

# Implementing Cryptographic Algorithms to Achieve Secure LAN Communication

**Seneca College  
Informatics and Security  
Bachelor Thesis – Research Paper**

**Jonathan Chiappetta  
April 16, 2012**

## **Abstract**

This work attempts to cover all areas regarding any peers wishing to achieve quick and secure communication over a Local Area Network (LAN). This implies that a series of cryptographic algorithms need to be used in conjunction with each other in order to provide a peer this capability.

The methods described in this paper will try to implement a standard and minimal set of tools that one can easily make use of on most Operating Systems (OS). For example, this paper includes the use of socket objects, which are typically included in most modern programming languages. For this particular application, the language chosen will be C due to its efficiency and power.

The algorithms chosen for this paper will help to provide a peer with a basic set of security on a LAN. This means that it should be possible for them to achieve basic properties such as Confidentiality, Integrity, and Authentication with regards to the data transmitted and received. These properties will help protect a peer from various attacks from a possibly evil party.

The sections covered will be laid out in an order that tries to produce the best logical flow of ideas and concepts. For example, the goals of the paper will be stated at the beginning and in addition, the tools needed to achieve those goals will be listed and described afterwards. The theory behind each algorithm will be demonstrated along with the steps necessary to implement them. Lastly, the final application and results will be published and discussed in the form of implementations and demonstrations.

# Table of Contents

---

<b>1</b>	<b>Introduction</b> .....	<b>- 1 -</b>
1.1	Motivation.....	- 1 -
1.2	Outline.....	- 2 -
1.3	Goals.....	- 3 -
1.4	Approach .....	- 3 -
1.5	Tools Used.....	- 4 -
1.5.1	Software requirements .....	- 4 -
1.5.2	Hardware requirements.....	- 4 -
<b>2</b>	<b>Related Work</b> .....	<b>- 6 -</b>
2.1	<b>Transport Layer Security</b> .....	<b>- 6 -</b>
2.1.1	Certificate Authorities .....	- 6 -
2.1.2	Certificate Signing Request .....	- 7 -
2.1.3	From Key Exchange to CIA .....	- 8 -
2.1.4	Man-In-The-Middle Attacks.....	- 8 -
2.2	<b>Secure Shell</b> .....	<b>- 9 -</b>
2.2.1	Verify-Once Key-Exchange .....	- 9 -
2.2.2	Perfect Forward Secrecy .....	- 10 -
2.2.3	Replay Attacks .....	- 10 -
2.3	<b>Off The Record Messaging</b> .....	<b>- 11 -</b>
2.3.1	Unauthenticated DHKE .....	- 11 -
2.3.2	Bulk Message Protection.....	- 11 -
2.3.3	Socialists Millionaires Protocol .....	- 12 -
<b>3</b>	<b>Theory and Algorithms</b> .....	<b>- 13 -</b>
3.1	<b>Key Exchange</b> .....	<b>- 13 -</b>
3.1.1	Diffie-Hellman Key Exchange.....	- 13 -
3.2	<b>Confidentiality</b> .....	<b>- 16 -</b>
3.2.1	Alleged Rivest Cipher.....	- 16 -
3.2.2	Replay Attacks .....	- 19 -
3.3	<b>Integrity</b> .....	<b>- 20 -</b>
3.3.1	Secure Hash Algorithm .....	- 20 -
3.4	<b>Authentication</b> .....	<b>- 25 -</b>
3.4.1	Hash-based Message Authentication Code.....	- 26 -
3.5	<b>Protocol Design</b> .....	<b>- 29 -</b>
3.5.1	Operator Definitions .....	- 30 -
3.5.2	Functions and Methods.....	- 30 -
3.5.3	Peer Notation .....	- 30 -
3.5.4	Program Variables .....	- 31 -
3.5.5	Authenticated Key Exchange .....	- 32 -
3.5.6	Bulk Message Exchange.....	- 33 -
<b>4</b>	<b>Application and Implementation</b> .....	<b>- 34 -</b>
4.1	<b>Program Execution</b> .....	<b>- 34 -</b>
4.1.1	Alex.....	- 34 -
4.1.2	Bob.....	- 35 -

<b>4.2</b>	<b>Packet Capture Analysis</b> .....	<b>- 35 -</b>
4.2.1	Key Exchange Initiation.....	- 35 -
4.2.2	Key Exchange Response .....	- 36 -
4.2.3	Bulk Message Exchange.....	- 36 -
<b>5</b>	<b>Conclusion</b> .....	<b>- 37 -</b>
5.1	Results.....	- 37 -
<b>6</b>	<b>Appendix</b> .....	<b>- 40 -</b>
6.1	seccom.c.....	- 40 -

# Figures

Figure 1 .....	- 8 -
Figure 2 .....	- 14 -
Figure 3 .....	- 15 -
Figure 4 .....	- 17 -
Figure 5 .....	- 17 -
Figure 6 .....	- 18 -
Figure 7 .....	- 21 -
Figure 8 .....	- 22 -
Figure 9 .....	- 23 -
Figure 10 .....	- 25 -
Figure 11 .....	- 27 -
Figure 12 .....	- 28 -
Figure 13 .....	- 30 -
Figure 14 .....	- 32 -
Figure 15 .....	- 33 -
Figure 16 .....	- 34 -
Figure 17 .....	- 35 -
Figure 18 .....	- 35 -
Figure 19 .....	- 36 -
Figure 20 .....	- 36 -

# 1 Introduction

## 1.1 Motivation

Most computers in a lab are connected together via some sort of layer-2, Ethernet-based network switch. Communication between them is typically allowed unless the switch is off, Virtual LANs (VLANs) are enabled or a firewall is between them. This means it is likely possible for various parties on the LAN to talk with each other in a secure fashion whether or not this action is desirable by others.

These observations brought up an interesting challenge of implementing an application that is lightweight, easy to use, and hopefully secure enough to take advantage of these facts. This means that knowledge in various areas such as, networking, programming, security and cryptography is required to accomplish such a goal. In addition, an application like this could be used as a basis for future implementations of secure LAN communication services.

The work in this paper could allow for the peers on a LAN to violate certain institutional policies put in place wishing to prevent peer communication. For example, this includes a lab room setting where a test may be taking place or even a work place which hosts employees dealing with confidential, sensitive information. One of Seneca's policies is to disable the Wide Area Network (WAN) part of the router but this does not necessarily include the LAN part of the switch. As stated above, this means that students could be capable of cheating during a test. The material covered could encourage organizations to improve their network

configurations and policies used when connecting a series of computers together and wishing to limit the sharing of information between peers.

## 1.2 Outline

This paper will begin with an observation of a potential problem along with a clear overall goal. The sub-steps of the overall solution will be explained in detail first. Once each step is explained, the theory or math behind the various topics will be described as much as possible. Lastly, the final application of each idea or concept will be implemented so one can actually achieve what is discussed in this paper.

The first step in describing this research starts with choosing a language based on its capabilities. This means it has to be capable of network communication along with basic data processing methods and operations. In addition, an overall protocol design must be decided on ahead of time along with the specific cryptographic components it will make use of.

The next part includes describing how two clients will begin discovery and communication of one another. Since security is desired here, a strong key exchange method will have to be chosen which typically includes a Diffie-Hellman Key Exchange (DHKE).

Once a secret session key is shared, the peers or clients can then move on to exchanging confidential messages with each other. This includes the use of some type of enciphering method such as the Alleged Rivest Cipher (ARC).

In order to ensure integrity, each end must first verify any message before it is transmitted or after it is received. This task requires the use of a cryptographic-based hash function such as the Secure Hash Algorithm (SHA).

Lastly, each peer must verify the authenticity of each message by using a hash function in a special manner. The algorithm combines a hash function along with a shared, secret key to create a unique, authenticate fingerprint of some data. This is often referred to as a Hash-based Message Authentication Code (HMAC).

Once all of these parts are covered, the actual code to implementing them will be made available by the end of this paper. This means that anyone with the capable tools of compiling this software should be able to make use of this information.

### **1.3 Goals**

The goal of this work will attempt to point out a semi important and possibly undesirable capability of computer lab rooms. This paper could help enlighten an organization or institution in creating guidelines for what is considered a secure lab environment. An invalid assumption, for example, includes the thought that peers on a network without internet access are unable to communicate to one another securely and locally. Often, the best way to prove a point is by making an application that demonstrates your idea as simply and precisely as possible along with being easy to operate by anyone wishing to use it.

