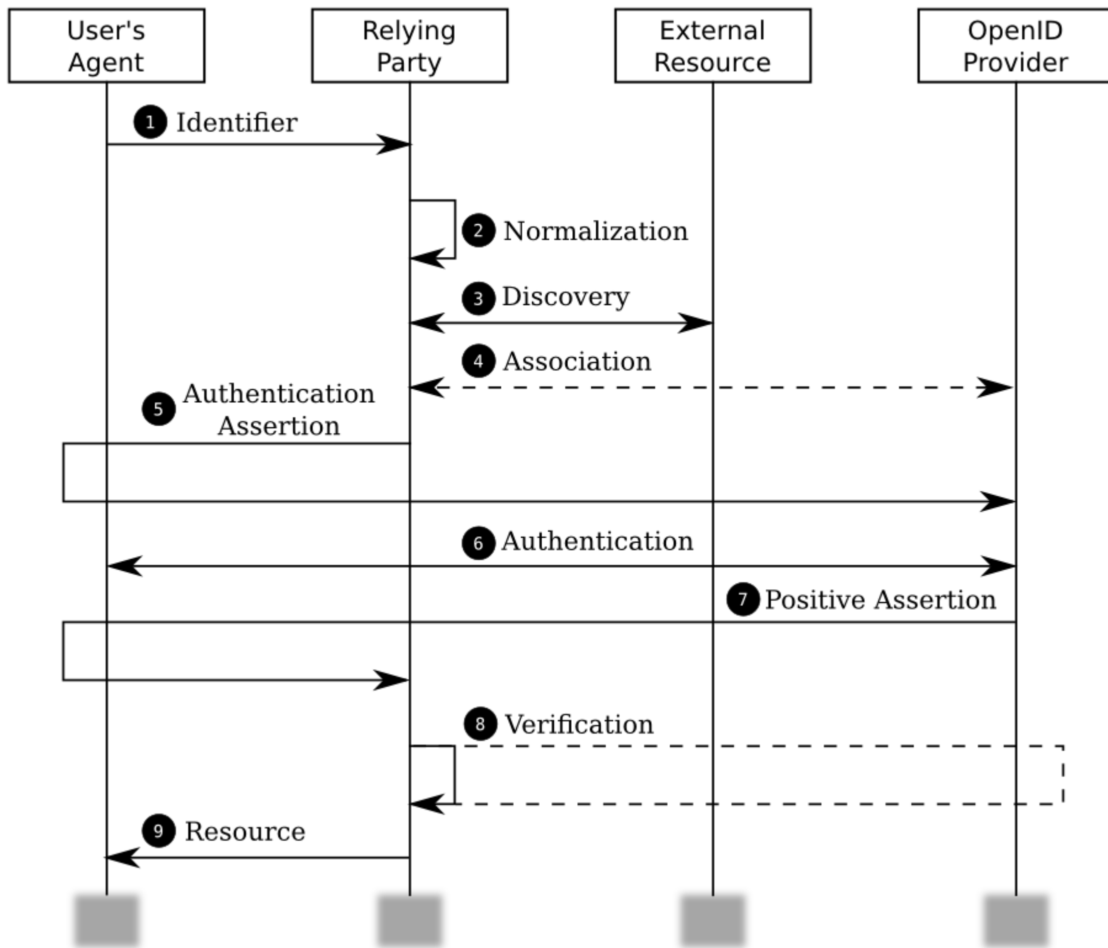

Figure 2.13: The verification phase of the OpenID protocol (Dashed lines represent optional messages.)

OpenID Provider or a Relying Party and does not need to be approved by a third party. Many different OpenID implementations provide OpenID Provider and Relying Party capabilities. It is up to the individual user to decide which OpenID Provider they wish to place their trust in and which Relying Parties they wish to make use of. The OpenID specification provides an approach for solving the single sign-on problem for the web domain.

Open Standard	Yes
Specification Style	Single Specification
Implementations	Multiple open source implementations available.
Decentralized	Yes
Trust	None

Table 2.4: The different characteristics of the OpenID protocol.

2.5.2 Shibboleth

Shibboleth enables resource sharing between parties with different authentication systems by presenting a solution to the single sign-on problem. Sharing resources between organizations is a difficult problem to solve, as each organization will have their own authentication system. Shibboleth approached this problem by designing a solution to the single sign-on problem. In Shibboleth, a group of organizations will combine to form a federation. Each organization will maintain their own authentication system and user identities. Using Shibboleth, a user from an organization may access a resource owned by another organization in the federation using their own identifier.

Shibboleth provides a federated identity to its users. In a federated identity system, a user has an identity and attributes stored with one entity and is able to use this identity across a group of other entities, the federation. The single sign-on problem is a subset of the federated identity problem as it only addresses the problem of authenticating the user across different parties. A federated identity solution solves both authenticating identity and

exchanging attributes such that the user's entire identity can be utilized across the federation. Shibboleth has built a federated identity system in which the federation is implemented as a whitelist. The rest of this section will discuss the historical background of the Shibboleth system, the known issues with the current version, and the details of the Shibboleth protocol.

History

The MACE working group, Middleware Architecture Committee for Education, released Shibboleth 1.0 in July of 2003. Shibboleth 1.0 had multiple minor releases and is an implementation of the Shibboleth web single sign-on and attribute exchange profile defined in the Shibboleth Architecture [36] working draft. The working draft is based on SAML 1.1 [28] and defines the messages exchanged during the Shibboleth single sign-on protocol. Shibboleth 1.0 and the working draft were used to define and grow the Shibboleth single sign-on profile.

The Shibboleth single sign-on profile was integrated into SAML 2.0, specifically the SAML profiles document [31], in March of 2005 as the Web Browser Single Signon Profile. Shibboleth 2.0 was released in March of 2008 and is an implementation of the SAML 2.0 Web Browser Single Signon Profile. The changes in Shibboleth 1.0 to Shibboleth 2.0 were made to increase the interoperability of the Shibboleth implementation with other SAML 2.0 implementations. Shibboleth 2.0 maintains full backwards compatibility with Shibboleth 1.3 and is an open source single sign-on solution under the Apache 2 license.

Shibboleth is susceptible to information leakage and phishing attacks. There are points in the protocol where information leakage can occur. The authentication request message is used to request authentication of a user's identity from their Identity Provider. This message has a target parameter that is set to the restricted resource that the user is attempting to

access. The target parameter is returned in the authentication response message, which allows the Service Provider to direct the user to the resource they want to access. The Identity Provider does not need to know which resource the user is attempting to access and it is recommended that the Service Provider use either a token or encryption such that the restricted resource is not passed in plain text to the Identity Provider.

Phishing attacks are a danger to all single sign-on protocols because of their distributed nature. For example, a malicious Service Provider can redirect a user to a fake website that looks like their Identity Provider's website. If the user fails to notice a difference in the website then they will inadvertently give their user name and password to the malicious Service Provider in an attempt to authenticate themselves. This provides the malicious Service Provider with the information it needs to impersonate the user. There is no known method for Shibboleth to defend against a phishing attack and it is up to the user to remain vigilant and aware of where they enter their credentials.

Protocol Details

The Security Assertion Markup Language, SAML, is a set of specifications that defines how to transfer information between business partners. [32] SAML defines an XML based syntax that all SAML messages conform to. Protocols define the content of a SAML message and the order in which the messages are exchanged between entities. Bindings define the format of the SAML messages as they are transferred across a communication channel between two entities. Bindings will often reference well-known protocols such as HTTP or SOAP. Profiles define a set of protocols and bindings that can be used to accomplish an overall goal.

The SAML Core specification [29] defines the generic syntax that all SAML messages conform to. The SAML Core specification also includes protocol definitions like the Au-

thentication Request Protocol and the Artifact Resolution Protocol used in Shibboleth. The SAML Bind specification [30] defines SAML bindings including the HTTP Redirect binding, the HTTP POST Binding, and the HTTP Artifact Binding. The Web Browser Single Signon Profile is defined in the SAML Profiles specification [31] and uses the Authentication Request Protocol, Artifact Resolution Protocol, HTTP Redirect binding, HTTP POST binding, and the HTTP Artifact binding to provide a solution to the single sign-on problem.

The Authentication Request Protocol defines messages that are used to authenticate a user to an external party. The messages defined are the Authentication Request message and the Response message. The Authentication Request message is sent when a relying party needs the user to be authenticated. The Response message is returned with an authentication statement. The authentication statement informs the relying party that the user was either successfully or unsuccessfully authenticated.

The Artifact Resolution Protocol defines a way to pass SAML message by reference. The messages defined are the ArtifactResolve message and the ArtifactResponse message. Prior to this protocol being used a SAML entity will send an artifact to a second SAML entity. The ArtifactResolve message includes that artifact and is sent to the SAML entity which generated the artifact. The artifact acts as a key that the first SAML entity uses to reference stored content. The ArtifactResponse message includes the stored content and is sent as a response to the ArtifactResolve message. This protocol is normally used to obtain messages that are passed through an intermediate party but the intermediate party has size or security constraints that do not allow the message to be passed directly through.

The bindings define how the protocol messages are transferred across a communication channel. The HTTP Redirect binding defines how SAML messages are embedded in HTTP GET requests as URL parameters. The HTTP POST binding defines how SAML messages

are embedded in an HTML form control. It is possible to create an auto-submit form using the HTTP POST binding which uses javascript to automatically press the submit button of the HTML form when loaded into the user's agent. The HTTP Artifact binding defines how to encode an artifact as a URL parameter. The receiver would use the Artifact Resolution Protocol to resolve the artifact to a SAML message.

The Web Browser Single Signon Profile uses the protocols and bindings described above to solve the single sign-on problem. There are multiple actors involved in this profile including the Principal, the User Agent, the Identity Provider, and the Service Provider. The Principal is the human user trying to gain access to a restricted resource. The User Agent is the program that the Principal is using to access the restricted resource and is usually a web browser. The Identity Provider owns the Principal's identity and attributes. The Service Provider owns a private resource that the Principal is attempting to access. The Identity Provider is responsible for authenticating the Principal at the request of the Service Provider. The following details describe the Web Browser Single Signon Profile while highlighting the Shibboleth implementation choices.

The two phases of the Web Browser Single Signon Profile are Identity Provider discovery and authentication. Figure 2.14 depicts the Identity Provider discovery phase and the beginning of the authentication phase. The Principal requests a resource owned by the Service Provider in step 1. The Service Provider will check to see if an existing security context exists. The security context in Shibboleth is a session id cookie that is cached in the User Agent. When there is no security context the Service Provider needs to determine which Identity Provider the Principal uses.

The SAML identity provider discovery profile is used to determine which Identity Provider the Authentication Request message is sent to. Shibboleth implements the discovery profile

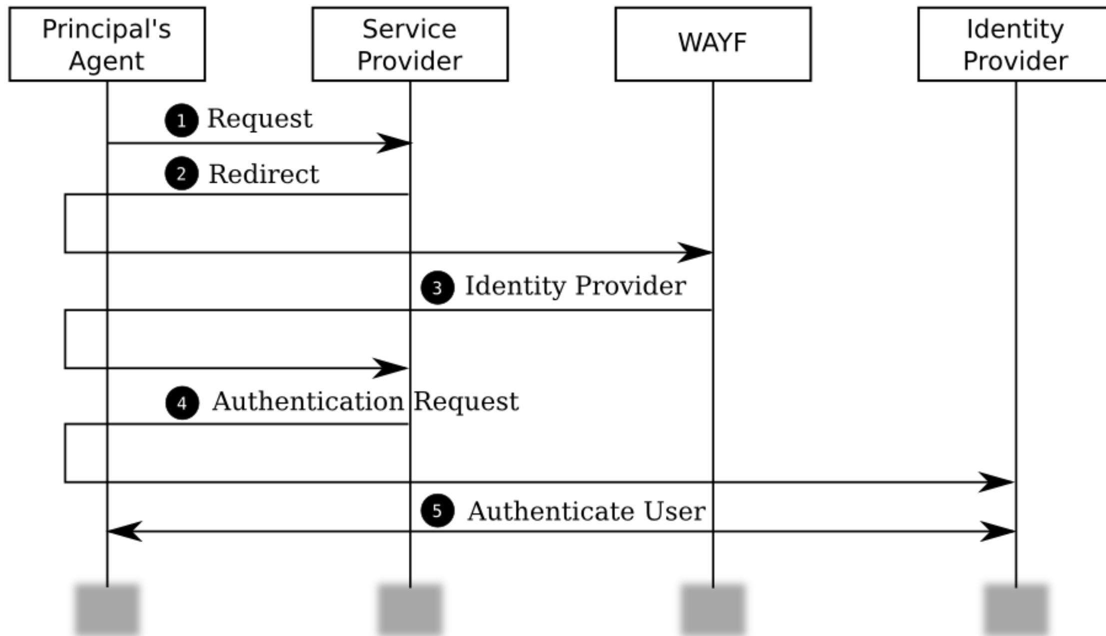

Figure 2.14: Identity Provider discovery and the beginning of authentication for the Web Browser Single Signon Profile (Adapted from Figure 1, page 15 of [31].)

as a Where Are You From, WAYF, server. The Service Provider uses an HTTP redirect to send the Principal to the WAYF server in step 2. The WAYF server presents an HTML webpage to the Principal asking them to choose which Identity Provider they belong to. The WAYF server redirects the Principal back to the Service Provider with the chosen Identity Provider in step 3. The WAYF server is a whitelist of accepted Identity Providers in the federation.

