

UNIVERSITY OF CALIFORNIA
RIVERSIDE

A Scalable Architecture for Public Key Distribution Acting in Concert with Secure DNS

A Thesis submitted in partial satisfaction
of the requirements for the degree of

Master of Science

in

Computer Science

by

Daniel Francis Berger

August 2004

Thesis Committee:

Dr. Chinya Ravishankar, Chairperson

Dr. Mart Molle

Dr. Thomas Payne

Copyright by
Daniel Francis Berger
2004

The Thesis of Daniel Francis Berger is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

I owe an uncountable debt to my wife and best friend, Dawnise. Without her unwavering support and indulgence, this document, and the study and exploration it represents, would have been utterly impossible.

My deepest thanks to my parents, Gwen and Joel, for all the things parents deserve thanks for but so rarely get. And to my Uncle Denis; for the sage advice offered to a toddler years ago — “always ask why.” It’s hard to imagine three small words capable of causing so much parental consternation.

Many thanks to everyone who reviewed early drafts of this manuscript, and especially to Dr. Mart Molle, Dr. Peter Fröhlich, John Jones, Titus Winters, Tony Espy, and Win Treese, for their thoughtful comments and critique. A sincere thank you to Olaf Kolkman of RIPE NCC, for taking the time to answer questions about turning DNSSEC principle into practice.

To my committee members Dr. Mart Molle and Dr. Thomas Payne, thank you for conspiring to facilitate my return to grad school, and for taking time from your schedules to sit on my thesis committee. Finally, thank you to Dr. Ravishankar and DARPA for supporting my studies and for advising my journey along the path.

This work sponsored in part by $C_8H_{10}N_4O_2$.

ABSTRACT OF THE THESIS

A Scalable Architecture for Public Key Distribution Acting in Concert with Secure DNS

by

Daniel Francis Berger

Master of Science, Graduate Program in Computer Science
University of California, Riverside, August 2004
Dr. China Ravishankar, Chairperson

In this thesis, we present a distributed infrastructure for authenticated distribution of public keys, building on Secure DNS as a foundation. The lack of such an infrastructure has hampered widespread adoption of public key cryptography, despite its great promise. Successful deployments of public key cryptography to date — such as SSL/TLS — have in effect been centralized, since they put a select number of commercial firms in charge of issuing cryptographic certificates, the fundamental unit of digital identity.

Our architecture is scalable and general, and our strategy of using Secure DNS (DNSSEC), an existing infrastructure, makes adoption more tractable. Our goal is to bridge theory with practice, and to enable pervasive end-to-end encryption and data authentication for networked applications.

Clients wishing to register or retrieve public keys use DNSSEC to securely discover the

identity of relevant key registration/distribution servers. Key registration/query messages are directed to the servers using a special-purpose protocol. Clients can validate and authenticate keys retrieved from key servers using public key commitments published in DNSSEC. Applications can establish secure end-to-channels between named endpoints (such as users or named hosts) by using our infrastructure to perform mutual authentication, based on the trust placed in the domains containing the endpoints.

We make the following contributions. First, we specify concise key registration and query protocols which allow users and applications to register, retrieve, and authenticate keys in concert with DNSSEC. Second, we present the design and implementation of a prototype, to demonstrate the ease deployment and use of our model.

Contents

List of Tables	xiii
List of Figures	xiv
1 Introduction	1
1.1 The Need for Digital Privacy and Authenticity	1
1.2 Simplifying End-To-End Privacy	2
1.3 Problem Statement: Steps Toward A Solution	3
1.4 Why Key Distribution?	4
1.5 Roadmap	4
2 Cryptography Primer	6
2.1 Symmetric and Asymmetric Ciphers	7
2.2 Digital Signatures	9
2.3 Secure Hash Functions	11
2.4 Key Authentication	12

2.4.1	Certifying Authorities	12
2.4.2	Webs Of Trust	13
2.4.3	Identity Based Encryption (IBE)	13
2.5	Key Revocation	14
3	The Domain Name System (DNS)	16
3.1	The DNS Namespace	17
3.2	DNS Record Types	18
3.3	DNS Query Resolution	20
3.4	Efforts to Secure DNS	21
3.4.1	DNSSEC Overview	23
3.4.2	DNSSEC Implementation Status	26
4	Previous Approaches to Key Distribution	27
4.1	In-Band Key Transmission	27
4.1.1	Secure Shell (SSH)	28
4.1.2	Secure Socket Layer (SSL)/Transport Layer Security (TLS)	28
4.2	Dedicated Public Key Distribution Services	29
4.3	Distribution by Directory Service	30
4.3.1	X.500	30
4.3.2	DNS	31

5	Proposed Solution	36
5.1	High Level Solution Summary	37
5.2	Design Philosophy	38
5.2.1	Requirements/Constraints	39
5.3	System Architecture	42
5.3.1	Scalability	44
5.3.2	Query Interface	45
5.3.3	Administrative Authority	45
5.4	Envisioned Use Cases	46
5.4.1	Key Lookup	46
5.4.2	Key Registration	49
6	Protocol Design	53
6.1	Locating a Key Server	54
6.1.1	DNS SRV Resource Records	54
6.1.2	Key Server Service and Protocol Names	55
6.2	Message Marshalling and Transport	56
6.2.1	Marshalling	56
6.2.2	Transport	58
6.2.3	Key Query and Key Registration Target URIs	58
6.3	Locating Response-Signing Keys	59

6.4	Authentication During Key Registration	61
6.4.1	Targeted Authentication Mechanisms	62
6.4.2	Candidate Authentication Schemes	63
6.4.3	Key Registration Authentication Process	64
6.4.4	HTTP Authentication	65
6.4.5	External Authentication	66
6.4.6	Client Key Management	67
7	Protocol Messages	69
7.1	Key Query	70
7.1.1	Key-Query Parameters	70
7.1.2	Key-Query Response	72
7.2	Key Registration	78
7.2.1	Key-Registration Parameters	78
7.2.2	Key-Registration Response	81
7.3	Key Revocation	82
7.3.1	Key-Revocation Request	82
7.3.2	Key-Revocation Response	83
8	Implementation	85
8.1	Overview	85

8.2	DNSSEC: Theory into Practice	86
8.2.1	Securing a Zone	86
8.2.2	Secure Resolution	89
8.2.3	Resolver Client Interface	92
8.2.4	Storing DSA Key-Signing Public Keys	93
8.3	Key Server	96
8.3.1	Software Dependencies	96
8.3.2	Architecture	96
8.3.3	Key Registration	97
8.3.4	Key Retrieval	99
8.4	Client Library Application Programmer Interface	100
8.4.1	Key Query	101
8.4.2	Key Registration	102
8.5	A Simple Client	102
9	Conclusion	104
	Bibliography	107
A	Service Interface Descriptions (WSDL)	116
A.1	Key Query Service	116
A.2	Key Registration Service	120

B	Key Server Database Schema	125
C	Client Library API Definition	127
C.1	Key Query	127
C.2	Key Registration	129
C.3	Service Location	133
C.4	Low-Level DNS Resolver	133

List of Tables

7.1	Key Query Parameters	72
7.2	Query Response Header Contents	74
7.3	Query Response Key Record Contents	77
7.4	Key Registration Parameters	81
7.5	Key Revocation Parameters	83

List of Figures

3.1	Obligatory DNS tree diagram	18
5.1	System Architecture Diagram	43
5.2	Key Query Process	46
5.3	Client Query Confirmation Dialog	49
5.4	Key Registration Process	50
5.5	Client Registration Confirmation Dialog	52
7.1	SHA-1 Input from Key Query Response Key Record (ASN.1)	78
7.2	SHA-1 Input from Key Query Response Revocation Record (ASN.1)	78
8.1	DNS Query Results	89
8.2	DNS Query Results with DNSSEC Key and Signature Records	90
8.3	DNSSEC Resolution Actors	91
8.4	DSA Public Key DNS Records	94
8.5	Generating DSA Public Key DNS Records	95

8.6 Server Architecture Layer Diagram 97

8.7 Client Library Layer Diagram 100

Chapter 1

Introduction

“No right of private conversation was enumerated in the Constitution. I don’t suppose it occurred to anyone at the time that it could be prevented.”

Whitfield Diffie

1.1 The Need for Digital Privacy and Authenticity

As networking and miniaturization technologies converge to facilitate personal and ubiquitous computing [64], the need for ubiquitous verifiability and privacy of digital communications is larger than ever. Messages of all types would benefit from privacy and authentication — improving existing networked applications and enabling new applications yet to be considered.

Additionally, 2004 saw the first arrest publicly attributed to passive email monitoring [3].

Long presented as a hypothetical scenario by privacy advocates, theory has suddenly become reality. While large-scale message interception is hardly a new phenomenon¹, the increased funneling of communication into the digital domain, coupled with the relative ease of searching even large volumes of digital content for “interesting” patterns, increases the need to take appropriate measures to secure private communications. Due to the nature of most deployed Internet protocols, any node along the path from source to destination can observe (and alter) message content without detection.

1.2 Simplifying End-To-End Privacy

While cryptography has been successfully deployed to mitigate passive eavesdropping, application to Internet protocols has, in part, been hampered by a lack of infrastructure focused on simplifying construction of cryptographically enabled tools. Cryptography has been most successfully deployed in protocols, such as SSL/TLS [46, 28], and SSH [127], where a clear client-server relationship exists. While proposals exist for securing less hierarchical applications, such as the Privacy Enhanced Mail [78, 70, 10, 68], and S/MIME [53] specifications for securing email, there has been little adoption by the Internet community at large.

In recent years attention has begun to shift to problems of usability and adoption — in cryptography and security-focused software systems specifically — and systems software and configuration in general. Analysis and observation have shown that security and cryp-

¹Interested readers are encouraged to read [12] and [11] for details from America’s recent past.

tographic software systems assume too much user knowledge [58], have poor user interfaces [124], and often (unintentionally) misuse cryptography [4], undermining any achieved gains.

1.3 Problem Statement: Steps Toward A Solution

In this work we focus on a capability that is necessary to ensure pervasive application of cryptography: **simple, scalable, authenticated key distribution**. We describe a practical, deployable, architecture for providing scalable and application-agnostic public key distribution in cooperation with Secure DNS (DNSSEC). We provide a general key management infrastructure to a variety of applications by being flexible and extensible with respect to the specific sorts of key data stored and served. Additionally, we strive to simplify deployment and aid adoption by strategically leveraging existing infrastructure. The goal of this work is to help bridge the state of the art with the state of practice, and to enable pervasive end-to-end encryption and data authentication for networked applications.

Our results are in two parts. First, we specify concise key registration and query protocols which, in concert with DNSSEC, permit users and applications to perform key registration, key location and key authentication. Second, we design, construct and examine a prototype implementation, guided by HCI and usability concerns, to demonstrate deployment and use.

1.4 Why Key Distribution?

A large body of work exists in cryptography — and while many seemingly esoteric problems, such as group key agreement, have received much attention by researchers, relatively little effort has gone into making widespread use of cryptography feasible. Phil Zimmerman’s release of PGP in 1991 is arguably the last major effort in this direction; but 13 years of experience has shown that a minuscule portion of the Internet population uses, or is even aware of, this powerful privacy tool.

Observers have pointed out that despite the failed efforts of crypto-evangelists who touted cryptography as a goal unto itself, cryptography is becoming “a background feature of collaborative workspaces” [109]. As more application developers look for ways to integrate encryption into communication tools, the problem of key distribution and authentication is being repeatedly solved in an ad-hoc manner. It is our intent to provide a general, reusable solution; allowing application developers to focus on the specifics of their application, rather than re-inventing cryptographic key distribution.

1.5 Roadmap

The remainder of this document is organized as follows: Chapters 2 and 3 provide necessary background for understanding our work and contribution — a brief overview of critical cryptographic concepts, the fundamentals of the domain name system (DNS), and a brief

look at previous and current efforts to secure DNS. Chapter 4 examines other approaches to the problems of key distribution and authentication - surveying the in-band key distribution used by secure socket layer (SSL) and secure shell (SSH) protocols; dedicated key distribution services such as the MIT PGP key server; and efforts at distributing keys via directory services such as X.500, LDAP, and DNS.

Chapter 5 gives a high-level view of our proposed solution and explores the design philosophy and constraints, before walking through the two major use cases motivating the design. Chapter 6 delves deeper into the protocol design — detailing the server location mechanism, message marshaling and transport, and authentication during key registration. Chapter 7 describes the client-server message exchange during key lookup, key registration, and key revocation. Chapter 8 describes our proof-of-concept implementations of the client and server components of the scheme, as well as briefly describes configuring necessary prerequisites — notably DNSSEC support for the Berkeley Internet Name Daemon (bind). In Chapter 9 we review our findings, summarize our contributions and work still ahead, and offer conclusions.

Chapter 2

Cryptography Primer

“gentlemen do not read each other’s mail”

Henry Stimson, former US Secretary of State

There are a few key cryptographic concepts that must be understood to appreciate our contribution. This chapter provides brief overviews of symmetric and asymmetric cryptography, digital signatures, key authentication, and key revocation. Aside from the basics presented here, we assume only a rudimentary understanding of cryptographic concepts.

Readers well-versed in cryptographic concepts such as public-key cryptography, digital signatures, and key authentication may proceed directly to Chapter 3, which discusses DNS and DNSSEC.

2.1 Symmetric and Asymmetric Ciphers

All cryptosystems, or ciphers, provide the same fundamental capabilities — namely: given a plaintext P , an encryption key K_e , and an encryption function¹ E , applying E to the plaintext P with key K_e produces a ciphertext, C ; i.e.

$$E(P, K_e) \rightarrow C$$

Decryption is similar — given a ciphertext C , a decryption key K_d , and a decryption function D , applying D to ciphertext C with key K_d produces the plaintext P ; i.e.

$$D(C, K_d) \rightarrow P$$

In a traditional symmetric cipher, the same key is used to both encrypt and decrypt — that is $K_e = K_d$. The need for participants to share keys prior to secure message exchange is highly problematic. The problem becomes how to communicate the shared key securely so as not to undermine the very purpose of encryption.

Public key cryptosystems, also called asymmetric cryptosystems, provide a significant improvement over traditional symmetric cryptosystems. Public key cryptosystems first appeared in the literature in the mid 1970's when Diffie and Helman's innocuously titled

¹We intentionally ignore security parameter selection and key generation in this discussion, interested readers should consult a cryptography text, such as [105], for details.

“New Directions in Cryptography” was published in *IEEE Transactions on Information Theory* [29].² Diffie and Hellman described a method, based on modular arithmetic, for two parties to securely compute a common key while passing all necessary information over an insecure channel and leaking no information to a listening adversary. Shortly thereafter, Rivest, Shamir and Adelman published their own public key cipher — based on similar principles — which was to become known as RSA [99].

In a public key cryptosystem, the encryption and decryption keys are different — that is $K_e \neq K_d$. Further, knowledge of K_e provides no knowledge about K_d , allowing K_e to be distributed publicly — hence “public key” — without weakening the security offered by the cryptosystem. In the 30 years since its development, public key cryptography has been well studied — literally hundreds of public key cryptographic schemes have been proposed by cryptographic researchers.

A significant problem that public key cryptosystems do not directly solve is *key authentication*: a participant, upon receipt of a public key reported to be from a specific individual, must verify that the key truly belongs to the party claimed, i.e. that the key-to-name binding is accurate. An example of this problem can be seen by considering a so-called *man-in-the-middle* attack. An attacker interposes himself between two parties wishing to securely communicate and defeats their security by providing both sides with public keys for which he

²Recent revelations from GCHQ (British Intelligence) indicate that they had independently developed public key methods a number of years before the Diffie-Hellman result but chose not to publish their findings for reasons of national security [37, 22, 125].

holds the corresponding private keys. By performing the necessary decryption/re-encryption, the attacker is able to hide his presence from the two communicating parties, while gaining full control over the content of their messages.

A number of approaches to mitigating the key authentication problem exist. Explaining and contrasting solutions requires a brief discussion of secure hash functions and digital signatures.

2.2 Digital Signatures

While public key cryptography would seem to be sufficient for private message exchange, it turns out to be insufficient in one critical respect. Imagine Alice wants to send a private message to Bob. She obtains Bob's public key³, encrypts the message using that key, and sends it to Bob. Upon receipt, Bob recovers the message by decrypting using his private key. Unfortunately, since Bob's public key is public, Bob has no inherent reason to believe the message he's just decrypted actually originated with Alice.

Alice and Bob need something to attest to the authenticity of a document — a digital analogue to a signature in the physical world; a digital signature. At a high level, a digital signature scheme consists of two operations, **sign** and **verify**. Given a message M , a signing key K_s and a signature function **SIGN**, applying **SIGN** to the message M with signing key

³How, precisely, Alice obtains Bob's public key is often relegated to a footnote such as this one, in which this problem is defined to be outside the scope of interest. This is precisely the oversight we intend to remedy.

K_s produces a signature $S_{(M,K_s)}$.

$$\text{SIGN}(M, K_s) \rightarrow S_{(M,K_s)}$$

Verify is the logical complement of sign. Given a message M , a signature $S_{(M,K_s)}$, a verification key K_v , and a verification function **VERIFY**, applying **VERIFY** to message M , signature $S_{(M,K_s)}$, and verification key K_v results in a boolean success value.

$$\text{VERIFY}(M, S_{(M,K_s)}, K_v) \rightarrow (\text{success} \mid \text{failure})$$

Armed with such a signature scheme, and appropriate public/private signing keys, imagine Alice again wants to send a private message to Bob — and she wants Bob to be able to confirm that the message, in fact, came from Alice. Alice again obtains Bob's public encryption key. Prior to encrypting the message to Bob, she first digitally signs the message using her private signing key, then encrypts the message and resulting signature using Bob's public key. The resulting message is sent to Bob. Upon receipt, Bob decrypts the message using his private key and finds a message reporting to be from Alice with an accompanying signature. Bob uses the included message and Alice's public verification key to verify the included signature. Only someone in possession of Alice's private key (hopefully Alice) and the message Bob received could have generated a correct signature.

2.3 Secure Hash Functions

The last necessary cryptographic building block we describe are secure hash functions. All hash functions have the property that given an input (sometimes called a pre-image) M of arbitrary length, and a hash function H , applying H to M produces a fixed length output h .

$$H(M) \rightarrow h$$

A secure, also referred to as one-way, hash function has three additional properties [105]:

1. Given M , it is easy to compute h .
2. Given h , it is hard to compute M such that $H(M) = h$.
3. Given M , it is hard to compute another message, M' , such that $H(M) = H(M')$.

Additionally, it is typically desirable for a secure hash function to be *collision-resistant*; that is, it is difficult to find two random messages, M and M' , such that $H(M) = H(M')$.

Referring back to the discussion of digital signatures in Section 2.2, it is worth noting that in practice, rather than signing the entire message, often a hash, or *digest*, of the message is signed. If the digest is created using an appropriate one-way hash function, this provides comparable security with much less computation required during verification.

One-way hash functions can also be used in conjunction with symmetric keys to achieve a form of digital signature which can be verified only by those in possession of the appropriate

key. Conceptually, the hash function H becomes a function of both the message M and the key K . In practical terms, this can be achieved by concatenation of the key and message prior to hashing. This use of one-way hash functions is known as a message authentication code (MAC).

2.4 Key Authentication

Key authentication is the process of validating the binding of a cryptographic key to a named entity. Recall from Section 2.1 that while public key cryptosystems simplify key distribution, they do not magically solve the problem of key authentication, as demonstrated by the existence of man-in-the-middle attacks. The two most widely deployed solutions to the key authentication problem are the *web-of-trust* approach, as exemplified by Pretty Good Privacy (PGP) [129], and the certifying authority model, as exemplified by the Secure Socket Layer (SSL) [46] and its successor, Transport Layer Security (TLS) [28].

2.4.1 Certifying Authorities

The certifying authority (CA) model works on the idea that there are a limited number of highly trusted individuals (or organizations) in the community. When a new key is received, it is accompanied by an assertion (digital signature) from the appropriate trusted certifier that the provided key is associated with the named individual. All participants wishing to verify keys signed by a given certifying authority must somehow obtain (and trust) that certifying

authorities public key. In practice, a small set of so-called **root certificates** — public keys for various recognized certifying authorities — are typically pre-loaded into the cryptographic application during installation.

2.4.2 Webs Of Trust

The web-of-trust model, in contrast, essentially relies on peers vouching for the validity and trustworthiness of other peers. No individual in the community is inherently more important than any other. Given an unfamiliar key, the recipient is also given a list of affirmations (digital signatures) from community members who assert that the provided key is associated with the claimed identity. Only if enough verifiable affirmations come from individuals that the recipient trusts is the association between the key and named individual created. Researchers have examined the characteristics of these sorts of trust systems in different contexts, including large-scale distributed systems [96] and ad-hoc networking [63].

2.4.3 Identity Based Encryption (IBE)

Identity-based cryptography — a field of cryptography research spawned by Adi Shamir in 1984 [108] — provides a solution to the key authentication problem by allowing participants to derive an entity's public key directly from that entity's name. In an identity-based cryptographic system, there exists a trusted participant — the private key generator — who is capable of generating any entities private key using the combination of the entities name, sys-

tem parameters and a system secret. Identity-based cryptosystems require this set of system parameters be learned (in an authenticated manner) by all participants. Further, the built-in reliance on key escrow prevents identity-based cryptography from being globally applicable.