### **1.4 Approach**

Reading a series of Request For Comments (RFC) articles published by the Internet Engineering Task Force (IETF) provides an in-depth look at what math is



required behind these specific algorithms. In addition, the Federal Information Processing Standards (FIPS), which is published by the Institute of Electrical and Electronics Engineers (IEEE) and hosted by the National Institute of Standards and Technology (NIST), provides a similar level of deep algorithmic explanation.

Once the math is understood for the various concepts required by this paper, implementation can begin given that proper Application Programming Interfaces (API) exist for the chosen language. The Open Group, which was merged with the Open Software Foundation (OSF) contains simple, basic references to certain headers, libraries, method, and function calls in the C programming language.

## 1.5 Tools Used

### 1.5.1 Software requirements

- vim
- gcc (ld and as)
- gdb
- gmp (Stallman)
- sockets
- string(s)
- stdio
- stdlib

### 1.5.2 Hardware requirements

- Layer-2 network switch
- 3 or more computers

- CAT-5e network cabling

## 2 Related Work

### 2.1 Transport Layer Security

The Transport Layer Security (TLS) protocol (the successor to Secure Sockets Layer) has been around for years, as the need for online security was very much desired. The big proponents pushing the development of secure online communication channels consisted of mainly banks and electronic commerce companies. These companies knew their users would one day want to perform such information-sensitive transactions as it brings the added convenience of being able to login from anywhere at any time. These companies knew that customers would be hesitant in using such services unless they were ensured that their personal information and communications were indeed secure. In addition, it helps these companies from being subject to liability if something goes wrong as they could demonstrate that they followed all of the related standards and procedures.

#### 2.1.1 Certificate Authorities

TLS starts off with requiring that a series of centralized and controlled Certificate Authority (CA) organizations are established which can authenticate other certificates issued by intermediate parties. This means that it employs a generalized model for signing other certificates which allows users to perform automated and authenticated key exchanges. Each CA will generate a unique public and private asymmetric key pair. Once this is done, a client application (for example a web browser) will then collect a list of public keys from each CA party along with

their basic information such as name, address, email, country, domain, etc. The private keys mentioned in any instance must be kept secret from unwanted and undesirable parties.

### **2.1.2 Certificate Signing Request**

The intermediate certificate holders wishing to make use of TLS need to submit a Certificate Signing Request (CSR) to any legitimate CA in order to get their certificate validated and authenticated. The process starts with choosing a verified CA since there is typically many of them available. Once a CA is chosen, one has to generate a CSR, which includes all related information such as the domain name and public key. As a note, the public key is generated with an asymmetric private key part at the same time. Once the CSR is submitted, the CA will cryptographically hash all of the submitted certificate data (along with a basic constraints field stating that it can only be used for authentication, not signing) and then use its private key to encrypt the hash value itself. This newly signed certificate is sent back to the certificate holder for use in any new connections to them.

### 2.1.3 From Key Exchange to CIA

Once the initial certificates and public keys are installed, a similar approach is followed as seen in the table below.

Client	Medium	Server
Application with CA list Init Conn to Domain name		
	C -> S : ???	
		crt = signed cert
	C <- S : crt	
a = hash(cert_info) b = decr(sign_info, ca_pub_key) Verify -> a equals b		
q = encr(DHKE, crt_pub_key)		
	C -> S : q	
		s = decr(q, crt_pri_key)
		r = encr(DHKE, crt_pri_key)
	C <- S : r	
s = decr(r, crt pub key)		
m = plaintext message i = random IV c = encr_cbc(i, m, s) h = hmac_hash(m, s) p = (i    c    h)		
	C -> S : p	
		i = p[0] m = decr_cbc(i, p[1], s) h = hmac_hash(m, s) Verify -> h equals p[2]

**Figure 1**

An official description of the above can be found in the related RFC describing all of the TLS protocol (such as the RSA and DHKE algorithms) in great detail (Dierks, appendix-F.1.1.2&3).

### 2.1.4 Man-In-The-Middle Attacks

Since the initial public keys of each CA are manually installed by a vendor, assuming that no private keys have been compromised, TLS can verify with a high amount of certainty if a Man-In-The-Middle (MITM) attack is taking place or not.

This means that any MITM taking place can be detected and stopped before any sensitive data is transmitted assuming that self-signed certificates are not in use.

## 2.2 Secure Shell

The SSH protocol was designed to replace an insecure, plaintext remote-terminal client and server application named telnet. This protocol allows an authorized user of a computer system to access a server remotely via a command line (or GUI) interface.

### 2.2.1 Verify-Once Key-Exchange

The OpenSSH developers took an interesting stance on how key exchanges should take place. They decided that it was the user's responsibility in "first verifying the association between the server host key and the server host name" (Ylonen, rfc4251 section-9.3.4) before a trusted and secure connection can take place. When a client attempts to connect to a server for the first time, the server sends its asymmetric public key in the clear while retaining the private key in secrecy as described in (Ylonen, rfc4253 section-8). The user's client will either present a hashed fingerprint of the public key, or more recently, an ASCII-art drawing of it. The assumption is that the user will take responsibility and verify this fingerprint manually with a system administrator through some other channel (for example the telephone system). However, once a public key is transferred and verified, a MITM attack is unable to take place afterwards as the initially agreed upon hash fingerprint of the key exchange will differ afterwards. A trade off was made between an initial user inconvenience and greater protection against active

attackers. The protocol then follows a typically CIA model (like the general one described in TLS) to achieve end-to-end security.

### **2.2.2 Perfect Forward Secrecy**

“PFS is essentially defined as the cryptographic property of a key-establishment protocol in which the compromise of a session key or long-term private key after a given session does not cause the compromise of any earlier session” (Ylonen, rfc4251 section-9.3.7). Since the SSH protocol is based on a standard DHKE, a new session key is generated for every connection that is established. This means that even if a long-term asymmetric key is disclosed, any previous session key exchanged underneath it is still protected. On another note, if any single session key is ever determined, no previous or future communications will be compromised as they have made use (or will make use) of a different key entirely for that session.

### **2.2.3 Replay Attacks**

The Diffie-Hellman key exchange that takes place helps to prevent long-term replay attacks against the SSH protocol as the session keys established are never used again in the future. The Transmission Control Protocol (TCP) also helps in preventing spoofed, short-term replay attacks as a unique session ID is assigned to each peer and port and is only increased throughout the connection. “The transport protocol provides a unique session identifier (bound in part to pseudo-random data that is part of the algorithm and key exchange process) that can be used by higher level protocols to bind data to a given session and prevent replay of data from prior

sessions” (Ylonen, rfc4251 section-9.3.3). In addition, random Initialization Vectors (IV) and monotonically increasing sequence numbers can be used with the encryption keys to also help in eliminating instant replay attacks.

## **2.3 Off The Record Messaging**

The Off The Record (OTR) messaging protocol is a widely adopted and greatly respected protocol, which implements state-of-the-art algorithms to achieve very high security. This protocol is generally used in instant messaging applications as the ability for plausible deniability is specifically achieved here as well.

### **2.3.1 Unauthenticated DHKE**

The protocol starts off by performing an unauthenticated Diffie-Hellman Key Exchange (DHKE) combined with the complicated usage of asymmetric keys, Advanced Encryption Standard (AES), hash functions and Hash-based Message Authentication Codes (HMAC). The details of how they perform the new version of the initial key exchange (Alexander, section 2.3) versus the old method (Borisov, section 4.3) is beyond the scope of this paper but is covered in great detail by the mentioned sources.

### **2.3.2 Bulk Message Protection**

A monotonically increasing index number is initialized for every new DH session key that is exchanged. This is done to prevent short-term replay attacks from taking place by an active attacker. A small note includes the interesting fact that AES is used in Counter mode (CTR) which forces it to act like a stream cipher instead of a typical block cipher. The AES-CTR algorithm is given the session key



plus the index number as the encryption key which allows a message to become confidential. Lastly, an HMAC is calculated with hash of the encryption key to not only provide authentication and integrity but also allowing it to be announced publically after the message is transmitted. Since this key is a one-way hash of the encryption key, it can be verified by each peer to be authentic without giving away the encryption key and yet it can also be published to everyone allowing anyone to also sign messages afterwards. This allows a peer to achieve plausible deniability as nothing permanent or personal is ever tied to the messages they've signed (unlike a long-term public/private cryptographic keying system).