The Service Provider sends an Authentication Request to the Identity Provider to begin the authentication phase in step 4. The Authentication Request message includes a set of requested attributes about the Principal. The Authentication Request message should have a signature attached to it because it is passed through the User's Agent. The signature is used to verify the integrity of the message contents by the Identity Provider when the message is received. The Authentication Request can be sent with the HTTP Redirect binding, the

HTTP POST binding, or the HTTP Artifact binding. Shibboleth uses the HTTP POST binding to send the Authentication Request in an automatically submitted HTML control form to the Identity Provider.

The Identity Provider verifies the integrity of the Authentication Request message and attempts to authenticate the Principal in step 5. How the Identity Provider authenticates the Principal is out of scope of the SAML Authentication Request Protocol. A biometric scanning device, two-factor authentication or simply a user name and password could all be used to prove that the Principal owns the claimed identifier.

An Authentication Response message is sent from the Identity Provider to the Service Provider once the authentication is completed. The Authentication Response message contains an authentication statement that indicates success or failure and includes the set of attributes requested by the Service Provider. The Authentication Response message can be sent with the HTTP POST binding, Figure 2.15, or the HTTP Artifact binding, Figure 2.16. By default, Shibboleth will use the HTTP POST binding to send the Authentication Response in an automatically submitted HTML control form to the Service Provider.

The Service Provider verifies the integrity of the Authentication Response message and processes the attributes that were requested. Finally, the Service Provider should establish a security context for the Principal and return the restricted resource. Shibboleth does this by redirecting the Principal to the restricted resource and installing a new Shibboleth session id as a cookie in the user agent. The user agent will process the redirect and send an HTTP GET with the new Shibboleth session id. The security context will be available in this request and the restricted resource will be returned to the Principal.

Shibboleth is fully compliant with the SAML 2.0 Web Browser Single Signon Profile. The Shibboleth specific actions described above occur when the Shibboleth Identity Provider and Service Provider are used. The Shibboleth Identity Provider can interact with a different SAML 2.0 Service Provider implementation and the Shibboleth Service Provider can interact with a different SAML 2.0 Identity Provider implementation. This means that the Shibboleth implementation is fully capable of handling an Authentication Request and Authentication Response message through the HTTP Redirect binding, HTTP POST binding, or the HTTP Artifact binding.

The Web Browser Single Signon Profile depicted in Figure 2.15 shows the messages sent when the HTTP POST binding is used. To review, in step 1 the Principal attempts to access a resource held by the Service Provider. In step 2 the Service Provider redirects the Principal to the WAYF server to perform Identity Provider discovery because the Principal did not have a security context needed to access the resource. The Principal chooses their Identity Provider in step 3 and is redirected back to the Service Provider. In step 4 the Service Provider forms an Authentication Request and sends it to the Identity Provider. The Identity Provider will authenticate the Principal in step 5 and return a Response message to the Service Provider in step 6. Finally, in step 7, the Principal is allowed to access the resource if the authentication was successful.

The Web Browser Single Signon Profile depicted in Figure 2.16 shows the messages sent when the Identity Provider uses the HTTP Artifact binding. Steps 1 through 5 are exactly the same steps as seen in Figure 2.15 and will not be discussed again. After authenticating the Principal, the Identity Provider will store the authentication result and generate an artifact that is associated with the stored authentication result. In step 6 the Identity Provider will send the Service Provider the artifact using the HTTP Artifact binding. The Service Provider must use the Artifact Resolution protocol to resolve the artifact. In step

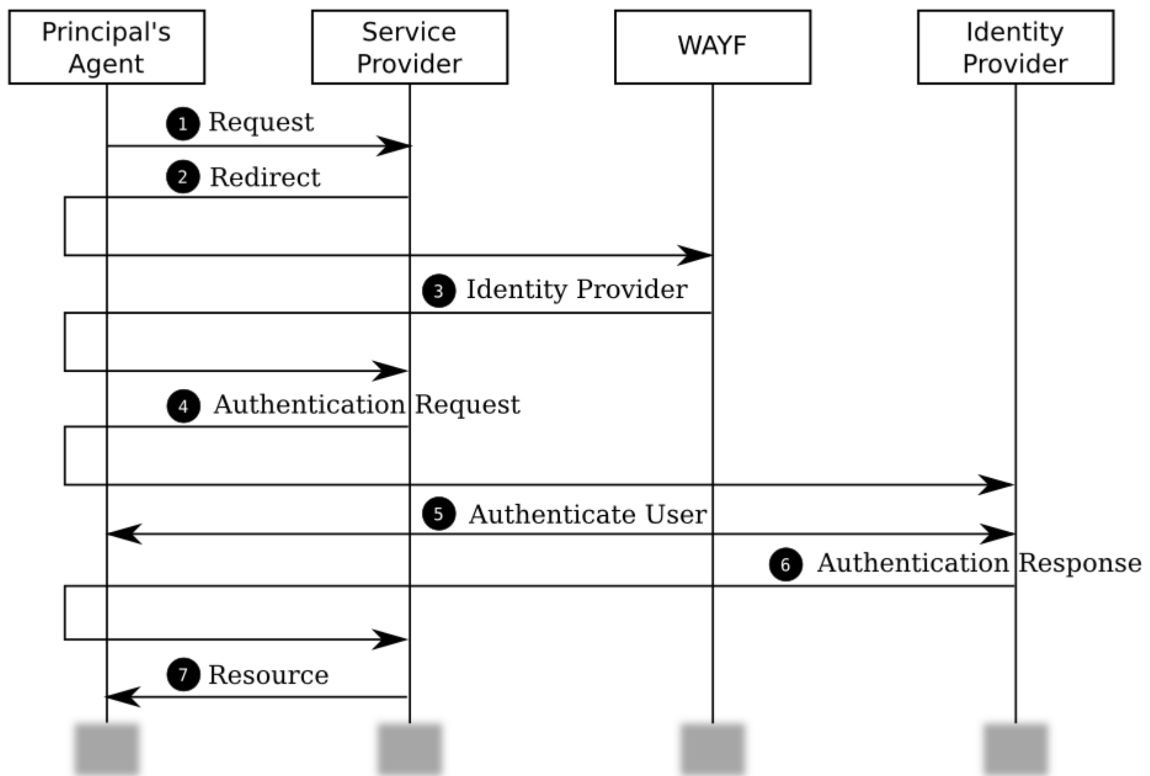

Figure 2.15: The Web Browser Single Signon Profile using the HTTP POST binding (Adapted from Figure 1, page 15 of [31].)

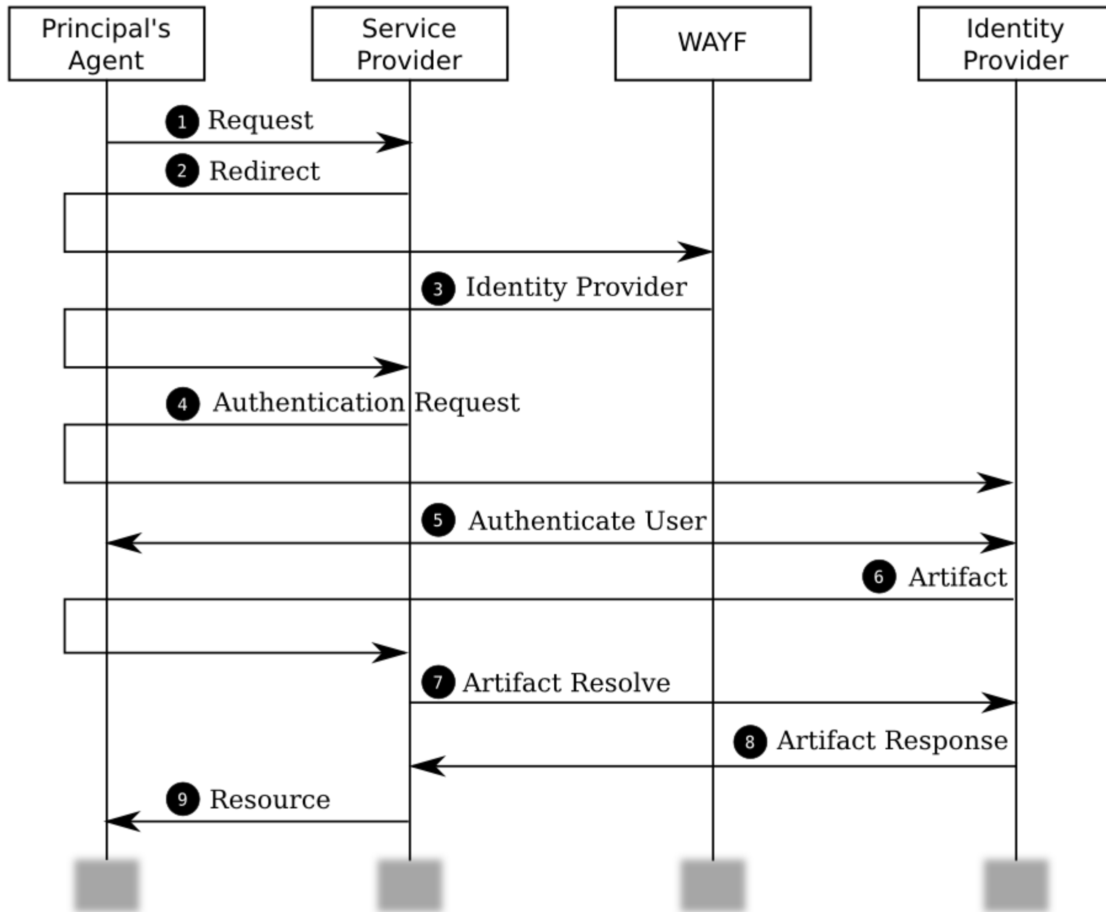

Figure 2.16: The Web Browser Single Signon Profile when the Identity Provider uses the HTTP Artifact binding (Adapted from Figure 1, page 15 of [31].)

7 the Service Provider creates a direct communication channel with the Identity Provider and sends an ArtifactResolve message. The Identity Provider matches the artifact to the stored authentication result and responds with an ArtifactResponse message in step 8. The ArtifactResponse message contains the Response message and the authentication statement. Finally, in step 9, the Principal will be allowed to access the resource if the authentication was successful.

Security is defined by the SAML message signatures and encryption. Signatures ensure data integrity and encryption ensures confidentiality and secrecy. The SAML signature

definition is based on the XML Signature specification [11] and can be used on all SAML messages to ensure data integrity. The SAML specification recommends that a signature be attached to all SAML messages that are not passed over a direct secure communications channel. For example, any message being passed through the User Agent should be signed to ensure that any change to the information by the intermediate is caught. Sensitive information in SAML protocols are encrypted, but a binding or profile can expand the use of encryption by defining their own set of encryption standards. For example, the SOAP binding defines the use of existing encryption techniques by relying upon either TLS or SOAP Message Security.

The Web Browser Single Signon Profile provides a level of trust between the Service Provider and the Identity Provider. The Identity Providers in the federation are trusted to provide authentication. The federation is a whitelist of accepted Identity Providers and is implemented as the WAYF server in Shibboleth. If the Principal does not have an identity with any Identity Providers in the federation then they cannot access the resource. The trust is determined outside of the protocol by the administrators that formed the federation. Comparatively, this type of trust is not present in the OpenID protocol.