Recently, a group at Xerox PARC published work that describes a domain-level key-distribution scheme [112] using the identity based encryption scheme by Boneh and Franklin in 2001 [17]. Since their proposed solution leverages identity based encryption, which inherently implies key escrow, they side-step the problem of key registration. Requiring key escrow is often undesirable or unacceptable, and even careful and correct implementation carries significant risks [2].

2.5 Key Revocation

In a perfect world, private keys remain private and public keys expire before they can fall to cryptanalysis. Reality, unfortunately, is rarely so elegant. Private keys are occasionally lost or compromised — and a mechanism must exist to warn potential users of these compromised keys. The process of “un-publishing” a key is known as key revocation, and is a necessary function of any practical cryptosystem.

In principle, a key revocation is simply a message that states the public key in question is no longer trustworthy and should be discarded. Key revocation procedures depend heavily on the key authentication scheme being used.

In the Web of Trust model, the owner of the private key may issue a key revocation

certificate. Each revocation certificate is a stand-alone object, and may be distributed by whatever methods are used to distribute public keys. To prove that the publisher of the key revocation message (often called a key revocation certificate) has the right to revoke the key in question, the revocation is signed with the private half of the public key being revoked. Note that in the event that a private key is lost, rather than compromised, issuing a revocation certificate becomes impossible. Participants who have retrieved the key revocation certificate consider all signatures on public keys created with the revoked private key invalid.⁴

In the X.509 certifying authority model, there are two private keys that may be compromised — the certificate-holder and the certifying authority. If the certificate holder's private key has been compromised, the certifying authority adds the affected certificate to its certificate revocation list (CRL), which enumerates all revoked certificates issued by that certifying authority. Clients are responsible for checking each key against the appropriate authority-provided CRL prior to use.⁵ The Online Certificate Status Protocol (OCSP) defined in [87] is intended to facilitate this check in real-time. If the certifying authority private key is compromised, the situation is more dire: all certificates issued using the compromised key must be replaced, and the public half of the compromised key-pair removed from all instances of client software in which it exists.

⁴Note that this removal of signatures from other public keys can, in principle, cause other public keys to fall below the configured trust threshold.

⁵In practice, many clients, including version 1.6 of Mozilla and version 6 of the Microsoft Internet Explorer web browsers, as tested by the author, ship with this check disabled

Chapter 3

The Domain Name System (DNS)

“To call up a demon you must learn its name. Men dreamed that, once, but now it is real in another way...”

William Gibson; *Neuromancer*

In the early days of the Internet, participating hosts were identifiable primarily by their network address — a sequence of bytes unique to the machine being addressed. Though functional, addresses were difficult to remember and generally not user friendly. For a time a mapping table from network address to more human friendly name was maintained and published by SRI.¹ In 1983 Paul Mockapetris published a Request For Comment (RFC) entitled *Domain Names — Concepts and Facilities* [84] which described the architecture for a distributed, scalable name translation service.

This work leverages DNS heavily, so a more detailed introduction follows. Readers fami-

¹Formerly known as the Stanford Research Institute.

lar with the structure and workings of DNS may choose to proceed directly to the discussion of DNSSEC in Section 3.4.

Mockapetris proposed a name space in the form of a hierarchical rooted tree, divided into zones that could be managed by different organizations. Further, he defined a query protocol for clients to translate names into resources — a process called “name resolution.” Resources were typically addresses and other names, though several resource types were defined in the specification, a relevant subset of which is discussed in Section 3.2. Though the details and implementation have evolved over the last 20 years, the core concepts in the Domain Name System — typically referred to only by the acronym DNS — has changed relatively little.

3.1 The DNS Namespace

The domain name system specifies a hierarchical name space structured as a rooted tree. A portion of the name space is illustrated in Figure 3.1. In DNS, both interior nodes and leaves contain useful information. A DNS name is the ordered collection of name parts obtained by following the path from the the node of interest up to the root of the tree, delimited by dots (“.”). The root of the tree, from which all names originate, is simply “.” and is often omitted when describing domain names. Following the bold links and shaded nodes in Figure 3.1 results in the name *www.cs.ucr.edu*.

Each node in the tree may be managed by a different organization, and in recent years there are many nodes which are further subdivided and are themselves managed by more than

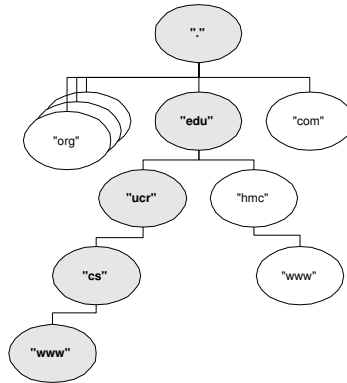


Figure 3.1: Obligatory DNS tree diagram

one organization. The details of this division of authority are not important, the key concept is that a node closer to the root (for example *edu* in the figure) can delegate authority for a subtree rooted at it to another node (e.g. *ucr*).

3.2 DNS Record Types

While DNS is often thought of as simply a way to map names to addresses, the original DNS specification included a set of fourteen different resource types which DNS could manage [85]. This list has mutated over the years as features have been added and deprecated. We briefly discuss a handful of record types in common use and relevant to our work.

- Address (A) Records

As their name suggests, a DNS A record stores the network address of a named resource. DNS was designed with the capability to house different forms of addresses — originally ARPA Internet and CSNET — and the type of address is both stored in the

DNS database and returned to a requesting client. This mapping from name to address is typically referred to as a “forward-mapping,” and its resolution is a “forward-lookup” or “forward-resolution.”

- Canonical Name (*CNAME*) Records

Quite often a single host is referred to by multiple names, or aliases. The *CNAME* record maps an alias to its canonical name. When a requesting client looks up a named resource and obtains a *CNAME* record as the response, it repeats the resolution process on the returned *CNAME* to obtain the actual data of interest.

- Mail Exchange (*MX*) Records

Email, being one of the earliest and most persistent “killer-apps” on the Internet, warranted special treatment in DNS. An *MX* record maps a name to a set of prioritized *A* and/or *CNAME* records identifying hosts capable of handling email for the domain or host of interest.

- Pointer (*PTR*) Records

It is often useful to answer the question “*what name is associated with the following address?*” The *PTR* record provides a pointer from an address back to a name. Since the maintainer of the IP address space (typically an Internet Service Provider) is a distinct entity from the maintainer of the DNS name space (typically a customer of the Internet Service Provider), the *PTR* records mapping a given network address to

a name are typically not managed by the same organization. These mappings from address to name are typically referred to as a “reverse-mapping”, and resolution from address to name is known as a “reverse-lookup” or “reverse-resolution”. *PTR* records will be revisited during the discussion on securing DNS in Section 3.4

3.3 DNS Query Resolution

When a client (known in DNS terminology as a “resolver”) wishes to translate a name into some form of resource location information (called a “resource record”), it breaks the name into parts — appending the implicit “.” if necessary. Beginning at the root (“.”), the client walks down the tree toward the name of interest, following delegation links from each level of the tree to the one below it, finally finding a DNS server which is **authoritative** for the name being resolved and can return the requested information, or state with authority the requested record does not exist.

For example, consider a client wishing to translate *www.cs.ucr.edu* into its network address. The client asks one of the DNS servers responsible for the root (“.”) “*I’m looking for ‘www.cs.ucr.edu.’ - who is responsible for ‘edu.’?*” The root server consults its database and responds by directing the client to a server with authority over the *edu* portion of the tree. The client repeats the procedure — connecting to a server responsible for *edu* and asking who is responsible for *ucr.edu.*, finally connecting to the server responsible for *cs.ucr.edu.* that can authoritatively answer queries regarding *www.cs.ucr.edu.*

Notice that there are multiple lookups to multiple servers involved with the seemingly simple task of resolving a single name. DNS employs pervasive caching to reduce the number of actual server-queries required.

3.4 Efforts to Secure DNS

Security was not a primary consideration during the design and implementation of DNS. As the Internet has grown — both in terms of user population and in terms of the number of participating hosts — DNS has become an absolutely critical component of the Internet infrastructure. As of 2001 the DNS root servers were handling a peak load of over 5000 queries per second [19].

In 1995, the USENIX Security Symposium featured two key papers regarding DNS security. Steven Bellovin published a paper entitled *Using the Domain Name System for System Break-ins* [13] and Paul Vixie published *DNS and BIND Security Issues* [120]. Bellovin’s paper had been written five years earlier but voluntarily withheld from publication until the security flaws he described had been somewhat mitigated. In the paper Bellovin described attacks made possible by a combination of poor authentication and authorization techniques and inherent limitations of the DNS.

The fundamental problem discussed by Bellovin related to “name-based” authentication and authorization. At the time it was commonplace for systems to trust incoming connections based almost entirely on the (DNS-supplied) name of the connecting system. By taking

advantage of the disconnection between forward mappings (recall from Section 3.2 that a forward mapping is from name to network address and a reverse mapping is from network address to name) and reverse mappings, and exploiting the trusting nature of the client software, an attacker could appear to be connecting from any arbitrary name of his choosing and easily gain unauthorized access to the target system. Further, the obvious solution — to check that the forward and reverse mappings appear consistent — fell prey to a relatively simple cache poisoning attack.

Bellovin’s conclusion was that a healthy dose of skepticism and cryptographic authentication would go a long way towards eliminating the described threat. Due to the design and implementation of DNS, responses from the naming service had to be treated as suspect, as there was no mechanism to authenticate or verify the retrieved data.

At the same conference, Paul Vixie — one of the maintainers of BIND, the de-facto standard DNS server implementation — published work in which he detailed a number of improvements made in, and planned for, BIND to help mitigate some of the identified threats against DNS in general and BIND in particular. Vixie acknowledged, however, that working within the constraints of the existing DNS design precluded certain solutions, and stated

“we are counting on the IETF DNSSEC effort to bring us a DNS protocol revision that authoritatively signs responses. . . . Until DNSSEC is finished and in wide use, there are some things we’re just going to have to live with.”

The effort Vixie referenced is an Internet Engineering Task Force (IETF) effort to se-

cure DNS. Launched in late 1993 by members of the DNS working group, the DNSSEC group published their first request for comment (RFC) regarding DNS security in 1997 [36]. Over 10 years after formation, the DNSSEC working group proposal is nearing operational readiness, bringing with it the promise of a trustworthy name service.²

3.4.1 DNSSEC Overview

DNSSEC is a collection of proposals intended to secure the data stored in DNS. Using cryptographic techniques, queries and associated responses can be strongly authenticated by the server and requesting client respectively — greatly reducing the potential for abuse present in the current DNS. Atkins and Austein have written an IEFT draft enumerating the threats DNSSEC is intended to guard against [7]. We focus here on the portions of DNSSEC relevant to our work. Interested readers can find a more detailed overview in [5].

Zone Signing

DNSSEC offers two fundamental improvements over traditional DNS: data origin authentication and data integrity verification. The idea is that a DNSSEC-enabled DNS server responsible for a given domain (referred to as a “zone”) cryptographically signs the zone’s resource records with a public/private key pair bound to that zone³, and includes crypto-

²As one might expect, there are skeptics including [14].

³Since the signing key is not associated with a particular DNS server, but with the zone, compromise of a secondary DNS server, which does not have a copy of the signing key, or of a primary server, if the signing key is kept offline, does not permit tampering with zone data.

graphic signatures in responses to clients.

Resource record signatures are stored in a newly defined DNS record type called an *RRSIG*. A *RRSIG* record contains a cryptographic signature, which authenticates a specific named set of resource records (known as an “RRSet”) for a specific duration. Each named resource in a secured DNS zone will have at least one *RRSIG* record associated with it.

When a DNSSEC-enabled, or “security-aware”, client requests a DNS record for a given name from a DNSSEC-enabled server, the server will return the requested record along with the associated *RRSIG* record. The client obtains the public key associated with the zone containing the retrieved record and verifies the provided signature. If the signature is valid, the client can trust that the response was provided by the authoritative source.

A more subtle issue exists if the requested resource record does not exist in the queried zone. Rather than returning a simple “no such record” response — which could be easily replayed if static, or would require the zone signing key to be kept on-line if dynamically generated — the server pre-computes and signs responses that indicate the “gaps” in the zone. By employing these *NSEC* (next secure) records, a client can obtain authenticated proof, valid for a specific duration, that a requested name does not exist in the queried zone.

One of the requirements during the development of DNSSEC was that security not come at the cost of caching. A caching name server can cache and replay signed responses; as long as the data has not been tampered with, and the time-to-live not exhausted, clients can verify that the response obtained from a caching server was generated by the authoritative source.

There is ongoing discussion regarding how a resolver should indicate the authenticity of retrieved data to client applications [123, 51].

Key Distribution in DNSSEC

In order for a client to verify signatures provided with query responses, the client must have either been statically configured with, or be able to obtain and authenticate, the public key for the queried zone (the zone key). To facilitate distribution of these public keys, DNSSEC defines a *DNSKEY* resource record type. Interestingly, the *KEY* resource record — the predecessor to the *DNSKEY* type — was originally intended as a general purpose public key distribution mechanism [31] but was subsequently restricted to holding only DNSSEC keys [80] for reasons discussed in Section 4.3.2.

A DNS client can query for a zone key in the same way it queries for any other DNS record type. Unlike other resource record types, it makes little sense to sign a public zone key with the private component of the same key. Instead, the zone signing keys are signed by the parent domain⁴. In order to authenticate the retrieved key, the *DNSKEY* chain must ultimately be signed by a key that the client has previously authenticated. By recursively requesting keys while moving up the DNS name hierarchy, the client will either reach a trusted key, or exhaust the name space without reaching such a key — causing the current key authentication attempt to fail.

⁴While this description is conceptually sufficient, it is not technically faithful. Detail seekers should refer to [56].

3.4.2 DNSSEC Implementation Status

Recently, DNSSEC has matured from a set of related RFCs to an implementable system. DNSSEC has been deployed in a medium scale [50] and the signing hierarchy has been revised based on this operational experience [56]. Additionally, a handful of documents considering operational and security concerns have been written and published [30].

An IETF draft exists which updates RFC 2535 and details the protocol changes required to support DNSSEC [6]. The current release candidate version of BIND [119] (9.3) supports the designated signer [56] structure, which replaces the RFC 2535-based implementation found in the current stable version (9.2.x). A DNSSEC deployment working group has been formed with support of NIST and ICANN, and in April of 2004, Steve Crocker and Russ Mundy published an invitation to interested parties to build a road map for DNSSEC deployment [26]. Consensus is growing that DNSSEC is largely ready for deployment, and that 2005 may see the beginnings of wide-spread adoption.

Chapter 4

Previous Approaches to Key Distribution

“If I have seen further, it is by standing on the shoulders of giants.

Sir Isaac Newton (1642-1727)

This work is not the first to look at the problem of key distribution, other approaches have been proposed and some are in use today — either within a single organization, or for a specific cryptographic application. Here we briefly survey previous approaches and work.

4.1 In-Band Key Transmission

Perhaps the most common approach to key distribution is to relegate it to the communication protocol. The secure shell (SSH) [127], secure socket layer (SSL) [46] and its successor the transport layer security protocol (TLS) [28], transmit the necessary public key during connection setup, but use different schemes for authenticating the received key.

4.1.1 Secure Shell (SSH)

In the case of SSH, the initial key authentication is performed by asking the user if they wish to associate the provided key with the named host. If the response is affirmative, a hash of the public key (typically called a “key fingerprint”) is stored locally. Provided the key supplied by that host during subsequent connections matches the stored fingerprint, no further user intervention is required.

The primary issue with this approach is that it assumes that the end-user will know the appropriate key fingerprint during initial connection setup. While this approach does limit the window for a successful man-in-the-middle attack to the initial connection, it does not completely eliminate the threat.

In support of SSH, where the trust relationships are fairly static (users tend to repeatedly connect to the same small set of hosts), this is often an acceptable level of risk mitigation. In the case of end-user to end-user communication, however — where the relationships are far more dynamic — the requirement that key identifiers be transmitted and confirmed via a manual, out-of-band, process is not viable.

4.1.2 Secure Socket Layer (SSL)/Transport Layer Security (TLS)

SSL/TLS uses the certifying authority model, and the connecting client is provided with the servers’ certificate, signed by one or more certifying authorities. Recall from Section 2.4 that clients (such as web browsers) are typically pre-configured with a number of root certificates.

If the certificate provided by the server has been signed by one of the statically-known certifying authorities, the connection is established without user intervention. In principle the SSL/TLS model permits intermediate signatories, in which the host certificate is validated with another certificate that is, in turn, validated by a trusted root certificate. However, this is rarely used in practice.

Verisign, in their corporate introduction to public key cryptography [118], states:

“Users of RSA technology typically attach their unique Public Key to an outgoing document, so the recipient need not look up that Public Key in a public key repository.”

They go on to point out the issue of authenticating keys received in this fashion, and propose that a certificate — signed by a certifying authority (preferably Verisign, of course) as the appropriate solution.

4.2 Dedicated Public Key Distribution Services

Not all cryptographic applications are suited to in-band key distribution. A number of application-specific key distribution services have been built, probably the best known being the MIT PGP key-server at <http://pgp.mit.edu>. Written by Marc Horowitz as his advanced undergraduate thesis [61], it was the first public PGP key server not implemented by using PGP’s internal keyring management features. GnuPG [95] — an RFC compliant

OpenPGP implementation [20] — has built-in support for locating keys in a public PGP keyserver and adding them to the local keyring.

Relatively little scholarly work exists in this area. A University of Michigan Technical Report written in 1998 described a scheme for a hierarchical set of certificate servers [81] similar in capabilities to the certification authority requirements outlined in the Privacy Enhanced Email (PEM) specification [70]. They describe their design, which is based on a well-connected trust graph, and examine its behavior under hypothesized load. It is unclear from their presentation if they have considered operational issues such as off-line signing keys and heterogeneous keys.

4.3 Distribution by Directory Service

The idea that public keys are just another attribute of a named entity leads to the idea of storing keys in a directory service. The two most notable efforts involve X.500 and DNS, and are briefly explored here.

4.3.1 X.500

In the late 1980's the ISO and CCITT (now known as ITU) released a lengthy set of recommendations for building distributed replicable directory services under the umbrella name X.500 [66] and RFC 1487 proposed a simplified “lightweight directory access protocol” (LDAP) in 1993 [126].

Today, X.500/LDAP directories are fairly common in large organizations. PKI (Public Key Infrastructure) solutions — a much heralded technology suite in the mid 1990’s — typically employ a centrally managed directory for hosting user information, including public keys.

Configuration and maintenance of an X.500/LDAP directory is perceived as difficult and complex. Although standard schemas exist for a wide variety of object types — including X.509 certificates [16] — implementors often ignore, or are unaware of, these standards. This results in directories that address the issues of the implementing organization, but are not interoperable across administrative domains. Additionally, the LDAP protocol is often not permitted across network boundaries, resulting in disconnected islands of information.

Perhaps most damagingly, this complexity is not isolated to the implementor and administrator: users wishing to search an X.500/LDAP directory — assuming they have a search tool — must specify appropriate values for unfamiliar terms such as “Search Base” and “Search Scope.” Correct values are required to obtain useful search results, and in most tools guidance goes no further than “consult your system administrator.” These and other factors have prevented X.500/LDAP from becoming a practical Internet-wide key distribution tool.

4.3.2 DNS

Over the years, efforts have been made to standardize storing X.509 certificates [35] and keys of various types [32, 35, 33, 34, 104] in DNS. Recently, Mark Delany at Yahoo! has

submitted a draft to the IETF entitled *Domain-based Email Authentication Using Public-Keys Advertised in DNS (DomainKeys)* [27] which uses DNS to distribute public keys for the purpose of authenticating email delivery.

The FreeS/WAN Project [94] — an open source IPsec [71] implementation — supports what they call “opportunistic encryption.” By automatically retrieving a host’s public keys from DNS, end-to-end IPsec encryption could be setup without user intervention. In March of 2004 the FreeS/WAN project ceased active development, citing the poor adoption of opportunistic encryption as a major contributing cause¹.

James Galvin, one of the early contributors to the DNSSEC effort, published a paper in USENIX '96 that provided an overview of DNSSEC and briefly discussed the potential for using DNSSEC to distribute end-user public keys [49].

Of particular interest to this work is an informational RFC² published in 1997 entitled *Key Exchange Delegation Records for the DNS* [8]. The author describes a key exchange (KX) record with semantics similar to the mail exchange (MX) record discussed in Section 3.2. The intent was to permit IPsec endpoints to delegate key exchange authority to other nodes. The author briefly describes the potential for this mechanism to delegate authority to a key distribution center.

¹We would argue that while they solved half the problem — simplifying key lookup — they did not adequately address key registration — which may have stymied potential users.

²An informational RFC, as the name suggests, is not proposed as a standard for widespread implementation and adoption.

Why Key Distribution via DNS Doesn't Work

It would seem, at first blush, that DNSSEC makes authenticated key distribution a solved problem. As mentioned in Section 3.4.1, the *KEY* record defined by DNSSEC was originally intended to house keys of many sorts, both those used internally by DNSSEC as well as end-user application specific keys. This decision was explicitly reversed in RFC 3445 [80]. There are three primary reasons for this reversal, which we briefly explore. None of them are insurmountable, but the Internet community is, perhaps justifiably, risk-averse to changes in and around DNS; which is part of the reason that DNSSEC has taken so long to reach fruition.

Scalability

Although the original DNS RFC anticipated using DNS to house per-user information [84], the growth experienced by the Internet user population — approximately 945 million in 2004 [114] — has completely surpassed the growth in DNS-registered host systems — estimated at 230 million the same year [23].