### **2.3.3 Socialists Millionaires Protocol**

There is a complicated procedure that is performed in OTR, which helps to detect or verify if a MITM attack is taking place. This specific algorithm is said to allow two millionaires to determine if they are equally as rich as each other. "This is itself a variant of Yao's original Millionaires' Problem, where the millionaires wish to know who is richer without revealing any other information about their wealth" (Alexander, section 4). This is means that some sort of irreversible calculation is performed on the numbers, which retains their respective sizes, but masks their original values thus allowing them to be compared logically. This protocol can be used here to help two peers determine if their shared session keys match or not. If they algorithm's output produces a mismatch, that is a likely indicator that a MITM attack is currently taking place.

## 3 Theory and Algorithms

### 3.1 Key Exchange

One of the main reasons to perform an extended key exchange rather than just using a shared, secret passphrase includes the fact that the calculated keys will provide the security property called Perfect Forward Secrecy, as they are always re-calculated and different for each peer and session. This means that if any given session key is discovered, it does not compromise any previous or future keys calculated versus a static password which could be discovered by an attacker. It's important to note that it is the user's responsibility to select a strong passphrase as the initial key exchange depends on its strength along with the rest of the communication.

#### 3.1.1 Diffie-Hellman Key Exchange

The main reasoning behind why the DHKE is considered secure relies on the current mathematical fact that it is extremely hard to perform the inverse of exponentiation when given large enough exponents. Performing the inverse of exponentiation is referred to as calculating a logarithm and this is a hard, indirect process to perform.

A prime is used a modulus here because not only does it help to keep the numbers produced relatively small after a huge calculation but primes are also capable of producing an evenly-random distribution of "wrapped" numbers as they contain no multiples of any other number.

### 3.1.1.1 Modular Exponentiation

Notes:

- / denotes integer division
- % denotes modulus

```
function modexp(bnum g, bnum x, bnum p)
{
    o = 1
    while (x > 0)
    {
        if ((x % 2) == 1)
        {
            o = ((o * g) % p)
        }
        x = (x / 2)
        g = ((g * g) % p)
    }
    return o
}
```

**Figure 2**

This function of calculating big number exponentiation while modulating the result can be found in more detail from the following source (Schneier, chapter 11.3). As you can see however, the use of some sort of big number library would be required here thus making its implementation a bit more difficult.

### 3.1.1.2 Key Calculation and Exchange

One is able to perform a DHKE by first publically agreeing upon a shared base (g) along with a safe prime (p). Each party then generates a random, large exponent (x) and performs the following calculations where rand(size) returns a random array of bits with size length:

Alice	Medium	Bob
<code>x = rand(1024)</code>		
<code>q = modexp(g, x, p)</code>		
	Alice -> Bob : q	
		<code>y = rand(1024)</code>
		<code>r = modexp(g, y, p)</code>
	Alice <- Bob : r	
<code>s = modexp(r, x, p)</code>		<code>s = modexp(q, y, p)</code>

**Figure 3**

“The protocol begins with one party, Alice, creating a random number (x) and sending the exponential (a<sup>x</sup>) to the other party, Bob ... Bob creates a random number (y) and uses Alice’s exponential to compute the exchanged key K = ((a<sup>x</sup>)<sup>y</sup>). Bob responds with the exponential (a<sup>y</sup>) and ... Alice computes K” (Diffie, section 5.1).

### 3.1.1.3 Man in the Middle Attacks

The DH key exchange on its own is vulnerable to a MITM attack as it is just a simple calculation performed between two parties. There is nothing involved in the basic calculation that helps to identify or authenticate each member. This means that an active attacker in the middle of a connection is able to impersonate each peer during a key exchange and effectively perform two separate calculations. Once

this happens, the attacker is now spliced in the middle of the connection and is able to decrypt, read and re-encrypt every message while passing them on as if everything is still secure.

A common practice to help prevent an attack like this is to sign or authenticate the Diffie-Hellman transmissions themselves. This means one is able to use either a Hash-Based Message Authentication Code (HMAC) algorithm with a shared secret key or an asymmetric encryption algorithm with a pre-shared public key to help sign and authenticate each calculation. Note, if a public key is also transmitted at the time of connection, it is also subject to a MITM attack itself just like the DHKE.

## **3.2 Confidentiality**

Encryption and decryption methods allow peers to achieve secrecy for the information they are transferring and this is often referred to as confidentiality. The typical model generally includes parties first sharing or calculating some secret key material and then applying some sort of method on the given data. This method should be easily reversible to those with the key and hard to decode to those without it. In an ideal algorithm, the only attack that should be available to an attacker should be a brute force attack as it forces them to try and guess through an entire set of key combinations.

### **3.2.1 Alleged Rivest Cipher**

ARC is a symmetric stream based cipher, which can be used to protect messages from being viewed by unwanted parties.

The XOR operator is used in the ARC algorithm because it is capable of producing an even distribution of output values based on every unique combination of input items per each key type. This property allows the XOR operator to achieve reversibility given the original output and key item. The output below demonstrates the various states that bitwise XOR produces where (a XOR b EQUALS c):

a	b	c
0	0	0
0	1	1
1	0	1
1	1	0

**Figure 4**

### 3.2.1.1 Key Schedule

Notes:

- % denotes the modulus operator

```
ARC4 Key Schedule

function keys(array pass, numb size)
{
    S = array[256]
    for (i from 0 to 255)
    {
        S[i] = i
    }
    j = 0
    for (i from 0 to 255)
    {
        j = ((j + S[i] + pass[i % size]) % 256)
        t = S[i]
        S[i] = S[j]
        S[j] = t
    }
    return S
}
```

**Figure 5**

This method can be described by a procedure which states that you “allocate an 256 element array of 8 bit bytes to be used as an S-box ... Initialize the S-box. Fill each entry first with it's index ... Fill another array of the same size (256) with the key, repeating bytes as necessary ... Set j to zero and initialize the S-box” (Kaukonen, section 3.1).

### 3.2.1.2 Pseudo Random Number Generation

Notes:

- ^ denotes the Exclusive-OR operation

```
ARC4 PRNG

function prng(array inpt, numb size, array S)
{
    i = 0
    j = 0
    o = array[size]
    for (x from 0 to (size - 1))
    {
        i = ((i + 1) % 256)
        j = ((j + S[i]) % 256)
        t = S[i]
        S[i] = S[j]
        S[j] = t
        K = S[(S[i] + S[j]) % 256]
        o[x] = (inpt[x] ^ K)
    }
    return o
}
```

Figure 6

This method also simply swaps values of the key state before using a random value from it for encryption. This can further be described by saying, “for either encryption or decryption, the input text is processed one byte at a time. A pseudorandom byte K is generated ... To encrypt, XOR the value K with the next byte

of the plaintext. To decrypt, XOR the value K with the next byte of the ciphertext” (Kaukonen, section 3.2).

### **3.2.1.3 Predictive Key Stream Attacks**

The key-scheduling algorithm is said to be vulnerable to having its initial byte stream values predicted by an attacker. This paper does not go into the detail of this attack as it is published elsewhere. However, this paper will choose to implement ARC-drop[4096] which is designed to drop the first 4096 bytes worth of the key stream generated just to be safe.

### **3.2.1.4 Key Stream Recovery Attacks**

If an attacker knows the plaintext of a message but the key still remains secret, the key-stream for that particular message can be determined. As a side note, it is a required property of stream ciphers to prevent the recovery of the original keying material if the key stream is ever discovered. In general, there should always be measures in place to prevent the transmission of any message given a previously used key-stream. This typically implies that the assistance of an IV is necessary.

### **3.2.2 Replay Attacks**

If no IV is used (or the same IV is used more than once), an attacker is able to perform a replay attack on the given message and possibly have it appear valid. A common method to prevent such attacks is to ensure that a random Initialization Vector (IV) is used for each message encrypted along with a monotonically increasing counter such as the current timestamp of the system measured in seconds for example.