Table 2.5 shows the different characteristics of the Shibboleth implementation. Shibboleth provides a decentralized solution to the single sign-on problem. The Shibboleth implementation provides an open source solution that anyone can use to setup both an Identity Provider and a Service Provider. The Identity Provider is a standard Java web application that can be installed in Jetty 7, Apache Tomcat or JBoss Tomcat. The Service Provider has implementations to plug into the Apache webserver or the Internet Information Services (IIS) webserver. Anyone can install and configure these systems and have a working Shibboleth federation without the need to be approved by a third party.

Open Standard	Yes
Specification Style	Multiple Modular Specifications
Implementation	Open source
Decentralized	Yes
Trust	Whitelist

Table 2.5: The different characteristics of the Shibboleth implementation.

Chapter 3

SCURL Authentication

The first line of defense in protecting sensitive information is to ensure the authenticity of the remote party. The TLS protocol and the SFS key negotiation protocol that we presented provide authentication and secure channel establishment. The focus of SCURLs and the SCURL authentication protocol will be to explicitly authenticate two CREST islands to stop the attempts of attackers as early as possible. This will minimize the resources consumed by the CREST island when being attacked.

The CREST framework will use SCURL's, the SCURL authentication protocol, and the CREST revocation system to authenticate CREST islands. The SCURL authentication protocol and the CREST revocation system are a decentralized approach to performing explicit authentication between CREST islands. The authenticity established by SCURL's and the SCURL authentication protocol is based upon public key cryptography. The CREST revocation system denies access to compromised SCURL's with revocation certificates. It is assumed that each CREST island generates a public key pair that is used to create their SCURL during configuration. This SCURL is available to the CREST framework during execution and is used during the SCURL authentication protocol.

Secure channel establishment is not part of the SCURL authentication protocol, but the CREST framework will need to establish a secure communication channel. It is a simple task to create a secure communication channel once the SCURL authentication protocol has completed. The method presented by the SFS key negotiation protocol or the DiffieHellman-Merkle key exchange protocol are excellent examples of establishing a secure communication channel over an insecure connection. The following sections will describe the design and implementation of the CREST SCURL, the SCURL authentication protocol, and the CREST revocation system.

3.1 Design

The authenticity of a CREST island is based upon the wealth of knowledge found in SFS and the SFS-HTTP system. The SCURL presented in SFS-HTTP proved that self-certifying pathnames could be leveraged in the web domain. The problem is that the SCURL seen in SFS-HTTP is not compliant with the URL syntax. Minor changes were made to the SFS-HTTP SCURL to create the CREST SCURL, which does conform to the URL syntax. Conforming to the URL syntax makes it possible to build a browser based CREST island and allows CREST developers to take advantage of the existing URL parsing libraries.

The SFS-HTTP SCURL syntax has been modified to ensure that CREST SCURLs are syntactically well-formed URLs. The host id in the SFS-HTTP SCURL is placed in the port element position and blocked the use of any port other than the SFS-HTTP server's internally defined port. The host id in the CREST SCURL is a part of the path element. The path element in a CREST SCURL must contain the word "scurl" as the first path element and the host id as the second path element. The CREST SCURL format is depicted in Figure 3.1 where the domain and port specify the network address of the remote server and

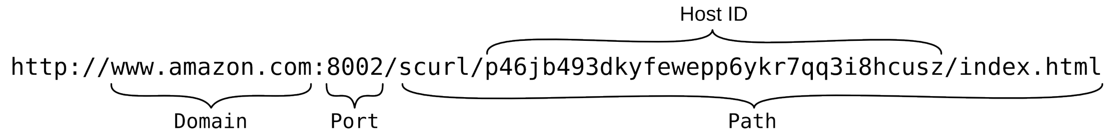

Figure 3.1: A CREST self-certifying URL.

the host id is a cryptographic hash of the server's public key, domain, and port components. The CREST SCURL is used to identify and determine the authenticity of a CREST island.

The host id in the CREST SCURL is easily formed by any party with access to a cryptography library. The formula used to produce a host id in a CREST SCURL is:

$$HostID = DIGEST\left(DIGEST\left(Location,\ Port,\ PublicKey\right), Location,\ Port,\ PublicKey\right)$$

The location and port act as constant inputs to the formula and have no bearing on security. The *DIGEST* value is the hash function specified by the SCURL itself and can change between SCURLs. To create the host id, the public key is appended to the location and port values and is passed through the *DIGEST* hash function. The location, port, and public key values are appended to the result of the first hash function and passed through the *DIGEST* hash function again to produce the final byte format of the host id. Two hash functions are used to increase the strength against cryptanalysis.

The *DIGEST* used in the host id formula is defined in the SCURL instead of using a globally defined digest for all SCURLs. This accounts for future security issues that may arise with the currently available hash functions. This formula produces the compact byte format of the host id and the method presented in Section 2.3 is used to convert between the compact byte format, expanded byte format, and string format of the host id. The number of bytes produced by a hash function can vary and it is possible for a SCURL to use a hash

function that produces more than 20 bytes of output. The formula used to convert between the compact, expanded, and string formats expects exactly 20 bytes of input data and to account for this the hash function output is trimmed to be exactly 20 bytes.

Trimming the output of the hash function is a security issue. It was realized that trimming the hash function output to be 20 bytes decreases the number of possible outputs of the hash function and increases the risk of collision. This increases the possibility that two SCURLs with different public keys will end up having the same host id. The solution to this problem is to use all of the bytes produced by the hash function and to modify the formula that converts the compact byte format into the expanded byte format.

The expanded byte conversion formula must be modified to work with a variable length byte array. To start, a five bit boundary is found by dividing the number of bits by five. All bits within the boundary will be expanded in the same way as the old formula. Each set of five bits is expanded into an eight bit byte where the upper three bits are padded with zeros. There will be between zero and four bits remaining outside of the five bit boundary. These remaining bits are expanded into an eight bit byte where the upper bits are padded with zeros. The last byte needs to indicate the number of bits used in the second to last byte. The last byte will be a value from zero to four and is used when the conversion is being reversed.

For example, the SHA-512 hash function will produce a host id with a compact byte format length of 512 bits. The five bit boundary includes the first 510 bits. They will be expanded into eight bit bytes where the upper three bits are padded with zeros. The remaining two bits are used to form the second to last byte, which will be a value from zero to three. The last byte is a value of two to indicate that two bits are used from the second to last byte when reversing the conversion. The length of the host id for the expanded byte format and

the string format is increased because of the last byte needed to reverse the conversion. The SCURL authentication protocol, presented next, uses the CREST SCURL to authenticate remote CREST islands.

3.1.1 SCURL Authentication Protocol

The SCURL authentication protocol authenticates two parties to each other over an insecure communication channel. The SFS key negotiation protocol is the inspiration for the SCURL authentication protocol, but there are major differences because of the different goals of each protocol The SFS key negotiation protocol provided one-sided, implicit authentication that was tied to the establishment of a secure communication channel. The SCURL authentication protocol provides explicit authentication for both parties and does so without relying upon the creation of a secure communication channel.

The SCURL authentication protocol relies on public key cryptography and digital signatures to explicitly authenticate a remote party. Each island issues a request for the remote island's public key and a digital signature. The public key is used to generate the remote island's host id, which proves that they are claiming ownership of the expected public key, and the digital signature is verified to prove ownership of the associated private key. The authenticity of the remote island is explicitly verified by generating the expected host id and verifying the digital signature.

Before beginning the SCURL authentication protocol, detailed in Figure 3.2, the client island and server island will generate public key pairs, create a host id from the public key, and form their SCURL. The client island establishes an insecure connection to the server island using the network address and port specified by the server island's SCURL. The client island initiates the protocol by sending a client hello message to the server island that

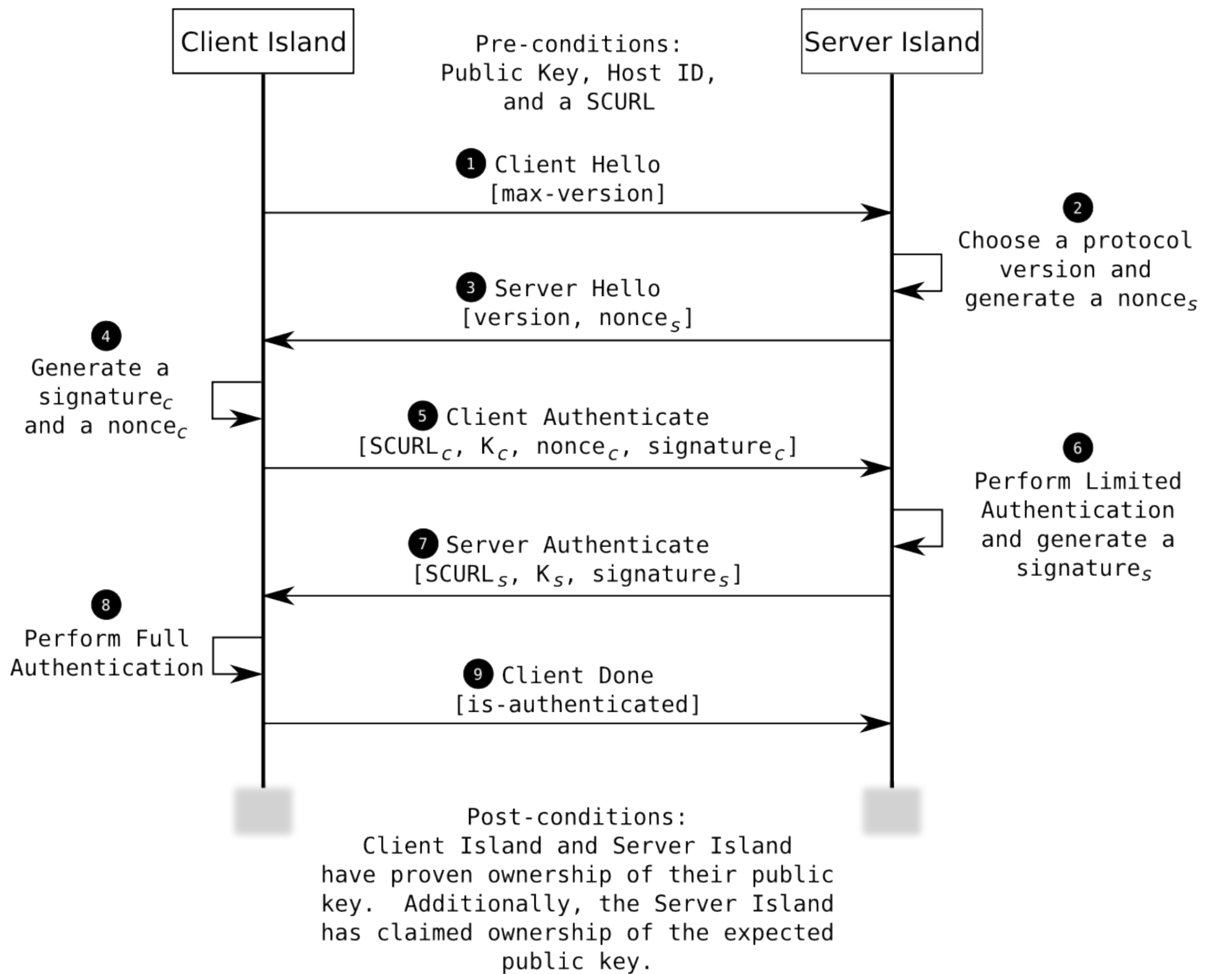

Figure 3.2: The SCURL authentication protocol.

includes the largest protocol version number that the client island supports. The server island responds to the client hello message with a server hello message that contains the chosen protocol version and the server island generated nonce.

The server hello message is an authentication challenge from the server island to the client island. The client island sends its SCURL, public key, a client island generated nonce, and a digital signature to the server island in the client authenticate message. The digital signature is created using the server island generated nonce and the client island's host id. The client authenticate message serves two purposes: it contains the information needed to authenticate the client island to the server island and it is an authentication challenge from the client island to the server island.

The server island performs limited authentication of the client island when it receives the client authenticate message. The limited authentication verifies the authenticity of the digital signature using the server island generated nonce and the received client island's public key. The limited authentication proves that the client island owns the private key associated to the claimed public key. The limited authentication cannot prove that the claimed public key is the expected public key because the server island does not know the SCURL of the client island at this time. It is up to the implementation to fully authenticate the client island's SCURL by verifying that the claimed public key is the same as the expected public key.