Adding DNSSEC signature records to a zone increases the size of the zone data by a factor of 8 or 9 [50], and adding per-user keys and their signatures would further increase the size of the zone data. Current versions of BIND store zone data in flat text files, making managing zone databases with millions of entries problematic.

Finally, from a network perspective, DNS has been designed and optimized for very small

query/response exchanges — with the average response being on the order of 256 bytes and the protocol defined maximum payload of 512 bytes [86]. Returning key data (and associated signatures) in DNS responses is expected to significantly increase network load, as would zone transfers between primary and secondary servers.

Query Interface

The second reason for the reclamation of the DNSSEC *KEY* record was a mismatch between the resolver query interface and the requirements of an application seeking a particular key.

Different types of keys stored in *KEY* records were to be differentiated by subtype — a single named entity might have multiple key records, each storing a different type of key (i.e. suitable for a different cryptographic algorithm, application, or purpose). Unfortunately, the DNS resolver interface does not support query by sub-type, so the client was forced to retrieve all key records present for the named entity and find the (potentially non-existent) “right one” on the client. Since DNSSEC internally required keys retrieved from foreign servers, this affected not only applications, but the efficiency of the name service itself.

Administrative Authority

In addition to the volume and query interface issues described above, the fact that the administrative model for DNS doesn't match the requirements for managing end-user keys poses an additional hurdle to effectively distributing end-user keys via DNS.

DNS data tends to change slowly and is under the control of a domain administrator.

Allowing end-users some level of direct control over their keys — i.e. facilitating end-user initiated update of DNS key records — is problematic and breaks down the existing administrative model. Supporting dynamic DNS update in the context of DNSSEC is difficult in general; RFC 3007 [122] discusses it in detail and several researchers have contributed solutions [40, 121].

Chapter 5

Proposed Solution

Treese's First Law of Key Management: "The complexity of key management can not be lessened — it can only be moved around."

Win Treese [116]

As outlined in Chapter 4, previous efforts to provide an ubiquitous key distribution infrastructure have failed for a number of reasons. Of particular note is X.500/X.509 which, while conceptually complete, has proven difficult to implement and has failed to reach critical mass. We are hardly the first to observe that X.500 has been hampered by its complexity (see [72] for another perspective). Another barrier to adoption seems to be the name space provided by X.500: it is related, but *not* identical, to the name space which Internet end-users interact with on a regular basis (DNS). Further, the name space and access tools introduce structure and terminology unfamiliar to Internet users.

A natural reaction to this observation is piggybacking on the existing directory service

(DNS) which, for reasons described in Section 4.3.2, has the potential to compromise the stability of a critical piece of Internet infrastructure. In designing a new solution to the key distribution problem, we attempt to sail the straights between Scylla and Charybdis¹ — leveraging the DNS as appropriate while being deliberate to avoid compromising its primary function.

5.1 High Level Solution Summary

We propose a dedicated key registration and distribution service capable of storing and publishing keys bound to DNS names — the sort of names familiar to users of Internet communication tools. By using DNSSEC to securely delegate key distribution activities to a dedicated service, we benefit from the scalability and hierarchy of the name service while not placing significant additional load — in terms of query/response as well as in terms of data storage — on the DNS infrastructure.

End-clients wishing to register or retrieve a public key use DNSSEC to learn and authenticate the identity of the responsible key registration/distribution² server. All registration/query messages are directed to the identified server using a specialized protocol. Server responses can be authenticated by way of digital signatures — the public key required to validate signatures being published via DNSSEC. Any application that communicates between named

¹From Homer’s *Odyssey*, Scylla was a six-headed serpent guarding a narrow water passage, and Charybdis was a powerful whirlpool on the other side of the same passage.

²Henceforth we omit the “key” specifier unless required for clarity.

endpoints (users, named hosts, or a combination of the two) can use this key distribution infrastructure to perform mutual authentication based on the trust placed in the domains containing the endpoints, and thus establish a secure end-to-end channel.

Our proposed solution is similar to X.500 in that it places keys and associated meta-data in a hierarchical name space. X.500 has failed to gain significant traction, even in tightly-coupled corporate environments³ due to model mismatch, implementation complexity and over-ambitious goals. We aim to provide a simple solution — focused specifically on key distribution — suitable for deployment across a user population numbering in the hundreds of millions. By distributing the responsibility for key registration and retrieval to DNS-level organizations, and allowing them to differentiate between query and registration requests, we allow the system to scale and meet demand in a distributed fashion.

5.2 Design Philosophy

As with DNS, while an implementation of the server is a necessary step toward adoption, the primary contribution is captured in the overall architecture and protocols; in our case protocols for key-registration and key-query. A standardized protocol, flexible and extensible enough to satisfy varying end-user needs, helps ensure that different server and client

³Those who would argue that X.500/X.509 has been successful need only refer back to nearly 10 years of press releases by various firms including Novell [65], RSA [107], and others claiming to have finally “removed the barriers to widespread adoption” of directory services and encryption, and recent acknowledgments from industry groups such as OASIS [43] that the problem is far from solved.

implementations can exist and interoperate. More importantly, while standardizing the client-server protocol does not *guarantee* pervasive adoption, *failure* to standardize the protocol will almost certainly preclude it.

5.2.1 Requirements/Constraints

To proceed towards a design, we first enumerate the primary requirements and constraints. We focus here on requirements which directly affect the client-server protocol. Requirements internal to the implementation of the key server are deferred to the implementation discussion in Chapter 8.

1. Users must be authenticated during key registration. This authentication will be specific to a domain, and potentially to a particular application. Thus the registration protocol needs to facilitate different types of authentication. Some services will lend themselves to programmatic (automated) authentication — for example email, instant messenger, or other services accessed via a client that has (easy access to) a secret shared with the service. For example, a username and password. Other services — such as registering public SSH keys for a machine on the Internet or creating a “domain-signed” SSL/TLS certificate — may require out-of-band authentication (e.g. a domain administrator confirming that the registration request was initiated by the appropriate party).

2. As implied by Constraint 1, it must be possible for a given registration server to authenticate clients in different ways. Different services protected by the same registration server may use different authentication mechanisms or, if a common mechanism, have separate credential databases.
3. Constraints 1 and 2 notwithstanding, the protocol should simplify the common case. It is desirable that the client application be able to complete the authentication dialog with minimal or no interaction from the user. Further, the number of required network round-trips should be kept to a minimum both for latency and reliability reasons.
4. None of the key query or registration mechanisms can be application-specific. We imagine applicability to a wide range of services such as IPSec, Email, Peer-to-Peer, Instant Messaging, SSL/TLS, ssh. Anything that communicates between named entities that are naturally contained in, or can be assigned to, a DNS domain. Note that an application such as instant messaging could define user keys to be stored in the domain of the sponsor/host. For example, all AOL Instant Messenger (AIM) users could implicitly have names of the form user@aim.aol.com. In keeping with the end-to-end principle [102], we should not care what the application is; just provide a key storage and retrieval service.
5. The protocol should avoid enumerating valid options. Rather — in the interest of future-proofing — it should provide a mechanism for reliably generating canonical

versions of potential options. This can be seen as another application of the end-to-end principle: by isolating the protocol and server from details subject to change, knowledge is only necessary in the endpoints.

6. As in DNSSEC, it should be possible for intermediate nodes to cache and replay unmodified key query responses until their time-to-live is exhausted.
7. It must be possible to authenticate and validate all protocol messages from a registration or query operation. Further, it must be possible to facilitate this validation without requiring the server to have on-line access to private signing keys.
8. Changes to DNS must be avoided whenever possible. In particular we must avoid creation of new resource record types, as such a change requires additional *per-client* software deployment and re-configuration.
9. DNS *TXT* records — often used as a “catch-all” record type in DNS, are inefficient from a storage and transmission perspective. During transmission of most DNS records, repeated strings are transmitted only once. This compression — while simple — is effective, but not applied to *TXT* records. The need to parse and interpret the contained data introduces additional complexity at the retrieving client. DNS *TXT* records should be avoided when possible.
10. We need to be conscious of the differences between key-authentication guarantees expected by an end-user and the guarantees actually provided by the system. Mismatches

between the system model and user model [89] can be disastrous, a point we revisit in Section 5.4.1.

5.3 System Architecture

Distilling the high-level summary presented in Section 5.1 and filtering it through the constraints enumerated in Section 5.2.1, we can now present the overall architecture of our proposed solution.

Each DNS domain delegates responsibility for handling registration and query requests to an arbitrary named host, potentially outside that domain. This delegation is performed by the addition of resource records to the DNS zone for the domain being delegated and is under the domain administrator's direct control (detailed in Section 6.1.2).

Clients wishing to locate or register keys for a named entity within a given domain learn of the delegation by performing DNS queries and authenticating the responses via DNSSEC. Clients communicate with the specified host via SOAP datagrams transmitted over HTTP or HTTPS (detailed in Section 6.2).

To allow clients to validate retrieved responses, each server cryptographically signs keys that it distributes with a named key-signing key. The public half of the signing key (also referred to as the verification key) is published in DNSSEC. Clients use DNSSEC to securely obtain and authenticate the verification key. Using this verification key, the client can verify the signature on query results, ensuring that the retrieved key has not been modified.

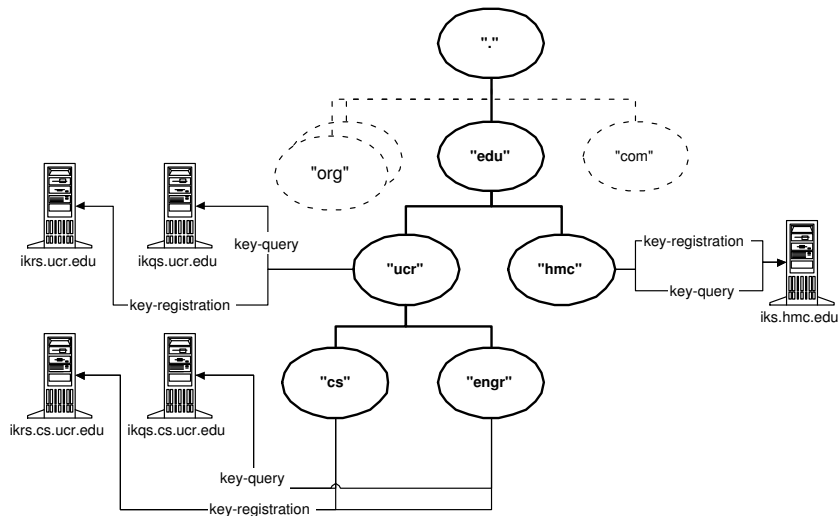


Figure 5.1: System Architecture Diagram

Figure 5.1 illustrates some of the possible forms of delegation. The domain “*hmc.edu*” has delegated both registration and query processing to a single host (“*iks.hmc.edu*”). On the other hand “*ucr.edu*” has delegated registration request handling to “*ikrs.ucr.edu*”, and query request handling to “*ikqs.ucr.edu*”. Similarly, “*cs.ucr.edu*” has delegated registration and query handling to a set of hosts, which also service registration and query requests for the “*enrg.ucr.edu*” domain. The delegation need not be one-to-one — a domain can specify a set of query or registration hosts.

It is worth noting that there is no implicit delegation in this scheme. A domain that does not explicitly publish delegation records is essentially choosing not to participate in key distribution. Requesting clients will fail to find suitable service endpoints to conduct registration or query transactions. Also, since DNSSEC provides verifiable responses — even in the case where the requested name does not exist — we can be sure that clients can

learn of attempts to block legitimate delegations.

By using DNSSEC to provide authenticated delegation, we provide a secure hand-off between DNSSEC and our key-management services, as well as a trustworthy mechanism for retrieving verification keys. The additional data placed in DNS is negligible — consisting of delegation records and verification key publication — and does not increase with the number of keys active in the system. The DNS requests required to learn of the delegation and retrieve verification keys are comparable to other service publication (e.g. publishing the location of a website or FTP server).

Our scheme is conceptually de-coupled from DNSSEC — it relies only on the presence of a trustworthy name-service, not any particular implementation. If the DNSSEC proposals undergo additional evolution, any impact on our scheme should be minimal. Further, unlike previous attempts to distribute keys via DNS(SEC), our proposal does not suffer from any of the three pitfalls enumerated in Section 4.3.2: scalability, poor query-interface, and mismatch of administrative authority.

5.3.1 Scalability

By adopting the hierarchical structure of DNS, we inherently de-centralize the task of key-distribution. Though we depend on the DNS root name servers, we do not introduce any new scalability bottle-necks. As is detailed in Sections 6.1.1 and 6.1.2, the delegation mechanism permits simple weighted load-distribution across an arbitrary set of hosts, adding to

the scalability of the system. By providing domain administrators wide latitude in distributing the work-load for handling key-management requests, we allow each deployment site to determine the approach that best fits their particular requirements.

5.3.2 Query Interface

As detailed in Chapter 6, we propose a specialized query and registration protocol which provides the appropriate level of expressiveness for key-registration and key-distribution. Clients can perform narrow searches based on relevant attributes such as key length, cryptographic algorithm, and key container format — allowing applications to locate available keys suitable for their purposes.

5.3.3 Administrative Authority

Unlike proposals to distribute keys in DNS, our proposal places minimal burden on DNS administrators and does not imply rapid changes to DNS zone data. The delegation records for a given domain are expected to be static, and publication or revocation of key-signing keys are expected to be an infrequent event, and not driven by end-user behavior.

5.4 Envisioned Use Cases

To further motivate and illustrate the proposed design, we examine the two primary use cases: fetching a key from the system, and registering a key to a name. We are intentionally light on details in these use cases, deferring to the design discussion in Chapter 6.

5.4.1 Key Lookup

Suppose Alice has a messaging application that supports our proposed key distribution service. We assume, without loss of generality, that “messaging” in this case refers to email. Alice wishes to send a private message to Bob. She composes the message using her client, and addresses it with Bob’s address — *bob@example.com*. When Alice presses “Send,” her client software attempts to find a suitable encryption key. Figure 5.2 illustrates the process.

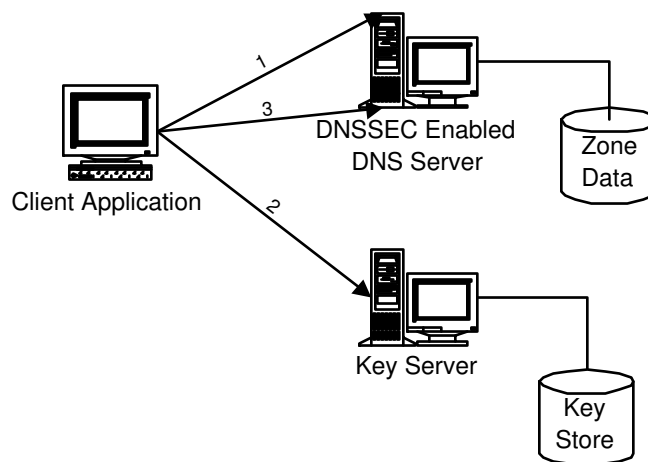


Figure 5.2: Key Query Process

Alice’s client first decomposes Bob’s address into the user part (*bob*) and the domain part

(*example.com*). The query then proceeds as follows:

1. Alice's client constructs a DNS query looking for the key-server(s) registered to handle queries for *example.com*.
2. Assuming a suitable response is obtained and can be authenticated via the DNSSEC zone-signing mechanisms described in Section 3.4.1, Alice's client constructs a query message specifying the named entity (*bob@example.com*) and appropriate query criteria (key strength, algorithm, etc.), contacts the indicated key-server, and transmits the query.
3. A query response — including any registered public keys matching Alice's provided query criteria — is generated and returned to Alice's client software. Portions of this response are cryptographically signed by the key-server with an asymmetric key pair named in the response message, the public half of which is published in DNS.
4. To validate the key(s) retrieved for *bob@example.com*, Alice's client software constructs a second DNS query — this time for the public half of the key signing indicated in the query response. Again, the information retrieved from DNS is validated using DNSSEC zone signatures. The retrieved key is used to validate the digital signature on the key(s) contained in the query response message.

Placing Trust In Query Responses

At this point we return to a topic first raised in Section 5.2.1 as design constraint 10. It is important to explicitly call out what guarantee(s) are and are not offered by the key-server signing the query response message.

In essence, the key-server signing a query response indicates

“The key distribution server for *domain* currently asserts that the key provided is bound to *name*. Further, the administrators of *domain* have performed what **they** consider reasonable verification that the key in question was provided by the individual in control of *name* or an agent operating on their behalf.”

It is equally important to call out what this signature does **not** convey. A signed query response does not indicate to the querying client any details of the verification performed during key registration. Upon receipt of a signed key query response, the user — or their agent — needs to determine if the key(s) retrieved from *domain* should be trusted. This decision must be based on the level of trust the end user places in the domain in which the remote endpoint resides.

Assuming the validation of the DNS response and subsequent validation of the query response are successful, Alice’s software might present a dialog indicating that a key is available and prompting the user to permit or deny its use (visualized in Figure 5.3).

Here the user is prompted to accept or reject the retrieved key, potentially making this decision persistent. The option is provided to apply this decision to all keys retrieved from

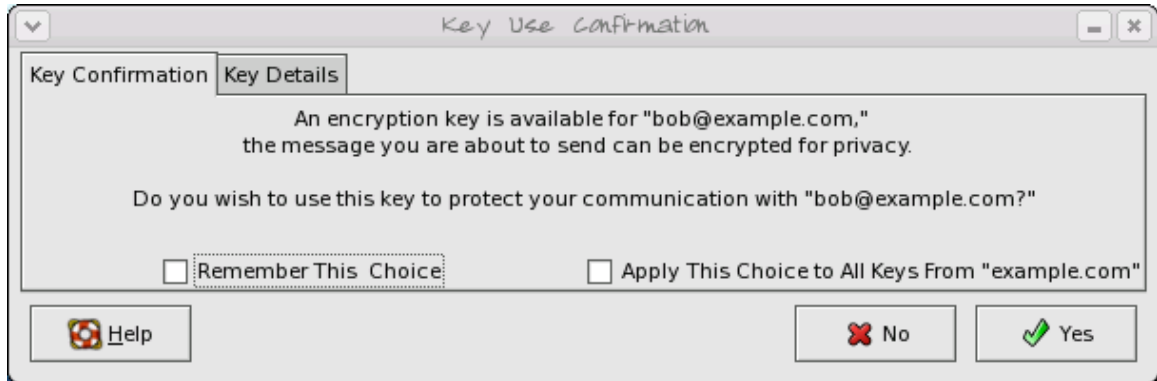


Figure 5.3: Client Query Confirmation Dialog

the domain in question (which implies persistence).

Though our scheme aims to insulate the end-user from the complexities of key distribution, the final decision to trust a key seems to require explicit (guided) action. This is similar to the approach taken by Rivest and Lampson in [98] in which binding of global names into a user's local name space is placed under the control of the end user.

5.4.2 Key Registration

Of course, for Alice to successfully retrieve public keys for Bob, there must be keys registered for Bob in *example.com*'s key server. Compared to query, registration is more challenging, primarily owing to the requirement that Bob be suitably authenticated to the key-server before being authorized to register a key bound to a specific name.

In a similar vein we now describe key registration, again being intentionally light on details and revisiting them in the design discussion in Chapter 6.

Suppose Bob has a messaging application that supports our proposed key distribution

service. Again we assume that “messaging” refers to email. Key registration could be initiated automatically (e.g. during configuration of the messaging client) or explicitly (e.g. by the user wishing to create and publish new keys). Regardless of how and why registration is initiated, the process is the same as illustrated in Figure 5.4.

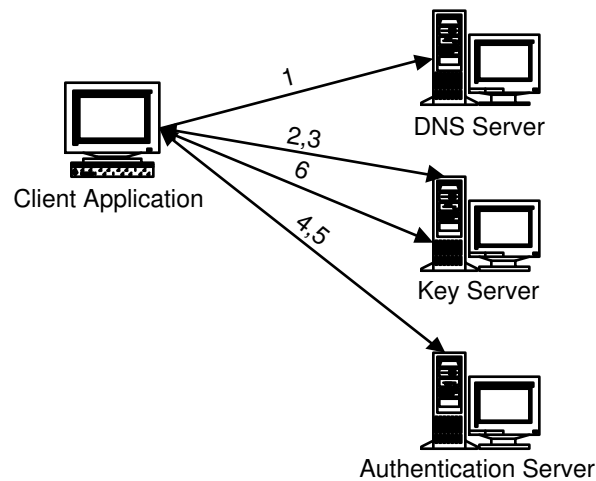


Figure 5.4: Key Registration Process

Bob’s client retrieves his address from its configuration (*bob@example.com*) — prompting the user if necessary — and decomposes it into the name part (*bob*) and domain part (*example.com*). Given a suitable public/private keypair, registration proceeds as follows:

1. Bob’s client constructs a DNS query looking for key-server(s) registered to handle registration for *example.com*.
2. Assuming a suitable response is obtained and can be authenticated via the DNSSEC mechanisms described in Section 3.4.1, the client constructs a registration message including the name and public key to be registered and required meta-data (permitted

uses, key expiration date, etc.), contacts the indicated key-server, and transmits the registration request.

3. The key-server responds to Bob's client with a registration challenge, potentially directing him to an external authentication server.
4. Bob's client marshals the credentials required to authenticate his claim to the name *bob@example.com* — prompting Bob for these credentials if necessary — and creates an appropriate response. The challenge response is sent to the authentication service.
5. The authentication service processes the challenge response and provides an authentication response.
6. Bob's client again contacts the key distribution server and re-sends the registration message, this time including the authentication response. The key distribution server reports either success or failure to Bob's client.

