USING IDENTITY-BASED ENCRYPTION TO ELIMINATE CERTIFICATES IN SSL TRANSACTIONS

A Thesis
In TCC 402

Presented to

The Faculty of the
School of Engineering and Applied Science
University of Virginia

In Partial Fulfillment

of the Requirements for the Degree

Bachelor of Science in Computer Engineering

by

J. Adam Sowers

26 March 2002

On my honor as a University student, on this assignment I have neither given nor received unauthorized aid as defined by the Honor Guidelines for Papers in TCC Courses.

_____

Approved _____ (Technical Advisor)
David Evans

Approved _____ (TCC Advisor)
Patricia Click

"DEAR SIR -- A favorable and a confidential opportunity offering by Mr. Dupont de Nemours, who is revisiting his native country gives me an opportunity of sending you a cipher to be used between us, which will give you some trouble to understand, but, once understood, is the easiest to use, the most indecipherable, and varied by a new key with the greatest facility of any one I have ever known."

- Thomas Jefferson, in a letter to Robert R. Livingston, Apr. 18, 1802

Table of Contents

Table of Figures

Glossary of terms

Asymmetric key pair - A key pair with a private and public key; used in modern ciphers because of the difficulty in determining the private key from the public key

Ciphertext – A message that has been encrypted so that its contents cannot be determined without decrypting

Cryptography - (from Greek "secret writing") the process of obscuring text or data by changing it using a method that can allow the changes to be undone with a key

Cryptology - The study of cryptography

Key - In a cryptographic algorithm, the secret portion of the algorithm that encrypts and decrypts the message

Plaintext – A message in its original readable form

Private Key - In an asymmetric key pair, the key that is known only to the owner of the key pair

Public Key - In an asymmetric key pair, the key that is known to all

Symmetric key - a single key that is used both to encrypt and decrypt messages

Abstract

This thesis report discusses an alternative implementation of the current Secure Sockets Language (SSL) protocol in use for secure communications on the Internet. Using a different cryptographic protocol than the current SSL standard, this new implementation uses identity-based encryption (IBE) to eliminate the need for server-side certificates. The new system, called IBE-SSL, involves the use of a private key generator (PKG) to create a private key for the server. The server can then use its private key to decrypt any messages sent to it by a client using the server's DNS name as a public key. The system can be implemented into modern browsers, and would provide an alternative security system for web servers and clients.

The report includes a history of the new system along with the underlying mathematical basis which provides the security of the system. The report also includes the IBE system and its method of securely generating keys and the encryption and decryption functions. Then an overview of the IBE-SSL system implementation is given. An efficiency analysis is provided to gauge the feasibility of using the implementation in an industrial setting.

A simple, yet complete, implementation for the IBE-SSL system was completed and the source code is available online (See the conclusion section for the URL.) The system includes a sample private key generator, as well as test client and server.

I. Introduction

Purpose

The purpose of this project is to demonstrate that traditional SSL can be supplanted with identity-based encryption to eliminate the need of site certificates. Code for the proof-of-concept is given, and the report includes security and speed analyses.

Background

Throughout history there has been a race in the field of cryptology: cryptographers strive to find new ciphers that are increasingly hard to break, and cryptanalysts work to break the ciphers. In traditional (symmetric) ciphers, there is a single key that allows the sender to encrypt a message, and the recipient must use this same key to decrypt the message. Most of these ciphers were hard to break in their time, but a large problem remained: the single key [9]. If two parties were unsure about the security of their connection, they might think to encrypt their messages using cryptography. However, they first had to meet and exchange a common key in order to decrypt each other's messages. This was a problem if the pair could not meet in person [6].

Cryptographers Ron Rivest, Adi Shamir and Leonard Adleman developed a revolutionary new method in 1977. Their method uses two keys: a private

key known only to the owner and a public key that can be given to anyone. The system is based on a complicated mathematical formula involving large prime numbers and exponentiation [8]. When someone wishes to send a message to a person, he or she must find that person's public key and encrypt the message. The recipient then decrypts the message with his or her private key and receives the plaintext. When online retailers appeared on the World Wide Web, experts realized that this new form of cryptography (commonly called public-key cryptography) could allow these businesses to conduct secure transactions. This new algorithm, called Secure Sockets Language (SSL), quickly made its way into most web browsers.

In an SSL system, the server generates its own public and private key pair and then publishes the public key to the Internet. Anyone wanting to transfer sensitive data can then use the server's public key to encrypt the data. However, if a hacker compromised the server and replaced the server's public key with the hacker's public key, then the hacker could intercept incoming ciphertext messages and decrypt them [6]. To avoid this, the creators of SSL introduced certificates. One of the unique properties about the public / private key pair is that a person can encrypt a message with his or her *private* key, and then anyone can get the person's public key and decrypt the message. Since the person's public key decrypts the message, the corresponding private key must have encrypted it, so anyone who decrypts the message knows that the person sent it. This process is known as

"signing." In SSL, a trusted party (called a certificate authority or CA) will issue a certificate for a server. Essentially, the certificate is the public key of the server encrypted with the private key of the signer. Web browsers have the public key of the CA embedded in their code and these keys are implicitly trusted on SSL's root-level certificate trust model. Since web browsers (and users) trust that the CA has not given away its private key, they can decrypt the server's public key by decrypting with the CA's public key with the assurance that a hacker has not compromised the system security [9].

In 2001 D. Boneh and M. Franklin developed a similar method of public key cryptography, except that it allowed the sender to pick one of the keys using the Weil pairing. The Weil pairing is another system that is based on a complicated mathematical formula - for the purposes of this explanation let us assume that the mathematical function describes an ellipse. Since any ellipse has two axes of symmetry, any point on the ellipse has an "opposite." If the mathematical function is "hard enough" – meaning it cannot be determined by knowing only its inputs and outputs, then the output of this ellipse function (its opposite) will be an effective key [2]. The advantage to this system is that a person can pick his or her own public key, and rely on a trusted third party called the Private key generator (PKG) to generate the proper private key to decode the message. Since the public key can be chosen, the scheme is called Identity-Based Encryption (IBE).

The Problem

In the most common example, suppose that someone wants to buy a product from an online retailer. With traditional SSL, the web browser connects to the server and downloads the certificate. It checks to see that it has been signed by a trusted party, and extracts the public key of the server. It then encrypts data with the public key and sends it to the server, which decrypts the data and completes the transaction. The main disadvantage to this scheme is that certificates rely on the user to manage them, and most web users are not even aware of SSL technology, much less certificates. Certificates can also be revoked, but most users do not check the status of the certificate before transferring secure data.

If SSL used the IBE system instead, there would only be the need for one certificate: the master certificate embedded in web browsers for the PKG. The browser would then encrypt data with the URL of the website (e.g. "amazon.com") and send the encrypted text to the server. The server would get the encrypted message from the client and decrypt it with its private key that it has received from the PKG. There is no need for the server to have its own certificate: it merely needs the

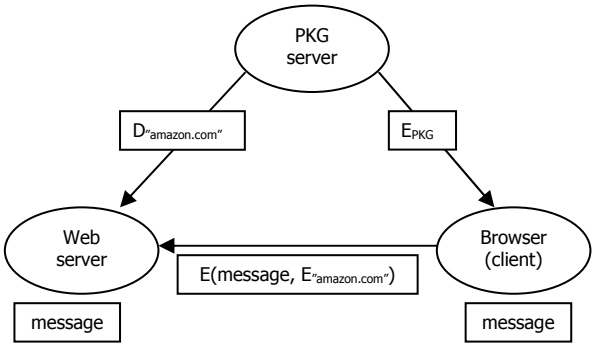

Figure 1. Diagram of IBE SSL interactions.

private key generated from the PKG to decrypt any message sent to it. This process is shown graphically in Figure 1.