The server island responds to the authentication challenge from the client island when the client island's authenticity is verified. The client island generated nonce and the server island's host id is used to generate a digital signature. The server island sends its SCURL, public key, and the digital signature to the client island in the server authenticate message. The client island performs full authentication of the server island when it receives the server authenticate message. The server island's claimed public key is used to generate the server

island's host id and is compared against the known host id found in the SCURL used to create the insecure connection. The digital signature is verified using the client island generated nonce and the received server island's public key.

The full authentication of the server island proves that we have contacted an island that claims to be who we wish to communicate with and that the island has access to the correct private key. These two pieces of information guarantee that the server island is the entity identified by the SCURL used to create the insecure connection. The client done message is sent when the server island has passed authentication and indicates that the client authentication portion is complete. The connection should be closed without further communication if the authenticity of either island fails. When the protocol is succesfully completed the server island has proven ownership of expected public key and the client island has proven ownership of their claimed public key.

It is important to note that the SCURL authentication protocol can only provide full authentication of the server island. Full authentication verifies that the server island claims and proves ownership of the expected public key. The protocol provides full authentication of the server island because the client island starts the protocol with a known SCURL. The server island does not start the protocol with a known SCURL, which means that the protocol does not know the expected public key of the connecting island. It is up to the protocol implementation for the server island to complete the limited authentication provided by the protocol.

The implementation, described below, defines an extension point directly after the limited authentication is performed. This extension point is a revocation function that is called with the client island's claimed SCURL and is used to determine if the remainder of the protocol is allowed to continue for this client island. In the CREST framework this extension

point is used to tie in a higher level procedure called revocation, discussed in Section 3.1.2, which is used to complete the limited authentication of the client island before continuing the SCURL authentication protocol. The rest of this section will discuss the elements of the SCURL authentication protocol that provide protection against known attacks and the security concerns that we have identified with this design.

Replay attacks are defended against through the use of a nonce, or number used once. A replay attack occurs when an attacker records messages between two authenticating parties and uses the recorded messages in the future to trick an island into trusting them. A nonce is a value that is never repeated and in the SCURL authentication protocol will ensure that the digital signature must be generated fresh with every round. The replay of an old authenticate message is guaranteed to fail because the old signature can never be the same as the new signature. The nonce used is a combination of a timestamp and a random value. When the timestamp and random value are concatenated together the probability of a repeated nonce is extremely low.

The order of authentication was established to provide a defense against a denial of service attack. In the SCURL authentication protocol the server authenticates the client first followed by the client authenticating the server. The order of operations was established such that the server island can break the connection and stop the authentication protocol without having to generate a digital signature when a malicious client sends a client authenticate message with an invalid signature. The resources needed to generate a digital signature may not be much but anything helps when defending against a denial of service attack.

Issues

Data modification and protocol blocking are two security concerns that were identified after designing and implementing the SCURL authentication protocol. Data modification occurs when an attacker captures a message, changes the value of an element, and injects the modified message into the system. The digital signature in the protocol messages is a means to authenticate the remote island and acts as a message authentication code for all elements that are used to generate the signature. The message authentication code ensures the data integrity of the elements included in its creation. The elements that are not included in the digital signature are susceptible to modification. These elements include the version numbers in the client hello message and the server hello message and the authenticated element in the client done message. An attacker could modify these elements to make the protocol fail or cause a downgrade attack.

A solution to the data modification problem is to create the digital signature out of all the message elements previously sent in the protocol. The client island would include all of the elements in the client hello message, server hello message, and client authenticate message in the client generated digital signature. The server island would include all of the elements in the client hello message, server hello message, client authenticate message, and server authenticate message in the server generated digital signature. Finally, the client done message needs a message authentication code to ensure the data integrity of its elements.

Protocol blocking is the second security issue found during implementation of the SCURL authentication protocol. The implementation will wait forever for the next expected message in the protocol. This allows an attacker to block the island from further processing and provides an attacker a way to lock a resource that is never released. This issue is addressed with a timeout that will break out of the message read and release the connection if the

next message is not received within the desired amount of time. The timeout can be at the message level where each message must be received before the timeout is reached or at the connection level where each connection must be authenticated before the timeout is reached.

Table 3.1 provides the characteristics of the CREST SCURL and the SCURL authentication protocol. The SCURL authentication protocol provides explicit authentication between two parties over an insecure communication channel. The authenticity of the client and server is explicitly verified in a decentralized manner and is based on SCURLs and public key cryptography. The SCURL authentication protocol is a feasible approach to entity authentication in the CREST framework.

Public Key based Authentication	Yes
Server Authentication	Yes
Explicit Server Authentication	Yes
Client Authentication	Yes
Explicit Client Authentication	Yes
Certificate Revocation	N/A
Root Certificate Revocation	Yes
Decentralized	Yes
Privacy	No
Data Integrity	No

Table 3.1: The different characteristics of the SCURL authentication protocol.

3.1.2 Revocation

The CREST revocation system blocks access to a CREST island when it is deemed unsafe to communicate with. There will be malicious CREST islands trying to damage CREST based systems and the CREST revocation system will be used to block the malicious islands when they are found. The CREST revocation system is a configurable blacklist that allows the CREST developer to decide which islands are not trusted. An island is blocked because of a compromised private key or because the island has been deemed unsafe by another means.

The CREST revocation system is based on the SFS revocation system presented in Section 2.4 with minor technical changes made to provide the same functionality in the CREST framework.

The CREST revocation system is positioned at a higher level than the SCURL authentication protocol. The CREST developer has access to the revocation system and will use it as a tool in their CREST tool belt. The CREST revocation service prevents a CREST island from communicating with a remote CREST island when an authentic revocation certificate is found or when the remote island is deemed unsafe by flagging it for host id blocking. The revocation certificate is issued when the private key associated with a SCURL is compromised.

The revocation certificate in CREST is based on the revocation certificate presented in SFS. Minor changes to the revocation certificate were made to replicate the functionality using a SCURL instead of a self-certifying pathname. The host id associated with a compromised private key should not be trusted to authenticate a CREST island. Revocation certificates are used to invalidate a SCURL once the private key has been compromised. A revocation certificate contains the SCURL, a public key, and a digital signature. The SCURL identifies the host id that should not be trusted and the digital signature can be verified using the included public key to prove the authenticity of the revocation certificate.

The digital signature in a revocation certificate is formed by combining the "PathRevoke" symbol, the location, the port, and the public key of a SCURL. The following formula shows how the digital signature in a revocation certificate is formed.

$$\{\text{“PathRevoke”}, Location, Port, K\}_{K^{-1}}$$

Any party can verify the signature by extracting the location and port values from the SCURL, combining them with the public key, and compare it against the decrypted digital signature to prove that the revocation certificate was generated by a party with access to the private key. There is no need for a central authority to provide revocation certificates and there is no need for the CREST island using the certificates to trust the source of the certificate because revocation certificates are fully self-authenticated.

Host id blocking is used when the CREST island is a known attacker, but there is no revocation certificate available. Host id blocking can be enabled at any time for any island. How and why a CREST island is deemed dangerous is up to the CREST developer. The CREST revocation system only provides a way for the CREST developer to enforce their decisions. The rest of the section will discuss the details of the CREST revocation system and how it differs from the SFS revocation system.

The CREST revocation system is configured with revocation programs. A revocation program is defined as a tuple with four elements:

$$\{block, filter, exclude, function\}$$

The block flag is a boolean value, the filter and exclude values are PERL style regular expressions, and the function is a programmatically defined function. The filter and exclude regular expressions are matched against the SCURL. The revocation program is applied when the SCURL matches the filter value and fails to match the exclude value. The revocation function is executed when the revocation program applies to a SCURL.

The revocation system will revoke the SCURL when the revocation function completes execution without exception and returns an authenticated revocation certificate. The revo-

cation system will check the block flag to determine the status of host id blocking when the function completes execution without exception and does not return a revocation certificate. Host id blocking will be enabled if the block flag is set. The revocation system will perform this task for each applicable revocation program that it is configured with. The revocation system will indicate that the SCURL is not revoked when all applicable revocation programs have been applied and neither a revocation certificate nor host id blocking is found. The CREST revocation system performs the exact same process as the SFS revocation system to determine when a SCURL applies to a revocation program and returns the same status when it completes.

Modifications

There are three differences between the CREST revocation system and the SFS revocation system. The first difference is the change made to the revocation program. The second difference is how the revocation system is configured. The third difference is when the revocation system is run.

The revocation program was changed to use a programmatic function instead of an externally executable process. This change was made because the CREST framework wants to encourage the development of CREST based applications whereas SFS is a complete application. The target audience of the CREST framework is developers and the target audience of SFS is system administrators. A CREST developer could build a revocation mechanism in their CREST application that mimics the SFS functionality by creating a function that executes an external process.

The CREST revocation system is configured programmatically through the CREST framework. There is no external program that is run similar to the sfskey utility used to configure

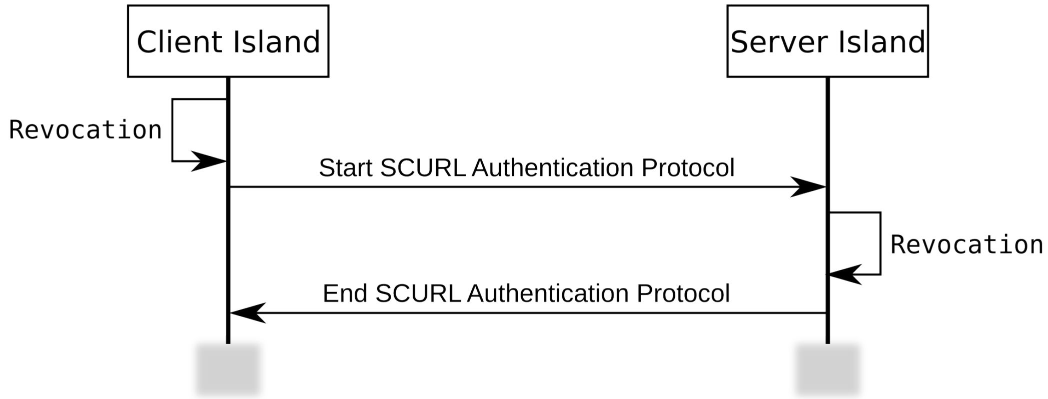

Figure 3.3: The time at which the CREST revocation service is run.

the SFS revocation system. This change was made for the same reasons as the move to a revocation function. The CREST framework is targeted at developers and not a system administrator and it is up to the CREST developer to build a configuration mechanism into their application.

The SFS revocation system is run only on the client side prior to establishing an insecure connection to the SFS server. The CREST revocation system must be executed for both the client island and the server island to completely authenticate both islands. The time at which the revocation service is executed is depicted in Figure 3.3 and is different for the client island and the server island. The client island runs the revocation service directly before an insecure connection is established to the server island. If the server island's SCURL is revoked then a socket connection will not be attempted. The server island runs the revocation service after a socket connection has been established and limited authentication has been provided by the SCURL authentication protocol. If the client island's SCURL is revoked then the socket connection will be closed without further communication and the connection will not be accepted.

The CREST revocation service is how CREST developers can complete the limited authentication provided by the SCURL authentication protocol for the client island. The CREST revocation service is a configurable blacklist through which SCURLs can be flagged for host id blocking and compromised CREST islands can be revoked. This revocation service can allow only known SCURLs access to the CREST island or it can allow any SCURL access to the CREST island depending upon its configuration. There are many ways that a CREST developer may wish to determine if a CREST island should be trusted and the CREST revocation service is a powerful tool to configure, manage, and manipulate a SCURL blacklist.

3.2 Implementation

The CREST framework is built in the Scheme programming language. The CREST SCURL, SCURL authentication protocol and the CREST revocation system are also built in the Scheme programming language to ensure compatibility with the CREST framework. The OpenSSL cryptography library [4] provided by the mzcrypto [16] Racket module is used to perform all of the cryptography in this implementation. This section focuses on the implementation details of the CREST SCURL, the SCURL authentication protocol, and the CREST revocation system.

The SCURL structure contains an URL, a host id, a digest type, a public key type, and a public key. The URL is an URL data structure that contains the textual representation of the SCURL. The host id is stored in its compact byte format as a byte array. The digest type is an mzcrypto digest object which indicates the digest used when creating the host id. The public key type is an mzcrypto pkey identifier that indicates the type of public key that is associated with this SCURL. The public key is an mzcrypto pkey object that contains

the public key. The pkey object contains just the public key when the SCURL represents a remote CREST island. The pkey object contains both the public key and the private key when the SCURL represents the local CREST island.