Bob's software, during installation or reconfiguration, might present a dialog indicating that a registration server is available and prompting the user to generate or supply a suitable public key (visualized in Figure 5.5).

If Bob chooses to generate a key, the software should gather any required information, generate a key pair, and register the public half. It should also be possible for Bob to use an existing key — either from a previous installation of this software, or any key pair for which the public and private halves are available in a standard format.



Figure 5.5: Client Registration Confirmation Dialog

Chapter 6

Protocol Design

“Protocol is everything”

Francois Giuliani

As described in the use cases in Section 5.4, a client wishing to interact with either the query or registration services has to perform the following steps:

1. Locate the server responsible for queries or registrations for a specific domain.
2. Construct a query or registration message in the format required by the server.
3. In the case of a query request, locate and retrieve the cryptographic keys used to verify the server response.
4. In the case of a registration request, successfully authenticate to the server.

In this chapter we explore each of these in greater depth.

6.1 Locating a Key Server

The first issue which we must address is determining the server handling requests for the target domain. The obvious possibilities are (a) invent a new DNS resource record specifically for delegation of key distribution services (as proposed in RFC 2230 [8]), (b) represent the delegation record as a DNS *TXT* record, or (c) use the existing *SRV* record type — intended for service location. In keeping with Constraint 8 from Section 5.2.1, we leverage the DNS service (*SRV*) resource record type [57], which we briefly explain in the next section.

6.1.1 DNS SRV Resource Records

DNS *SRV* records are intended to allow clients to perform service discovery using DNS. To find a server in domain D , for a given service, S , running over a given protocol¹, P , (as defined in RFC 1700 [97]), a client issues a DNS query for $_{S}_{P}.D$. Responses include a list of host, port pairs, along with a priority for each matching record and a weight used to distribute load across servers of the same priority.

For example, to find an *ftp* server in *example.com* a client would issue a query for *ftp.tcp.example.com*. The response would indicate the host providing *ftp* service and the port the service is available on (e.g. 21). If multiple *ftp* servers have been configured, the client will receive all matching records, each with a weight and priority. Clients contact

¹Note that “protocol” in this context refers to a transport layer protocol (such as TCP), and not an application layer protocol (such as FTP).

servers in priority order, and distribute load across servers of the same priority in proportion to their specified weight.

6.1.2 Key Server Service and Protocol Names

In order to locate a server using DNS *SRV* records, a client must first know the service and protocol names as described in Section 6.1.1. Since we anticipate the desire to be able to direct query requests to a different set of servers than registration requests, we need for two distinguishable service names. The choice of service name is entirely capricious. We adopt *ikqs*, an acronym for “Internet Key-Query Service”, and *ikrs*, an acronym for “Internet Key-Registration Service.”

For reasons to be discussed in subsequent sections, we adopt HTTP [42] as the query transport mechanism, and HTTPS (HTTP over a Secure Socket Layer [46] connection) as the registration transport mechanism. We point out that the SSL certificate used to protect the key registration service could be signed by the domain, rather than a Certifying Authority, using our proposed scheme to perform certificate verification.

Given these service and protocol names, a client wishing to locate a server capable of handling queries for names contained in domain D would query for an *SRV* record matching *_ikqs._tcp.D*. Similarly, to locate a server capable of handling registration requests for names contained in *domain*, *SRV* records matching *_ikrs._tcp.D* would be requested.

Note that nothing restricts the query or registration services for a given domain to actually

be hosted in or by the owner of the domain. All that is required is that the domain administrator delegate this function to the service provider of their choosing by adding the appropriate *SRV* records. So the query or registration services for *example.com* could be provided by someone completely outside the *example.com* domain. We see the potential for an organization, possibly existing certifying authorities who already have a good understanding of the operational security issues involved in key management, to offer key distribution services to domain administrators.

6.2 Message Marshalling and Transport

6.2.1 Marshalling

The World Wide Web Consortium (W3C) has published a multi-part recommendation called SOAP (Simple Object Access Protocol) which specifies an interoperable means of using XML to exchange structured and typed information in a distributed environment or application [54, 55].

In essence SOAP specifies message formatting — including the overall structure of the message as an XML document (called the SOAP *envelope*), the structure of an optional SOAP header carrying non-payload information, and the structure of the SOAP body carrying the message payload. SOAP is transport-independent, dictating only the high-level message format and providing bindings to various transport mechanisms, including HTTP and SMTP

(Simple Mail Transport Protocol) [93]. A more complete overview of SOAP, with supporting examples, can be found in [83].

Loath to re-invent the wheel unnecessarily, we choose to encode the protocol message using SOAP. Relevant details of the SOAP message structure will be explained as they are required.

The W3C has also issued a note describing a Web Service Definition Language (WSDL) which provides a mechanism for “describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information” [21]. Essentially, WSDL provides a standard mechanism for performing service discovery — allowing applications to determine the capabilities offered by a service endpoint and message marshalling requirements. Appendix A contains our query and registration service interface definitions, expressed in WSDL.

Canonical Names

In the spirit of design constraint 5 (Section 5.2.1), we propose a protocol free from enumerated types. Rather than attempting to anticipate all possible useful values of a given protocol message field, we present a simple set of guidelines to produce consistent values at independent endpoints.

1. **Remove Non Alpha-Numeric Characters:** Any non alpha-numeric characters must be removed from the name being canonicalized. So to canonicalize PKCS #7, a cryp-

tographic message syntax standard, the white-space and “#” are removed from “PKCS #7” resulting in “PKCS7”.

2. **Convert to Lowercase:** “PKCS7” is further transformed into “pkcs7” — the canonical representation of “PKCS #7”.

While in general this scheme may increase the occurrence of collisions, this seems unlikely due to the relatively sparse input-space of interest.

6.2.2 Transport

With the increasing popularity of the Web Services model of remote service invocation, HTTP and HTTPS are fast becoming the defacto standard transport protocols for remote procedure call. One of the primary reasons for this adoption is that HTTP and HTTPS are typically permitted through firewalls and across different administrative domains within an organization. This is reinforced by the relative simplicity of the protocol as well availability of client and server implementations.

Given SOAP’s transport independence, other transport mechanisms — such as SMTP — may be supported in the future.

6.2.3 Key Query and Key Registration Target URIs

Unfortunately, knowing the name of the host providing the desired service (obtained from DNS *SRV* records as described in Section 6.1.2) is insufficient. Since SOAP services are

differentiated by URI, we use the service names established in Section 6.1.2: the path “/ikqs” references the key query service and “/ikrs” references the key registration service. Given the host and port information obtained from the *SRV* records, a client can construct complete URI’s for the desired service. Modern HTTP servers provide flexible mechanisms to map the published URI space into an arbitrary server side structure, so the mandate of public endpoint names does not dictate server side details.

6.3 Locating Response-Signing Keys

As will be discussed in Section 7.1.2, portions of query response messages are cryptographically signed by the responding server. In order to verify this signature, the client must fetch the public half of the signing keypair from DNSSEC. In order to avoid polluting a domains DNS name space with a slew of standardized names, we allow each server to identify the signing key by name in the query response, and require only that the named key be published in DNSSEC by the domain being queried.

While it is tempting to store these keys using the *KEY* resource record format for DSA keys described in RFC 2536 [32], this runs afoul of RFC 3445 [80] which prohibits storing keys not directly consumed by DNSSEC in *KEY* records. As an alternative we explored breaking the DSA public key, which consists of four parameters (pub, p, q, g), into a set of four *TXT* records logically linked by their keyname.

A key K , the parameters needed to reconstruct the key are stored in DNS records with

names pub_K , p_K , q_K , and g_K . Each record contains a hexadecimal representation of the numeric value in big-endian byte-order. An example of these records is shown in Section 8.2.4.

Limitations in BIND, however, force key parameters stored in this manner to be relatively small (255 bytes). Encoding a 768 bit key with this method is acceptable, while encoding a 1024 bit key exceeds the 255 byte record length limit. Without supporting changes from BIND (and potentially DNS clients), this limitation on key length was deemed unacceptable.

To overcome this limitation, we decided to store a key commitment (a SHA-1 hash) of the verification key in DNS, instead of the entire verification key.² All the guarantees provided by DNSSEC apply equally and, barring a break in the one-way hash function, the achieved security is comparable. This storage solution eliminates the key length limitation discussed above, as the key hash is a fixed length — 160 bits in the case of SHA-1 — regardless of the key length.

Given a key K in domain D , the verification key must be a DSA key in PEM format stored on the key-server serving queries for domain D with name K . The hash of the PEM key data is stored in a DNS text record with the name sha1_K . The record contains a hexadecimal representation of the SHA-1 hash value in big-endian byte-order.

To verify the results of a query, the client first obtains the verification half of specified key-signing key by requesting the key named in the query response from the server. Subsequently,

²Cryptanalytic results against SHA-1, presented at CRYPTO 2004, just as this manuscript was being finalized may mandate re-evaluating the use of SHA-1 as a secure hash function.

the client retrieves the hash of that key-signing key from DNSSEC and confirms that the published hash matches the retrieved key-signing key. Finally, the key-signing key is used to verify the query results.

Note that regardless whether the key or a hash of the key is stored key-signing keys do not explicitly have a lifetime. The key-signing key lifetime is bound by the validity period of DNSSEC signatures and appearance in query responses. If a key-signing key is compromised, or is being routinely refreshed, stored records signed with the deprecated key must be re-signed with the replacement key. Clients, however, may cache the previously published key until its DNSSEC signature expires, and use this cached key to validate any retrieved records that reference it as the signing key.

6.4 Authentication During Key Registration

The problem of authenticating the registration client to the registration server has several interesting aspects. In keeping with design constraints 1 and 2 (Section 5.2.1), there is the need to give the registration server wide latitude with respect to required and supported authentication mechanisms. Flexibility, however, works against design constraint 3 (simplify the common case). Flexibility often comes at the price of complexity and not only is complexity the enemy of security, it likely makes handling the authentication process with minimal human involvement very difficult.

6.4.1 Targeted Authentication Mechanisms

To better understand the design challenge, it is helpful to consider the various authentication schemes that the service may be asked to support:

- Many services are currently protected by a shared secret — typically username and password. While considered “weak” by cryptographic standards, username/password authentication is ubiquitous and has proven difficult to displace for a variety of reasons. Biometrics, while promising, have been slow to gain adoption outside corporations.
- Variations on a simple password scheme abound, including one-time password schemes such as S/Key [59] and OTP [60], as well as two-factor schemes involving hardware devices such as SecurID [106]. In some cases, use of these authentication schemes looks identical to username/password schemes from the system perspective, differing only in the logic used to test the password.
- A number of organizations have deployed Kerberos [73] as their centralized authentication and authorization mechanism, and adoption has accelerated with the inclusion of Kerberos as a supported authentication protocol in Microsoft Windows Active Directory [24].
- Perhaps the most degenerate form of authentication we concern ourselves with is “proxy” authentication whereby a trusted, authenticated party is able to vouch for the identity of others. In our scenario this could take the form of an identified system

administrator confirming credentials for the requesting client out of band, and authenticating themselves to the system to authorize the request.

The over-arching goal is to permit these and other authentication schemes to be incorporated in a flexible yet manageable manner. Authentication in this context is dictated by the policies of and, in some respects, specific to the domain involved in registration. In standardizing the protocol, a primary goal is to facilitate building clients that can operate correctly regardless of the domain into which they are ultimately deployed.

6.4.2 Candidate Authentication Schemes

Given the above list of authentication techniques, we turn to potential ways to realize support for these and as-yet unanticipated schemes in the registration protocol.

HTTP supports authentication and authorization: the basic framework is defined in RFC 2616 [42], while RFC 2617 [45] details the standard HTTP “basic” and “digest” authentication mechanisms. Microsoft has an undocumented, proprietary, HTTP authentication mechanism called NT Lan Manager (NTLM) [52] — allowing pass-through authentication leveraging existing operating system credentials on Windows — which has been reverse engineered and incorporated into non-Microsoft servers and HTTP user-agents.

An expired IETF draft [18] provided details of Microsoft’s adaptation of GSSAPI [77, 79] (Generic Security Services API), and SPNEGO [9] to implement Kerberos [73] authentication over HTTP (which they call “Negotiate” authentication) in their Internet Explorer

browser and Internet Information Server HTTP server. Despite not being an RFC, support for the described authentication mechanism exists as a plug-in for Mozilla [38], and server side support exists for Apache web servers [75].

The Simple Authentication and Security Layer (SASL) [82] was designed to solve authentication mechanism negotiation for connection-oriented protocols where the client and server can effectively carry on an interactive dialog.³ The SASL standard defines multiple mechanisms that a client and server can use to achieve authentication. The base standard includes mechanisms built on Kerberos version 4, the GSSAPI, S/Key as well as a mechanism that delegates to an authenticator “external” to SASL. RFC 2595 [88] defines a “PLAIN” SASL mechanism for using plaintext usernames and passwords over a secured communications channel (such as provided by SSL/TLS); RFC 2831 [76] defines how HTTP Digest Authentication [45] can be used as a SASL mechanism; and RFC 2808 [90] is an informational RFC describing a mechanism for supporting SecurID authentication.

6.4.3 Key Registration Authentication Process

At a high level there are two ways a server can choose to ask for authentication from a client: via the HTTP authentication mechanism described above or by some method outside the HTTP protocol.

In an effort to leverage existing deployed client code, we choose to use HTTP authen-

³Although HTTP is often thought of as a one-message-per-connection protocol, the specification explicitly contains provisions for pipelining multiple requests and responses over a single connection.

tication as the primary authentication protocol. All clients must support at least basic authentication and should support digest authentication, both as detailed in RFC 2617. A brief overview of HTTP authentication is provided in Section 6.4.4.

Since we anticipate HTTP authentication as insufficient to facilitate the range of possible authentication processes, we provide a secondary mechanism that must be implemented by all clients and leverages the redirection facilities of HTTP to handoff a client to an external authentication process. This alternate mechanism, which also leverages the HTTP standard, will be described in more detail in Section 6.4.5.

The decision to base authentication on HTTP increases the level of coupling between our protocol and its underlying transport. Given the ubiquitous nature of HTTP clients and servers, we don't see this as a significant liability. If necessary, a subsequent version of the protocol can revisit this decision, replacing HTTP authentication with an alternative mechanism to support transport protocols other than HTTP.

6.4.4 HTTP Authentication

The basic HTTP authentication mechanism is straightforward. Upon receipt of a request which requires authentication and authorization, the server returns a specific status code (401) in lieu of the requested resource. The response also includes one or more challenges, any one of which must be correctly responded to by the client to gain access. Upon receipt of a list of challenges, the client can choose one to answer and may resubmit the request with the

appropriate response, potentially gathering the information needed from the end user. Once a challenge has been successfully responded to, access to the requested resource is granted.

6.4.5 External Authentication

Rather than attempt to rigorously specify interactions with unanticipated authentication processes, we specify the handoff mechanism to an external authentication process. All clients must support this external authentication process, though as described later, the client may choose to delegate much of the interaction to an external application.

The HTTP 1.1 specification [42] includes a number of status codes (3xx) which indicate that a requested resource can be found at an alternate location. We allow a server wishing to initiate external authentication to respond to the registration message with an HTTP 307 status which indicates that the requested resource resides temporarily under a different URI, a so-called “temporary redirect.”⁴

Key Server Requirements

A server wishing to initiate an authentication process other than the built-in HTTP mechanism must issue an HTTP 307 response, providing the client with the URI of the external authentication process. The server must provide some mechanism, transparent to the client, for the external authentication process to indicate successful authentication to the server.

⁴Implementors are cautioned that constructing a secure multi-party authentication scheme is non-trivial and should be undertaken with care [47].

Client Requirements

A client, upon receipt of an HTTP 307 response to a registration request, must resubmit the registration message to the specified URI (per the HTTP specification). The resulting HTTP response must be handled — either internally or by handing off the received data to a program associated with the content type of the response. In the event that the client delegates control to another process, it should provide a mechanism to determine the status of the registration request.

Authentication Server Requirements

An external authentication process must retain the registration request and, upon completion of the authentication process, cause the client to resubmit the request, along with appropriate authentication credentials, to the server.

6.4.6 Client Key Management

In an effort to minimize the need for the server to rely on username/password authentication, we propose a simple key-management key approach. In this approach a client first registers a key-management key (i.e. the public half of a signing keypair) with the server using the standard or external authentication mechanisms described previously. Subsequently, the client uses the private half of the key-management key to sign all registration (and revocation) requests.

Since our scheme already makes use of DSA, we restrict keys intended for this purpose to be DSA keys of at least 1024 bits in length. To be able to identify the key-management key, it must be identified as protecting service “*ikrs*” for the named entity. Only one key of this sort may be active for a given named entity at any time. Once a user has successfully registered a key-management key, a server should treat future registration or revocation requests signed with the private half of this key-management key as sufficiently authenticated.

Chapter 7

Protocol Messages

“When I use a word,” Humpty Dumpty said in rather a scornful tone, “it means just what I choose it to mean — neither more nor less.”

Lewis Carroll; *Alice Through the Looking Glass*

As an intermediate step in defining the protocol, we examine each of the major operations supported by the service, and the messages exchanged to perform those operations. For each, we enumerate the message contents at a high level, deferring low-level details to the WSDL interface definitions found in [Appendix A](#).

7.1 Key Query

7.1.1 Key-Query Parameters

As described in the use case in Section 5.4.1, a client attempting to retrieve a key for a named entity constructs a query message and transmits the message to the appropriate key-server. In this section, we detail the various query parameters useful in locating keys usable by a requesting application. Each parameter further narrows the query, thereby reducing the potential result set.

These parameters are summarized in Table 7.1, including the range of values the parameter may take on, and if server support for each parameter is optional or required. No restrictions are placed on which parameters a client must provide in a given query. The server will use all supplied parameters it supports to narrow the search, returning matching results.

- **Name:** The fully-qualified name of the entity for which a key is being requested. Any name that can be mapped into DNS (i.e. split into name and domain parts) should be accepted. RFC 822 [25] describes the acceptable name format.
- **Service:** The service that this key protects, and an optional port number to distinguish between multiple instances of a given service on the same named endpoint. RFC 1700 [97] describes the officially registered service list, but implementations should not restrict registration or query to this list.
- **Unique Key Id:** An identifier (globally) unique to a specific key. It is suggested,

though not required, that this be implemented using a UUID as described in [67].

- **Key Fingerprint:** The fingerprint of the key that is being queried. Not all key formats define key-fingerprint generation, so this field is optional and may be ignored by an implementation.
- **Key Formats:** The key container format(s) acceptable to the requesting client. Several well-known key formats exist (listed in Table 7.1) and should be supported. Unrecognized formats should be passed through. Supplied key-format names must be canonicalized per the guidelines in Section 6.2.1.
- **Algorithms:** The algorithm(s) acceptable to the requesting client. Well-known algorithms (listed in Table 7.1) should be supported. Unrecognized algorithms should be passed through. Supplied algorithm names must be canonicalized per the guidelines in Section 6.2.1.
- **Key Length:** The minimum key length, in bits, acceptable to the requesting client.
- **Permitted Use:** The use(s) for which the client is requesting the key (authenticity, privacy, both, none).
- **Valid After:** A POSIX [92] time-stamp relative to UTC. If supported, any returned keys must be valid for use at times greater than or equal to the provided value.
- **Valid Until:** A POSIX time-stamp relative to UTC. If supported, any returned keys

must be valid for use at times less than or equal to the provided value. If both Valid After and Valid Until are provided, they are combined as a logical AND.

Parameter Name	Value Range (<i>Sample Values</i>)	Server Support Required?
Name	Per RFC 822 (<i>bob@example.com, www.cs.ucr.edu</i>)	Required
Service	Per RFC 1700[:port] (<i>smtp, https:4430</i>)	Required
Unique Key Id	[0-9a-f]+ (<i>76f3caf87da549db9651e1d58b45efd4</i>)	Required
Key Fingerprint	[0-9a-f]+ (<i>1b27a0b9c9faa7d8ea9e76b4847c95c551a12856</i>)	Optional
Key Formats	one or more of [0-9a-z]+ (<i>openpgp [20], x509v3 [62], secsh [48], pem [78], pkcs7 [69]</i>)	Required
Algorithms	one or more of [0-9a-z]+ (<i>dh, rsa, dsa, elgamal</i>)	Required
Key Length	[0-9]+ (<i>1024</i>)	Required
Permitted Use	none, privacy, authenticity, privacy+authenticity (<i>privacy</i>)	Required
Valid After	POSIX time-stamp relative to UTC (<i>1085611960</i>)	Optional
Valid Until	POSIX time-stamp relative to UTC (<i>1085611960</i>)	Optional

Table 7.1: Key Query Parameters

7.1.2 Key-Query Response

Upon successful evaluation of a submitted query message, the server returns a query response message containing any keys matching the query, and potentially meta-data about the query execution.

The overall structure of the query response is a response header followed by zero or more key records.

Query Response Header

The response header is not cryptographically signed, and hence all data it contains must be considered non-authoritative. This decision was made to allow query servers to operate without requiring signing keys be kept online. Table 7.2 summarizes the various data potentially appearing in a query response header. Clients must be capable of gracefully ignoring unexpected header information.