Scope

In this project, I have developed a method of using IBE in SSL to eliminate the need for certificates. I have provided proof-of-concept code to set up the PKG, for the server to connect to the PKG and request its private key, and for the client to encrypt with the server's public key and connect to the server. I anticipate that this project will have a positive impact in making Internet transactions more secure. Although this solution does have certain advantages over SSL, I do not expect the outcome of this project to serve as a replacement; SSL has become an industry standard and completely replacing SSL with an IBE SSL algorithm would prove unfeasible in modern e-commerce.

Overview of Thesis Report

In the following chapters I will provide a more in-depth discussion of the mathematics behind the systems and its applications in the algorithms. I will present my code for the proof-of-concept demonstration, data on speed and processor requirements, and a security analysis of the method. Finally, I will conclude the report with a project summary and an interpretation of the data, as well as recommendations for the future of this system.

2. Review of Relevant Literature

It would be unwise to start any project using public-key cryptography without researching the original papers on the subject. The first paper on a simple and secure public-key cryptographic system was "New Directions in Cryptography," published in 1976 by Whitfield Diffie and Martin Hellman. In this paper, the groundwork for public-key cryptography began: the authors discovered the first way for two people to share a key over an insecure connection. No longer would people have to resort to trusted couriers to exchange keys for encrypted messages [3]. This proved to be a monumental step in public-key cryptology, but still a significant problem remained: how could one person verify that the other person was in fact the intended recipient and not an imposter?

This question was answered by R. L. Rivest, A. Shamir and L. M. Adelman in 1978 in a paper entitled "A Method for Obtaining Digital Systems and Public-Key Cryptosystems." The paper expanded on the concept of the Diffie-Hellman algorithm to develop a (nearly) complete system of secure communication over an insecure line using a matched set of keys [9]. The researchers explained the mathematical basis behind their findings and outlined the manner in which these methods could be used to generate encrypted messages between any two people without fear of the message being intercepted. This paper revolutionized the world of cryptology, as it succeeded where others had failed for hundreds of years in the quest of

developing secure communication between perfect strangers. Without this paper, reliable security on the then fledgling Internet would not have been possible, and modern e-commerce and a majority of other Internet technologies would not exist [10]. As a side note, in 1997 the British government declassified documents revealing that British cryptologists developed virtually the same methods of the Diffie-Hellman and RSA algorithms in the early 1970's, but due to governmental security issues, they were not allowed to publish their results [5]. Since the work was classified until 1997, Rivest, Shamir, Adleman, Diffie, and Hellman are still credited with the invention of modern cryptography since they developed it independently and were the first to publish their findings [10].

The RSA algorithm (as it is commonly called) works perfectly well and turns out to be very secure. However, the main drawback to the algorithm at the time of its invention was that it was rather slow. Around the same time, the NSA approved the use of another encryption algorithm called DES (Data Encryption Standard). This algorithm suffered from the age-old problem of key distribution but had a great advantage over RSA in that it was notably faster [7]. Despite the key-distribution problem, DES was implemented in banking systems and governmental security systems, among other uses [1].

In 1991, a programmer named Philip Zimmermann developed a simpler, yet equally effective system using RSA for e-mail that he called PGP (Pretty Good Privacy). Since the RSA algorithm was very slow to compute,

Zimmermann decided to encrypt the text using a random key, which is much faster for a PC to calculate. However, the key itself is encrypted with the RSA algorithm and since the key size is relatively small, PGP takes less time to encrypt than the original RSA encryption. Zimmermann worried that the government would block him from distributing his program for several reasons. The US government had listed cryptographic systems as non-exportable munitions at the time, and the RSA algorithm is patented. Putting the public interest of privacy over potential legal ramifications, he asked a friend to publish the source code on the Internet to guarantee its rapid spread [1]. Zimmermann's program and his actions made him the subject of a three-year investigation from the government. Nevertheless, at this time all legal issues have been cleared up and now PGP is freely available to anyone in the world thanks to more lax cryptographic export restrictions [1].

Due to the work of these cryptographic pioneers, users of the Internet can benefit from high security while sending e-mails and online commerce. However, there are still some drawbacks. For example, suppose that one wants to send an e-mail with sensitive information to someone else over the Internet. Using the PGP system, you can use the other person's public key to encrypt the data, but they might not have a PGP key, and even if they do, how would one find it? Ron Shamir asked for a system in which the sender can choose the public key. This remained unanswered until 2001 when two researchers from Stanford, D. Boneh and M. Franklin, developed a system

which does exactly that. For example, say Alice wants to send a message to Bob, whose e-mail address is "bob@virginia.edu." Alice simply encrypts the message using Bob's e-mail address as the public key and Bob decrypts it with the help of the Private key generator [2].

This system uses many concepts of the previous algorithms but implements a new system using a specialized form of elliptic curve cryptography, in which a private key generator can be used to generate the private key based on the public key by using mathematical equations on special forms of ellipses. The Stanford team has shown that this system can effectively work for e-mailing sensitive data [2].

The previous research in the field of public-key cryptography has proven that strong cryptosystems can provide security over insecure networks in e-mail and e-commerce. My research explores the possibilities of using the new Weil pairing encryption system for SSL transactions with the removal of site certificates. In my investigation, I implemented a system that improves security on the Internet without adding unnecessary complexity to existing systems, and avoids complications due to server authentication.

3. Mathematical Background

RSA Public-Key Cryptography

The SSL system, as well as many public-key systems in use today, relies on the RSA scheme. This relatively simple scheme uses known mathematical challenges to provide the system's security. The algorithm is as follows:

1. Pick two large prime numbers p and q.
2. Multiply these two numbers together to yield $n = pq$.
3. Choose two numbers e and d such that

   $$ed \equiv 1 \bmod (p-1)(q-1) \qquad \textbf{(3.1)}$$

   and d is relatively prime to $(p-1)(q-1)$.
4. To encrypt a message M,

   $$E(M) = M^e \bmod n \qquad \textbf{(3.2)}$$

5. To decrypt the ciphertext message C,

   $$D(C) = C^d \bmod n \qquad \textbf{(3.3)}$$

The encryption and decryption functions are both based on the numbers d and e which are derived from the original prime numbers p and q. (A mathematical proof of the assumptions in (3.2) and (3.3) is given in Appendix A.) In the RSA system, n and e are public while p, q, and d are kept private. The underlying security of the system rests in the fact that since p and q are very large prime numbers, n is difficult to factor into p and q. So far, no

method exists to factor a large number into its prime factors in any reasonable amount of time, so *p* and *q* are hidden to outsiders and *e* cannot be determined. Note also that *e* and *d* are interchangeable, so while (3.2) and (3.3) are true, they both still hold if *d* and *e* are exchanged. This forms the basis for both encryption and message signing in the RSA system.