The SCURL structure always contains a valid URL and host id component, but the digest type, public key type, and public key do not need to be present. Given the string representation of a SCURL the URL component can be parsed and the compact byte format of the host id can be formed from the string format of the host id. This is all the information needed to verify the authenticity of a remote CREST island prior to contacting them. The SCURL authentication protocol transfers the digest type, public key type, and public key that are needed to authenticate the remote CREST island. The digest type and public key type elements are present to allow changes to the SCURL structure when advances in cryptography are made. The SCURL structure is not tied to a specific digest or public key and can be different from one CREST island to the next. In this implementation, the SHA-256 digest is used as the digest type and the RSA public key algorithm is used as the public key type.

When forming the host id the location is represented as an array of UTF-8 bytes, the port is represented as a two byte value, and the public key is represented as an array of bytes. These byte arrays are used as the location, port, and public key inputs to the host id formula described in Section 3.1. The compact byte format of the host id is stored in the SCURL host id component and is always available when a SCURL object exists. The rest of this section will discuss the details of the SCURL authentication protocol and the CREST revocation system implementation.

3.2.1 SCURL Authentication Protocol

The CREST framework handles marshalling Scheme primitives to and from their byte representation when transferred over a wire. The SCURL authentication protocol messages are defined as lists of Scheme primitives because there is no need to explicitly define their byte representation. The five messages used in the SCURL authentication protocol are the client hello message, the server hello message, the client authenticate message, the server authenticate message, and the client done message. The message formats are described in Tables A.1 through A.5 in Appendix A. The SCURL authentication module defines two functions that provide the client island functionality and the server island functionality.

The client island function requires the SCURL of the client island, the SCURL of the server island, an input port, and an output port. The SCURL of the client island must have a public key that contains both the public key component and the private key component. The SCURL of the server island must contain the URL representation of the SCURL. Protocol messages are read from the input port and written to the output port. The messages are read and written as Scheme lists containing primitive values. All data structures are marshaled from their programmatic definition to a primitive value when they are written to the output port and marshaled back to their programmatic definition when read from the input port.

The client island initiates the protocol by writing the client hello message to the output port. The client hello message indicates the maximum protocol version number that this client can handle. The client island waits for the server hello message that is read from the input port. The server hello message contains the chosen version number, a timestamp, and a nonce. The client authentication fails if the chosen version is not recognized or the timestamp is outside of a configurable delta. The client island proceeds by creating a digital signature using the client island's host id, the timestamp, and the nonce that was retrieved

from the server hello message. The client authenticate message is written to the output port and includes the client island's SCURL, a new timestamp, a new nonce, and the digital signature.

The client island waits for the server authenticate message to arrive on the input port once the client authenticate message is sent. The server authenticate message will only arrive if the server island successfully authenticates the client island with the information in the client authenticate message. The server authenticate message contains the server island's SCURL and a digital signature. The client island generates the server island's host id from the information in the server authenticate message and compares it against the known server island's host id. The digital signature is verified to prove that the server island's owns the private key associated with its host id. The server island is fully authenticated when both the host id matches and the digital signature is verified. The client island function writes the client done message to the output port and returns the server island's broadcasted SCURL. Any value other than a SCURL returned by the client authenticate function is an indication of failure.

The server island function requires the SCURL of the server island, a revocation function, an input port, and an output port. The SCURL of the server island must have a public key that contains both the public key component and the private key component. The revocation function is a function that accepts the SCURL of the connecting island as the only argument. The revocation function must return a boolean value of true to indicate that the island should not be authenticated. Protocol messages are read from the input port and written to the output port.

The server authentication function reads the client hello message from the input port. The chosen protocol version is determined by picking the latest protocol version that both the

client and server island support. A server hello message is written to the output port that includes the chosen protocol version, a timestamp, and a nonce. The server island waits for the client authenticate message to arrive on the input port.

The client authenticate message contains the client island's SCURL, a timestamp, a nonce, and a digital signature. The server authentication fails when the timestamp in the client authenticate message is outside of a configurable second delta. The server island performs limited authentication by verifying the authenticity of the digital signature in the client authenticate message. The revocation function is executed with the client island's SCURL when the digital signature is successfully verified. The server authentication fails when the revocation function raises an exception or returns a boolean value of true. The server island sends the server authenticate message when the client island passes the revocation function which includes the server island's digital signature. The digital signature is created using the server island's host id and the timestamp and nonce in the client authenticate message. The server island will now wait for a client done message from the input port.

The client done message indicates that the client island has successfully authenticated the server island and that the SCURL authentication protocol is complete. The client island's SCURL is returned to indicate that the client was successfully authenticated. Any value other than a SCURL returned by this function is an indication of failure.

The four Racket modules that make up the SCURL authentication library are the host id module, the SCURL module, the message definition module, and the authentication module. The host id module defines functions that are used to convert the host id between its byte format and its string format. The SCURL module defines the SCURL structure and utilities that create and manage a SCURL. The message definition module defines all of the messages used in the SCURL authentication protocol in their primitive format. Each message is

defined as a list and has creation functions that handle converting complex data types into primitive types as well as accessor functions that convert the primitive types into a complex data type. Finally, the authentication module defines the client authentication function and server authentication function described above. All modules have correlating test suites defined that were used to test the implementation.

The test suites are not enough to simulate testing the implementation in a working environment. To test the protocol in a more realistic manner a test framework was built that used TCP sockets to connect to remote nodes which were running instances of the test framework as well. The test framework is a placeholder for the CREST framework and handled marshalling Scheme primitives to and from a byte format. The Racket serialization module was leveraged to convert primitives to and from a byte format. The test framework provides a meaningful example of how the authentication protocol could be used in the CREST framework.

3.2.2 Revocation Service

The CREST revocation service is a configurable blacklist for the CREST framework. Revocation certificates and host id blocking give the power of revocation to the CREST developer. A revocation certificate structure is composed of a SCURL structure and a digital signature. The SCURL structure identifies the compromised SCURL and host id that should no longer be trusted. The digital signature is used to verify the authenticity of the revocation certificate without reliance upon a third party.

The digital signature is verified by forming the signature base used to create the digital signature and comparing it against the decrypted signature. The signature base used to create the digital signature is formed by combining the "PathRevoke" symbol, the location,

the port, and the public key of the SCURL. The "PathRevoke" symbol is represented as a UTF-8 formatted byte array of the "PathRevoke" string, the location is represented as an array of UTF-8 bytes, and the port is represented as a two byte value. The digital signature is decrypted using the public key contained in the revocation certificate and compared against the signature base. The authenticity of the revocation certificate is successfully verified when the two values match.

The revocation service is configured with revocation programs. The five attributes of a revocation program are an identifier, a revocation function, a block flag, a filter regular expression, and an exclude regular expression. The identifier is a number assigned to the revocation program that is used to remove the revocation program from the revocation service. The revocation function is a Scheme function that accepts a single string argument. The argument passed to the revocation function will always be the string formatted host id of the SCURL in question. The block flag is a boolean value which indicates whether host id blocking is enabled or not. The filter and exclude regular expressions are Scheme regular expression objects.

The revocation service is a stateless module and uses a revocation list to help manage and store the revocation service configuration. The developer using the revocation service handles storing the revocation list needed when configuring and executing the revocation service. The revocation list is composed of a list containing all registered revocation programs and a variable containing the next identifier that is used when a revocation program is registered. Revocation programs can be added and removed from the revocation list and the revocation system is executed with a revocation list and a SCURL.

The add revocation program function requires a revocation list and a revocation function. The host id blocking flag, filter regular expression, and exclude regular expression are

optional arguments to the add revocation program function. A new revocation program is created out of the given information and assigned a new single use identifier. A new revocation list is returned which contains the old revocation programs and the new revocation program appended to the end of the list of revocation programs. Figure 3.3 depicts how an initialization service would use the revocation service managed by the CREST framework. The remove revocation program function requires a revocation list and a revocation program identifier. The first revocation program with a matching identifier is removed and a new revocation list is returned.

The revocation service is run by executing the run revocation programs function. This function requires a revocation list and a SCURL. The run revocation function attempts to apply each revocation program to the SCURL in question. A revocation program is applied when the SCURL's textual representation matches the revocation programs filter regular expression and fails to match the revocation programs exclude regular expression. If the revocation program is applicable then the SCURL's host id in string format will be passed to the revocation program's revocation function. If the revocation function returns a verifiable revocation certificate then the revocation service will complete and return the revocation certificate. The state of the block flag is checked when the revocation function does not return a revocation certificate and does not raise an exception during execution. If the block flag is set to true then the revocation service will complete and return a boolean value of true to indicate that the SCURL was flagged for host id blocking. The revocation service will return a boolean value of false when no revocation program produces a revocation certificate or has their block flag enabled.

The revocation service is defined in the program module and the revocation module. The program module includes common definitions for programs and program lists as well as functions that provide basic functionality for adding, removing, and executing a program

list. The program module is not available for general consumption and should not be used by developers using the revocation service. The revocation module defines the revocation program, revocation list, and the revocation service functions described above.

The program module and the revocation module have correlating test suites. The revocation system was integrated into the test framework built to test the SCURL authentication protocol. The test framework acting as the client island uses the revocation service prior to establishing a TCP socket with the server island. The test framework acting as the server island executes the revocation service when the revocation function handed to the SCURL authentication protocol is invoked to complete the client island authentication.

The test framework defined and held the revocation list state during execution. A CREST developer does not need to understand the concept of the revocation list as it is hidden from them by the CREST framework. The CREST developer will only need to understand the concept of a revocation program and how the service is run. During testing, it was possible to build instances of the test framework that revoked access to all unknown SCURLs and instances of the test framework that only revoked SCURLs with known revocation certificates. The test framework has proved that the revocation service built is a viable method of providing a configurable blacklist to any system.

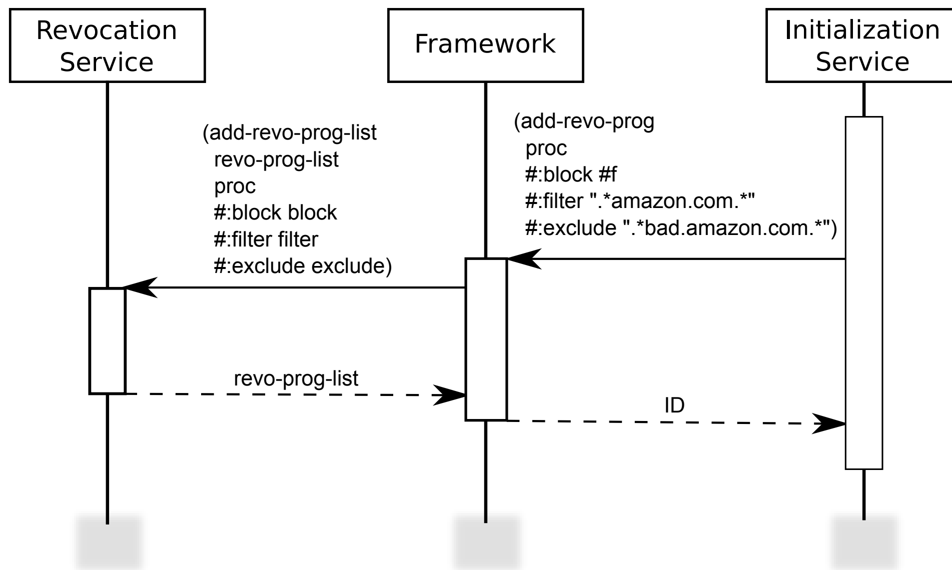

Figure 3.4: The CREST revocation configuration.

Chapter 4

OpenID

CREST developers must be able to leverage existing user authentication libraries for the CREST framework to be considered a feasible development framework. The OpenID protocol was chosen for integration into the CREST framework. This section will analyze the OpenID protocol, discuss the design of the OpenID library libopkele [19], and provide the details of the CREST based OpenID implementation. The analysis of the OpenID protocol provides the reasoning behind the design decisions made by the OpenID architects. The discussion of the libopkele OpenID library design provides background information needed to understand the details of the CREST based OpenID implementation.