- **Match Count:** An unsigned integer indicating the number of keys that matched the provided query.
- **Partial Result:** A boolean indicating if the returned keys are a subset of the matching keys. A server may choose to limit the number of keys returned for a given query. If a server implements such a policy, it should also support this header field. A client that receives a partial response should understand it to mean its query was too broad to be answered completely. A more specific query may be issued to narrow the result set.
- **Ignored Parameters:** A list of any optional query fields that this server does not implement and hence were not included in query evaluation.
- **Query Time:** A POSIX time-stamp relative to UTC. If present, represents the time (relative to the server clock) when the query was received.
- **Response Time:** A POSIX time-stamp relative to UTC. If present, represents the time (relative to the server clock) when the response was generated. Query and response

times are primarily for instrumentation and performance monitoring, and are not essential for functionality.

Field Name	Value Range (<i>Sample Values</i>)	Signed/ Unsigned
Match Count	Unsigned Integer (<i>5</i>)	Unsigned
Partial Result	Boolean (<i>true</i>)	Unsigned
Ignored Parameters	List of String	Unsigned
Query Time	POSIX time-stamp relative to UTC. (<i>1085611960</i>)	Unsigned
Response Time	POSIX time-stamp relative to UTC. (<i>1085611960</i>)	Unsigned

Table 7.2: Query Response Header Contents

Query Response Key Records

Portions of the query response are cryptographically signed. Unsigned information should not be trusted. Table 7.3 summarizes the various data potentially appearing in a query response. The “signed/unsigned” column indicates if the named field is included in the record signature computation. Details regarding verifying signatures of received keys are described later in this section.

- **Name:** The fully-qualified name to which the key is bound. RFC 822 [25] describes the acceptable name format.
- **Service:** The service that this key protects, and an optional port number to distinguish between multiple instances of a given service on the same named endpoint. RFC 1700 [97] describes the officially registered service list.

- **Unique Key Id:** An identifier (globally) unique to a specific key.
- **Key Format:** The container format of the provided key. Supplied key format names must be canonicalized per the guidelines in Section 6.2.1.
- **Algorithm:** The algorithm applicable to the provided key. The supplied algorithm name must be canonicalized per the guidelines in Section 6.2.1.
- **Key Length:** The length, in bits, of the included key.
- **Key:** The key bits, in the format specified by the Key Format attribute. Binary key formats must be represented in Base64 [101] encoding.
- **Permitted Use:** The permitted use(s) for the returned key (authenticity, privacy, both, none).
- **Valid After:** A POSIX time-stamp relative to UTC. Optional. If present, indicates the beginning of the key validity period.
- **Valid Until:** A POSIX time-stamp relative to UTC. Optional. If present, indicates the key expiration time.
- **Revoked At:** A POSIX time-stamp relative to UTC. Optional. If present, indicates the key was revoked at the specified time (relative to the server clock). In this case, the *Key* field will be present, but must be empty. Support for key revocation is described in Section 7.3.

- **Revocation Certificate:** If the requested key has been revoked, and a revocation certificate is available, it will be provided in the query response. This certificate is in the “logical” compliment of the reported Key Format.
- **Signature Create Time:** A POSIX time-stamp relative to UTC. Indicates when the provided signature was created.
- **Signature Expire Time:** A POSIX time-stamp relative to UTC. Indicates when the provided signature expires.
- **Signing Key Name:** The DNS name of the key used to sign the key data, as discussed in Section 6.3.
- **Signature:** The signature computed across all signed fields in the key part, Base64 encoded.

Cryptographic Signing and Verification

To ensure interoperability between implementations, we must carefully specify the cryptographic operations involved in verifying retrieved keys. Upon receipt of a query response, the client must retrieve the named signing key from DNS (or a local cache). Once the named signing key has been retrieved from DNS as described in Section 6.3, the client must recreate the precise structure that was signed by the server for verification to be meaningful.

Field Name	Description (<i>Sample Values</i>)	Signed/ Unsigned
Name	Per RFC 882 (See Table 7.1)	Signed
Service	Per RFC 1700[:port] (See Table 7.1)	Signed
Unique Key Id	[0-9a-f]+ (see Table 7.1)	Signed
Key Format	[0-9a-z]+ (see Table 7.1)	Signed
Algorithm	[0-9a-z]+ (See Table 7.1)	Signed
Key Length	[0-9]+ (1024)	Signed
Key	As specified by “Key Format”	Signed
Permitted Use	none, privacy, authenticity, privacy+authenticity (<i>privacy</i>)	Signed
Valid After	POSIX time-stamp relative to UTC (1085611960)	Signed (If Present)
Valid Until	POSIX time-stamp relative to UTC (1085611960)	Signed (If Present)
Revoked At	POSIX time-stamp relative to UTC. (1085611960)	Signed (If Present)
Revocation Certificate	As appropriate based on Key Format.	Signed
Signature Create Time	POSIX time-stamp relative to UTC. (1085611960)	Signed
Signature Expire Time	POSIX time-stamp relative to UTC. (1085611960)	Signed
Signing Key Name	Per RFC 882. (<i>ksk_1</i>) [See Section 6.3]	Signed
Signature Algorithm	[0-9a-z]+ (<i>dsa</i>)	Signed
Signature	Base64 [101]	N/A

Table 7.3: Query Response Key Record Contents

All DSA signatures are computed across a SHA-1 hash of the signed fields. The signed fields are fed into the hash function in a specific order. In the case of a query response, Figure 7.1 gives an ASN.1 [1] expression indicating the structure over which the SHA-1 hash is to be computed for retrieved keys; Figure 7.2 indicates the structure for retrieved revocation records. The field names correlate with those in Table 7.3. *OPTIONAL* indicates that the field should only be included in the hashed data if it is present.

```

Hash-Input ::= SEQUENCE {
    PrintableString Name,
    PrintableString Service,
    PrintableString Unique Key Id,
    PrintableString Key Format,
    PrintableString Algorithm,
    NumericString Length,
    BIT STRING Key,
    PrintableString Permitted Use,
    NumericString Valid After OPTIONAL,
    NumericString Valid Until OPTIONAL,
    NumericString Signature Create Time,
    NumericString Signature Expire Time
}

```

Figure 7.1: SHA-1 Input from Key Query Response Key Record (ASN.1)

```

Hash-Input ::= SEQUENCE {
    PrintableString Name,
    PrintableString Service,
    PrintableString Unique Key Id,
    BIT STRING Key Revocation Certificate,
    NumericString Signature Create Time,
    NumericString Signature Expire Time
}

```

Figure 7.2: SHA-1 Input from Key Query Response Revocation Record (ASN.1)

7.2 Key Registration

7.2.1 Key-Registration Parameters

As described in the use case in Section 5.4.1, a client attempting to register a key for a named entity constructs a registration message and transmits the message to the appropriate key server. As for the query component, we detail the various parameters necessary for

establishing a binding between a name and a public key.

These parameters are summarized in Table 7.4, which includes the range of values the parameter may take on, and if the presence of each parameter in the registration message is optional or required.

- **Name:** The name to which the key is to be bound. RFC 822 [25] describes the acceptable name format.
- **Service:** The service this key protects, and an optional port number to distinguish between multiple instances of a given service on the same named endpoint. RFC 1700 [97] describes the officially registered service list.
- **Key Format:** The container format of the provided key. Supplied key format names must be canonicalized per the guidelines in Section 6.2.1.
- **Key:** The key being registered in the format specified, encoded in Base64.
- **Key Revocation Certificate:** A revocation certificate for the key being registered in the logically appropriate format. For example, OpenPGP defines a revocation certificate packet format which is appropriate for revoking OpenPGP keys. One recurring issue with OpenPGP key management is that persons who have lost their private key are incapable of issuing a key revocation certificate after the loss is discovered. We therefor encourage, but do not require, client implementations to generate and include

a revocation certificate during key registration. Revocation is not supported for key formats that do not define revocation certificate structures.

- **Algorithm:** The algorithm applicable to the provided key. Supplied algorithm names must be canonicalized per the guidelines in Section 6.2.1. For key container types (such as OpenPGP) which contain multiple subkeys, each potentially for a different algorithm, the key must be registered once for each subkey to appear in query results. In this case, requests for any registered subkey will return the container and all subkeys.
- **Key Length:** The length, in bits, of the key being registered.
- **Key Fingerprint:** The fingerprint of the key being registered. If no fingerprint method is defined for the key type being registered, this field should be empty.
- **Permitted Use:** The use(s) for which the client is registering the key (authenticity, privacy, both).
- **Valid After:** A POSIX time-stamp relative to UTC. Optional. If present, indicates the beginning of the key validity period.
- **Valid Until:** A POSIX time-stamp relative to UTC. Optional. If present, indicates the key expiration time.
- **Authorizing Signature:** A DSA signature computed across all present fields in the registration message. (See Section 6.4.6.)

Parameter Name	Value Range (<i>Sample Values</i>)	Required/ Optional
Name	Per RFC 882 (See Table 7.1)	Required
Service	Per RFC 1700[:port] (See Table 7.1)	Required
Key Format	[0-9a-z]+ (see Table 7.1)	Required
Key	Base64	Required
Key Revocation Certificate	Base64	Optional
Algorithm	[0-9a-z]+ (See Table 7.1)	Required
Key Length	[0-9]+	Required
Key Fingerprint	[0-9a-f]+	Optional
Permitted Use	none, privacy, authenticity, privacy+authenticity (<i>privacy</i>)	Required
Valid After	POSIX time-stamp relative to UTC (1085611960)	Optional
Valid Until	POSIX time-stamp relative to UTC	Optional
Authorizing Signature	Base64	Optional

Table 7.4: Key Registration Parameters

7.2.2 Key-Registration Response

Once authentication has been successfully negotiated, the key-server processes the registration request and returns a status message indicating success or failure and, on success, the unique identifier of the registered key.

In the case of external authentication, this message may be delivered to an external application incapable of informing the client of the transaction status. In this case the client may choose, upon indication from the user that the process is complete, to issue a query for the newly registered key to confirm successful registration. It is important to note that the key may not be immediately available due to, for example, batching of new key signings (revisited in Chapter 8). Clients must be capable of handling this condition gracefully.

7.3 Key Revocation

As mentioned briefly in Section 7.2.1, we support distributing key revocation certificates. A client wishing to revoke a published key needs to know the unique identifier of the key in question, and needs to be able to authenticate itself to the appropriate registration server using the mechanisms described in Section 6.4. A server may choose to use a different form of authentication for authorizing key revocation requests — for example, it is suggested that a revocation request signed with the private half of the key being revoked be considered authenticated. Once revoked, a key is purged from the key store and all subsequent queries for that key result in a response indicating the time of the revocation as described in Section 7.1.2.

7.3.1 Key-Revocation Request

A client wishing to revoke a specific key constructs a key-revocation message and transmits the message to the appropriate key-server. We detail the various parameters necessary to identify the key to be revoked. Note that the message contents are not intended to authenticate the revocation request — only to uniquely identify the key that the client wishes to revoke.

These parameters are summarized in Table 7.5, which includes the range of values the parameter may take on, and if the presence of each parameter in the revocation message is optional or required.

- **Name:** The name to which the key is bound. RFC 822 [25] describes the acceptable name format.

- **Service:** The service that the key protects, and an optional port number to distinguish between multiple instances of a given service on the same named endpoint. RFC 1700 [97] describes the officially registered service list.
- **Unique Key Id:** The server assigned identifier (globally) unique to the specific key being revoked.
- **Key Revocation Certificate:** A revocation certificate for the key being revoked in the logically appropriate format.
- **Authorizing Signature:** A DSA signature computed across all present fields in the revocation message (See Section 6.4.6).

Parameter Name	Value Range (<i>Sample Values</i>)	Required/ Optional
Name	Per RFC 882 (see Table 7.1)	Required
Service	Per RFC 1700[:port] (see Table 7.1)	Required
Unique Key Id	[0-9a-f]+ (see Table 7.3)	Required
Key Revocation Certificate	Base64	Optional
Authorizing Signature	Base64	Optional

Table 7.5: Key Revocation Parameters

7.3.2 Key-Revocation Response

Once authentication has been successfully negotiated, the key-server processes the revocation request and returns a status message indicating success or failure and, on success, the unique id of the registered key.

In the case of external authentication, this message may be delivered to an external application incapable of informing the client of the transaction status. In this case the client may choose, upon indication from the user that the process is complete, to issue a query searching for the newly-revoked key to confirm successful revocation.

Chapter 8

Implementation

In theory there's no difference between theory and practice. In practice there is.

Anonymous

8.1 Overview

We now turn our attention to our proof-of-concept implementation of our proposed key distribution service. We break the discussion into three parts:

1. **DNSSEC Theory into Practice:** In this section we describe the practice of configuring and operating a DNSSEC enabled DNS server as an island of security from both the server and client perspectives. We also discuss the encoding of DSA key signing keys, providing a practical example of creating the requisite *TXT* records.

2. **Key Server:** Here we elucidate relevant portions of the design and construction of the server component of the system. We describe and justify various design decisions in the hopes they assist others interested in improving on this work.
3. **Client Library:** Finally, we describe the implementation of a client library for performing queries and registrations which encapsulates the protocol described earlier in this document. We briefly discuss a simple client built using this library.

A decision was made early on to leverage as much existing code as was feasible in the construction of the system prototype. In some cases reuse was obvious: using the latest release candidates of BIND, which track the current DNSSEC IETF drafts, and using existing cryptographic libraries. In other cases the decision was made to speed development: leveraging existing SOAP frameworks on both client and server. Each section identifies any third-party components used, and points out any relevant details of each.

8.2 DNSSEC: Theory into Practice

8.2.1 Securing a Zone

The first hurdle to overcome was the provisioning of a security-aware DNS server. The latest release candidates of BIND (9.3rc1 at the time of this writing) are closely tracking the latest DNSSEC IETF drafts, and include utilities to perform key generation and zone signing [119].

Olaf Kolkman at RIPE (Réseaux IP Européens) has written an operational HOWTO [74]

that provides guidance in configuring secure resolvers and zones. This HOWTO, however, described “cutting edge” code at the time of its writing, and subsequent implementation changes are not reflected. This was observed as a general problem: while the bits implementing DNSSEC seem to be maturing quickly, the documentation for implementors is often nonexistent or inaccurate.

While the details are guaranteed to change over time and are therefore omitted, the overall process is worth describing. The primary requirement is to have an instance of BIND configured and compiled with DNSSEC support (which requires the presence of the OpenSSL [128] library and development files).

We describe here the process of establishing a so-called “island of security,” meaning that the set of secured zones do not form a fully-connected graph within the DNS namespace. The high-level steps required on the server side are:

1. Construct a standard zone file containing the resource records to be served. Many documents exist that describe the format of a BIND zone file, this information is not repeated here. For our purposes, it is important that the appropriate *SRV* records (see Section 6.1.1) be included in the zone data.
2. Generate key-signing and zone-signing keys. BIND includes utilities to generate keys usable for signing zone data, allowing the domain administrator to select from suitable algorithms and key strengths. Once keys are generated, they are included in the zone file created in step 1. Multiple zone-signing keys can be generated, each identified

by a unique (within the zone) id. The key-signing key is primarily used for secure delegation across zones and is not described further here.

3. Sign the zone file generated in step 1. The zone-signing utility (also included in the BIND distribution) performs the necessary steps to sign a provided zone file with a specified signing key. It performs the necessary ordering and grouping of resource records to form resource record sets (RRSets, see Section 3.4.1), computes signatures for each RRSet and emits a new zone file including these signature records.
4. Configure BIND to enable DNSSEC responses. By default, the current incarnation of BIND will not include signature records in query responses, even if the client has indicated DNSSEC capability, unless explicitly configured to do so. This is one of the many places where the documentation is missing or misleading; the BIND documentation currently makes no mention of this configuration.

Once these steps have been performed, responses to queries that indicate support for DNSSEC will include the *DNSKEY* and *RRSIG* records required to cryptographically validate the requested resource record.

A normal query and response, as realized using the *dig* utility included in the BIND distribution, is listed in Figure 8.1. With DNSSEC support enabled (the `+dnssec` flag to *dig*), the response appears as Figure 8.2 (the actual signature and key data have been omitted for brevity and are represented by ellipsis in the *dig*). Note that when a query indicates support for DNSSEC, the server includes signature and signing-key records in the response.

```
; <<>> DiG 9.3.0rc1 <<>> @localhost walkabout.cs.ucr.edu
;; global options: printcmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 31682
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 2, AUTHORITY: 1, ADDITIONAL:
;0

;; QUESTION SECTION:
;walkabout.cs.ucr.edu.          IN      A

;; ANSWER SECTION:
walkabout.cs.ucr.edu.  600     IN      A      138.23.174.213
walkabout.cs.ucr.edu.  600     IN      A      138.23.174.216

;; AUTHORITY SECTION:
cs.ucr.edu.           3600    IN      NS      walkabout.cs.ucr.edu.

;; Query time: 1 msec
;; MSG SIZE  rcvd: 84
```

Figure 8.1: DNS Query Results

8.2.2 Secure Resolution

Facilitating secure resolution by clients is more involved than server configuration. To understand the complexity, it is helpful to first understand the secure resolution model. There are three actors involved in secure resolution, as pictured in Figure 8.3: the security enabled server, a verifying resolver trusted by the client, and the requesting client.

A client, potentially unaware of DNSSEC, submits a query to the verifying resolver that performs the necessary steps to provide validated results. This involves contacting potentially many security-aware DNS servers to gather the needed signatures and signing keys, verifying signatures and trust during each step. When the process is complete, the resolver responds with the requested records, setting the authenticated data (AD) bit to indicate the data has been verified.

```

; <<>> DiG 9.3.0rc1 <<>> @localhost +dnssec walkabout.cs.ucr.edu
;; global options: printcmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 61371
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 3, AUTHORITY: 2, ADDITIONAL:
;5

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags: do; udp: 4096
;; QUESTION SECTION:
; walkabout.cs.ucr.edu.          IN      A

;; ANSWER SECTION:
walkabout.cs.ucr.edu.  600     IN      A      138.23.174.213
walkabout.cs.ucr.edu.  600     IN      A      138.23.174.216
walkabout.cs.ucr.edu.  600     IN      RRSIG  A 5 4 600 \
    20040804211423 20040705211423 49061 cs.ucr.edu. bdOJ7vl+...dPA=

;; AUTHORITY SECTION:
cs.ucr.edu.           3600    IN      NS      walkabout.cs.ucr.edu.
cs.ucr.edu.           3600    IN      RRSIG  NS 5 3 3600 \
    20040804211423 20040705211423 49061 cs.ucr.edu. XERqq25...LnE=

;; ADDITIONAL SECTION:
cs.ucr.edu.           3600    IN      DNSKEY  256 3 5  AQP...w+M2Yw==
cs.ucr.edu.           3600    IN      DNSKEY  257 3 5  AQP...9I9yIQ==
cs.ucr.edu.           3600    IN      RRSIG  DNSKEY 5 3 3600 \
    20040804211423 20040705211423 49061 cs.ucr.edu. MbhV...UPU=
cs.ucr.edu.           3600    IN      RRSIG  DNSKEY 5 3 3600 \
    20040804211423 20040705211423 58047 cs.ucr.edu. B1f+u...c+M=

;; Query time: 78 msec
;; MSG SIZE rcvd: 1067

```

Figure 8.2: DNS Query Results with DNSSEC Key and Signature Records

If the verification process fails due to missing, expired, or corrupt keys or signatures, the verifying resolver reports a server failure (*SRVFAIL*) to the requesting client.¹

The verifying resolver can take a number of forms: it could be a trusted caching name server close (in terms of network geography) to the client, a caching name server running on the client host, or even code within the client application which implements the DNSSEC resolution and verification algorithms. Due to the potential amount of work (in terms of

¹There are a number of opponents to this overloading of the failure response code, the author included. We will revisit this when we discuss the client interface to a secure resolver in Section 8.2.3.

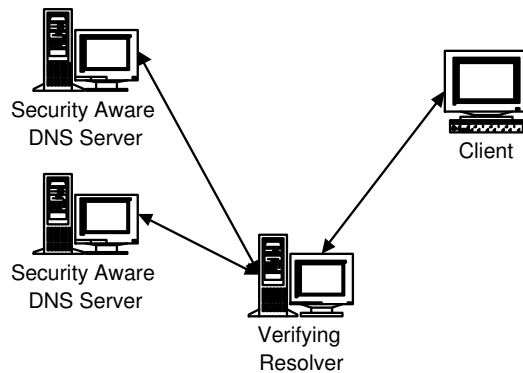


Figure 8.3: DNSSEC Resolution Actors

additional DNS queries and cryptographic operations) required to verify the response to an arbitrary query, the desire is to share verifying resolvers when possible and to allow those shared resolvers to perform caching to reduce work load.

The BIND distribution includes a lightweight resolver daemon, called *lwresd*. This daemon, intended to be run on each client host and service only local requests, provides implementations of the standard resolution interfaces (discussed in Section 8.2.3) as well as extensions that give clients additional control over name resolution. Currently, as is true for much of the code surrounding DNSSEC, the documentation on *lwresd* is nearly nonexistent, both in terms of configuring the daemon as well as in terms of using its extended API.

One of the more confounding behaviors currently present in BIND is that a security-enabled server that is authoritative for a given domain will not perform validation of generated query responses and set the authenticated data (AD) bit in response to clients. This means, for example, that clients local to a domain cannot use the primary name server for that domain as the verifying resolver — a dedicated verifying resolver is required. While this may

initially seem counterintuitive, a verifying resolver does a nontrivial amount of computation during verification, off-loading the authoritative servers.