Elliptic Curve Cryptography (ECC)

Elliptic curve cryptography, which the IBE system is based on, utilizes certain mathematical properties of elliptic curves, that is, curves of the form

$$y^2 + xy = x^3 + ax^2 + b\,. \qquad \textbf{(3.4)}$$

Specifically, IBE uses the curve

$$y^2 = x^3 + 1 \qquad \textbf{(3.5)}$$

in its algorithm. In the traditional sense, the quantities for the variables are real numbers, but in ECC these numbers may be any numbers in a given finite field. In this field, numbers can be added and multiplied to yield other numbers also in the field. In ECC, a person picks a particular elliptic curve and a particular point on the elliptic curve (denoted as F). These parameters can be shared in public between two users. Then each person *i* picks a private key K_i (a random integer) and computes K_iF. This number serves as the public key for person *i*. Say that Alice and Bob have agreed on an elliptic curve and a point on the curve. Alice sends Bob KaF and Bob sends Alice KbF.

Alice and Bob can now agree on a common key by multiplying the public key of the other person by their private key. Alice would compute the key as

$$K = Ka(KbF) = KaKbF \qquad \textbf{(3.6)}$$

and Bob similarly calculates the key as

$$K = Kb(KaF) = KbKaF = KaKbF\ . \qquad \textbf{(3.7)}$$

Note that this approach is similar to the Diffie-Hellman key exchange algorithm, but using elliptic curve fields instead of prime integers. Since it is hard to calculate Ki from KiF (based on the defined addition and multiplication rules of the elliptic curve's field), Alice and Bob can use these keys to communicate securely. Whereas this method involves key exchange and communication over a symmetric cryptosystem, the IBE system exploits a property of the Weil pairing to allow for asymmetric cryptography.

4. The IBE Scheme

The IBE scheme operates on a specialized form of ECC using the Weil pairing. While the majority of this algorithm is beyond the range of this text, essentially the Weil pairing can be used in ECC to create a bilinear map which satisfies certain properties desirable for a public-key cryptosystem. Using this pairing and its resulting bilinear map

$$\hat{e} : \mathbb{G}_1 \times \mathbb{G}_1 \rightarrow \mathbb{G}_2 \qquad \textbf{(4.1)}$$

one can create a system which takes an arbitrary point on the curve and generates the complement of the point based on the bilinear pairing. The IBE system has four major algorithms, as follows:

1. Setup – Generates private- and public-key parameters for the particular IBE private key generator (PKG) server, along with a master key *s.*

2. Extract – Maps an arbitrary string $ID \in \{0,1\}^*$ to a point on the elliptic curve, calculates a private key $d_{ID} = sQ_{ID}$

3. Encrypt – Encrypts a message string *M* with public key mapped from $ID \in \{0,1\}^*$ along with a randomized parameter *r.*

4. Decrypt – Determine if ciphertext message *C* is on elliptic curve; if so, decrypt with extracted private key and remove *r* randomization.

The IBE system implements these algorithms to provide a complete solution to the selected public key problem posed by Shamir. In the paper

"Identity-Based Encryption from the Weil Pairing," the Stanford team argues convincingly that the IBE system is secure from the standpoint of chosen ciphertext security in the random oracle model, that is, that given any private keys that are not the same private key as the server and messages encoded with these keys, an attacker still cannot learn anything useful about the server's private key.

5. IBE-SSL Implementation

In this section, I will outline the steps taken to show that the IBE system can be applied to a traditional SSL implementation. While SSL is traditionally performed implicitly in modern browsers, I have chosen to write demonstration programs to show a proof of concept. This demonstration could be used at a later date in an actual open-source browser, such as Mozilla, to test its usefulness in real-world applications. The user must execute three main tasks for the system to work properly. These tasks are outlined below.

Setting up the master PKG

The IBE system provides a program named *gen* which generates the master system parameters for the PKG (i.e. the setup algorithm). In a real SSL system, *gen* would be run on the master PKG (e.g. Verisign). The program reads its system parameters from a configuration file (gen.cnf) and generates a public and private set of parameters, much like Certificate Authorities in SSL generate a master public and private key-pair. I have made no changes in this program other than setting up the system parameters in the configuration file to reflect the IBE-SSL system. Once the user has run *gen*, he or she can distribute the public parameters freely for embedding into browsers as long as the private parameters are held securely on the PKG computer.

Getting a private key

The IBE system also provides a tool called *pkghtml* which acts as an SSL server and issues private keys to servers. The tool originally served as an email-based system, but I modified it to write the private keys to files stored on the PKG server. The original program also allowed anyone to have a private key generated for this specific IBE system, but I again modified it to only allow authenticated servers by use of an access file on the PKG server.

To set up the *pkghtml* program, the PKG needs public and private keys generated for its SSL encryption. These keys can be generated from the OpenSSL package available online. The public key file certificate (ca.cert) and the private key (ca.priv) need to be placed in the same directory as the program. The administrator for the *pkghtml* server also needs to create a pkaccess file which includes the ID, password, and location to write the private key file. As with *gen*, the *pkghtml* program has a configuration file (pkg.cnf) which defines the basic parameters of the program. The administrator runs *pkghtml* and server administrators can connect and generate their private keys. Figure 2 shows the output of the *pkghtml* server running on a current web browser over an SSL connection.