### 3.3 Integrity

The ability for two or more parties to ensure that some data set has not been appended, modified, or shortened implies that one has the ability to verify the integrity of that information. There are algorithms which are capable of starting off with a random hash set and mixing in some given data in a way that is unpredictable, yet consistent each time and producing what is referred to as a “unique fingerprint”. This fingerprint should not be subject to easy reproduction and should be at least fifty percent different if any single bit of the input message changes.

#### 3.3.1 Secure Hash Algorithm

A cryptographic based hashing algorithm is essentially likened to a method that produces a unique fingerprint for some given data set. Any hash digest, which is considered to be secure, must be able to retain the following properties:

- Given a hash (h), one should not be able to find a message (m) such that  $\text{hash}(m) = h$  (Preimage resistance)
- Given some message (m1), one should not be able to find another message (m2) such that  $\text{hash}(m1) = \text{hash}(m2)$  (Second-preimage resistance)
- It should be infeasible to find any two messages such that  $\text{hash}(m1) = \text{hash}(m2)$  (Collision resistance)

### 3.3.1.1 Preliminary Methods

Notes:

- $\sim$  represents bitwise negation
- $\&$  denotes bitwise AND operation
- $\wedge$  represents bitwise XOR operation
- $\gg$  and  $\ll$  symbolize bitwise shifting bits with zero fill

```
Hash Function Methods

function Safe(numb x)
{
    return (x & 0xffffffff)
}

function Rotr(numb x, numb n)
{
    x = Safe(x)
    return ((x << (32 - n)) | (x >> n))
}

function Ch(numb x, numb y, numb z)
{
    return ((x & y) ^ ((~x) & z))
}

function Maj(numb x, numb y, numb z)
{
    return ((x & y) ^ (x & z) ^ (y & z))
}

function Sum0(numb x)
{
    return (Rotr(x, 2) ^ Rotr(x, 13) ^ Rotr(x, 22))
}

function Sum1(numb x)
{
    return (Rotr(x, 6) ^ Rotr(x, 11) ^ Rotr(x, 25))
}

function Sig0(numb x)
{
    x = Safe(x)
    return (Rotr(x, 7) ^ Rotr(x, 18) ^ (x >> 3))
}

function Sig1(numb x)
{
    x = Safe(x)
    return (Rotr(x, 17) ^ Rotr(x, 19) ^ (x >> 10))
}
```

Figure 7

These function definitions can be found in the related source (Gallagher, fips180-2 section 4.1.2).

### 3.3.1.2 Initial Constants

```
Hash Function Constants

K =
[
  0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5,
  0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,
  0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3,
  0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,
  0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc,
  0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
  0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7,
  0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,
  0x27b70a85, 0x2e1b2138, 0x4d2c6dfe, 0x53380d13,
  0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,
  0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3,
  0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,
  0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5,
  0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6fff3,
  0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208,
  0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2
]
```

Figure 8

This constant definition can be found in the related source (Gallagher, fips180-2 section 4.2.2).

### 3.3.1.3 Message Padding

```
function padd(array inpt, numb size)
{
    s = (size * 8)
    o = array[size + 128]
    for (x from 0 to (size - 1))
    {
        o[x] = inpt[x]
    }
    o[size] = char(0x80)
    size = (size + 1)
    while ((size % 64) != 56)
    {
        o[size] = char(0x00)
        size = (size + 1)
    }
    for (x from 0 to 3)
    {
        # ((2^32) - 1) / 8 sizes only
        o[size] = char(0x00)
        size = (size + 1)
    }
    for (x from 0 to 3)
    {
        t = (24 - (x * 8))
        o[size] = char((s >> t) & 0xff)
        size = (size + 1)
    }
    p = ""
    for (x from 0 to (size - 1))
    {
        p = (p + o[x])
    }
    return p
}
```

Figure 9

### 3.3.1.4 Digest Algorithm

#### Hash Function Core

```
function hash(array inpt, numb size)
{
    m = padd(inpt, size)
    x = 0
    H =
    [
        0x6a09e667,0xbb67ae85,0x3c6ef372,0xa54ff53a,
        0x510e527f,0x9b05688c,0x1f83d9ab,0x5be0cd19
    ]
    while (x < len(m))
    {
        W = array[64]
        i = x
        for (y from 0 to 15)
        {
            W[y] = 0
            for (z from 0 to 3)
            {
                t = (24 - (z * 8))
                W[y] = (W[y] | (numb(m[i]) << t))
                i = (i + 1)
            }
        }
        for (y from 16 to 63)
        {
            W[y] = (Sig1(W[y - 2]) + W[y - 7] + Sig0(W[y - 15]) + W[y - 16])
        }
        a = array[8]
        for (y from 0 to 7)
        {
            a[y] = H[y]
        }
        for (y from 0 to 63)
        {
            t1 = (a[7] + Sum1(a[4]) + Ch(a[4], a[5], a[6]) + K[y] + W[y])
            t2 = (Sum0(a[0]) + Maj(a[0], a[1], a[2]))
            a[7] = a[6]
            a[6] = a[5]
            a[5] = a[4]
            a[4] = (a[3] + t1)
            a[3] = a[2]
            a[2] = a[1]
            a[1] = a[0]
            a[0] = (t1 + t2)
        }
        for (y from 0 to 7)
        {
            H[y] = (H[y] + a[y])
        }
        x += 64
    }
    o = ""
    c = "0123456789abcdef"
    for (x from 0 to 7)
    {
        for (y from 0 to 7)
        {
            t = (28 - (y * 4))
            u = ((H[x] >> t) & 0xf)
            o = (o + c[u])
        }
    }
}
```

```
    }  
    return o  
}
```

**Figure 10**

This core function algorithm can be found in the related source (Gallagher, fips180-2 section 6.2.2).

### **3.3.1.5 Birthday Attack**

The birthday attack is based on a problem which asks, “How many random people would it take until the probability of finding two people with the same birth date is at least 50 percent?” It turns out that people often underestimate how fast or quickly a duplicate birthday appears in this little hypothetical situation. This relates to cryptography as the strength of a hash function is related to this problem or paradox as it’s known. The reasoning is, since a hash function produces a deterministically random output of static size, it is subject to the question, “How many random input messages would it take before the probability of finding two identical output hashes is at least 50 percent?” It turns out that this attack reduces the strength of most hash functions to only half its output size in bits given some complicated math that is beyond the scope of this paper.

## **3.4 Authentication**

The ability for one peer to verify that another communicating party is who they say they are is referred to as being able to perform authentication. This means you should be able to verify that some piece of data came from someone you expected and not a possible attacker. There are a few ways one can sign a message,

including the use of public/private keys, but this paper deals with algorithms that are based on cryptographically secure hash functions.

#### **3.4.1 Hash-based Message Authentication Code**

HMAC's are based on a special combination of message and keying material in order to produce a fingerprint, which is not only unique but authenticated as well. The reason why one should not just simply hash a message with key includes the risk that if a weakness is ever found in the underlying hash function, it may be easier for an attacker to forge signed messages, even if the key is still unknown.

### 3.4.1.1 Preliminary Methods

```
function htoa(array inpt, numb size)
{
  o = ""
  x = 0
  c = "0123456789abcdef"
  while ((x + 1) < size)
  {
    n = 0
    for (y from 0 to 15)
    {
      if (c[y] == inpt[x])
      {
        n = (n + (y << 4))
      }
      if (c[y] == inpt[x + 1])
      {
        n = (n + y)
      }
    }
    o = (o + char(n))
    x = (x + 2)
  }
  return o
}
```

**Figure 11**



### 3.4.1.2 Signing Algorithm

```
function hmac(array inpt, numb ilen, array skey, numb slen)
{
    b = 64
    if (slen > b)
    {
        skey = hash(skey, slen)
        skey = htoa(skey, leng(skey))
        slen = leng(skey)
    }
    while (slen < b)
    {
        skey = (skey + char(0))
        slen = (slen + 1)
    }
    ipad = ""
    opad = ""
    for (x from 0 to (b - 1))
    {
        ipad = (ipad + char(0x36 ^ numb(skey[x])))
        opad = (opad + char(0x5c ^ numb(skey[x])))
    }
    ihsh = hash(ipad + inpt, b + ilen)
    ihsh = htoa(ihsh, leng(ihsh))
    ohsh = hash(opad + ihsh, b + leng(ihsh))
    return ohsh
}
```

Figure 12

The algorithm above can be described by the following procedure, “append zeros to the end of K to create a B byte string ... XOR (bitwise exclusive-OR) the B byte string computed in step (1) with ipad ... append the stream of data 'text' to the B byte string resulting from step (2) ... apply H to the stream generated in step (3) ... XOR (bitwise exclusive-OR) the B byte string computed in step (1) with opad ... append the H result from step (4) to the B byte string resulting from step (5) ... apply H to the stream generated in step (6) and output the result” (Krawczyk, section 2).