8.2.3 Resolver Client Interface

Various name resolution libraries are commonly available to client applications. Possibly the most ubiquitous of these libraries is the *resolver* library which provides the *gethostbyname()* and *getaddrinfo()* family of functions. Name resolution, from human-friendly name to network address, is often hidden by networking libraries, allowing developers to program in terms of conceptually higher-level network operations.

Unfortunately, DNSSEC throws a proverbial monkey wrench into the gears. Existing applications, using existing interfaces, are unaware of DNSSEC; they have no way to interpret information about the validity of name query responses, and the application programmer interfaces (APIs) they use have no means to express this information. This dilemma leads directly to the need to overload an existing DNS response code (server failure) to report security verification failure, as briefly mentioned in Section 8.2.2.

In the short term, while DNSSEC gains acceptance and adoption, it is unclear how applications that are interested in the security state of their name-resolution requests will interface with the verifying resolver. For the reasons alluded to in Section 8.2.2 surrounding caching of DNS query results, it is undesirable to have each application embed a verifying resolver, but the standard library interfaces are inadequate for the task at hand. Further, performing vali-

dation requires cryptographic code which is unlikely to find its way into the standard system libraries due to export restrictions and other related issues. From a practical perspective, this has two key undesirable effects:

1. Leveraging existing networking libraries is difficult. Name resolution has been (correctly) seen as a task that a library should insulate its user from. Often calls to resolver functions are buried deep in libraries, making it difficult to use the functionality offered by the library while retaining the required level of control over name resolution. We return to this issue when we discuss our client library implementation in Section 8.4.
2. Applications wishing to have control over, or at least insight into, name resolution are likely to invent their own interface to a verifying resolver, if not the resolver itself. By exposing themselves to the “guts” of name resolution these applications become more fragile.

The interface between clients and resolvers is still the topic of active development and an evolving IETF draft [51].

8.2.4 Storing DSA Key-Signing Public Keys

Initial design called for storing DSA public keys as PEM key blocks, as generated and consumed by many cryptographic libraries, including OpenSSL [128]. Unfortunately, due to DNS packet format and size limits, BIND limits *TXT* record contents to 255 bytes.

```

...
pub_ksk_1 IN      TXT   "674301d3901f6e13fb0b60bb35ba55994d23f36815... \
705a457769ccae17dacc5ee7ef65977acb6d738e8a02"
p_ksk_1  IN      TXT   "00d5a3e833e360f439bbef341a2387e49012f42410... \
95040e3ac69e7bb2b2d85569990d3a6a1cd137b24f8d71"
q_ksk_1  IN      TXT   "00e813778c56bb9a4ca6ed43516b3ff51347b7a15d"
g_ksk_1  IN      TXT   "182ffa2f14f9d8c0590a892e772f337f9a2cc0c37c... \
8e63779bead8b5903311487db7c7bccdc8ecdef9cc78"
...

```

Figure 8.4: DSA Public Key DNS Records

To side-step this limitation, and in keeping with the spirit of RFC 2536, it was decided to store DSA public keys as a collection of the DSA parameters required for cryptographic operations. Four parameters are required to reconstruct a DSA public key — often labeled p , q , g and the public key itself (pub). Figure 8.4 shows an excerpt from a BIND zone file containing the four records corresponding to the key-signing key with logical name “ksk_1” (key data has been truncated and split across lines for clarity).

Fortunately, given a DSA public key in PEM format, it is straightforward to generate the necessary DNS records, as the Python code fragment in Figure 8.5 illustrates.

During experimentation it was determined that this storage technique placed unacceptable restrictions on the length of (and the security provided by) key-signing keys. It was therefore decided to store the actual verification key on the key server itself, and store a cryptographic hash of the key — a key commitment of sorts — in DNSSEC.

```

import M2Crypto
import M2Crypto.m2
import binascii

# a PEM file containing the DSA keypair (public and private) as
# generated by OpenSSL
keyfile="dsapriv.1.pem"

keyname="ksk_1"

dsa = M2Crypto.DSA.load_key(keyfile)

# the result of dsa_get_[g|p|q|pub] is an mpi - 4 bytes of length, and
# the number in big endian, so loose the first four bytes to get just
# the number we care about

pub = M2Crypto.m2.dsa_get_pub(dsa.dsa)
g = M2Crypto.m2.dsa_get_g(dsa.dsa)
p = M2Crypto.m2.dsa_get_p(dsa.dsa)
q = M2Crypto.m2.dsa_get_q(dsa.dsa)

print 'pub_%s IN TXT "%s"' % (keyname, binascii.b2a_hex(pub[4:]))
print 'p_%s IN TXT "%s"' % (keyname, binascii.b2a_hex(p[4:]))
print 'q_%s IN TXT "%s"' % (keyname, binascii.b2a_hex(q[4:]))
print 'g_%s IN TXT "%s"' % (keyname, binascii.b2a_hex(g[4:]))

```

Figure 8.5: Generating DSA Public Key DNS Records

Similar to the four-part scheme described above, the key hash is stored as a text record. The content of the record is the SHA-1 hash of the PEM representation of the public half of the signing key. The actual public key is stored on the key-server, with the specified name, in PEM format.

While this makes verifying a retrieved public key slightly more complex, requiring an additional request/response to the key server to retrieve the verification key, clients should cache the retrieved verification key, amortizing the additional work across multiple requests.

8.3 Key Server

8.3.1 Software Dependencies

We chose to implement the key-server in Python [100], using the Zolera Soap Infrastructure (ZSI) [103], *mod_python* [117] and the Apache [44] web server. The M2Crypto [111] wrapper provided Python access to the cryptographic functionality in the OpenSSL [128] library. The server used SQLite [113], an embeddable SQL'92 compliant RDBMS engine, and *pysqlite* [91], a Python DB-API [110] compliant interface layer, for underlying data storage.

During our work, we identified and corrected defects in the ZSI framework code, and added additional functionality to the M2Crypto python cryptographic library. In both cases, our changes were contributed back to the maintainers of those projects.

8.3.2 Architecture

The server functionality is composed of two components — one that processes query requests, and one that processes registration requests.² The two components have a similar structure: a handler for incoming requests that uses the ZSI framework to parse the incoming SOAP request into a request object and dispatches to the appropriate implementation method based on the operation requested. Figure 8.6 illustrates the component layers. The implementation extracts needed information from the request object, performs the requested

²Support for key revocation requests is not currently implemented.

operation, and creates an appropriate response object that is serialized by the ZSI layer into a SOAP response.

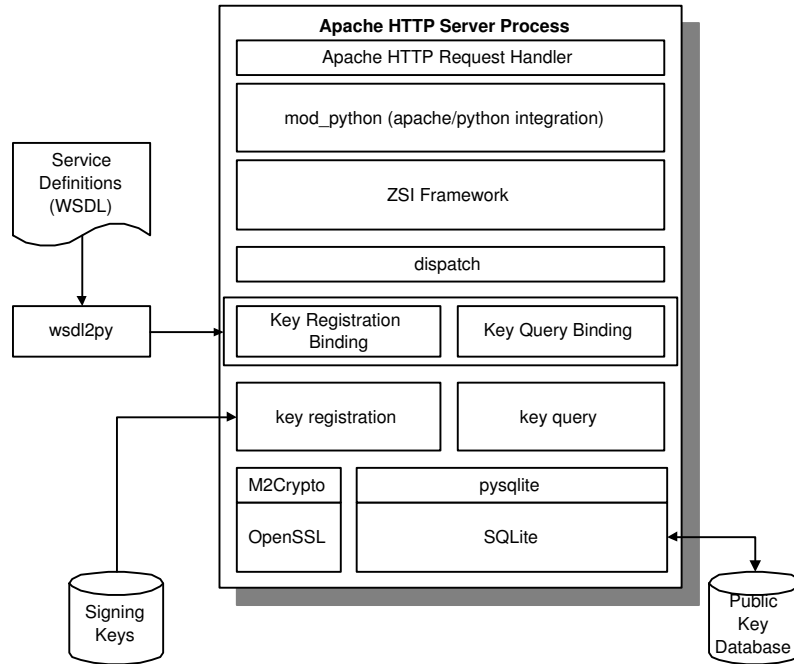


Figure 8.6: Server Architecture Layer Diagram

Both the query and registration components of the key server use a relational database for storage and data access. The schema (expressed in SQL'92) is included as Appendix B.

8.3.3 Key Registration

The registration server component is invoked by Apache for all requests destined to `http://server.name/ikrs`. Currently, the key registration server assumes the Apache instance is handling authentication, and processes incoming requests without additional authentication steps. The request object, including the supplied key, is handed through a set

of configured key-parsers, each capable of extracting meta-data from a specific key format. These key parsers can ensure, for example, that meta-data supplied in the registration request (such as key bit length, container type, and, in the case of X.509 certificates, bound names) actually match the attributes of the supplied key.

In our current implementation, the resulting key is signed at registration time, and the key, associated meta-data and signature are transformed into a set of SQL *INSERT* statements and stored.

Performance

Running on a single CPU (1.8Ghz Pentium 4M) machine, our initial implementation, running in debug mode and driven by a Python client running on the same host, performs two point eight (2.8) key registrations per second. Much of this time is spent in serialization/deserialization on both sides, and each request results in five (5) kilobytes of SOAP messages exchanged between client and server. A simple C client implementation, also with debug code, performs seven (7) registrations per second — a significant improvement over the Python version and suggesting that the Python serialization/deserialization code (shared with the server) is a source of inefficiency.

8.3.4 Key Retrieval

The query server component is invoked by Apache for all requests destined to `http://server.name/ikqs`. The request object is transformed into an SQL query against the underlying data store. The result-set is transformed into a response object that is returned to the ZSI layer, that performs serialization into the XML SOAP format. By design, the key query server component performs no updates against the database, allowing it to run against a read-only database replica. In our current implementation no explicit caching is performed.

Performance

Running on a single CPU (1.8Ghz Pentium 4M) machine, our initial implementation, running in debug mode and driven by a simple Python client running on the same host, performs one point six (1.6) key queries per second. A simple C client implementation performs 6.3 queries per second — again, a significant improvement over the Python version. As with key registration, much of the execution time is spent in serialization/deserialization — each request results in six point five (6.5) kilobytes of SOAP messages exchanged between client and server. One reason for the reduction in performance of key retrieval compared to key registration seems to be the complexity of the SQL query performed to evaluate the submitted key query. It may be possible to reduce this complexity by simplifying the database schema.

8.4 Client Library Application Programmer Interface

In the interest of encouraging adoption by providing potentially reusable client code, we implemented our client library in ANSI C, and were conscious to limit external dependencies. We leverage the cryptographic routines in OpenSSL [128] for key query verification, and SOAP client stubs generated using gSOAP [39].³ Figure 8.7 illustrates the various component layers of the client library.

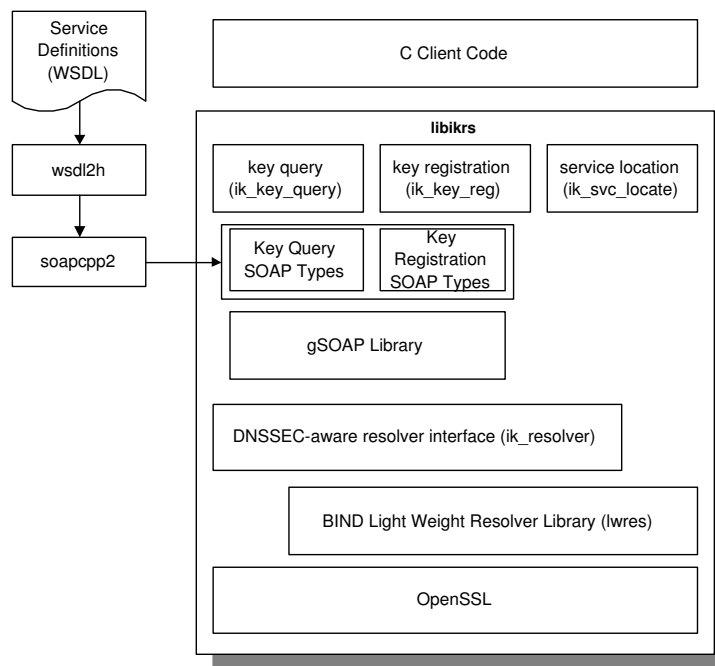


Figure 8.7: Client Library Layer Diagram

The gSOAP library hides name resolution from the caller, and uses the standard resolver interfaces which are unaware of DNSSEC. To address this, we modified the library to allow

³We note that, due to time constraints, we did not perform a careful audit of the gSOAP codebase. A careful inspection should be undertaken before any real-world deployment based on this code is considered.

a client to register a call-back function to perform name resolution. This and other small modifications were submitted to the gSOAP maintainers and are expected to appear in a future release.

In the following sections, we briefly describe the APIs provided by our client library for performing key lookup and key registration. The APIs are intentionally as simple as possible, providing “course grain” operations and reducing the number of library calls required to accomplish the desired operation.

8.4.1 Key Query

The basic process for performing a query consists of four steps:

1. creating and populating a *KeyQueryRequest* structure — which contains all the various query parameters (name, service, key format, etc.),
2. locating the query service host(s) for the domain of interest,
3. submitting the query request to the identified hosts, and
4. extracting the results from the resulting *KeyQueryResponse* structure.

Helper functions are provided to process query responses — including performing cryptographic validation. The specifics of the query interface appear in Appendix C.1.

8.4.2 Key Registration

The registration interface is highly similar to the query interface — requiring the client to create and populate a *KeyRegistrationRequest* message, locate the appropriate registration target URI for the domain of interest, submit the registration request and process the response. The calling client is responsible for handling authentication — both “internal,” as realized by HTTP basic authentication, and “external,” as realized by the HTTP redirection mechanism described in Section 6.4.5. At present, the library supports only HTTP “basic” authentication, digest authentication is not supported, and only preliminary support for external authentication exists. The specifics of the query interface appear in Appendix C.2.

8.5 A Simple Client

Using the developed client library, simple command-line registration and query clients were developed. Not suitable or intended for actual use, they were developed primarily to demonstrate that the client library provided all necessary features.

The client constructs a registration (or query) message from statically configured data and uses the client library described in Section 8.4 to locate the appropriate key-server, submit the request, and validate the response (including, in the case of queries, fetching verifying keys from DNSSEC and cryptographically validating returned query results).

Excluding argument parsing, the clients consisted of approximately 100 lines of code,

including rudimentary error handling. Based on this evidence, we feel confident that developing an “industrial-strength” client, potentially integrated with the graphical user interface of a communication tool, is completely feasible.

Chapter 9

Conclusion

“Begin at the beginning,” the King said gravely, “and go on till you come to the end: then stop.”

The King of Hearts,

Lewis Carroll; *Alice’s Adventures In Wonderland*

In this work we attempt to aid development and widespread adoption of cryptographically-enabled applications by facilitating distributed, authenticated key distribution; an often overlooked requirement for practical cryptographic tools. We design and specify a protocol for performing key query and registration, including service location, query, registration, and revocation facilities. The protocol has been specified to provide the expressiveness required by the problem domain, and built using current industry best practices and standards for remote service location and invocation.

By relying on DNSSEC to provide authenticated and trustworthy name resolution and

delegation, we keep the function of key distribution outside the critical name server infrastructure. This allows us to use the name service infrastructure to provide authenticity guarantees while avoiding scalability, efficiency, and administrative pitfalls of previously proposed, DNS-based, mechanisms. Furthermore we operate on DNS names, rather than inventing a new namespace, eliminating a crucial conceptual barrier for end-users.

We described our construction of a prototypical server and client access library in support of this protocol. The resulting implementation, consisting of registration and query server components, and a client registration and query library, represents approximately 1000 lines of Python and 2000 lines of C code and demonstrates performance adequate to encourage further study. Developing a simple client using this infrastructure shows that the client-side development effort is minimal, suggesting that incorporating this functionality into existing collaborative tools is tractable.

It is our hope that as DNSSEC gains adoption and penetration, our contribution can be used to facilitate ubiquitous cryptographic key location and exchange, improving the security of existing network applications and protocols, and enabling new developments as yet unimagined. In the future, when Alice needs to locate Bob's key, the protocol for doing so should be well defined and realizable.

Future Work

We intend to continue improving the existing prototype implementation, readying it for production in tandem with DNSSEC. The query performance of the Python prototype is particularly concerning — reimplementing the server in a language with a more robust tool-chain will allow more effective profiling and optimization. A higher-level, possibly object-oriented, interface to the client library, which was written in C in this effort for maximum applicability, would simplify integration.

A more drastic change could be to move more query processing to the client, rather than the server. While our thoughts in this area are still preliminary, it might be possible to re-think portions of the query protocol to allow the server to provide pre-signed static data, moving the query work to the client — retrieving specific keys of interest (and their signatures) by unique id only. This could potentially vastly improve the scalability of the scheme, by pushing work to the edges and allowing the server to serve static data.

Ultimately, we intend to submit the protocol definition to the IETF for widespread review and comment — hopefully ultimately publishing a draft or standards-track RFC.

Bibliography

- [1] Abstract Syntax Notation One (ASN.1) Specification of Basic Notation. ITU-T Rec. X.680 (2002), ISO/IEC 8824-1:2002, 2002.
- [2] H. Abelson, R. Anderson, S. Bellovin, J. Benaloh, M. Blaze, W. Diffie, J. Gilmore, P. Neumann, R. Rivest, J. Schiller, and B. Schneier. The Risks of Key Recovery, Key Escrow, and Trusted Third-Party Encryption. <http://www.cdt.org/crypto/risks98/>, 1998.
- [3] D. Akin. Arrests key win for NSA hackers. The Globe And Mail, April 2004. <http://www.globetechnology.com/servlet/story/RTGAM.20040406.gtterror06/BNStory/Technology/>.
- [4] R. Anderson. Why Cryptosystems Fail. In *Proc. First ACM Conference on Computer and Communications Security*, pages 215–227, 1993.
- [5] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. DNS Security Introduction and Requirements. <http://www.ietf.org/internet-drafts/draft-ietf-dnsext-dnssec-intro-09.txt>, February 2004.
- [6] R. Arends, M. Larson, R. Austein, D. Massey, and S. Rose. Protocol Modifications for the DNS Security Extensions. <http://www.ietf.org/internet-drafts/draft-ietf-dnsext-dnssec-protocol-06.txt>, May 2004. Updates RFC 2535.
- [7] D. Atkins and R. Austein. Threat Analysis of the Domain Name System. <http://www.ietf.org/internet-drafts/draft-ietf-dnsext-dns-threats-07.txt>, April 2004.
- [8] R. Atkinson. Key Exchange Delegation Records for the DNS. <http://www.ietf.org/rfc/rfc2230.txt>, November 1997.
- [9] E. Baize and D. Pinkas. The Simple and Protected GSS-API Negotiation Mechanism. <http://www.ietf.org/rfc/rfc2478.txt>, December 1998.
- [10] D. Balenson. Privacy Enhancement for Internet Electronic Mail: Part III: Algorithms, Modes, and Identifiers. <http://www.ietf.org/rfc/rfc1423.txt>, February 1993.

- [11] J. Bamford. *Body of Secrets: Anatomy of the Ultra-Secret National Security Agency*. Doubleday, 2001.
- [12] J. Bamford. *The Puzzle Palace: A Report on America's Most Secret Agency*. Viking Press, 2001.
- [13] S. Bellovin. Using the Domain Name System for System Break-ins. In *Proc. Fifth USENIX Security Symposium*, June 1995.
- [14] D. J. Bernstein. DNS Forgery. <http://cr.yp.to/djbdns/forgery.html>.
- [15] P. Biron and A. Malhotra. XML Schema Part 2: Datatypes. <http://www.w3.org/TR/xmlschema-2/>, May 2001.
- [16] S. Boeyen, T. Howes, and P. Richard. Internet X.509 Public Key Infrastructure LDAPv2 Schema. <http://www.ietf.org/rfc/rfc2587.txt>, June 1999.
- [17] D. Boneh and M. Franklin. Identity-based Encryption from the Weil Pairing. In *Proc. of CRYPTO 01*, pages 213–229, 2001.
- [18] J. Brezak. HTTP Authentication: Kerberos Authentication As implemented in Microsoft Windows 2000. <http://www.ietf.org/internet-drafts/draft-brezak-kerberos-http-01.txt>, September 2001. Expired and removed, archival copies may be found at <http://www.google.com/search?q=draft-brezak-spnego-http-00.txt>.
- [19] N. Brownlee, kc Claffy, and E. Nemeth. DNS Measurements at a Root Server. In *Proc. of Globecom 2001*. IEEE, November 2001.
- [20] J. Callas, L. Donnerhackle, H. Finney, and R. Thayer. OpenPGP Message Format. <http://www.ietf.org/rfc/rfc2440.txt>, November 1998.
- [21] E. Christensen, F. Curbera, G. Meredit, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. <http://www.w3.org/TR/wsdl>, March 2001.
- [22] C. Cocks. A Note on Non-Secret Encryption. <http://www.cesg.gov.uk/site/publications/media/notense.pdf>, November 1973.
- [23] Internet Software Consortium. Internet Domain Survey Host Count. <http://www.isc.org/index.pl?ops/ds/>, May 2004.
- [24] Microsoft Corp. Active Directory Overview. <http://www.microsoft.com/windows2000/server/evaluation/features/dirlist.asp>, June 1999.
- [25] D. Crocker. Standard for the Format of ARPA Internet Text Messages. <http://www.ietf.org/rfc/rfc822.txt>, August 1982.