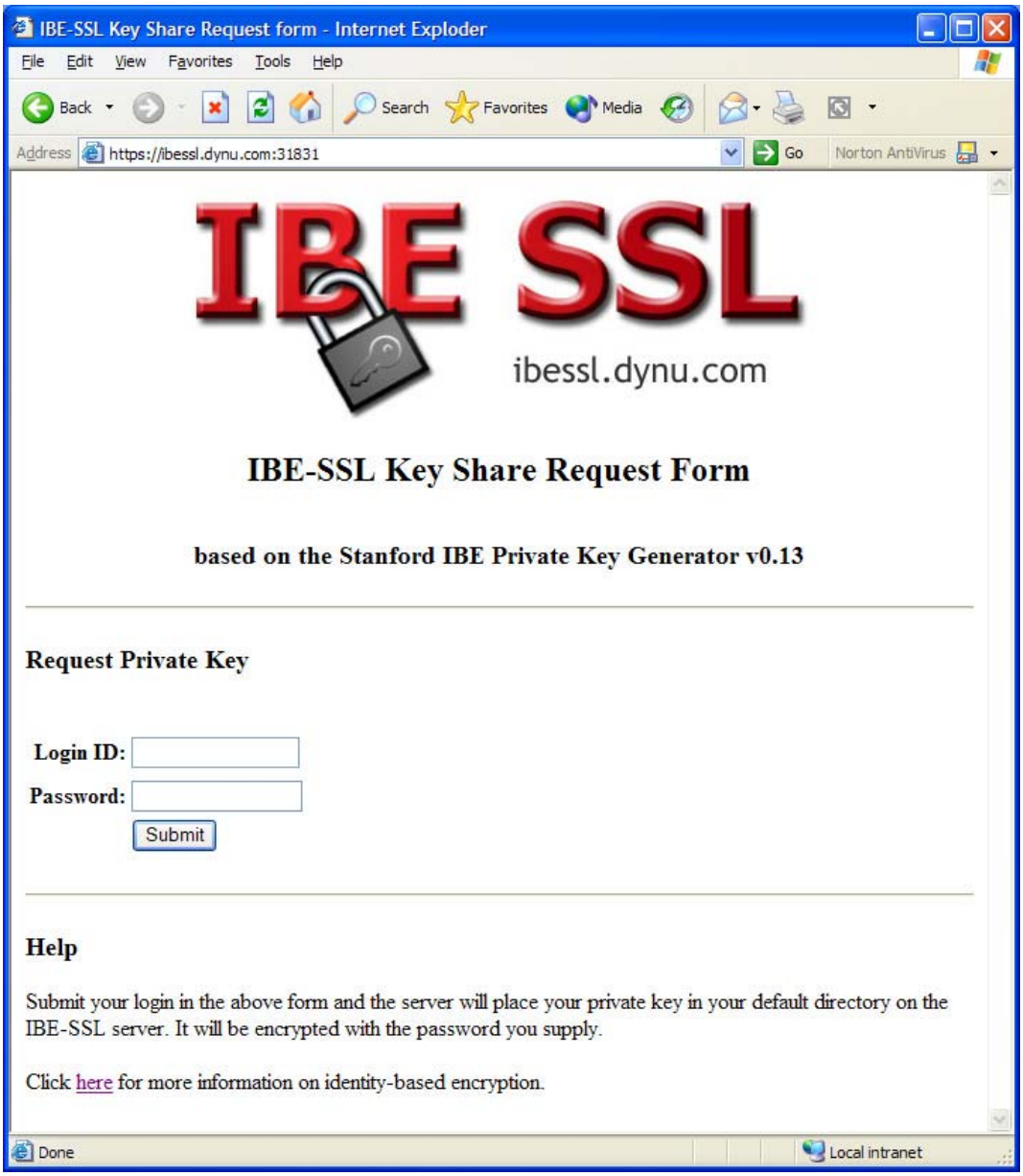

Figure 2. The initial SSL page generated from *pkghtml*

Once the server administrator has entered a valid id (usually the DNS name of the server which will be receiving encrypted IBE-SSL messages) and password for that ID, the *pkghtml* server will generate a new page (as shown in Figure 3) confirming the key creation.

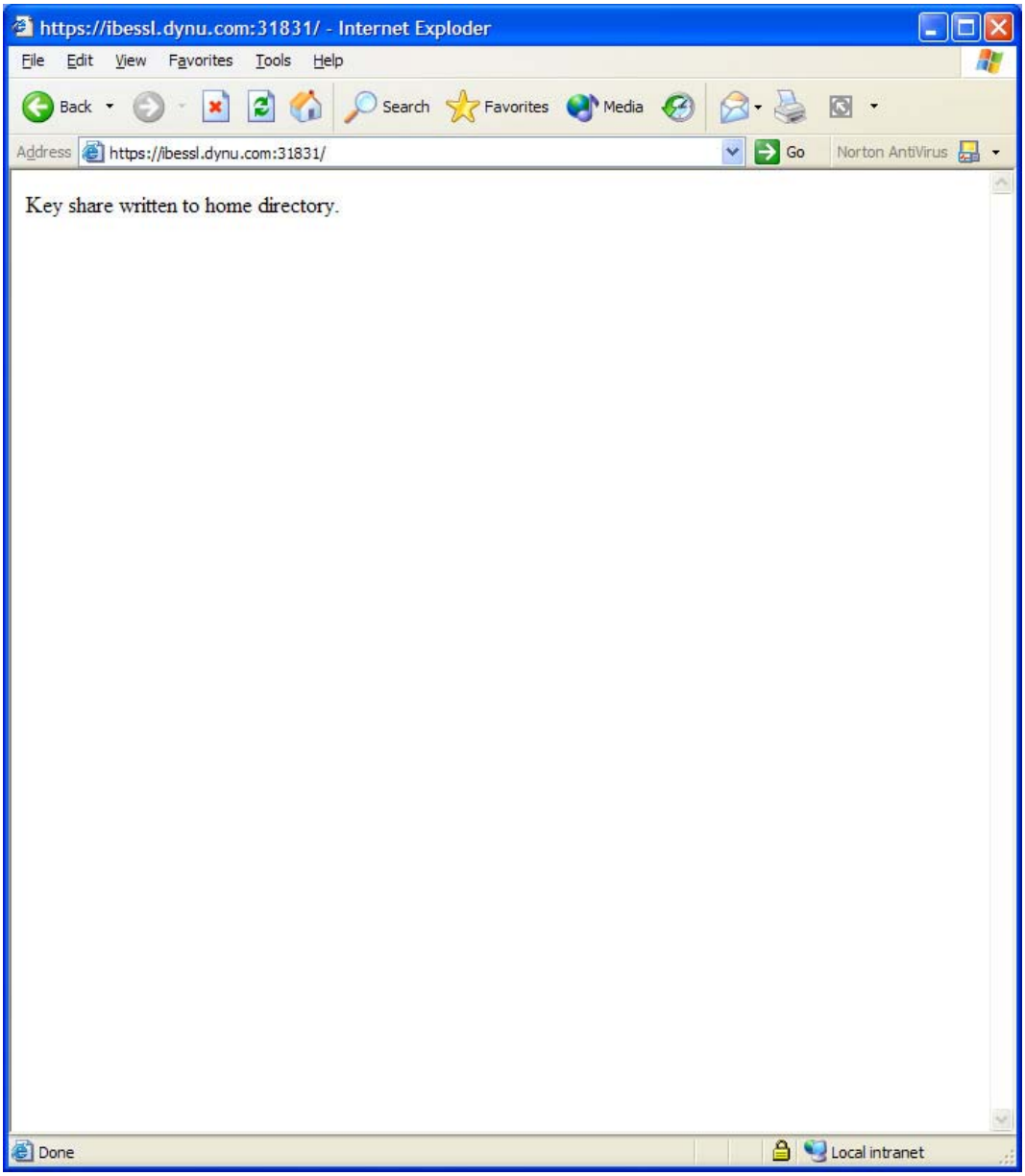

Figure 3. The second generated page showing the key share written to the specified directory.

At this point, the server administrator can log into the PKG server in an agreed-upon way (i.e. ssh, secure ftp, etc) and download the *params* file. The administrator then needs to run *ibe combine <ID> params* to remove

the password from the *params* file for use in the final step of the IBE-SSL implementation.

Communicating over insecure networks

Once the server has received and decrypted the private key, it can use the key to decrypt any message encrypted with its ID. To demonstrate the IBE's encrypt and decrypt functions in real time, I have implemented a simple client / server system using standard sockets. As a test, the user can run these two programs on the same machine, or on two machines connected over the Internet. Of course, in a real SSL scheme, the client and server would certainly be different computers, but the client and server programs I have developed are merely meant to show that the system can quickly encrypt and decrypt messages sent over an insecure network. The server begins by reading parameters from a configuration file (server.cnf) – the port to listen on, the file containing the private key generated from the PKG, and the password to the file. Again, the password provided in the configuration file is meant to be a convenience to the tester and in a real system should not be placed in any file that might be intercepted by an intruder. The server then listens for connections from the client on the specified port. The user runs the client as *client <server DNS name> <ID> <file>.* The client program connects to the server specified on the command line and reads in the specified file. It encrypts the file with the given ID and sends the

encrypted message to the server. The server then receives the encrypted file and, using the private key for the ID, decrypts the file and sends the unencrypted message back to the client to demonstrate that it has decrypted the message successfully. This action would be completely unnecessary in a real SSL implementation, but in these test programs, the user can verify that the implementation can successfully encrypt messages, send them securely over an insecure network, and decrypt the ciphertext at the server side. On the client's side, the unencrypted file will be stored with the same name as the original file, except with ".out" appended to the end. The user can verify that the contents of the files are the same by visual inspection or by using the Unix command *diff.*