### 3.5 Protocol Design

A number of design decisions were taken from and based off of the OTR protocol which implements fairly strong anonymity and security features. The protocol here starts off by requiring that all peers wishing to communicate with each other share a common, secret passphrase. This is done so that any initial communication can be secured, authenticated, and verified amongst all peers. This helps to prevent any passive or active attacks from interfering with an initial key exchange process which is helping to set up the keying material for future communication. The protocol also makes use of a random IV and incremented timestamp for each message being encrypted. This helps to prevent key or message leakage along with providing replay attack prevention.

The first phase of communication includes any ready peers to participate in performing an authenticated DHKE. Since the DHKE is vulnerable to an active MITM attack, the pre-shared passphrase is used to help authenticate any key exchange sent. In addition, since the DHKE is used here first, it provides the cryptographic property called Perfect Forward Secrecy as mentioned before.

The next phase of communication includes the peers participating in the exchange of bulk messages. These messages are always encrypted with their related Diffie-Hellman session key which each peer exchanged with one another previously. All messages are also signed with the HMAC algorithm making use of that same session key. This allows messages to remain confidential, with integrity, and be authenticated.

The protocol assumes anonymity is achievable as long as any recorded traffic is not tied to any network port and not tied to any peer in particular. In addition, a peer may be able to spoof their MAC and IP addresses ahead of time which won't interfere with the ability to send UDP based LAN broadcast traffic on the switch. This is true since the "protocol provides a procedure for application programs to send messages to other programs with a minimum of protocol mechanism. The protocol is transaction oriented..." (Postel, Introduction section).

### 3.5.1 Operator Definitions

^ = exponentiation operator  
 % = modulus operator  
 || = byte concatenation

### 3.5.2 Functions and Methods

Function	Description
rand(size)	return a random set of bits of size length
modexp(g, p, m)	perform an efficient calculation of $((g \wedge p) \% m)$
append(array, addr, item)	append or update element (item) for entry (addr) in the array (array)
remove(array, addr, item)	remove element (item) for entry (addr) from array (array)
check(array, addr, time)	verify the seconds (time) given for entry (addr) is greater than the one store in array (array)
hmac(inpt, skey)	calculate the hmac of data (inpt) with key (skey)
vrfy(inpt, skey, hash)	verify hmac (hash) with data (inpt) and key (skey)
encl(ivec, inpt, skey)	encrypt data (inpt) with key (skey) and seed (ivec)
decr(ivec, inpt, skey)	decrypt data (inpt) with key (skey) and seed (ivec)
time()	returns the current time in seconds since epoch
print(outp)	display the given output data (outp) to the user

**Figure 13**

### 3.5.3 Peer Notation

<c-> = unauthenticated client  
<c~> = authenticating client  
<c+> = authenticated client  
<bc> = broadcast connecting all clients together  
<cn> = any number of listening potential clients  
<x> -> <y> :: <z> = peer <x> sending peer <y> data <z>

### 3.5.4 Program Variables

b = pre-shared DH base  
p = pre-shared DH safe prime modulus  
a = ipv4 address of the local client  
c = ipv4 address of the remote client  
k = pre-shared secret passphrase  
l = the list of authenticated client session information

### 3.5.5 Authenticated Key Exchange

Client A	Client B
<pre> &lt;c-&gt; x = rand(size) q = modexp(b, x, p) t = time() i = rand(size) n = hash(k    a    t    i) h = hmac(q, n) e = encr(q, n) m = (0    t    i    h    e) &lt;c~&gt; -&gt; &lt;bc&gt; :: m </pre>	
	<pre> &lt;cn&gt; y = rand(size) r = modexp(b, y, p) t = time() i = rand(size) n = hash(k    a    t    i) h = hmac(r, n) e = encr(r, n) m = (1    t    i    h    e) &lt;cn&gt; -&gt; &lt;c~&gt; :: m </pre>
<pre> &lt;c+&gt; for (each) m (from) &lt;cn&gt; check(l, c, m[1]) n = hash(k    c    m[1]    m[2]) d = decr(m[4], n) h = hmac(d, n) verfy(d, n, m[3]) s = modexp(d, x, p) i = (c    m[1]    s) append(l, c, i) </pre>	<pre> &lt;cn&gt; check(l, c, m[1]) n = hash(k    c    m[1]    m[2]) d = decr(m[4], n) h = hmac(d, n) verfy(d, n, m[3]) s = modexp(d, y, p) i = (c    m[1]    s) append(l, c, i) </pre>

**Figure 14**

### 3.5.6 Bulk Message Exchange

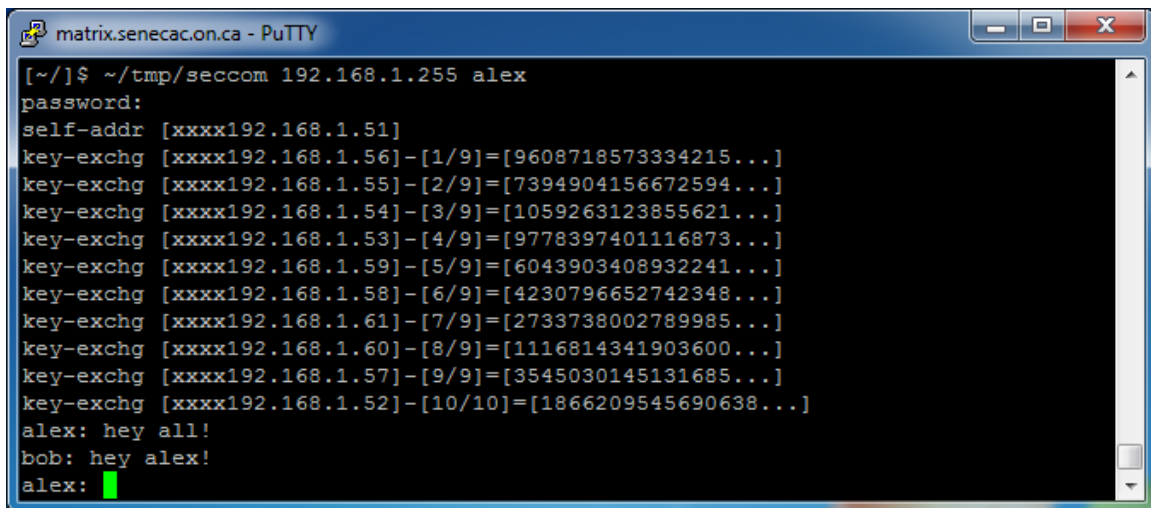
Client A	Client B
<pre>&lt;c+&gt; o = &lt;message&gt; for (each) j (in) l   t = time()   i = rand(size)   k = hash(j[2]    a    t    i)   h = hmac(o, k)   e = encr(o, k)   m = (2    t    i    h    e) &lt;c+&gt; -&gt; &lt;j[0]&gt; :: m</pre>	<pre>&lt;cn&gt; check(l, c, m[1]) for (each) j (in) l   n = hash(j[2]    j[0]    m[1]    m[2])   d = decr(m[4], n)   vrfy(d, n, m[3])   print(d)   j[1] = m[2]</pre>