4.1 OpenID Protocol

The OpenID protocol solves the single sign-on problem. A single sign-on protocol allows a user to access resources across multiple domains while using one identity to do so. The Identity Provider authenticates the user's identity for the Relying Party. The user's credentials are maintained with the Identity Provider and sensitive information is exchanged between

the user and the Identity Provider. The Relying Party allows the user to access a resource once the Identity Provider authenticates the user for the Relying Party.

The OpenID protocol has been guided through its development by its target domain, the focus on security, and the format and meaning of their identifiers. The OpenID 2.0 specification is a single sign-on solution in the HTTP domain. The protocol defines all OpenID messages as HTTP GET or POST commands and utilizes HTML documents to deliver auto-submitted HTTP POST messages. Security has played a large role in guiding the design of the OpenID protocol and is seen by its use of associations and realms. OpenID identifiers are XRIs or URLs and have undergone significant changes from previous versions of the protocol. Lastly, we will discuss how OpenID views identification and authorization.

HTTP Domain

The OpenID protocol is designed to be used with applications in the web domain. The information in OpenID messages are defined as key/value pairs. These key/value pairs are encoded as HTTP GET and HTTP POST messages. The use of HTTP to encode OpenID messages is a direct reflection of the OpenID origins. OpenID started as a login solution for blog comments, which existed solely in the web domain. HTTP is a standard format that is handled by many libraries and systems in the world and should make it easier for parties to adopt and leverage the OpenID protocol.

The specification mentions that the OpenID authentication plays nicely with an ajax style setup. In an ajax style setup, the user is authenticated without having to leave the Relying Parties website. This style of OpenID authentication might increase the adoption of the protocol because Relying Parties will not have to direct the user away from their website and possibly lose a customer. The ajax setup will also increase the chances of a successful

phishing attack because the user will be comfortable with entering their credentials into an unknown website. The ajax setup is highly dangerous for OpenID users until there is a good defense against a phishing attack.

Security

Associations are used to ensure the integrity of the information that is passed through the user agent. The choice between a shared association and a private association affects the control flow of the protocol. A shared association is created through a direct connection between the Relying Party and the Identity Provider. The association is exchanged in plaintext or encrypted with a shared secret established with the Diffie-Hellman-Merkle protocol. It is recommended to use the TLS/SSL protocol to establish a secure channel when exchanging the shared association in plaintext. A private association is only known to the Identity Provider and used to verify an OpenID assertion through a direct connection between the Identity Provider and the Relying Party.

An association is meant to protect the data in an OpenID message from modification. The bulk of the OpenID messages are sent through the user agent, which increases the number of attack points. The OpenID solution treats the user agent as an insecure entity and enforces that all information sent through the user agent is validated for data integrity. The association does not provide any form of trust in the Identity Provider or the Relying Party. It is solely meant to defeat a man in middle attack when messages are sent through the user agent.

The OpenID protocol does not handle revocation of OpenID identifiers or OpenID endpoints. Revocation is pushed outside of the protocol and the Identity Provider and Relying Party must manage that on their own. A realm field is included in the OpenID protocol and

is meant to indicate the realm that the Relying Party belongs to. The OpenID specification suggests that the Identity Provider presents the realm to the user before sending a positive assertion. The realm could also be used by the Identity Provider to filter automatic assertions based upon user determined filters. This is not strong enough to provide any real form of trust when it is only a suggestion to implementations. It is up to the Identity Provider and Relying Party to build revocation into their OpenID implementations.

Identifiers

The OpenID 2.0 specification is the culmination of effort between the OpenID developers, the Light-Weight Identity protocol developers, and the XRI/i-names developers. All three of these collaborators have affected the way OpenID identifiers are defined and used. There are two different types of OpenID identifiers and two different formats for an OpenID identifier. This section will detail the differences in OpenID identifiers.

The OpenID 2.0 specification allows the use of two different types of identifiers. The OpenID Identity Provider's identifier or the OpenID user's identifier can be given to the Relying Party to start the OpenID protocol. This is a shift from earlier versions of the protocol where only the OpenID user's identifier could be used. This allows users to make use of different personas while only having to remember a single OpenID identifier.

A user can enter an OpenID Identity Provider identifier and they will be forwarded to that Identity Provider. The Identity Provider allows the user to choose an identifier to be authenticated with and returns the user's identity with the positive assertion. This makes it possible for the user to use different identities with different Relying Parties while only remembering a single OpenID identifier. It is important to note that the user's identifier will always be in a positive assertion sent in response to an authentication request for a user's

identity.

The URL format was the first identifier format available in the OpenID 1.0 specification and relied upon HTML-based discovery. Following the OpenID 1.0 specification the OpenID developers and the Light-Weight Identity developers collaborated to produce the Yadis discovery protocol. The Yadis discovery protocol was added to the OpenID 2.0 specification to help replace the dependence on HTML-based discovery. HTML-based discovery is restricted to discovery of a single OpenID endpoint and is eclipsed by well developed service definition formats that were available in the community.

The Yadis discovery protocol resolves the URL based identifier to an XRDS document. The XRDS document format was contributed by the XRI/i-names developers and contains different identity management services. Each service in the XRDS document has a priority and a type. OpenID services include a URI that is the discovered OpenID endpoint. The XRDS format offers a wider range of functionality when resolving an identifier to an Identity Provider than HTML-based discovery and was a natural growth for the OpenID 2.0 specification.

The second identifier format allowed is in the form of an XRI. The XRI/i-names developers use an abstract global namespace to assign non-reusable identifiers that are resolved through the XRI resolution protocol. The XRI/i-names developers helped integrate the use of XRIs into the OpenID 2.0 specification. XRI formatted identifiers provide a non-reusable identifier in an abstract global namespace and are resolved to an XRDS document similar to the Yadis discovery protocol. One of the major critics of OpenID is its use of URL's as identifiers. A URL is recyclable and will be resold to another owner when the payments are not maintained. This leads to different users accessing the same Relying Party with the same identifier.

For example, user A has the identifier “user.uci.edu” and is accessing a resource owned by the Relying Party, “www.relying-party.com”. The Relying Party creates an internal user account to track the user’s statistics and usages of their resources. After a period of time the URL identifying user A may be reassigned to user B. User B accesses the resource with the same identifier, successfully authenticates, and continues to the use the Relying Parties service. The information being tracked by the Relying Party is incorrect and should not be associated with this new user. It is even worse when the Relying Party is storing personal information about User A that is now accessible by User B.

The OpenID 1.0 specification used URL based identifiers to indicate the user identity. The OpenID 2.0 specification has grown with the use of OpenID identifiers to indicate either a user identifier or an Identity Provider identifier in either an XRI or URL format. The OpenID 2.0 specification listened to the concerns of the community and took actions to modify the specification to improve the protocol.

Identification and Authorization

Many user authentication protocols do not distinguish between identification and authorization. Identification is often overused when authorization would suffice. Authorization allows an action to occur. For example, a Relying Party may provide news articles to an end user. The Relying Party does not need to know who the end user is, but only that they are authorized to view the news articles. On the other hand, identification allows two parties to continue a relationship. For example, a Relying Party offers a fantasy Hockey league service. The Relying Party must identify the end user that is trying to access the service because the end user wants to manage their fantasy Hockey team. The overuse of identification is a prevalent problem and leads to the decline of anonymity and privacy. [20]

The OpenID protocol is a single sign-on protocol that provides identification of the user to the Relying Party. The identification is performed by the Identity Provider and results in a token, the positive assertion, being given to the Relying Party. This token could act as authorization if not for the identity value contained in the positive assertion. The identity value contains the user's identifier for use with the Relying Party. This is a minor technical difference, but a major conceptual difference.

Providing the identification of a user allows the Relying Party to track the user's actions. This may provide a better experience for the end user because the Relying Party is able to custom tailor the user's experience based upon past actions. It will also build a relationship with the Relying Party that the user may not wish to partake in. The distinction of whether to identify or authorize should be with the user and not glossed over by the protocol.

The OpenID protocol may be designed to provide identification because of its origins as an authentication system for blog post comments. When the initial goal of a protocol is to attach an identity to a block of text then it is natural for it to grow into a system that provides identification. OpenID has grown larger and must grow a bit more to provide both identification and authorization. This is crucial for the continued privacy and anonymity that we enjoy on a daily basis.

The OpenID protocol could be modified to provide identification and authorization with a very small change. Make the identifier value in the positive assertion an optional value. A user could use the OpenID Identity Provider identifier to begin the OpenID process. The Identity Provider would either fill in or leave blank the identity field in the positive assertion based upon the user's choice of either identification or authorization. It would be up to the Relying Party to decide whether they required identification or authorization and what type of service they provide in each case. This change puts the choice of anonymity back into the

hands of the user.

The OpenID Identity Provider will be able to track which Relying Parties that the user uses. It is up to the user to find a trusted Identity Provider and to agree to their terms and conditions. Similarly, it is up to the user to agree to the terms and conditions when being identified to a Relying Party. The user would have more control than not using a service because they do not wish to identify themselves and Relying Parties would have more customers.

4.2 libopkele

The OpenID C++ library, libopkele, is an object-oriented implementation of basic OpenID functionality. The library requires the OpenSSL library [4], the cURL library [2], the Boost smart pointer library [1], the Expat XML Parser library [5], and the HTML Tidy library [3]. The OpenSSL library is used for all cryptographic utilities in the OpenID protocol during association establishment and verifying positive assertions. The cURL library is used to send HTTP data during discovery and shared association establishment. The Boost smart pointer library includes a shared pointer that libopkele uses for memory management during runtime. The Expat XML Parser library is used to parse XRDS documents retrieved through XRI resolution or the Yadis protocol during discovery. Finally, the HTML tidy library is used to correct the HTML output of the libopkele library.

The library contains definitions for OpenID data types and the basic functionality of a Relying Party and an Identity Provider. The association data type contains the information established when an association is created for verification of a positive assertion. The OpenID message class is a map of key to value pairs where both the key and value are stored as

string data types. The Relying Party functionality is contained in the *basic_RP* class and the Identity Provider functionality is contained in the *basic_OP* class. A concrete implementation will inherit from these abstract classes and provide the persistent storage implementation needed for their framework.

The *basic_RP* class defines an API using abstract functions that must be defined by the implementation. The global persistent store functions are used to store, retrieve, and invalidate associations that must be stored between the round trip. The session persistent store functions are used to store and retrieve OpenID endpoints that are found during discovery, but are not needed after the round trip. Lastly, the OpenID action functions are used to initiate a session, create an OpenID authentication assertion, and verify an OpenID authentication assertion.

The *basic_OP* class defines an API using abstract functions that must be defined by the implementation. The global persistent store functions are used to create and retrieve both associations and nonces. The association must be stored for later use to sign and possibly verify a positive assertion. The nonce must be stored for a reasonable amount of time to ensure that it cannot be reused in a replay attack. The OpenID action functions provide functionality for creating an association, creating a positive assertion, and creating a negative assertion.

The *basic_RP* and *basic_OP* use a combination of return values and exceptions to indicate state when calling the API functions. Memory management is handled by shared pointers, which wrap all return values. An exception is raised to indicate an error or an implementation choice. For example, the *basic_RP* class defines the *retrieve_assoc* function in the global persistent store API. A *failed_lookup* exception is raised to indicate that an association was not found and a *dumb_RP* exception is raised to indicate that this Relying Party

implementation does not support shared associations. A private association is created by the Identity Provider when the Relying Party does not support shared associations. Finally, an association is returned when it is found.

The libopkele library provides all of the basic OpenID functionality and an API that allows the developer to define the method of storage necessary for the OpenID protocol to work. It handles identifier normalization and discovery including XRI resolution, the Yadis discovery protocol, and HTML-based discovery. It uses cURL to perform direct connections with the Identity Provider when performing discovery, creating a shared association, and verifying a positive assertion.

The major issue with this library is the use of exceptions for control flow. This decreases the readability of the source code and makes it difficult to understand what values should be returned versus what exceptions should be thrown in the overridden methods. This issue increased the learning curve needed to understand the library and was an irritant rather than a significant issue with the operation of the library.

4.3 CREST OpenID Implementation

The CREST OpenID implementation proves that it is possible to port an existing user authentication protocol into the CREST framework. The CREST framework is a development environment that can utilize any library built in the same language. This boils the CREST OpenID implementation down to building a Scheme based solution for the existing OpenID user authentication protocol. Racket is an implementation of the Scheme language and was used to build the CREST OpenID implementation. The CREST OpenID implementation only provides the Relying Party functionality and was not built to support a CREST OpenID

Identity Provider. The act of creating the Relying Party is enough to prove that it is possible to utilize existing authentication libraries in CREST.