Efficiency Analysis

To test the efficiency of the system, I encrypted and decrypted sample files using the IBE system and measured the benchmark times for the encryption and decryption stages. All tests were performed on an Intel Pentium II CPU running at 450 MHz with 128MB of PC100 SDRAM and Redhat Linux 7.2 using a Pentium II-optimized kernel. I ran the client and server programs on the same machine to eliminate the possibility of lag between the two programs which could affect results in the initialization phase. The encryption phase benchmarks provided more results than the decryption phase benchmarks, but in both cases the total time was provided. I chose files of three sizes to send: 'short.txt' (a 35byte file), 'virginia.txt' (a 1.5 kilobyte file), and 'charter.txt' (a 6.6 kilobyte file.) I would have liked to test larger files, but the ciphertext for any file in the IBE system is longer than the plaintext, and IBE currently only supports ciphertexts up to 10 kilobytes. The tests were run 10 times, and the raw results are available in Appendix B. Figure 4 highlights the CPU times of encrypting the texts. The file size did not seem to affect the processing of the file, and indeed the decryption time did go down with the larger file. Realistically, this time should increase with much larger files, but in an actual IBE-SSL system, the user would not be transmitting much more than credit card information and mailing addresses, so the total amount of information sent would be relatively low. The only potential problem in this scheme is that the computer took about .15 seconds

to decrypt a file. This corresponds to about 7 decryptions possible per second, and a problem could arise on a busy e-commerce server which surely has more than 7 secure connections per second during peak times. However, these tests were performed on a single Pentium II without extensive speed optimizations, so it is possible that servers running at e-commerce sites would have much more processing power than my test server and could certainly handle more secure connections per second.

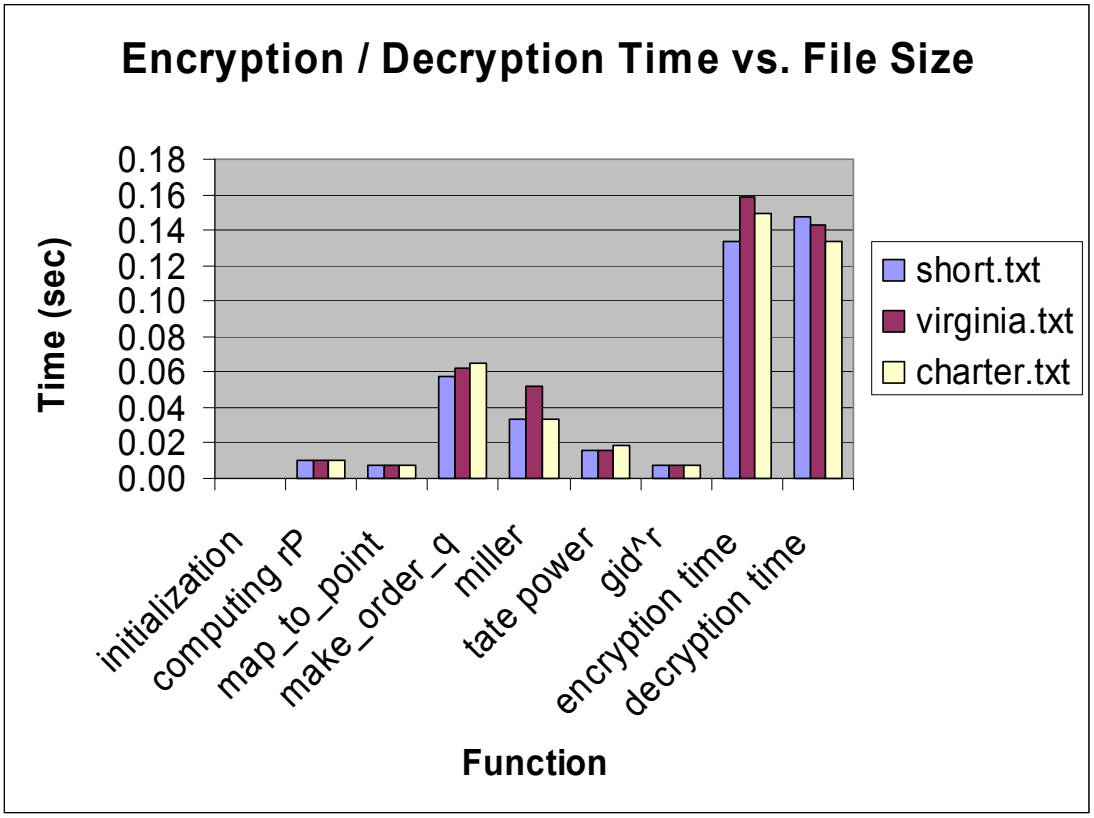

Figure 4. Encryption and Decryption times for different file sizes.

Conclusion

In this report, I have shown that a complete scheme for secure network transactions can be accomplished without the need for certificates. I have shown that an alternative cryptosystem to the current RSA system has comparable security and similar ease of use. I will present a summary of the findings, an interpretation of the data, and recommendations for the future of the system.

Summary

In the proceeding text, I have shown that for a cryptosystem to effectively operate as a secure network public-key cryptographic protocol, it must have certain properties. Of course, the underlying cryptosystem must be secure. I have shown the IBE team's proof that the system has all of the desired properties for a secure public-key cryptosystem. I have also shown how the IBE system differs from RSA in terms of its mathematical properties. The system must also exhibit some features for it to work similarly to SSL and I have outlined these features also.

The protocol must first have the ability to create public and private keys that are mathematically hard to break. The protocol must be able to quickly generate private keys for servers and have a secure method of distributing the private key to the server, and finally be able to decrypt messages sent to

it based on its ID. I have shown that these features are easily implemented, as well as sample output from a private key generator and a simple client / server system illustrating this system in action. Lastly, I have shown that these steps have been implemented in the system and that they work together to provide a complete solution to the problem without the use of certificates as in traditional SSL.

Interpretation

As the preceding results show, this system is feasible to implement as a secure method of sending data over insecure networks and comparable to the current SSL standard. However, as mentioned before, any system based on RSA encryption relies on the difficulty of factoring large numbers into their prime factors. Mathematicians have struggled without success to find a method to quickly factor large numbers, but with the discovery of a significantly more efficient number factoring method, the RSA algorithm would no longer provide a secure method of exchanging messages and the SSL system would fall apart. Mathematicians agree that in this event, security systems would require the use of elliptic curve cryptography, which provides the mathematical basis of IBE and has not yet shown any weakness to cryptographic attack. Therefore, the IBE-SSL system fulfills a definite need in current security measures.

In addition, I have shown that the IBE system works in a relatively efficient manner to generate private keys for servers from the PKG and to encrypt and decrypt data. While a typical e-commerce server would need to handle many requests at once, this server performed rather well on a modest Intel Pentium II system running at 450 MHz and Redhat Linux 7.2. I would imagine that industrial e-commerce servers would handle the IBE encryption much faster than my test system and that the encryption / decryption process might get a speed boost from extended code optimization. I believe that this system is implementable in an industrial setting, and should be included in future web browsers as an alternative to traditional SSL. Source code is available online at http://www.people.virginia.edu/~jas8qs/ibessl/ibessl.tar (as of 26 March 2002).