- [26] S. Crocker and R. Mundy. An Invitation: Building a Road Map for DNSSEC Deployment. <http://www.sdl.sri.com/other/dnssec/DNSSECDeploymentRoadMapChallenge.pdf>, April 2004.
- [27] M. Delany. Domain-based Email Authentication Using Public-Keys Advertised in the DNS (DomainKeys). <http://www.ietf.org/internet-drafts/draft-delany-domainkeys-base-00.txt>, May 2004.
- [28] T. Dierks and C. Allen. The TLS Protocol Version 1.0. <http://www.ietf.org/rfc/rfc2246.txt>, January 1999.
- [29] W. Diffie and M. E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.
- [30] D. Eastlake. DNS Security Operational Considerations. <http://www.ietf.org/rfc/rfc2541.txt>, March 1999.
- [31] D. Eastlake. Domain Name System Security Extensions. <http://www.ietf.org/rfc/rfc2535.txt>, March 1999.
- [32] D. Eastlake. DSA KEYS and SIGs in the Domain Name System (DNS). <http://www.ietf.org/rfc/rfc2536.txt>, March 1999.
- [33] D. Eastlake. Storage of Diffie-Hellman Keys in the Domain Name System (DNS). <http://www.ietf.org/rfc/rfc2539.txt>, March 1999.
- [34] D. Eastlake. RSA/SHA-1 SIGs and RSA KEYS in the Domain Name System (DNS). <http://www.ietf.org/rfc/rfc3110.txt>, May 2001.
- [35] D. Eastlake and O. Gudmundsson. Storing Certificates in the Domain Name System (DNS). <http://www.ietf.org/rfc/rfc2538.txt>, March 1999.
- [36] D. Eastlake and C. Kaufman. Domain Name System Security Extensions. <http://www.ietf.org/rfc/rfc2065.txt>, January 1997. Superseded by RFC 2535.
- [37] J. H. Ellis. The Possibility of Secure Non-Secret Digital Encryption. <http://www.cesg.gov.uk/site/publications/media/possnse.pdf>, January 1970.
- [38] *mozdev.org*. Mozilla Negotiate Auth Plugin. <http://negotiateauth.mozdev.org/>.
- [39] R. Englen and K. Gallivan. gSOAP: C/C++ Web Services and Clients. gsoap2.sourceforge.net.
- [40] P. Eronen. Applying Decentralized Trust Management to DNS Dynamic Updates. In *Proc. Third NordU/USENIX Conference*, 2001.

- [41] D. Fallside. XML Schema Part 0: Primer. <http://www.w3.org/TR/xmlschema-0/>, May 2001.
- [42] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. <http://www.ietf.org/rfc/rfc2616.txt>, June 1999.
- [43] Organization for the Advancement of Structured Information Standards. OASIS Mobilizes to Overcome Challenges to PKI Adoption. http://www.oasis-open.org/news/oasis_news_02_23_04.php, February 2004.
- [44] The Apache Foundation. The Apache HTTP Server Project. <http://httpd.apache.org>.
- [45] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. HTTP Authentication: Basic and Digest Access Authentication. <http://www.ietf.org/rfc/rfc2617.txt>, June 1999.
- [46] A. Freier, P. Karlton, and P. Kocher. The SSL Protocol Version 3.0. <http://wp.netscape.com/eng/ssl3/draft302.txt>, November 1996.
- [47] K. Fu, E. Sit, K. Smith, and N. Feamster. Dos and Don'ts of Client Authentication on the Web. In *Proc. 10th USENIX Security Symposium*, Aug 2001.
- [48] J. Galbrith. SSH Public Key File Format. <http://www.ietf.org/internet-drafts/draft-ietf-secsh-publickeyfile-05.txt>, April 2004.
- [49] J. Galvin. Public Key Distribution with Secure DNS. In *Proc. Sixth USENIX Security Symposium*, 1996.
- [50] R. Gieben. DNSSEC in NL. <http://www.miek.nl/publications/dnssecnl/segreg-report.ps>, January 2004.
- [51] R. Gieben, G. Guette, and O. Courtay. DNSSEC Resolver Interfacer to Applications. <http://www.nlnetlabs.nl/dnssec/draft-gieben-resolver.txt>, January 2004. Expired July 1, 2004.
- [52] E. Glass. The NTLM Authentication Protocol. <http://davenport.sourceforge.net/ntlm.html>, 2003.
- [53] IETF S/MIME Working Group. S/MIME RFC's. <http://www.ietf.org/html.charters/smime-charter.html>.
- [54] M. Gudgin, M. Hadley, N. Mendelsohn, J. Moreau, and H. Nielsen. SOAP Version 1.2 Part 1: Messaging Framework, June 23.

- [55] M. Gudgin, M. Hadley, N. Mendelsohn, J. Moreau, and H. Nielsen. SOAP Version 1.2 Part 2: Adjuncts, June 23.
- [56] O. Gudmundsson. Delegation Signer (DS) Resource Record (RR). <http://www.ietf.org/rfc/rfc3658.txt>, November 2003.
- [57] A. Gulbrandsen, P. Vixie, and L. Esibov. A DNS RR For Specifying the Location of Services (DNS SRV). <http://www.ietf.org/rfc/rfc2782.txt>, March 1999.
- [58] P. Gutmann. Lessons Learned in Implementing and Deploying Crypto Software. In *Proc. Eleventh USENIX Security Symposium*, August 2002.
- [59] N. Haller. The S/KEY One-Time Password System. <http://www.ietf.org/rfc/rfc1760.txt>, February 1995.
- [60] N. Haller, C. Metz, P. Nesser, and M. Straw. A One-Time Password System. <http://www.ietf.org/rfc/rfc2289.txt>, February 1998.
- [61] M. Horowitz. A PGP Public Key Server. Undergraduate Thesis, MIT, 1996.
- [62] R. Housley, W. Polk, W. Ford, and D. Solo. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. <http://www.ietf.org/rfc/rfc3280.txt>, April 2002.
- [63] J. Hubaux, L. Buttyán, and S. Capkun. The Quest for Security in Mobile Ad Hoc Networks. In *Proc. 2nd ACM International Symposium on Mobile ad hoc Networking and Computing*, pages 146–155, 2001.
- [64] IEEE Pervasive Computing: Mobile and Ubiquitous Systems. <http://www.computer.org/pervasive/>, May 2004.
- [65] Novell Inc. Novell Removes Barriers to Widespread Adoption of Directory Services. <http://developer.novell.com/research/devnotes/1997/january/11/>, January 1997.
- [66] ISO and CCITT, editors. *Recommendation X.500: The Directory: Overview of Concepts, Models and Services*. ITU, 1993.
- [67] ISO/IEC 11578 Information Technology – Open Systems Interconnection – Remote Procedure Call (RPC), August 2001.
- [68] B. Kaliski. Privacy Enhancement for Internet Electronic Mail: Part IV: Key Certification and Related Services. <http://www.ietf.org/rfc/rfc1424.txt>, February 1993.
- [69] B. Kaliski. PKCS #7: Cryptographic Message Syntax Version 1.5. <http://www.ietf.org/rfc/rfc2315.txt>, March 1998.

- [70] S. Kent. Privacy Enhancement for Internet Electronic Mail: Part II: Certificate-Based Key Management. <http://www.ietf.org/rfc/rfc1422.txt>, February 1993.
- [71] S. Kent and R. Atkinson. Security Architecture for the Internet Protocol. <http://www.ietf.org/rfc/rfc2401.txt>, November 1998.
- [72] J. Kobiellus. Simplification, not XML, is the key to PKI success. <http://www.nwfusion.com/columnists/2001/0507kobiellus.html>, May 2001.
- [73] J. Kohl and C. Neuman. The Kerberos Network Authentication Service (V5). <http://www.ietf.org/rfc/rfc1510.txt>, September 1993.
- [74] O. Kolkman. DNSSEC Operational HOWTO. <http://www.ripe.net/disi/>, November 2002.
- [75] D. Kouril. Kerberos Module for Apache. <http://modauthkerb.sourceforge.net/>.
- [76] P. Leach and C. Newman. Using Digest Authentication as a SASL Mechanism. <http://www.ietf.org/rfc/rfc2831.txt>, May 2000.
- [77] J. Linn. Generic Security Service Application Program Interface. <http://www.ietf.org/rfc/rfc1508.txt>, September 1993. Superseded by RFC 2078.
- [78] J. Linn. Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures. <http://www.ietf.org/rfc/rfc1421.txt>, February 1993.
- [79] J. Linn. Generic Security Service Application Program Interface, Version 2. <http://www.ietf.org/rfc/rfc2078.txt>, January 1997.
- [80] D. Massey and S. Rose. Limiting the Scope of the KEY Resource Record (RR). <http://www.ietf.org/rfc/rfc3445.txt>, December 2002.
- [81] P. McDaniel and S. Jamin. A Scalable Key Distribution Hierarchy. Technical Report CSE-TR-366-98, E.E. & C.S. Department, University of Michigan, March 1998.
- [82] J. Meyers. Simple Authentication and Security Layer (SASL). <http://www.ietf.org/rfc/rfc2222.txt>, October 1997.
- [83] N. Mitra. SOAP Version 1.2 Part 0: Primer, June 23.
- [84] P. Mockapetris. Domain Names – Concepts and Facilities. <http://www.ietf.org/rfc/rfc882.txt>, November 1983. Superseded by RFC 1034.
- [85] P. Mockapetris. Domain Names – Implementation and Specification. <http://www.ietf.org/rfc/rfc882.txt>, November 1983. Superseded by RFC 1035.

- [86] P. Mockapetris. Domain Names – Implementation and Specification. <http://www.ietf.org/rfc/rfc1035.txt>, November 1987.
- [87] M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams. X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP. <http://www.ietf.org/rfc/rfc2560.txt>, June 1999.
- [88] C. Newman. Using TLS with IMAP, POP3 and ACAP. <http://www.ietf.org/rfc/rfc2595.txt>, June 1999.
- [89] D. A. Norman. *The Design of Everyday Things*. MIT Press, 1998.
- [90] N. Nystrom. The SecurID(r) SASL Mechanism. <http://www.ietf.org/rfc/rfc2808.txt>, April 2000.
- [91] M. Owens and G. Häring. PySQLite. <http://pysqlite.sourceforge.net>.
- [92] *System Application Program Interface (API) [C Language]*. Information technology—Portable Operating System Interface (POSIX). 1990.
- [93] J. B. Postel. Simple Mail Transport Protocol. <http://www.ietf.org/rfc/rfc821.txt>, August 1982.
- [94] FreeS/WAN Project. FreeS/WAN. <http://www.freeswan.org/>.
- [95] Gnu Privacy Guard Project. <http://www.gnupg.org>.
- [96] M. Reiter and S. Stubblebine. Path Independence for Authentication in Large-Scale Systems. In *Proc. 4th ACM Conference on Computer and Communications Security*, pages 57–66, 1997.
- [97] J. Reynolds and J. Postel. Assigned Numbers. <http://www.ietf.org/rfc/rfc1700.txt>, October 1994.
- [98] R. Rivest and B. Lampson. SDSI – A Simple Distributed Security Infrastructure. <http://theory.lcs.mit.edu/~rivest/sdsi10.ps>, 1996.
- [99] R. L. Rivest, A. Shamir, and L. M. Adelman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. Technical Report MIT/LCS/TM-82, 1977.
- [100] G. Rossum. Python. <http://www.python.org>.
- [101] Ed. S. Joseffson. The Base16, Base32, and Base64 Data Encodings. <http://www.ietf.org/rfc/rfc3548.txt>, July 2003.
- [102] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-End Arguments in System Design. *ACM Transactions on Computer Systems*, 2(4):277–288, 1984.
- [103] R. Salz. Zolera Soap Infrastructure. <http://pywebsvcs.sourceforge.net>.

- [104] J. Schlyter and W. Griffin. Using DNS to Securely Publish SSH Key Fingerprints. <http://www.ietf.org/internet-drafts/draft-ietf-secsh-dns-05.txt>, September 2003.
- [105] B. Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, second edition, October 1995.
- [106] RSA Security. RSA SecurID Authentication. <http://www.rsasecurity.com/node.asp?id=1156>.
- [107] RSA Security. RSA Security To Help Accelerate Broad Adoption of PKI Digital Certificates with Next Generation RSA Keon Family. http://www.rsasecurity.com/press_release.asp?doc_id=252, August 2000.
- [108] A. Shamir. Identity-based Cryptosystems and Signature Schemes. In *Advances In Cryptography - Crypto '84*, volume 196 of *Lecture Notes in Computer Science*, pages 47–53. Springer-Verlag, 1984.
- [109] C. Shirky. RIAA Succeeds Where the Cypherpunks Failed. http://www.shirky.com/writings/riaa_encryption.html, December 2003.
- [110] Python Database SIG. Python Database API Specification 2.0. www.python.org/peps/pep-0249.html.
- [111] N. P. Siong. M2Crypto. <http://sandbox.rulemaker.net/ngps/m2>.
- [112] D. K. Smetters and G. Durfee. Domain-Based Administration of Identity-Based Cryptosystems for Secure Email and IPSEC. In *Proc. Twelfth USENIX Security Symposium*, pages 215–230, August 2003.
- [113] SQLite.org. SQLite. <http://www.sqlite.org>.
- [114] ClickZ Stats. Population Explosion! http://www.clickz.com/stats/big_picture/geographics/article.php/5911_151151, April 2004.
- [115] H. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. XML Schema Part 1: Structures. <http://www.w3.org/TR/xmlschema-1/>, May 2001.
- [116] W. Treese. Personal Correspondance. June 2004.
- [117] G. Trubetskoy. mod_python: Apache/Python Integration. <http://www.modpython.org>.
- [118] Verisign. Introduction to Public Key Cryptography. <http://www.verisign.com/repository/crptintr.html>, April 2004.
- [119] P. Vixie. Berkeley Internet Name Domain. <http://www.isc.org/index.pl?/sw/bind/>.

- [120] P. Vixie. DNS and BIND Security Issues. In *Proc. Fifth USENIX Security Symposium*, June 1995.
- [121] X. Wang, Y. Huang, Y. Desmedt, and D. Rine. Enabling Secure On-line DNS Dynamic Update. In *Proc. Annual Computer Security Applications Conference*, December 2000.
- [122] B. Wellington. Secure Domain Name System (DNS) Dynamic Update. <http://www.ietf.org/rfc/rfc3007.txt>, November 2000.
- [123] B. Wellington and O. Gudmundsson. Redefinition of DNS Authenticated Data (AD) bit. <http://www.ietf.org/rfc/rfc3655.txt>, November 2003.
- [124] A. Whitten and J. D. Tygar. Why Johnny can't Encrypt: A Usability Evaluation of PGP 5.0. In *Proc. 8th USENIX Security Symposium*, 1999.
- [125] M. Williamson. Non-Secret Encryption Using a Finite Field. <http://www.cesg.gov.uk/site/publications/media/secenc.pdf>, January 1974.
- [126] W. Yeong, T. Howes, and S. Kille. X.500 Lightweight Directory Access Protocol. <http://www.ietf.org/rfc/rfc1487.txt>, July 1993. Superseded by RFC 1777.
- [127] T. Ylonen and D. Moffat. SSH Protocol Architecture. <http://www.ietf.org/internet-drafts/draft-ietf-secsh-architecture-15.txt>, October 2003.
- [128] E. Young and T. Hudson. OpenSSL. <http://www.openssl.org>.
- [129] P. Zimmerman. *The Official PGP User's Guide*. MIT Press, 1995.

Appendix A

Service Interface Descriptions (WSDL)

WSDL makes heavy use of XML Schema [115, 15] as a typing language — readers unfamiliar with XML Schema may refer to the XML Schema Primer [41].

A.1 Key Query Service

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
  $Id: key-query-svc.wsdl,v 1.12 2004/08/12 21:36:30 dberger Exp $
-->
<definitions name="InternetKeyQueryService"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:tns="urn:keyquery.querysvc.ikaros"
  targetNamespace="urn:keyquery.querysvc.ikaros">

  <types>
    <xsd:schema xmlns="http://www.w3.org/2001/XMLSchema"
      targetNamespace="urn:keyquery.querysvc.ikaros">

      <xsd:simpleType name="KeyUseString">
        <xsd:restriction base="xsd:string">
          <!-- none is a placeholder, so the default isn't sensible -->
          <xsd:enumeration value="none"/>
          <xsd:enumeration value="privacy"/>
          <xsd:enumeration value="authenticity"/>
          <xsd:enumeration value="privacy+authenticity"/>
        </xsd:restriction>
      </xsd:simpleType>

      <xsd:complexType name="ArrayOfString">
        <xsd:complexContent>
```

```

        <xsd:restriction base="soapenc:Array">
            <xsd:attribute ref="soapenc:arrayType"
                wsdl:arrayType="xsd:string []"/>
        </xsd:restriction >
    </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="KeyQueryRequest">
    <xsd:sequence>
        <xsd:element name="version"
            nillable="false" type="xsd:decimal"/>
        <xsd:element name="entityName"
            nillable="true" type="xsd:string"/>
        <xsd:element name="serviceName"
            nillable="true" type="xsd:string"/>
        <xsd:element name="uid"
            nillable="true" type="xsd:string"/>
        <xsd:element name="fingerprint"
            nillable="true" type="xsd:string"/>
        <!--
            the list of acceptable formats and algorithms is
            treated as a logical or.
        -->
        <xsd:element name="formats"
            nillable="true" type="tns:ArrayOfString"/>
        <xsd:element name="algorithms"
            nillable="true" type="tns:ArrayOfString"/>
        <xsd:element name="length"
            nillable="true" type="xsd:unsignedInt"/>
        <xsd:element name="permittedUse"
            nillable="true" type="tns:KeyUseString"/>
        <xsd:element name="validAfter"
            nillable="true" type="xsd:unsignedInt"/>
        <xsd:element name="validUntil"
            nillable="true" type="xsd:unsignedInt"/>
    </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="KeyQueryResponseHeader">
    <xsd:sequence>
        <xsd:element name="matchCount"
            nillable="true" type="xsd:unsignedInt"/>
        <xsd:element name="partialResult"
            nillable="true" type="xsd:boolean"/>
        <xsd:element name="ignoredParams"
            nillable="true" type="tns:ArrayOfString"/>
        <xsd:element name="queryTime"
            nillable="true" type="xsd:unsignedInt"/>
        <xsd:element name="responseTime"
            nillable="true" type="xsd:unsignedInt"/>
    </xsd:sequence>
</xsd:complexType>

```

```

<xsd:complexType name="KeyQueryResponseRevocation">
  <xsd:sequence>
    <xsd:element name="revokedAt"
      nillable="false" type="xsd:unsignedInt"/>
    <xsd:element name="revocationCertificate"
      nillable="false" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="KeyQueryResponseKey">
  <xsd:sequence>
    <xsd:element name="keyBits"
      nillable="false" type="xsd:base64Binary"/>
    <xsd:element name="permittedUse"
      nillable="false" type="tns:KeyUseString"/>
    <xsd:element name="validAfter"
      nillable="true" type="xsd:unsignedInt"/>
    <xsd:element name="validUntil"
      nillable="true" type="xsd:unsignedInt"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="KeyQueryResponseSignature">
  <xsd:sequence>
    <xsd:element name="createTime"
      nillable="false" type="xsd:unsignedInt"/>
    <xsd:element name="expireTime"
      nillable="false" type="xsd:unsignedInt"/>
    <xsd:element name="signingKeyName"
      nillable="false" type="xsd:string"/>
    <xsd:element name="algorithm"
      nillable="false" type="xsd:string"/>
    <xsd:element name="signatureBits"
      nillable="false" type="xsd:base64Binary"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="KeyQueryResponseKeyRecord">
  <xsd:sequence>
    <xsd:element name="entityName"
      nillable="false" type="xsd:string"/>
    <xsd:element name="serviceName"
      nillable="false" type="xsd:string"/>
    <xsd:element name="uid"
      nillable="false" type="xsd:string"/>
    <xsd:element name="format"
      nillable="false" type="xsd:string"/>
    <xsd:element name="algorithm"
      nillable="false" type="xsd:string"/>
    <xsd:element name="length"
      nillable="false" type="xsd:unsignedInt"/>
  </xsd:sequence>
</xsd:complexType>

```

```

        <!-- either a key, or a revocation block -->
        <xsd:choice>
            <xsd:element name="key"
                nillable="true"
                type="tns:KeyQueryResponseKey"/>
            <xsd:element name="revocation"
                nillable="true"
                type="tns:KeyQueryResponseRevocation"/>
        </xsd:choice>
        <xsd:element name="signature"
            nillable="false"
            type="tns:KeyQueryResponseSignature"/>
    </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="ArrayOfKeyQueryResponseKeyRecord">
    <xsd:complexContent>
        <xsd:restriction base="soapenc:Array">
            <xsd:attribute ref="soapenc:arrayType"
                wsdl:arrayType="tns:KeyQueryResponseKeyRecord[]"/>
        </xsd:restriction>
    </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="KeyQueryResponse">
    <xsd:sequence>
        <xsd:element name="version"
            nillable="false" type="xsd:decimal"/>
        <xsd:element name="header"
            nillable="false"
            type="tns:KeyQueryResponseHeader"/>
        <xsd:element name="matches"
            nillable="true"
            type="tns:ArrayOfKeyQueryResponseKeyRecord"/>
    </xsd:sequence>
</xsd:complexType>
</xsd:schema>
</types>