Figure 15

## 4 Application and Implementation

### 4.1 Program Execution

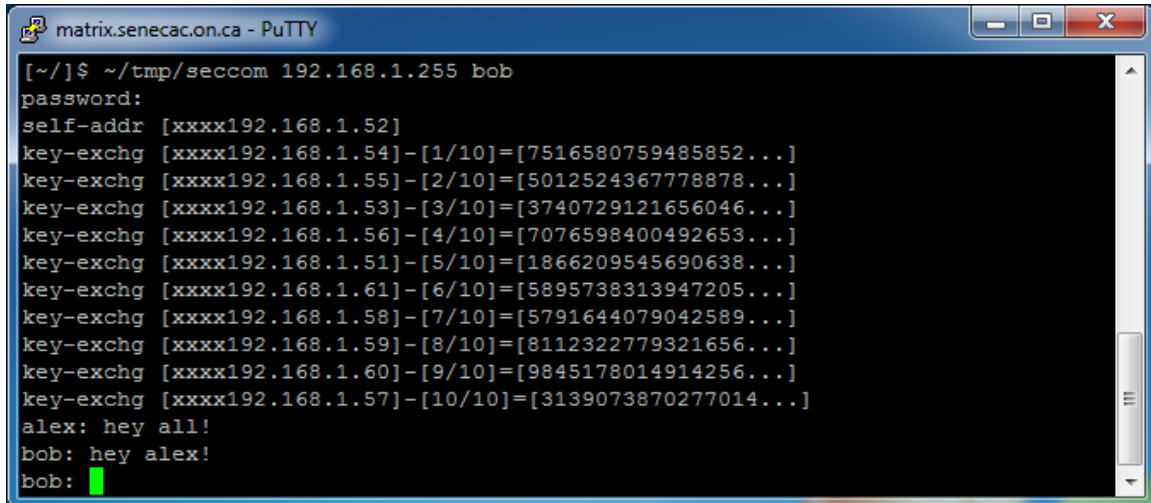
#### 4.1.1 Alex



```
matrix.senecac.on.ca - PuTTY
[~/]$ ~/tmp/seccom 192.168.1.255 alex
password:
self-addr [xxxx192.168.1.51]
key-exchg [xxxx192.168.1.56]-[1/9]=[9608718573334215...]
key-exchg [xxxx192.168.1.55]-[2/9]=[7394904156672594...]
key-exchg [xxxx192.168.1.54]-[3/9]=[1059263123855621...]
key-exchg [xxxx192.168.1.53]-[4/9]=[9778397401116873...]
key-exchg [xxxx192.168.1.59]-[5/9]=[6043903408932241...]
key-exchg [xxxx192.168.1.58]-[6/9]=[4230796652742348...]
key-exchg [xxxx192.168.1.61]-[7/9]=[2733738002789985...]
key-exchg [xxxx192.168.1.60]-[8/9]=[1116814341903600...]
key-exchg [xxxx192.168.1.57]-[9/9]=[3545030145131685...]
key-exchg [xxxx192.168.1.52]-[10/10]=[1866209545690638...]
alex: hey all!
bob: hey alex!
alex: █
```

Figure 16

## 4.1.2 Bob



```
matrix.senecac.on.ca - PuTTY
[~/]$ ~/tmp/seccom 192.168.1.255 bob
password:
self-addr [xxxx192.168.1.52]
key-exchg [xxxx192.168.1.54]-[1/10]=[7516580759485852...]
key-exchg [xxxx192.168.1.55]-[2/10]=[5012524367778878...]
key-exchg [xxxx192.168.1.53]-[3/10]=[3740729121656046...]
key-exchg [xxxx192.168.1.56]-[4/10]=[7076598400492653...]
key-exchg [xxxx192.168.1.51]-[5/10]=[1866209545690638...]
key-exchg [xxxx192.168.1.61]-[6/10]=[5895738313947205...]
key-exchg [xxxx192.168.1.58]-[7/10]=[5791644079042589...]
key-exchg [xxxx192.168.1.59]-[8/10]=[8112322779321656...]
key-exchg [xxxx192.168.1.60]-[9/10]=[9845178014914256...]
key-exchg [xxxx192.168.1.57]-[10/10]=[3139073870277014...]
alex: hey all!
bob: hey alex!
bob: █
```

Figure 17

## 4.2 Packet Capture Analysis

### 4.2.1 Key Exchange Initiation

```
0000a0: 6b 30 30 30 30 30 30 31 33 33 31 30 30 38 38 31
0000b0: 30 50 26 93 db 1c 87 1e bb 07 cf a9 71 39 8b ee
0000c0: 99 e9 9f ad 70 73 2f 5a 09 07 c5 ec 47 ae b0 df
0000d0: fe a0 47 ca 8e e1 b3 7c d2 88 7e ff 06 67 1a 8f
0000e0: f1 c4 b8 33 c0 8c 1b 2b 73 e7 88 d8 87 f3 8c 51
0000f0: 76 be c9 e3 c4 17 24 0d d0 6a c0 81 e4 20 41 ec
000100: 6b 18 75 c9 3f 59 9a 55 cb 6f b6 ce 9e 54 80 e5
*
0000a80: 62 18 cf 4d 09 43 58 a7 28 63 bd ec b9 63 bc 7b
0000a90: 2c c5 08

mode=k
pass=asdf
addr=xxxx192.168.1.56
time=0000001331008810
init=502693db * aeb0dffe
sign=a047ca8e * f38c5176
data=34500369 * 42190574
```

Figure 18



## 4.2.2 Key Exchange Response

```
00014a0: 6c 30 30 30 30 30 30 31 33 33 31 30 30 35 31 34
00014b0: 32 c2 17 1c 4d af 89 ee 2e 9e 04 de a8 0d 67 dc
00014c0: b9 2d 24 02 49 3b d9 a4 cb bb 23 0f ce ee 7b d6
00014d0: b0 22 f7 05 f8 a8 73 d9 8c 44 e8 4f bd 1d 2e 7a
00014e0: 59 26 37 e9 3d f1 d9 24 1b d9 bc e0 5e 85 6e a4
00014f0: cd c4 f7 93 74 7d a3 fb 81 ef fd b3 90 94 b1 67
0001500: 5f bf 66 11 2c 88 5b 72 1c 62 52 36 98 0e bc a0
*
0001e80: b1 4a 4a 75 35 cb 20 d1 dc 80 c5 e6 1b d0 6a 36
0001e90: c5 96 e5

mode=l
pass=asdf
addr=xxxx192.168.1.51
time=0000001331005142
init=c2171c4d * ee7bd6b0
sign=22f705f8 * 856ea4cd
data=68709269 * 99911521
```

Figure 19

## 4.2.3 Bulk Message Exchange

```
0000000: 6d 30 30 30 30 30 30 31 33 33 31 30 31 30 34 34
0000010: 30 24 47 cb 26 d8 a7 2c 03 a2 28 20 fb e5 38 93
0000020: 99 d6 b2 33 00 e5 74 15 53 e6 63 a0 b9 b8 1f 28
0000030: dc 52 c3 68 49 8f b7 5b a7 b9 2d 4d 76 c9 38 29
0000040: ec 3c a4 6a 04 89 14 21 4b 67 ca 9d b5 20 f3 40
0000050: 10 be 0c f3 e8 7d 37 f4 95 71 82 ea 8e d2 d3 76
0000060: 56 5c c6 8a

mode=m
pass=10866886 * 07707436
addr=xxxx192.168.1.51
time=0000001331010440
init=2447cb26 * b81f28dc
sign=52c36849 * 20f34010
data=alex: hey everyone!
```

Figure 20

## 5 Conclusion

### 5.1 Results

Assuming the limited testing done was correct, peers were able to successfully connect and communicate with one another using this protocol. The packet captures were able to help show what the transmitted data looked like exactly. The speeds for the key exchange, however, are a bit slow due to the massive size of the exponent and prime modulus numbers. In addition, the protocol and code base has not been analyzed yet for weaknesses or implementation flaws due to the limited time frame of this project. The algorithms were chosen based on their security and simplicity; however, they are not the newest or strongest to date. There exists a stronger key exchange algorithm called the Elliptic Curve Diffie-Hellman Key Exchange (ECDH) and a stronger encryption cipher named the Advanced Encryption Standard (AES) which can operate in Cipher Block Chaining (CBC) mode.

## Bibliography

Alexander, Chris and Ian Goldberg. 2007. Improved user authentication in off-the-record messaging. In: Proceedings of the 2007 ACM workshop on Privacy in electronic society - WPES '07, 41. Alexandria, Virginia, USA.  
doi:10.1145/1314333.1314340,  
<http://lcweb.senecac.on.ca:2126/citation.cfm?id=1314333.1314340&coll=DL&dl=ACM&CFID=10741576&CFTOKEN=25874701> (Accessed: 20. February 2011).

Borisov, Nikita, Ian Goldberg and Eric Brewer. 2004. Off-the-record communication, or, why not to use PGP. In: Proceedings of the 2004 ACM workshop on Privacy in the electronic society, 77–84. WPES '04. New York, NY, USA: ACM.  
doi:10.1145/1029179.1029200, .