Racket has a great foreign function interface, FFI, which allows the developer to call C functions from a racket module. This is done by defining the C function signatures in a Racket module and tying the Racket definition to the loaded library at runtime. There are many open source libraries that provide an OpenID implementation: hsopenid (Haskell), JOpenID (Java), OpenID4Perl (Perl), EasyOpenID (Python), mod_auth_openid (C++), and libopkele (C++). There are no OpenID libraries written in C and the best approach was to wrap the C++ library libopkele with a layer of C code.

C Layer

The mod_auth_openid library [27] is an apache module that allows developers to use OpenID as a Relying Party for their website by using libopkele [19] for basic OpenID functionality and sqlite for persistent storage. The mod_auth_openid library provided an example of how to utilize the libopkele library in a C framework and was used as a reference example for the stateless C layer that was built.

The libopkele library does not provide storage for OpenID endpoints, associations or nonces and it was important that the C layer held to the same premise. The C layer is solely meant to provide a Racket usable C interface which deals with memory management and conversion between an object based architecture and a functional architecture. The mixture of return values and exceptions to indicate state was not used when creating the C layer. This convention was not used because mixing exceptions between C, C++, and Racket is not recommended and not necessary. The C layer uses only return values to indicate success or failure.

The C layer defines a Relying Party implementation that inherits from the *basic_RP*. The C Relying Party implementation delegates all libopkele API functions to member function variables, which are defined during creation. When the libopkele library invokes an API function, the data is converted to the appropriate C data type and a delegate function is invoked. The return value is converted to the libopkele return type or exception and all memory is deallocated before returning control to the libopkele library. All C delegate functions provide feedback through their return value and all use of exceptions is restrained to the C layer.

The C layer contains a set of utility functions that help to manage C++ data types. The C++ class objects are passed as structure pointers through the C layer. The utility functions are accessors to the C++ class objects and allow the developer to set and extract the information hidden inside of the structure pointers. All libopkele API functions are being forwarded through the C layer by using delegate and accessor functions. A simple test case was built to utilize the C layer to ensure that all OpenID functionality worked correctly before building the Racket module. The test case exercises the C to C++ interface using in memory storage to retain information between round trips.

Racket Layer

A Racket module was created using the Racket FFI to invoke functions defined in the C layer. The Racket module is similar in design to the mod_auth_openid library in that it provides a concrete implementation of a Relying Party. The Racket module uses the C layer to interface with the libopkele library and uses a Racket sqlite module for persistent storage in a SQL database. The mod_auth_openid library was used as a reference guide for completing the final Racket OpenID implementation.

The Racket OpenID module is composed of the FFI definitions module and the Relying Party module. The FFI module defines the data structures and functions implemented in the C layer and ties them into the dynamically loaded library at runtime. The Relying Party module defines the delegate API functions that are handed down to the C layer. The delegate API functions use a Racket sqlite module to provide persistent storage. The SQL table structure used in the delegate API functions was taken from the mod_auth_openid library.

The Relying Party module provides two functions that a developer leveraging the Racket OpenID module would be concerned with: the start_authentication_session and the validate_authentication_session. These two functions handle initializing and calling the lower level C layer with the correct delegate functions defined in the Relying Party module. A developer using the Racket OpenID module only needs to use the start_authentication_session and the validate_authentication_session and all dangerous interface code calling the C layer is handled by the Racket OpenID module.

The Racket module defines a simple test similar to the C test case that provides in memory storage to retain information and is used to test the FFI module. A second test case is provided which uses the start_authentication_session and validate_authentication_session to prove that the Relying Party implementation module for the Racket OpenID module functions correctly.

The example in Figure 4.1 shows the interaction between the different layers during the start of an authentication session and will provide a better picture of how the Racket OpenID module works. In step 1, the test case starts the authentication by handing the Racket OpenID Module the identity, server, realm, and return-to values. In step 2, the Racket OpenID Module initializes and opens the database and invokes the start authentication

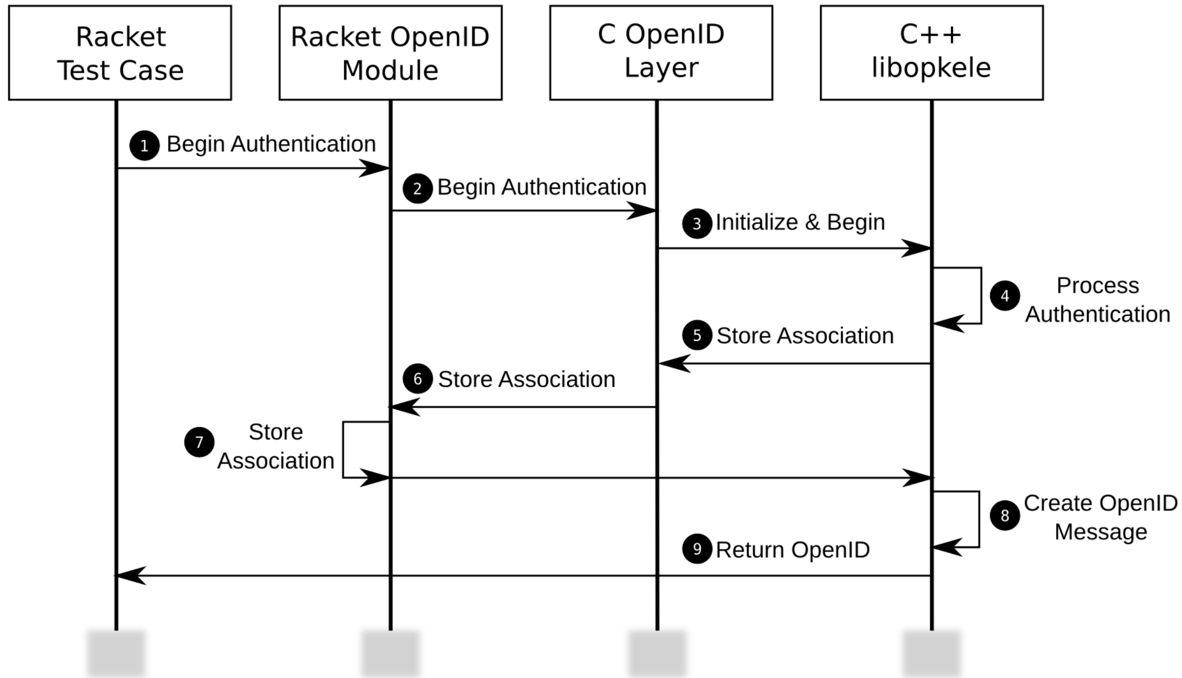

Figure 4.1: An example of how storing associations is handled in the Racket OpenID module.

function on the C layer passing the Racket delegate functions as well. In step 3, the C layer creates an object that extends the libopkele base class and initiates authentication on the object. In step 4, the libopkele base class performs normalization, discovery and creates a shared association with the discovered OpenID endpoint.

In step 5, the libopkele base class invokes the store association function that is overloaded in the C layer with the new association as an argument. In step 6, the C layer will convert the C++ data types to C data types and invoke the store association function pointer which is a delegate function provided by the Racket module. In step 7, the store association delegate function in the Racket OpenID module will be invoked with the association information. The association is stored using the Racket sqlite module and control is returned to the C layer. The C layer converts the return value of a boolean to indicate success or failure into the appropriate exception that is thrown in the libopkele library if needed. In step 8, the libopkele library will create and populate an OpenID message and return the value back to

the Racket test case.

Chapter 5

Evaluation

The SCURL authentication protocol is a feasible solution to entity authentication and the CREST framework is a fertile environment for the integration of existing user authentication protocols. The CREST SCURL and the SCURL authentication protocol use public key cryptography to authenticate two entities. An island is authenticated when it has been proved that they are claiming the expected public key and that they own the correlating private key. The client island achieves full authentication of the server island by verifying both claims. The server island achieves limited authentication because it only verifies that the client island owns the associated private key.

The CREST authentication protocol does not provide trust as to the intent of the island. As seen with the TLS/SSL protocol, it is not possible for an authentication protocol to assign trust to the intent of the island. The root certificates used in the TLS/SSL protocol are a single point of failure because there is no way to revoke their status. In the CREST framework, all CREST SCURLs can be revoked with the use of a revocation certificate. The revocation certificate is a decentralized method to publish a compromised private key.

Finally, the CREST revocation system is a crucial component of a decentralized authentication system. The CREST revocation system is used to complete the limited authentication on the server island and is the means to assign trust in the CREST framework. The CREST administrator will use the CREST revocation system to define the trust attempted by the TLS/SSL protocol. This trust can only be provided by an external authority and is not possible to establish with an authentication protocol.

The Racket OpenID module proves that it is possible to leverage an existing user authentication library in the CREST framework. Integration of a Scheme module into the CREST framework is as simple as including it in the development environment. The bulk of the effort for this module was expended to build the C wrapper around the existing C++ implementation that dealt with exception handling and return values in libopkele. The development of this module would have been considerable shorter if an existing C layer had been available. The metrics provided below show that the overhead added by the Racket FFI module become insignificant when compared with the latency added by the network and persistent storage of the OpenID module.

5.1 Benchmark

The SCURL authentication protocol needs to perform at a reasonable speed. The protocol needs to complete as quickly as possible and use as little resources as possible because it will be performed with each new connection. The two values that were of most interest are how long it takes the protocol to complete and how much CPU utilization the protocol is using. The first set of metrics presented was collected while running both the client island and the server island on the same machine. The second set of metrics presented was collected while running the client island and the server island on different machines.

The time measured, for both sets of metrics, was determined by capturing a timestamp directly before and after the SCURL authentication functions were called. The starting timestamp was captured after the TCP socket had been established and the ending timestamp was captured before the TCP socket was broken down. The only external influences are the latency introduced by the test frameworks conversion of Scheme primitives to bytes and the network latency.

The CPU utilization measured, for both sets of metrics, was determined by running the Linux *top* command in batch mode. The top command queries the process CPU utilization at a regular interval and its output was captured in a text file. The output of the top command was then parsed using a custom built bash script to provide an average CPU utilization presented in this paper. Only CPU percentages greater than zero were included in the average.

The machine used to collect both the client island and server island results is an Intel Core2 Duo 2.50GHz CPU with 3.9 Gigabytes of RAM. The remote machine is an AMD Athlon 3700 2.21 GHz with 1.0 Gigabyte of RAM. All tests were run twice to capture the results for both the client island and the server island from the Intel machine. The Racket test framework built to test the Racket SCURL modules was modified to instrument the SCURL authentication protocol. To run the tests a standalone executable was generated by DrRacket, which was then run as a command line process.

Connections	**Client Island (ms)**	**Server Island (ms)**
1	70.00	69.00
10	57.20	57.00
100	55.47	55.46
1000	55.74	55.68

Table 5.1: The average amount of time taken to perform the SCURL authentication protocol between two entities on a local machine.

The results presented in Table 5.1 show the average time taken to complete the SCURL authentication protocol when both parties are on the same machine. This test minimizes the network latency and provides a good view of the time used to complete the SCURL authentication protocol in a black box. As the number of connections grows the time for both the client island and server island converges to 55 milliseconds.

Connections	**Client Island (CPU %)**	**Server Island (CPU %)**
1	20	8
10	8	2
100	9	2
1000	9	2

Table 5.2: The average amount of CPU used to perform the SCURL authentication protocol between two entities on a local machine.

The results presented in Table 5.2 show the average amount of CPU utilization used by the client island and the server island when they are on the same machine. This test minimizes network latency when capturing timing metrics, but inflates CPU utilization because both processes are being run on the same machine. As the number of connections grow the client island CPU utilization converges to 9% and the server island CPU utilization converges to 2%.

The result for a single connection is higher because of the fidelity at which the metric was captured. There were results for the single connection test that produced a lower number, but the test for a single connection does not produce very accurate results. This test works better over a longer period of time when the utility can capture many points. The single connection result is included for completeness, but more weight should be placed in the tests with a larger sample set.

Connections	Client Island (ms)	Server Island (ms)
1	72.00	91.00
10	75.60	78.40
100	66.41	76.81
1000	66.726	77.41

Table 5.3: The average amount of time taken to perform the SCURL authentication protocol between two entities on different machines over a wired connection.