Recommendations

In this paper, I have presented only a simple implementation of a complete working IBE-SSL system. There are numerous modifications that could be made to this project in future designs. For example, an interesting feature of IBE is that the PKG could generate the ID as "ID | <date>". This would cause keys to automatically expire after the specified date, and the server could get a new key each day to further protect against attacks. I intend to release this software to the open-source community to expand upon it and port it from Linux to other systems. My hope is that other programmers

will obtain this system and refine it for uses I have not foreseen, and implement it as a standard along with traditional SSL. I believe that this project will have a definite use in the Internet security community as an alternative to SSL and as a potential replacement in the event that mathematicians find a way to expedite the factoring algorithm that protects RSA-based cryptosystems. I believe that this project has provided a worthwhile contribution to the security community and the results found here can aid future cryptographic protocols to increase the overall security of public networks.

Bibliography (works cited)

(Note: if a source has both a printed entry and online entry, the online entry has been provided as a secondary source for the reader's convenience.)

1. Back, Adam. "PGP Timeline." Online (Internet). 17 October 2001. Available: http://www.cypherspace.org/~adam/timeline/

2. Boneh, D. and Franklin, M. "Identity-Based Encryption from the Weil Pairing." Proceedings of Crypto '2001, Lecture Notes in Computer Science, Vol. 2139, Springer-Verlag, pp. 213-229, 2001. Available: http://crypto.stanford.edu/~dabo/papers/ibe.pdf

3. Diffie, W. and Hellman, M. "New Directions in Cryptography." IEEE Transactions on Information Theory, 1976.

4. Evans, David. "Lecture 8: RSA." Online (Internet). 23 March 2002. Available: http://www.cs.virginia.edu/~evans/cs588/lectures/lecture8.pdf

5. Ellis, J. H. "The Possibility of Non-Secret Encryption." CESG classified paper, 1970 (declassified 1997). Online (Internet). 17 October 2001. Available: http://www.cesg.gov.uk/publications/media/nsecret/possnse.pdf

6. Garrett, Paul. Making, Breaking Codes. New Jersey: Prentice Hall, 2001.

7. Network Associates, Inc. An Introduction to Cryptography (Chapter 1 and 2). PGP 6.5.1 documentation. Online (Internet). 17 October 2001. Available: ftp://ftp.pgpi.org/pub/pgp/6.5/docs/english/IntroToCrypto.pdf

8. Rivest, R, Shamir, A, and Adleman, L. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems." 1978.

9. Singh, Simon. The Code Book. New York: Anchor Books, 1999.

10. Wagner, D. and Schneier, B. "Analysis of the SSL 3.0 Protocol". *The Second USENIX Workshop on Electronic Commerce Proceedings*, USENIX Press, November 1996, pp. 29-40. Online (Internet). 17 October 2001. Available: http://www.counterpane.com/ssl.pdf

Appendix A: Mathematical proof of RSA system

(taken from "Lecture 8: RSA" from David Evans, a synopsis of "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems" by Rivest, Shamir, and Adleman)

For the RSA algorithm to work as a secure public key cryptosystem, the following property must be met:

$$D(E(M)) = M \quad \textbf{(A.1)}$$

2. *d* and *e* are easy to compute

This property is true because:

$$E(M) = M^e \bmod n \quad \textbf{(A.2)}$$

$$D(E(M)) = (M^e \bmod n)^d \bmod n \quad \textbf{(A.3)}$$

$$= M^{ed} \bmod n \quad \textbf{(A.4)}$$

and e, d, and n can be chosen such that

$$M \cong M^{ed} \bmod n \quad \textbf{(A.5)}$$

$$1 \cong M^{ed-1} \bmod n \quad \textbf{(A.6)}$$

based on Euler's totient (φ) function which states that for *a* and *n* relatively prime,

$$a^{\varphi(x)} \equiv 1 \bmod n \quad \textbf{(A.7)}$$

if a and n are relatively prime where

$\varphi(x) =$ the number of numbers < *n* not relatively prime to *n.*

Since *n* is the product of two primes, we can pick a number *d* relatively prime to *n* and its multiplicative inverse *e* such that

$$de \equiv 1 \bmod \varphi(n) \qquad \textbf{(A.8)}$$

and d and e satisfy the equation

$$ed - 1 = \varphi(n) = (p-1)(q-1). \qquad \textbf{(A.9)}$$

Then

$$1 \equiv M^{ed-1} \bmod n \qquad \textbf{(A.10)}$$

and correspondingly

$$M = M^{ed} \bmod n \qquad \textbf{(A.11)}$$

QED.

Appendix B: Raw data

encrypt (short.txt)

trial	initialization	computing rP	map_to_point	make_order_q	miller	tate power	gid^r	encryption time
1	0	0.010297	0.007058	0.057512	0.033426	0.015579	0.007133	0.133565
2	0	0.010142	0.007039	0.057685	0.033429	0.015572	0.00733	0.133717
3	0	0.009598	0.007057	0.056898	0.033482	0.015643	0.007416	0.13262
4	0	0.010298	0.007059	0.057494	0.03338	0.01557	0.007286	0.133655
5	0	0.01105	0.007055	0.057627	0.033437	0.015571	0.007428	0.134694
6	0	0.009914	0.007038	0.057676	0.033416	0.015586	0.007197	0.133383
7	0	0.009905	0.007058	0.057638	0.033404	0.015599	0.007164	0.133287
8	0	0.010293	0.007068	0.057647	0.033394	0.015572	0.007237	0.133733
9	0	0.009726	0.007039	0.057414	0.033427	0.015579	0.007266	0.132971
10	0	0.009537	0.007058	0.057618	0.033397	0.015594	0.007161	0.132886
average	**0**	**0.010076**	**0.0070529**	**0.0575209**	**0.0334192**	**0.0155865**	**0.0072618**	**0.1334511**

encrypt (test.txt)

trial	initialization	computing rP	map_to_point	make_order_q	miller	tate power	gid^r	encryption time
1	0	0.009217	0.007061	0.057576	0.086297	0.015613	0.007302	0.186867
2	0	0.009769	0.007045	0.057653	0.078618	0.015519	0.007262	0.18075
3	0	0.010461	0.007412	0.057825	0.033553	0.015476	0.007225	0.135711
4	0	0.009727	0.007044	0.057684	0.079077	0.015538	0.00716	0.18004
5	0	0.010885	0.007059	0.057622	0.033237	0.015461	0.00719	0.135143
6	0	0.008963	0.007065	0.057731	0.033513	0.015496	0.007204	0.133695
7	0	0.012025	0.00707	0.102341	0.033622	0.015489	0.007274	0.181644
8	0	0.009527	0.007062	0.057772	0.033505	0.016303	0.007127	0.135138
9	0	0.010086	0.007041	0.057902	0.033528	0.015479	0.007154	0.134887
10	0	0.009928	0.007043	0.057762	0.078537	0.015509	0.007191	0.179767
average	**0**	**0.0100588**	**0.0070902**	**0.0621868**	**0.0523487**	**0.0155883**	**0.0072089**	**0.1583642**

encrypt(charter.txt)