Cerf, Vinton. RFC 675 - Specification of Internet Transmission Control Program.  
<http://tools.ietf.org/html/rfc675> (Accessed: 3. October 2011).

Diffie, Whitfield, Paul C. Oorschot and Michael J. Wiener. 1992. Authentication and authenticated key exchanges. Designs, Codes and Cryptography 2, Nr. 2 (June): 107-125. doi:10.1007/BF00124891, (Accessed: 19. February 2011).

Gallagher, P. 2001. ADVANCED ENCRYPTION STANDARD (AES). Federal Information Processing Standards Publication 197 (26. November).  
<http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>.

Gallagher, P. 2002. Secure Hash Standard (SHS). Federal Information Processing Standards Publication 180-2 (1. August).  
<http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf>.

Gallagher, Patrick and Gary Locke. 2009. DIGITAL SIGNATURE STANDARD (DSS). Federal Information Processing Standards Publication 186-3 (June).  
[http://csrc.nist.gov/publications/fips/fips186-3/fips\\_186-3.pdf](http://csrc.nist.gov/publications/fips/fips186-3/fips_186-3.pdf).

Halevi, Shai and Hugo Krawczyk. 1998. Public-key cryptography and password protocols. In: Proceedings of the 5th ACM conference on Computer and communications security - CCS '98, 122-131. San Francisco, California, United States. doi:10.1145/288090.288118,  
<http://lcweb.senecac.on.ca:2126/citation.cfm?id=288118&CFID=10662528&CFTOKEN=41822796> (Accessed: 19. February 2011).

Kaukonen, K. and R. Thayer. 1999. A Stream Cipher Encryption Algorithm “Arcfour”. 14. July. <http://www.mozilla.org/projects/security/pki/nss/draft-kaukonen-cipher-arcfour-03.txt> (Accessed: 24. February 2011).

Kivinen, T. and M. Kojo. 2003. More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE). May. <http://www.ietf.org/rfc/rfc3526.txt> (Accessed: 24. February 2011).

Kravitz, David. 3.0 Elliptic Curve Groups over Fp. <http://www.certicom.com/index.php/30-elliptic-curve-groups-over-fp> (Accessed: 24. February 2011).

Krawczyk, H., M. Bellare and R. Canetti. RFC 2104 - HMAC: Keyed-Hashing for Message Authentication. <http://tools.ietf.org/html/rfc2104> (Accessed: 3. October 2011).

Postel, J. 1980. User Datagram Protocol. August. <http://tools.ietf.org/html/rfc768> (Accessed: 7. April 2011).

Schneier, Bruce (1996). Applied Cryptography: Protocols, Algorithms, and Source Code in C, Second Edition (2nd ed.). Wiley. ISBN 978-0471117094.

Stallman, Richard. The GNU MP Bignum Library. <http://gmplib.org/> (Accessed: 3. October 2011).

Tim Dierks <[tim@dierks.org](mailto:tim@dierks.org)>. 2008. The Transport Layer Security (TLS) Protocol Version 1.2. August. <http://tools.ietf.org/html/rfc5246> (Accessed: 7. April 2011).

Ylonen, Tatu and Chris Lonvick. 2006. The Secure Shell (SSH) Protocol Architecture. January. <http://tools.ietf.org/html/rfc4251> (Accessed: 7. April 2011).

Ylonen, Tatu and Chris Lonvick. 2006. The Secure Shell (SSH) Transport Layer Protocol. January. <http://tools.ietf.org/html/rfc4253> (Accessed: 7. April 2011).

## 6 Appendix

### 6.1 seccom.c

```
/*
gcc -O2 -Wall -W -pedantic -g -o ...
gcc -DGMP -O2 -Wall -W -pedantic -g -lgmp -o ...
gcc -DGMP -I./gmp -L./gmp/.libs -O2 -Wall -W -pedantic -g -lgmp -Wl,-no_pie -o ...
*/

#include <ctype.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>
#include <termios.h>
#include <time.h>
#include <unistd.h>

#include <arpa/inet.h>
#include <netinet/in.h>
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <sys/types.h>

#define SOCKETPORT 31337
#define SPADSIZE 16
#define HASHSIZE 32
#define SKEYSIZE 256
#define BUFFSIZE 8192

/* beg: hashing */

unsigned int rrot(unsigned int w, int n)
{
    w = (w & 0xffffffff);
    w = ((w << (32 - n)) | (w >> n));
    w = (w & 0xffffffff);

    return w;
}

void hash(unsigned char *outp, unsigned char *inpt, unsigned int ilen)
{
    /*
state buffer is in 512-bit (64-byte) blocks
message needs a padding 8-bits (1-byte) appended
last block must contain the message size in bits 64-bits (8-bytes)

[ 0 ... 54 ] [ 55 ] [ 56 57 58 59 ] [ 60 61 62 63 ]
ending of data padding MSB message size LSB message size
55-bytes 1-byte 4-bytes 4-bytes

e3b0c44298fc1c149afb4c8996fb92427ae41e4649b934ca495991b7852b855 -
ca978112calbbdcfac231b39a23dc4da786eff8147c4e72b9807785afee48bb - a
ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad - abc
f7846f55cf23e14eebeab5b4e1550cad5b509e3348fbc4efa3a1413d393cb650 - message digest
71c480df93d6ae2f1efad1447c66c9525e316218cf51fc8d9ed832f2daf18b73 -
abcdefghijklmnopqrstuvwxy
*/
}
```

```

248d6a61d20638b8e5c026930c3e6039a33ce45964ff2167f6eced419db06c1 -
abcdcbcedcdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq
db4bfcdbd4da0cd85a60c3c37d3fbd8805c77f15fc6b1fdfe614ee0a7c8fdb4c0 -
ABCDEFHGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789
f371bc4a311f2b009eef952dd83ca80e2b60026c8e935592d0f9c308453c813e -
12345678901234567890123456789012345678901234567890123456789012345678901234567890

```

```

limitation: max msg size = ((2 ^ 32) - 1) / 8) bytes
*/

unsigned int indx = 0, pflg = 0, sflg = 0;
unsigned int l, m, n, o, p, q, x, y;
unsigned int h[] = { 0x6a09e667, 0xbb67ae85, 0x3c6ef372, 0xa54ff53a, 0x510e527f,
0x9b05688c, 0x1f83d9ab, 0x5be0cd19 };
unsigned int k[] = { 0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b,
0x599f111f1, 0x923f82a4, 0xab1c5ed5,
0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74,
0x80deb1fe, 0x9bdc06a7, 0xc19bf174,
0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc, 0x2de92c6f,
0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3,
0xd5a79147, 0x06ca6351, 0x14292967,
0x27b70a85, 0x2e1b2138, 0x4d2c6dfe, 0x53380d13, 0x650a7354,
0x766a0abb, 0x81c2c92e, 0x92722c85,
0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819,
0xd6990624, 0xf40e3585, 0x106aa070,
0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3,
0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,
0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90befffa,
0xa4506ceb, 0xbef9a3f7, 0xc67178f2 };
unsigned int a[8], w[64];

while ((pflg == 0) || (sflg == 0))
{
    /* beg: buffer block - pack message */

    y = 0;

    for (x = 0; x < 64; ++x)
    {
        l = (x / 4);
        m = (x % 4);

        if (m == 0)
        {
            w[l] = 0;
        }

        if (indx < ilen)
        {
            n = (24 - (m * 8));
            w[l] |= (inpt[indx] << n);
            ++indx;
            ++y;
        }
    }

    /* end: buffer block - pack message */

    /* beg: buffer block - pad || size */

    if (indx >= ilen)
    {
        if ((pflg == 0) && (y < 64))
        {
            l = (y / 4);
            m = (y % 4);

            n = (24 - (m * 8));
            w[l] |= (0x80 << n);
            ++y;
        }
    }
}