The results presented in Table 5.3 show the average time taken to complete the SCURL authentication protocol when the client island and server island are on different machines. These machines were connected through a Linksys WRT54G2/GS2 router using the CAT-5 wired interface. As the number of connections grow the client island converges to 66 milliseconds and the server island converges to 77 milliseconds.

Connections	Client Island (ms)	Server Island (ms)
1	123.00	151.00
10	135.70	148.4
100	140.36	146.21
1000	142.77	150.84

Table 5.4: The average amount of time taken to perform the SCURL authentication protocol between two entities on different machines over a wireless connection.

The results presented in Table 5.4 show the average time taken to complete the SCURL authentication protocol when the client island and server island are on different machines. These machines were connected through a Linksys WRT54G2/GS2 router using the 802.11 B/G wireless support. The network latency is the main cause for the increase of average time. This test is a better example of how long the protocol will take to complete in a "real world" situation. As the number of connections grow the client island converges to 142 milliseconds and the server island converges to 150 milliseconds.

Connections	Client Island (CPU %)	Server Island (CPU %)
1	10	20
10	9	9
100	9	8
1000	8	9

Table 5.5: The average amount of CPU used to perform the SCURL authentication protocol between two entities on different machines over a wired connection.

The results presented in Table 5.5 show the average amount of CPU utilization used by the client island and the server island when they are on different machines over the wired connection. This test provides a better example of CPU utilization because there is only one island running on each machine. As the number of connections grow the client island CPU utilization converges to 8% and the server island CPU utilization converges to 9%.

Connections	Client Island (CPU %)	Server Island (CPU %)
1	8	4
10	3	3
100	3	3
1000	3	3

Table 5.6: The average amount of CPU used to perform the SCURL authentication protocol between two entities on different machines over a wireless connection.

The results presented in Table 5.6 show the average amount of CPU utilization used by the client island and the server island when they are on different machines over the wireless connection. As the number of connections grow both the client island and server island CPU utilization converges to 3%.

5.1.1 OpenID

The speed of the OpenID Racket module is not a major concern as the limiting factor will be the network latency and persistent storage. The major point of interest is the overhead

added by the Racket FFI used to leverage the C layer. The metrics presented in Table 5.7 show the time taken to invoke three delegate functions from the C Layer.

To get an accurate measure of the amount of time added by the Racket FFI it was important to remove delays caused by network latency and persistent storage. The C test case and Racket test case were used because they use in memory storage and do not perform persistent storage. The network latency was removed by starting the timing after discovery was performed. The timing starts after normalization and discovery are performed and ends with the output of the OpenID authentication assertion message. In these tests, the C layer calls the *get_endpoint* delegate function twice and the *append_params* delegate function once.

The machine used to collect the results is an Intel Core2 Duo 2.50GHz CPU with 3.9 Gigabytes of RAM. Both the C test case and the Racket test case were modified to capture timing results, display them to standard output, and exit directly after creating the OpenID Authentication Assertion message. The Racket test case was built as a standalone executable by DrRacket and run as a command line process.

Messages Generated	**C Test Case (μs)**	**Racket Test Case (μs)**
1	492	880
10	498	800
100	439	778
1000	473	764
3000	478	761

Table 5.7: The amount of time to invoke three delegate functions during creation of an OpenID Authentication Assertion message.

The results presented in Table 5.7 show the average time taken for the C layer to invoke three delegate functions defined in the test case. The C test converges to 478 μs and the Racket test case converges to 761 μs. This test shows that the Racket FFI adds around a

60% overhead to each delegate function invoked by the C layer. This overhead is insignificant when compared with network latency, persistent storage, and the benefit of using the Racket programming language.

Chapter 6

Conclusion

The TLS/SSL and the SFS key negotiation protocols were studied to understand two distinct methods of performing entity authentication. The TLS/SSL protocol provides authentication through digital certificates and relies heavily upon certificate authorities. The SFS key negotiation protocol provides authentication through self-certifying pathnames, which provide a decentralized approach to authentication. These protocols were both lacking in different areas, but each provided an excellent source of knowledge that was used to create the SCURL authentication protocol.

The TLS/SSL protocol relies upon certificate authorities to provide entity authentication. This model of centralized authentication was shown to have many flaws. The business model drives decisions based on increasing capital instead of providing security. A root certificate cannot be revoked and will eventually be compromised. Lastly, the trust provided by certificate authorities is a misnomer and must be provided by a decentralized group external to the authentication protocol.

The SFS key negotiation protocol used self-certifying pathnames and host ids to provide entity authentication. The SFS key negotiation protocol performed implicit, one-sided authentication. This worked well in a client-server architecture for a remote file system, but was not suitable for a peer-to-peer architecture like the CREST framework. Lastly, the self-certifying pathname does not fit in the web domain and led to research into the SFS-HTTP system, which presented the initial version of a self-certifying URL.

The SCURL authentication protocol uses CREST SCURLs to authenticate CREST islands. The CREST SCURL fully conforms to the URL syntax definition and holds the same utility as self-certifying pathnames. The protocol provides explicit, two-sided authentication in a decentralized manner. The intent of the island must be determined by an external party, but is applied through the CREST revocation system.

The CREST OpenID module was built following the development of the SCURL authentication protocol. OpenID and Shibboleth were studied to gain an understanding of solutions to the single signon problem. Both systems solve the single signon problem in a similar conceptual manner, but have distinctly different semantics and solutions. The goal of this research was to create a Racket OpenID module that could be leveraged in the CREST framework.

Future work could be done in regards to determining trust of a CREST island. The CREST revocation system provides a way to apply trust, but there needs to be a way to determine trust. A suitable web of trust architecture and implementation could be built around CREST SCURLs. Another area of work could be in modifying the OpenID protocol such that identification and authorization could be provided based upon a user choice. This would allow a wider scope of use for the OpenID protocol and possibly help to increase its usage in the web community.

Bibliography

[1] Boost Smart Pointers. http://www.boost.org/doc/libs/1_47_0/libs/smart_ptr/smart_ptr.htm.

[2] cURL. http://curl.haxx.se/.

[3] HTML Tidy Library Project. http://tidy.sourceforge.net/.

[4] OpenSSL. http://www.openssl.org/.

[5] The Expat XML Parser. http://expat.sourceforge.net/.

[6] P. K. Alan O. Freier and P. C. Kocher. The SSL Protocol Version 3.0. Technical report, Transport Layer Security Working Group, Nov. 1996.

[7] D. R. Brad Fitzpatrick and J. Hoyt. OpenID Authentication 2.0 - Final. Technical report, OpenID Foundation, Dec. 2007.

[8] T. K. C. Adams, S. Farrell and T. Mononen. Internet X.509 Public Key Infrastructure Certificate Management Protocol (CMP). RFC 4210, Internet Engineering Task Force, Sept. 2005. accessed July 14, 2011.

[9] Comodo. Comodo Fraud Incident 2011-03-23. http://www.comodo.com/Comodo-Fraud-Incident-2011-03-23.html, Mar. 2011.

[10] S. F. S. B. R. H. D. Cooper, S. Santesson and W. Polk. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280, Internet Engineering Task Force, May 2008. accessed July 14, 2011.

[11] D. Eastlake et al. XML Signature Syntax and Processing. Technical report, World Wide Web Consortium, Feb. 2002.

[12] David Mazières. Self-certifying File System. Technical report, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 2000.

[13] T. Dierks and C. Allen. The TLS Protocol Version 1.0. RFC 2246, Internet Engineering Task Force, Jan. 1999. accessed July 09, 2011.

[14] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.1. RFC 4346, Internet Engineering Task Force, Apr. 2006. accessed July 09, 2011.

[15] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, Internet Engineering Task Force, Aug. 2008. accessed July 09, 2011.

[16] Dimitris Vyzovitis. mzcrypto. http://planet.plt-scheme.org/package-source/vyzo/crypto.plt/2/0/planet-docs/manual/index.html.

[17] S. D. E. Rescorla, M. Ray and N. Oskov. Transport Layer Security (TLS) Renegotiation Indication Extension. RFC 5746, Internet Engineering Task Force, Feb. 2010. accessed July 09, 2011.

[18] L. C. W. T. Gabe Wachob, Drummond Reed and S. Churchill. Extensible Resource Identifier (XRI) Resolution V2.0. Technical report, OASIS, Feb. 2008.

[19] K. Group. libopkele, c++ openid implementation. http://kin.klever.net/libopkele/, Dec. 2009.

[20] J. Harper. *Identity Crisis: How Identification Is Overused and Misunderstood*. Cato Institute, 2006.

[21] K. E. Hickman. The SSL Protocol. Technical report, Netscape Communications Corp., Feb. 1995.

[22] J.R. Prins (CEO Fox-IT). DigiNotar Certificate Authority breach: Operation Black Tulip. http://www.rijksoverheid.nl/ministeries/bzk/documenten-en-publicaties/rapporten/2011/09/05/diginotar-public-report-version-1.html, Sept. 2011.

[23] Justin R. Erenkrantz, Michael M. Gorlick, Richard N. Taylor. CREST: A new model for Decentralized, Internet-Scale Applications. Technical report, Institute for Software Research, University of California, Irvine, Sept. 2009.

[24] M. Kaminsky and E. Banks. SFS-HTTP: Securing the Web with Self-Certifying URLs. Technical report, MIT Laboratory for Computer Science.

[25] A. M. S. G. M. Myers, R. Ankney and C. Adams. X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP. RFC 2560, Internet Engineering Task Force, June 1999. accessed July 14, 2011.

[26] J. Miller. Yadis Specification Version 1.0. Technical report, Mar. 2006.

[27] B. Muller. mod auth openid. http://findingscience.com/mod_auth_openid/, Feb. 2010.

[28] OASIS Security Services TC. Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) V1.1. Technical report, OASIS, Sept. 2003.

[29] OASIS Security Services TC. Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) V2.0. Technical report, OASIS, Mar. 2005.

[30] OASIS Security Services TC. Bindings for the OASIS Security Assertion Markup Language (SAML) V2.0. Technical report, OASIS, Mar. 2005.

[31] OASIS Security Services TC. Profiles for the OASIS Security Assertion Markup Language (SAML) V2.0. Technical report, OASIS, Mar. 2005.

[32] OASIS Security Services TC. Security Assertion Markup Language (SAML) V2.0 Technical Overview. Technical report, OASIS, Mar. 2008.

[33] D. Recordon and B. Fitzpatrick. OpenID Authentication 1.0. Technical report, OpenID Foundation.

[34] D. Recordon and B. Fitzpatrick. OpenID Authentication 1.1. Technical report, OpenID Foundation, May 2006.

[35] D. Reed and D. McAlpin. Extensible Resource Identifier (XRI) Syntax V2.0. Technical report, OASIS, Nov. 2005.

[36] Scott Cantor, Steven Carmody, Marlena Erdos, Keith Hazelton, Walter Hoehn, RL B̈obM̈organ, Tom Scavo and David Wasley. Shibboleth Architecture - Protocols and Profiles. Technical report, OASIS.

[37] Thomas Y.C. Woo, Raghuram Bindignavle, Shaowen Su and Simon S. Lam. SNP: An Interface for Secure Network Programming. Technical report, Department of Computer Sciences, The University of Texas at Austin, 1994.

Appendices

A Appendix A

The following tables define the message formats for the messages in the SCURL authentication protocol. The byte format is not specified because the protocol was written as a library to be used by the CREST framework, which handles conversion between scheme primitives to a byte format used over the wire.

Name	**Type**
identifier	integer
max-version	real

Table A.1: Client Hello Message Definition

Name	**Type**
identifier	integer
version	real
timestamp	integer
nonce	bytes

Table A.2: Server Hello Message Definition

Name	Type
identifier	integer
URL	string
digest type	integer
pkey type	integer
public key	bytes
timestamp	integer
nonce	bytes
signature	bytes

Table A.3: Client Authenticate Message Definition

Name	Type
identifier	integer
URL	string
digest type	integer
key type	integer
public key	bytes
signature	bytes

Table A.4: Server Authenticate Message Definition

Name	Type
identifier	integer
is-authenticated	boolean

Table A.5: Client Done Message Definition

Name	Type
identifier	integer
URL	string
digest type	integer
key type	integer
public key	bytes
signature	bytes

Table A.6: Key Revocation Response Message Definition