trial	initialization	computing rP	map_to_point	make_order_q	miller	tate power	gid^r	encryption time
1	0	0.009627	0.007045	0.057468	0.033366	0.041242	0.007252	0.16392
2	0	0.009318	0.007183	0.057133	0.033372	0.015681	0.007197	0.137669
3	0	0.011103	0.007099	0.057466	0.033401	0.015585	0.007401	0.139795
4	0	0.009501	0.007175	0.083838	0.033439	0.015581	0.007261	0.164581
5	0	0.009583	0.007043	0.057483	0.033386	0.015575	0.007372	0.13818
6	0	0.009487	0.007182	0.057403	0.033222	0.015556	0.007157	0.137789
7	0	0.010962	0.007064	0.083149	0.033432	0.015587	0.007309	0.165372
8	0	0.010902	0.007061	0.058185	0.033359	0.015572	0.007204	0.140057
9	0	0.009855	0.007043	0.083138	0.033395	0.015585	0.007143	0.164066
10	0	0.010603	0.007048	0.057456	0.033224	0.01558	0.007384	0.13931
average	**0**	**0.0100941**	**0.0070943**	**0.0652719**	**0.0333596**	**0.0181544**	**0.007268**	**0.1490739**

decrypt (short.txt)

trial	decryption time
1	0.153968
2	0.153687
3	0.119674
4	0.146095
5	0.124175
6	0.169278
7	0.14298
8	0.162181
9	0.153936
10	0.152241
average	**0.1478215**

decrypt (virginia.txt)

trial	dec time
1	0.13077
2	0.12518
3	0.154441
4	0.154871
5	0.148521
6	0.140944
7	0.149942
8	0.122283
9	0.152558
10	0.153792
average	**0.1433302**

decrypt (charter.txt)

trial	dec time
1	0.132094
2	0.173337
3	0.125117
4	0.129594
5	0.128571
6	0.138826
7	0.121921
8	0.129095
9	0.130597
10	0.129856
average	**0.1339008**

Appendix C: selected code and its output

client.c – simple IBE-SSL client program

Program code

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include "ibe.h"
#include "format.h"
#include "ibe_progs.h"

CONF_CTX *cnfctx;

int portnum;

#define MAXDATASIZE 1000 // max number of bytes we can get at once

int main(int argc, char *argv[])
{
    char defaultcnffile[] = "client.cnf";
    char tempfile[] = "temp.cli";
    char *cnffile = defaultcnffile;
    char *paramsfile;
    char *receivefile;
    int status;
     int sockfd, numbytes;
     char buf[MAXDATASIZE];
     struct hostent *he;
     struct sockaddr_in their_addr; // connector's address information

    byte_string_t M;
    char *id;
    unsigned char *buftemp;
    int pbufsize = 100;
    char *idarray[2];
    unsigned char *ptext;
    int ptextlen;
    char filebuf[MAXDATASIZE];
    int count;
    FILE *fp;

    printf("\nIBE-SSL test client v1.0\nby J. Adam Sowers
            (jasowers@virginia.edu)\n");
    printf("based on Stanford IBE 0.21\n\n");

     if (argc != 3) // need the server's DNS name and file to encrypt
    {
        fprintf(stderr,"usage: %s hostname filename\n", argv[0]);
        fprintf(stderr, "The client will automatically encrypt
            with the hostname provided.\n\n");
         exit(1);
     }

    printf("Loading config file...\n");
```

```
 cnfctx = LoadConfig(cnffile);

 if (!cnfctx)
{
    fprintf(stderr, "error opening %s\n", cnffile);
    exit(1);
 }

portnum = GetIntParam(cnfctx, "port", 0, 31832);
 paramsfile = GetPathParam(cnfctx, "params", 0, "params.txt");

 IBE_init();
 status = FMT_load_params(paramsfile);
 if (status != 1)
 {
    fprintf(stderr, "error loading params file %s\n", paramsfile);
    exit(1);
 }

// get the host info
 if ((he=gethostbyname(argv[1])) == NULL)
{
     perror("gethostbyname");
     exit(1);
 }

// set up the socket
 if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
     perror("socket");
     exit(1);
 }

// standard TCP/IP socket parameters
 their_addr.sin_family = AF_INET;    // host byte order
 their_addr.sin_port = htons(portnum);  // short, network byte order
 their_addr.sin_addr = *((struct in_addr *)he->h_addr);
 memset(&(their_addr.sin_zero), '\0', 8);  // zero the rest of the struct

printf("Connecting to %s...\n", argv[1]);

// connecto to socket
 if (connect(sockfd,(struct sockaddr *)
        & their_addr,sizeof(struct sockaddr)) == -1)
{
     perror("connect");
     exit(1);
 }

// receive server header string
 if ((numbytes=recv(sockfd, buf, MAXDATASIZE-1, 0)) == -1)
{
    perror("recv");
    exit(1);
}

buf[numbytes] = '\0';
printf("Received: %s",buf);

// make IBE id from argument specified
id = IBE_make_id(argv[1], NULL);

printf("opening %s...\n",  argv[2]);
fp = fopen(argv[2], "r");
if(!fp)
{
    printf("error opening file %s.\n", argv[2]);
    close(sockfd);
    return 1;
}

printf("encrypting file with public key %s...\n", argv[1]);
```

```
ptext = (unsigned char *) malloc(pbufsize);
ptextlen = 0;

// read in contents of file, put in buffer to encrypt
for(;;)
{
    if (feof(fp)) break;
    fgets(filebuf, 256, fp);
    count = 0;
    while((filebuf[count] != '\0') && !feof(fp))
    {
        ptext[ptextlen] = filebuf[count];
        count++;
        ptextlen++;
        if (ptextlen >= pbufsize)
        {
                pbufsize *= 2;
            buftemp = (unsigned char *) malloc(pbufsize);
            memcpy(buftemp, ptext, ptextlen);
            free(ptext);
            ptext = buftemp;
        }
    }
}

// close original file
fclose(fp);
printf("Encryption successful. Sending encrypted message...\n");

idarray[0] = id;
idarray[1] = NULL;
M->data = ptext;
M->len = ptextlen;

// open encrypted file for writing
fp = fopen(tempfile, "w");

fprintf(fp, "\n-----BEGIN IBE-----\n");
FMT_encrypt(fp, M, idarray);
fprintf(fp, "-----END IBE-----\n");

byte_string_clear(M);

fflush(fp);
rewind(fp);
fclose(fp);

// reopen the encrypted file to read in and send to server
fp = fopen(tempfile, "r");
fgets(filebuf, 256, fp);
while(!feof(fp))
{
    send(sockfd, filebuf, 256, 0);
    fgets(filebuf, 256, fp);
}

fclose(fp);
remove(tempfile);

printf("Done sending message.\nReceving unencrypted message...\n");

numbytes = 1;

receivefile = argv[2];
strcat(receivefile, ".out");

fp = fopen(receivefile, "w");

// receive server's response (unencrypted message; should be the same as
//the original file.
while(numbytes)
```

```
        {
            if ((numbytes = recv(sockfd, buf, MAXDATASIZE - 1, 0)) == -1)
            {
                perror("recv");
                exit(1);
            }

            buf[numbytes] = '\0';

            if(strcmp(buf, "!!!disconnect!!!") == 0)
            {
                numbytes = 0;
                printf("\nDone receving message. Server closing connection...");
            }
            else
                fprintf(fp, "%s", buf);
        }

        fflush(fp);
        fclose(fp);
        printf(" done. \nProgram exiting...\n");

        // all done, close the socket and exit
        close(sockfd);
         return 0;
    }
```

Program output

```
[root@ibessl clientserver]# ./client ibessl.dynu.com virginia.txt

IBE-SSL test client v1.0
by J. Adam Sowers (jasowers@virginia.edu)
based on Stanford IBE 0.21

Loading config file...
Connecting to ibessl.dynu.com...
Received: IBE-SSL server v1.0
opening virginia.txt...
encrypting file with public key ibessl.dynu.com...
Encryption successful. Sending encrypted message...
benchmarks:
0.000000 initialization
0.035897 computing rP
0.007169 first part of map_to_point
0.073525 make_order_q
0.033572 miller
0.015626 Tate power
0.007283 gid^r
elapsed time: 0.177013
Done sending message.
Receving unencrypted message...