<message name="KeyQueryMsg">
    <part name="request"
        type="tns:KeyQueryRequest"/>
</message>

<message name="KeyQueryResponseMsg">
    <part name="response"
        type="tns:KeyQueryResponse"/>
</message>

<portType name="KeyQueryPort">
    <operation name="lookupKey">
        <input message="tns:KeyQueryMsg"/>

```

```

        <output message="tns:KeyQueryResponseMsg"/>
    </operation>
</portType>

<binding name="KeyQueryBinding" type="tns:KeyQueryPort">
    <soap:binding style="rpc"
        transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="lookupKey">
        <soap:operation soapAction="lookupKey"/>
        <input>
            <soap:body use="encoded"
                encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                namespace="urn:keyquery.querysvc.ikaros"/>
        </input>
        <output>
            <soap:body use="encoded"
                encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                namespace="urn:keyquery.querysvc.ikaros"/>
        </output>
    </operation>
</binding>

<service name="KeyQueryService">
    <port binding="tns:KeyQueryBinding" name="KeyQueryPort">
        <soap:address location="http://localhost/key-query"/>
    </port>
</service>
</definitions>

```

A.2 Key Registration Service

```

<?xml version="1.0" encoding="UTF-8"?>
<!--
    $Id: key-registration-svc.wsdl,v 1.10 2004/08/18 18:46:37 dberger Exp $
-->
<definitions name="InternetKeyRegistrationService"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:tns="urn:keyreg.regsvc.ikaros"
    targetNamespace="urn:keyreg.regsvc.ikaros">

    <types>
        <xsd:schema xmlns="http://www.w3.org/2001/XMLSchema"
            targetNamespace="urn:keyreg.regsvc.ikaros">

            <xsd:simpleType name="KeyUseString">
                <xsd:restriction base="xsd:string">
                    <xsd:enumeration value="privacy"/>
                </xsd:restriction>
            </xsd:simpleType>
        </xsd:schema>
    </types>

```

```

        <xsd:enumeration value="authenticity" />
        <xsd:enumeration value="privacy+authenticity" />
    </xsd:restriction>
</xsd:simpleType>

<xsd:complexType name="ArrayOfString">
    <xsd:complexContent>
        <xsd:restriction base="soapenc:Array">
            <xsd:attribute ref="soapenc:arrayType"
                wsdl:arrayType="string []" />
        </xsd:restriction>
    </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="KeyRegistrationRequest">
    <xsd:sequence>
        <xsd:element name="version"
            nillable="false" type="xsd:decimal" />
        <xsd:element name="entityName"
            nillable="false" type="xsd:string" />
        <xsd:element name="serviceName"
            nillable="false" type="xsd:string" />
        <xsd:element name="format"
            nillable="false" type="xsd:string" />
        <xsd:element name="keyBits"
            nillable="false" type="xsd:base64Binary" />
        <xsd:element name="fingerprint"
            nillable="true" type="xsd:string" />
        <xsd:element name="revocationCert"
            nillable="true" type="xsd:base64Binary" />
        <xsd:element name="algorithm"
            nillable="false" type="xsd:string" />
        <xsd:element name="length"
            nillable="false" type="xsd:unsignedInt" />
        <xsd:element name="permittedUse"
            nillable="false" type="tns:KeyUseString" />
        <xsd:element name="validAfter"
            nillable="true" type="xsd:unsignedInt" />
        <xsd:element name="validUntil"
            nillable="true" type="xsd:unsignedInt" />
        <xsd:element name="authorizingSignature"
            nillable="true" type="xsd:base64Binary" />
    </xsd:sequence>
</xsd:complexType>

<xsd:complexType
    name="KeyRegistrationResponse">
    <xsd:sequence>
        <xsd:element name="version"
            nillable="false" type="xsd:decimal" />
        <xsd:element name="entityName"
            nillable="false" type="xsd:string" />
    </xsd:sequence>
</xsd:complexType>

```

```

        <xsd:element name="serviceName"
            nillable="false" type="xsd:string"/>
        <xsd:element name="uid"
            nillable="false" type="xsd:string"/>
        <xsd:element name="format"
            nillable="false" type="xsd:string"/>
        <xsd:element name="algorithm"
            nillable="false" type="xsd:string"/>
        <xsd:element name="length"
            nillable="false" type="xsd:unsignedInt"/>
        <xsd:element name="permittedUse"
            nillable="false" type="tns:KeyUseString"/>
        <xsd:element name="validAfter"
            nillable="false" type="xsd:unsignedInt"/>
        <xsd:element name="validUntil"
            nillable="false" type="xsd:unsignedInt"/>
    </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="KeyRevocationRequest">
    <xsd:sequence>
        <xsd:element name="version"
            nillable="false" type="xsd:decimal"/>
        <xsd:element name="entityName"
            nillable="false" type="xsd:string"/>
        <xsd:element name="serviceName"
            nillable="false" type="xsd:string"/>
        <xsd:element name="uid"
            nillable="false" type="xsd:string"/>
        <xsd:element name="revocationCert"
            nillable="true" type="xsd:base64Binary"/>
        <xsd:element name="authorizingSignature"
            nillable="true" type="xsd:base64Binary"/>
    </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="KeyRevocationResponse">
    <xsd:sequence>
        <xsd:element name="version"
            nillable="false" type="xsd:decimal"/>
        <xsd:element name="entityName"
            nillable="false" type="xsd:string"/>
        <xsd:element name="serviceName"
            nillable="false" type="xsd:string"/>
        <xsd:element name="uid"
            nillable="false" type="xsd:string"/>
    </xsd:sequence>
</xsd:complexType>

</xsd:schema>
</types>

```

```

<message name="KeyRegistration">
  <part name="request"
        type="tns:KeyRegistrationRequest"/>
</message>

<message name="KeyRegistrationResponse">
  <part name="response"
        type="tns:KeyRegistrationResponse"/>
</message>

<message name="KeyRevocation">
  <part name="request"
        type="tns:KeyRevocationRequest"/>
</message>

<message name="KeyRevocationResponse">
  <part name="response"
        type="tns:KeyRevocationResponse"/>
</message>

<portType name="KeyRegistrationPort">
  <operation name="registerKey">
    <input message="tns:KeyRegistration"/>
    <output message="tns:KeyRegistrationResponse"/>
  </operation>
  <operation name="revokeKey">
    <input message="tns:KeyRevocation"/>
    <output message="tns:KeyRevocationResponse"/>
  </operation>
</portType>

<binding name="KeyRegistrationBinding"
        type="tns:KeyRegistrationPort">
  <soap:binding style="rpc"
                transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="registerKey">
    <soap:operation soapAction="registerKey"/>
    <input>
      <soap:body use="encoded"
                encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                namespace="urn:keyreg.regsvc.ikaros"/>
    </input>
    <output>
      <soap:body use="encoded"
                encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                namespace="urn:keyreg.regsvc.ikaros"/>
    </output>
  </operation>
  <operation name="revokeKey">
    <soap:operation soapAction="revokeKey"/>
    <input>
      <soap:body use="encoded"
                encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                namespace="urn:keyreg.regsvc.ikaros"/>
    </input>
  </operation>
</binding>

```

```

        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="urn:keyreg.regsvc.ikaros"/>
    </input>
    <output>
        <soap:body use="encoded"
            encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
            namespace="urn:keyreg.regsvc.ikaros"/>
    </output>
</operation>
</binding>

<service name="KeyRegistrationService">
    <port binding="tns:KeyRegistrationBinding"
        name="KeyRegistrationPort">
        <soap:address location="http://localhost/key-registration"/>
    </port>
</service>
</definitions>

```

Appendix B

Key Server Database Schema

```
/*
   $Id: schema.sql,v 1.6 2004/08/17 23:06:50 dberger Exp $
*/

begin transaction;

drop table algorithm;
create table algorithm (id varchar primary key, name varchar not null);

drop table format;
create table format (id varchar primary key, name varchar not null);

drop table signing_key;
create table signing_key (id varchar primary key, name varchar not null);

drop table key;
create table key (id varchar primary key,
                 name varchar not null,
                 service varchar not null,
                 format_id integer not null,
                 algorithm_id integer not null,
                 length integer not null,
                 fingerprint varchar,
                 key_bits clob not null,
                 permitted_use integer not null,
                 valid_from unsigned bigint not null,
                 valid_until unsigned bigint,
                 signature_id varchar,
                 revocation_id varchar);

drop table revocation;
create table revocation (id varchar primary key,
                       revocation_bits blob not null,
                       revoked_at unsigned bigint);

drop table signature;
```

```
create table signature (id varchar primary key,  
    signing_key_id varchar not null,  
    signing_algorithm_id varchar not null,  
    signature_bits varchar not null,  
    create_time unsigned bigint not null,  
    expire_time unsigned bigint not null);  
  
commit;
```

Appendix C

Client Library API Definition

C.1 Key Query

```
/*
 * $Id: ik_key_query.h,v 1.13 2004/08/17 23:11:04 dberger Exp $
 *
 */

#ifndef _IK_KEY_QUERY_H
#define _IK_KEY_QUERY_H

#include "ik_common.h"
#include <lwres/netdb.h>

typedef struct {
    struct ns1__KeyQueryRequest *msg;
    char *domain;
    struct soap *ctx;
} KeyQueryRequest;

typedef struct {
    struct ns1__KeyQueryResponse *msg;
    char *domain;
    char *uri;
    struct soap *ctx;
} KeyQueryResponse;

/* create a new KeyQueryRequest structure , filling in the necessary
   fields. */
int ik_create_key_query(const char *ename, const char *domain,
                       const char *service, KeyQueryRequest **req);

int ik_create_key_query_ex(const char *ename,
                           const char *domain,
                           const char *service,
                           const char *format,
```

```

        const char *algorithm ,
        unsigned long key_len ,
        enum key_use uses ,
        KeyQueryRequest **req );

/* assorted get/set functions – fairly self explanatory. for all set
functions , copies are created of any provided pointers by the
library. for all get functions , the client is responsible for
making a copy of the returned value.

when not otherwise stated , a return value of 0 indicates success ,
and non 0 indicates error.
*/
int ik_set_key_query_name(const char *ename , KeyQueryRequest *req );
const char *ik_get_key_query_name(const KeyQueryRequest *req );

int ik_set_key_query_service(const char *service , KeyQueryRequest *req );
const char *ik_get_key_query_service(const KeyQueryRequest *req );

int ik_set_key_query_key_unique_id(const char *uuid ,
                                   KeyQueryRequest *req );
const char *ik_get_key_query_key_unique_id(const KeyQueryRequest *req );

int ik_set_key_query_key_fingerprint(const char *fprint ,
                                      KeyQueryRequest *req );
const char *ik_get_key_query_key_fingerprint(const KeyQueryRequest *req );

int ik_add_key_query_key_format(const char *format , KeyQueryRequest *req );

/* the return value is the number of formats in the char ** out
parameter) */
int ik_get_key_query_key_formats(const KeyQueryRequest *req ,
                                 char ***formats );

int ik_add_key_query_key_algorithm(const char *algo , KeyQueryRequest *req );

/* the return value is the number of formats in the char ** out
parameter) */
int ik_get_key_query_key_algorithms(const KeyQueryRequest *req ,
                                    char ***algorithms );

int ik_set_key_query_key_length(unsigned long len , KeyQueryRequest *req );
unsigned long ik_get_key_query_key_length(const KeyQueryRequest *req );

int ik_set_key_query_key_use(enum key_use use , KeyQueryRequest *req );
enum key_use ik_get_key_query_key_use(const KeyQueryRequest *req );

int ik_set_key_query_valid_after(unsigned long td , KeyQueryRequest *req );
unsigned long ik_get_key_query_valid_after(const KeyQueryRequest *req );

int ik_set_key_query_valid_until(unsigned long td , KeyQueryRequest *req );
unsigned long ik_get_key_query_valid_until(const KeyQueryRequest *req );

```

```

/* given a domain name, return a null-terminated sorted list of the
   key query uris available for that domain. They should be tried in
   returned order. The caller is responsible for calling
   ik_free_query_uri on the list when finished.*/

int ik_create_query_uris(const char *dname, char **uris []);

void ik_free_query_uris(char **uris []);

/* this actually does the work of submitting the query to the provided
   uri. the response is placed into rsp. */
int ik_submit_key_query(const char *uri, KeyQueryRequest *req,
                       KeyQueryResponse **rsp);

/* how many keys are in the response? allows the client to walk
   through them and validate them by calling
   ik_validate_key_query_keyrec() */
int ik_key_count(const KeyQueryResponse *rsp);

/* validates the idxth key in the key query response */
enum validate_result ik_validate_key_query_keyrec(int idx,
                                                  KeyQueryResponse *rsp);

/* return the ith key from the response – the caller is responsible
   for making a copy of this data, if needed */
unsigned const char *ik_get_key(int idx, const KeyQueryResponse *rsp,
                                unsigned int *keylen);

/* free resources allocated into the request and response objects by
   the library. */
void ik_cleanup_key_query(KeyQueryRequest *req,
                           KeyQueryResponse *rsp);

#endif

```

C.2 Key Registration

```

/*
 * $Id: ik_key_reg.h,v 1.7 2004/08/12 21:36:20 dberger Exp $
 *
 */

#ifndef _IK_KEY_REG_H
#define _IK_KEY_REG_H

#include <lwres/netdb.h>
#include "ik_common.h"

typedef struct {
    struct ns1__KeyRegistrationRequest *msg;

```

```

    struct soap *ctx;
} KeyRegistrationRequest;

typedef struct {
    struct ns1__KeyRegistrationResponse *msg;
    struct soap *ctx;
} KeyRegistrationResponse;

/* create a new KeyRegistrationRequest structure, filling in the
   necessary fields. */
int ik_create_key_registration(const char *ename, const char *service,
                              KeyRegistrationRequest **req);
int ik_create_key_registration_ex(const char *ename,
                                 const char *service,
                                 const char *format,
                                 const char *algorithm,
                                 unsigned long key_len,
                                 const unsigned char *key_bits,
                                 const unsigned char *revocation,
                                 enum key_use uses,
                                 KeyRegistrationRequest **req);

/* assorted get/set functions – fairly self explanatory. for all set
   functions, copies are created of any provided pointers by the
   library. for all get functions, the client is responsible for
   making a copy of the returned value.

   in all cases, a return value of 0 indicates success, and non 0
   indicates error.
*/
int ik_set_key_registration_name(const char *ename,
                                KeyRegistrationRequest *req);
char *ik_get_key_registration_name(const KeyRegistrationRequest *req);

int ik_set_key_registration_service(const char *service,
                                    KeyRegistrationRequest *req);
char *ik_get_key_registration_service(
    const KeyRegistrationRequest *req);

int ik_set_key_registration_key_fingerprint(const char *fprint,
                                             KeyRegistrationRequest *req);
char *ik_get_key_registration_key_fingerprint(
    const KeyRegistrationRequest *req);

int ik_set_key_registration_key_format(const char *format,
                                       KeyRegistrationRequest *req);
char *ik_get_key_registration_key_format(
    const KeyRegistrationRequest *req);

```

```

int ik_set_key_registration_key_algorithm(const char *algo ,
                                           KeyRegistrationRequest *req);
char *ik_get_key_registration_key_algorithm(
                                           const KeyRegistrationRequest *req);

int ik_set_key_registration_key_length(unsigned long len ,
                                         KeyRegistrationRequest *req);
unsigned long ik_get_key_registration_key_length(
                                           const KeyRegistrationRequest *req);

int ik_set_key_registration_key_use(enum key_use use ,
                                     KeyRegistrationRequest *req);
enum key_use ik_get_key_registration_key_use(
                                           const KeyRegistrationRequest *req);

int ik_set_key_registration_key(const unsigned char *keybits ,
                                 KeyRegistrationRequest *req);
unsigned char *ik_get_key_registration_key(
                                           const KeyRegistrationRequest *req);

int ik_set_key_registration_revocation_cert(
                                           const unsigned char *revocation ,
                                           KeyRegistrationRequest *req);
unsigned char *ik_get_key_registration_revocation_cert(
                                           const KeyRegistrationRequest *req);

int ik_set_key_registration_valid_after(unsigned long td ,
                                         KeyRegistrationRequest *req);
unsigned long ik_get_key_registration_valid_after(
                                           const KeyRegistrationRequest *req);

int ik_set_key_registration_valid_until(unsigned long td ,
                                         KeyRegistrationRequest *req);
unsigned long ik_get_key_registration_valid_until(
                                           const KeyRegistrationRequest *req);

/* provide a username and password to satisfy HTTP basic
   authentication */

int ik_set_key_registration_auth_credentials(const char *uname ,
                                              const char *passwd ,
                                              KeyRegistrationRequest *req);

/* given a domain name, return a null-terminated sorted list of the
   key registration uris available for that domain. They should be
   tried in returned order. The caller is responsible for calling
   ik_free_query_uri on the list when finished.*/

int ik_create_registration_uris(const char *dname, char **uris []);

void ik_free_registration_uris(char **uris []);

```

/ this actually does the work of submitting the query to the provided uri. the response is placed into rsp. The gsoap error code is propagated up – and may be a request for authentication, or redirect (per the protocol definition – a 307 can be used to hand off the registration client to some external authentication process. The client has to handle these cases – with code something like this:*

```

char *uri = NULL; // use default endpoint given in WSDL
int n = 10; // max redirect count
int ret;
KeyRegistrationResponse *rsp;
while (n--) {
    ret = ik_submit_key_registration(uri, ...);
    if (ret)
        if (ret == 307)
            // endpoint from HTTP 307 message
            uri = ik_get_key_registration_response_authT_uri(rsp);
        else if (ret == 401)
            // basic authentication required
        else {
            // ... report and handle error
            break;
        }
    }
else
    break;
}
*/

```

```

int ik_submit_key_registration(const char *uri,
                                KeyRegistrationRequest *req,
                                KeyRegistrationResponse **rsp);

```

/ the client is responsible for making any required copies of the returned data */*

```

char *ik_get_key_registration_response_unique_id(
                                const KeyRegistrationResponse *rsp);

```

/ in the event that the server issued a redirect – trying to push the client to a different URL to authenticate, this is how the client can get the new URL */*

```

char *ik_get_key_registration_response_authT_uri(
                                const KeyRegistrationResponse *rsp);

```

/ if the submit response was a 401 (authentication required) this will return the realm string provided by the server (in quotes). */*

```

char *ik_get_key_registration_response_authT_realm(
                                const KeyRegistrationResponse *rsp);

```

/ the content type of the response – normally text/xml (and the soap parser handles it) but possibly something else in the case of*

```

        external authentication.

char *ik_get_key_registration_response_content_type(
        const KeyRegistrationResponse *rsp);
*/

/* free resources allocated into the request and response objects by
   the library. */
void ik_cleanup_key_registration(KeyRegistrationRequest *req,
        KeyRegistrationResponse *rsp);

#endif

```

C.3 Service Location

```

#ifndef _IK_SVC_LOCATE_H
#define _IK_SVC_LOCATE_H

#include "ik_resolver.h"

/* provides a (sorted by priority) list of key query service hosts for
   the specified domain using ik_locate_svc. The caller is
   responsible for freeing the dynamic memory allocated for the
   response. */
int ik_locate_querysvc(const char *domain, ik_srv_info_t **resp,
        unsigned int *rcount);

/* provides a (sorted by priority) list of key registration service
   hosts for the specified domain using ik_locate_svc. The caller
   is responsible for freeing the dynamic memory allocated for the
   response. */
int ik_locate_regsvc(const char *domain, ik_srv_info_t **resp,
        unsigned int *rcount);

#endif

```

C.4 Low-Level DNS Resolver

```

#ifndef _IK_RESOLVER_H
#define _IK_RESOLVER_H

#include <netinet/in.h>
#include <arpa/nameser.h>
#include <lwres/netdb.h>
#include <sys/errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <string.h>
#include <stdlib.h> /* qsort */
#include <openssl/dsa.h>

```

```

#define DEBUG

#ifdef DEBUG
#include <assert.h>
#include <stdio.h>
#define ASSERT(t) assert(t)
#else
#define ASSERT(t)
#endif

#define ENONVALID (5)

#define SRV_PRIORITY (0)
#define SRV_WEIGHT (2)
#define SRV_PORT (4)
#define SRV_TARGET (6)

#define MAX_RESP (NS_PACKETSZ)

typedef struct {
    uint16_t priority;
    uint16_t weight;
    uint16_t port;
    char target[NS_MAXDNAME];
} ik_srv_info_t;

typedef struct {
    unsigned char addr[NS_INADDRSZ+1];
} ik_iaddr_t;

/* uses the lwres library to securely resolve the requested srv name
and provides a (sorted by priority) list of matching hosts. The
caller is responsible for freeing the dynamic memory allocated for
the response.*/
int ik_locate_svc(const char *domain, ik_srv_info_t **resp,
                 unsigned int *rcount);

/* uses the lwres library to securely resolve the requested name into
one or more A records, following CNAMEs if necessary. The caller is
responsible for freeing the dynamic memory returned in resp. */
int ik_resolve_host(const char *hname, ik_iaddr_t **resp,
                   unsigned int *rcount);

/* uses the lwres library to securely resolve the requested name into
a SHA-1 hash and verify the provided key. Expects the key to be
stored in a TXT record with the name sha1-<keyname> */
int ik_verify_dsa_key_by_name(const char *kname, const DSA *key);

#endif

```