Done receving message. Server closing connection... done.
Program exiting...
```

server.c – Simple IBE-SSL server program

Program code

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/wait.h>
#include <signal.h>
#include "ibe.h"
#include "format.h"
#include "ibe_progs.h"

#define MYPORT 3490        // the port users will be connecting to
#define MAXDATASIZE 1000   // the maximum num bytes to receive
#define BACKLOG 10         // how many pending connections queue will hold

CONF_CTX *cnfctx;

/* this function kills off zombies that occur from fork()'ed processes */
void sigchld_handler(int s)
{
    while(wait(NULL) > 0);
}


int main(int argc, char **argv)
{
    int sockfd, new_fd;  // listen on sock_fd, new connection on new_fd
    struct sockaddr_in my_addr;    // my address information
    struct sockaddr_in their_addr; // connector's address information
    socklen_t sin_size;
    struct sigaction sa;
    int yes = 1;
   int textlen;
   FILE *fp;
   int portnum;
   int numbytes;

   byte_string_t key;
   byte_string_t M;
   char *pw;
   char *privkeyfile;
   char *paramsfile;
   int status;
   char cnffile[] = "server.cnf";
   char tmpfile[] = "temp.svr";
    char buf[MAXDATASIZE];

   IBE_init();

   printf("\nIBE-SSL server v1.0\nby J. Adam Sowers (jasowers@virginia.edu)\n");
   printf("based on Stanford IBE 0.21\n\n");

   // load configuration file
   cnfctx = LoadConfig(cnffile);
   if (!cnfctx) {
          fprintf(stderr, "error opening %s\n", cnffile);
          fprintf(stderr, "using default values\n");
          cnfctx = constructCTX();
   }
```

```
paramsfile = GetPathParam(cnfctx, "params", 0, "params.txt");
status = FMT_load_params(paramsfile);
if (status != 1) {
        fprintf(stderr, "error loading params file %s\n", paramsfile);
        return(1);
}

// get parameters from configuration file
portnum = GetIntParam(cnfctx, "port", 0, 31832);
pw = GetStringParam(cnfctx, "password", 0, "");
privkeyfile = GetPathParam(cnfctx, "path", 0, "keyfile");

// set up socket
 if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
     perror("socket");
     exit(1);
 }
 if (setsockopt(sockfd,SOL_SOCKET,SO_REUSEADDR,&yes,sizeof(int)) == -1)
{
     perror("setsockopt");
     exit(1);
 }

// standard TCP/IP socket parameters
 my_addr.sin_family = AF_INET;         // host byte order
 my_addr.sin_port = htons(portnum);     // short, network byte order
 my_addr.sin_addr.s_addr = INADDR_ANY; // automatically fill with my IP
 memset(&(my_addr.sin_zero), '\0', 8); // zero the rest of the struct

// bind socket
 if (bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr)) == -1)
{
     perror("bind");
     exit(1);
 }

// listen on socket
 if (listen(sockfd, BACKLOG) == -1)
{
     perror("listen");
     exit(1);
 }

 sa.sa_handler = sigchld_handler; // reap all dead processes
 sigemptyset(&sa.sa_mask);
 sa.sa_flags = SA_RESTART;
 if (sigaction(SIGCHLD, &sa, NULL) == -1)
{
     perror("sigaction");
     exit(1);
 }

// main accept() loop
 while(1)
 {
     sin_size = sizeof(struct sockaddr_in);
     if ((new_fd = accept(sockfd, (struct sockaddr *)&their_addr,
                                                    &sin_size)) == -1)
    {
         perror("accept");
         continue;
     }

     // received a new connection
    printf("server: got connection from %s\n",
        inet_ntoa(their_addr.sin_addr));

    // test for child process
     if (!fork())
     {
```

```
 close(sockfd); // child doesn't need the listener

 if (send(new_fd, "IBE-SSL server v1.0\r\n", 21, 0) == -1)
     perror("send");

// open a temporary file for storing the encrypted text
fp = fopen(tmpfile, "w");

printf("Receving encrypted message...\n");

// store incoming message to the temp file
numbytes = 1;
while(numbytes)
{
        if ((numbytes = recv(new_fd, buf, MAXDATASIZE - 1, 0)) == -1)
        {
            perror("recv");
            exit(1);
        }

    buf[numbytes] = '\0';
        fprintf(fp, "%s", buf);

        textlen = 0;
        if(strcmp(buf, "-----END IBE-----\n") == 0) numbytes = 0;
}

fflush(fp);
fclose(fp);

printf("Finished receiving message.\n");

// reopen file to load text for decryption
fp = fopen(tmpfile, "r");

// load the private key file
printf("Loading private key file...\n");
status = FMT_crypt_load(privkeyfile, key, pw);
if (status != 1)
{
        fprintf(stderr, "error loading private key %s\n", privkeyfile);
        return 1;
}

// decrypt using IBE functions
printf("Decrypting message...\n");
status = FMT_decrypt(M, fp, key);
if (status != 1)
{
        fprintf(stderr, "error in decryption\n");
        return 1;
}

printf("Message decrypted.\n");

fclose(fp);
remove(tmpfile);

// open the file again, this time to write out the decrypted message
fp = fopen(tmpfile, "w");
byte_string_fprintf(fp, M, "%c");
fclose(fp);

// open once more, to send the message back to the client
fp = fopen(tmpfile, "r");
printf("Sending unencrypted message...\n");
fgets(buf, 256, fp);
while(!feof(fp))
{
        send(new_fd, buf, 256, 0);
        fgets(buf, 256, fp);
```

```
            }

            // send disconnection notice to client
            send(new_fd, "!!!disconnect!!!", 16, 0);

            // remove temporary file
            remove(tmpfile);

            // close up connection
            printf("Finished sending unencrypted message. Closing
                   connection...\n");
            close(new_fd);
             exit(0);
        }

        close(new_fd);  // parent doesn't need this
    }

    return 0;
}
```

Program output

```
[root@ibessl clientserver]# ./server

IBE-SSL server v1.0
by J. Adam Sowers (jasowers@virginia.edu)
based on Stanford IBE 0.21

server: got connection from 127.0.0.1
Receving encrypted message...
Finished receiving message.
Loading private key file...
Decrypting message...
dec time: 0.269544
Message decrypted.
Sending unencrypted message...
Finished sending unencrypted message. Closing connection...
```