```

```

        pflg = 1;
    }

    if ((sflg == 0) && (y < 57))
    {
        w[14] = 0;
        w[15] = (ilen * 8);
        sflg = 1;
    }
}

for (x = 16; x < 64; ++x)
{
    l = (rrot(w[x - 15], 7) ^ rrot(w[x - 15], 18) ^ (w[x - 15] >> 3));
    m = (rrot(w[x - 2], 17) ^ rrot(w[x - 2], 19) ^ (w[x - 2] >> 10));
    w[x] = (w[x - 16] + l + w[x - 7] + m);
}

/* end: buffer block - pad || size */

/* beg: core hashing loop */

for (x = 0; x < 8; ++x)
{
    a[x] = h[x];
}

for (x = 0; x < 64; ++x)
{
    l = (rrot(a[0], 2) ^ rrot(a[0], 13) ^ rrot(a[0], 22));
    m = ((a[0] & a[1]) ^ (a[0] & a[2]) ^ (a[1] & a[2]));
    n = (l + m);
    o = (rrot(a[4], 6) ^ rrot(a[4], 11) ^ rrot(a[4], 25));
    p = ((a[4] & a[5]) ^ ((~(a[4])) & a[6]));
    q = (a[7] + o + p + k[x] + w[x]);

    a[7] = a[6];
    a[6] = a[5];
    a[5] = a[4];
    a[4] = (a[3] + q);
    a[3] = a[2];
    a[2] = a[1];
    a[1] = a[0];
    a[0] = (n + q);
}

for (x = 0; x < 8; ++x)
{
    h[x] = (h[x] + a[x]);
}

/* end: core hashing loop */
}

for (x = 0; x < 8; ++x)
{
    l = (x * 4);

    for (y = 0; y < 4; ++y)
    {
        m = (24 - (y * 8));
        outp[l + y] = ((h[x] >> m) & 0xff);
    }
}

void hmac(unsigned char *outp, unsigned char *inpt, unsigned int ilen, unsigned char
*skey, unsigned int slen)
{
    /*
    limitation: hash()

```

```

*/

unsigned int x, blen = 64;
unsigned int clen = ((blen * 4) + (ilen + 8));
unsigned char *ipad = malloc(clen * sizeof(unsigned char));
unsigned char *opad = malloc(clen * sizeof(unsigned char));
unsigned char i[HASHSIZE], j[HASHSIZE];

if (slen > blen)
{
    hash(i, skey, slen);
    skey = i;
    slen = HASHSIZE;
}

for (x = 0; x < blen; ++x)
{
    ipad[x] = 0x36;
    opad[x] = 0x5c;

    if (x < slen)
    {
        ipad[x] ^= skey[x];
        opad[x] ^= skey[x];
    }
}

bcopy(inpt, &(ipad[blen]), ilen * sizeof(unsigned char));
hash(j, ipad, blen + ilen);

bcopy(j, &(opad[blen]), HASHSIZE * sizeof(unsigned char));
hash(outp, opad, blen + HASHSIZE);

free(ipad);
free(opad);
}

/* end: hashing */

/* beg: encryption */

void swap(unsigned char *keys, int idxa, int idxb)
{
    unsigned char t;

    t = keys[idxa];
    keys[idxa] = keys[idxb];
    keys[idxb] = t;
}

void cksa(unsigned char *outp, unsigned char *skey, unsigned int slen)
{
    unsigned int x, j = 0;

    for (x = 0; x < 256; ++x)
    {
        outp[x] = x;
    }

    for (x = 0; x < 256; ++x)
    {
        j = ((j + outp[x] + skey[x % slen]) % 256);
        swap(outp, x, j);
    }
}

void encr(unsigned char *outp, unsigned char *inpt, unsigned int ilen, unsigned char
*keys)
{
    /*
    enhancement: ARC4-drop[4096]

```



```

*/
unsigned int x, i = 0, j = 0;
unsigned char k;

for (x = 0; x < 4096; ++x)
{
    i = ((i + 1) % 256);
    j = ((j + keys[i]) % 256);
    swap(keys, i, j);
}

for (x = 0; x < ilen; ++x)
{
    i = ((i + 1) % 256);
    j = ((j + keys[i]) % 256);
    swap(keys, i, j);

    k = keys[(keys[i] + keys[j]) % 256];
    outp[x] = (inpt[x] ^ k);
}
}

/* end: encryption */

#ifndef GMPL
/* beg: EXPERIMENTAL bignum calculator */
void push(char *a, int i, int l)
{
    int x = 0;

    while (i < l)
    {
        if (a[i] > '0')
        {
            break;
        }

        ++i;
    }

    if (i > 0)
    {
        while ((x < l) && (i < l))
        {
            a[x] = a[i];
            ++x; ++i;
        }

        a[x] = '\0';

        if (x == 0)
        {
            a[x] = '0';
            a[x + 1] = '\0';
        }
    }
}

char *badd(char *a, char *b)
{
    int l = strlen(a), m = strlen(b), n = (l + m + 8);
    int s = n, o = 0, p;
    char *c = malloc(n * sizeof(char));

    bzero(c, n * sizeof(char));
    --l; --m; n -= 4;

    while ((l > -1) || (m > -1))

```

```

    {
        p = 0;

        if (l > -1)
        {
            p += (a[l] - '0');
            --l;
        }

        if (m > -1)
        {
            p += (b[m] - '0');
            --m;
        }

        o = (p / 10);
        p = (p % 10);

        c[n] = (p + '0');
        --n;
    }

    if (n > -1)
    {
        c[n] = (o + '0');
        --n;
    }

    push(c, n + 1, s);

    return c;
}

char *bsub(char *a, char *b)
{
    /*
     * limitation: only works for a and b such that a >= b
     */

    int l = strlen(a), m = strlen(b), n = (l + m + 8);
    int s = n, p;
    char *c = malloc(n * sizeof(char));

    a = strdup(a);
    bzero(c, n * sizeof(char));
    --l; --m; n -= 4;

    while (l > -1)
    {
        p = (a[l] - '0');
        --l;

        if (m > -1)
        {
            p = (p - (b[m] - '0'));
            --m;
        }

        if ((p < 0) && (l > -1))
        {
            a[l] -= 1;
            p += 10;
        }

        c[n] = (p + '0');
        --n;
    }

    push(c, n + 1, s);
    free(a);
}

```

```

    return c;
}

char *bmul(char *a, char *b)
{
    int l = strlen(a), m = strlen(b), n = (l + m + 8);
    int s = n, x, y, z;
    char *c = malloc(n * sizeof(char));

    bzero(c, n * sizeof(char));

    for (y = (m - 1); y > -1; --y)
    {
        for (x = (l - 1); x > -1; --x)
        {
            z = (x + y + 4);

            c[z] += ((a[x] - '0') * (b[y] - '0'));
            c[z - 1] += (c[z] / 10);

            c[z] = (c[z] % 10);
        }
    }

    z = ((l - 1) + (m - 1) + 4 + 1);

    for (x = 0; x < z; ++x)
    {
        c[x] += '0';
    }

    push(c, 0, s);

    return c;
}

int comp(char *a, char *b)
{
    int x, l = strlen(a), m = strlen(b);

    if (l < m)
    {
        return -1;
    }

    if (l > m)
    {
        return 1;
    }

    for (x = 0; x < l; ++x)
    {
        if (a[x] < b[x])
        {
            return -1;
        }

        if (a[x] > b[x])
        {
            return 1;
        }
    }

    return 0;
}

char *bdiv(char *a, char *b, int r)
{
    int l = strlen(a), m = strlen(b), n = (l + m + 8);
    int i, j, k, x, y, z;
    char *d = malloc(n * sizeof(char));

```

```

char *c = malloc(n * sizeof(char));
char *s = strdup(a);
char **t = malloc(10 * sizeof(char *));

bzero(c, n * sizeof(char));

/* beg: create a pre-calculated multiplication lookup table */
d[1] = '\0';

for (x = 0; x < 10; ++x)
{
    d[0] = ('0' + x);
    t[x] = bmul(b, d);
}

/* end: create a pre-calculated multiplication lookup table */

/* beg: core division loop (splits the dividend (a) in divisor (b) sized blocks) */
bzero(d, n * sizeof(char));
strncpy(d, a, m);

j = 0;
x = m;
z = 0;

while ((x <= 1) && ((z + m) < (n - 4)))
{
    /* beg: bring down one more dividend (a[...]) digit if this number is too small
*/
    if ((comp(t[1], d) > 0) && (x < 1))
    {
        d[m] = a[x];
        d[m + 1] = '\0';
        ++x;

        if (j != 0)
        {
            c[z] = '0';
            ++z;
        }
    }

    /* end: bring down one more dividend (a[...]) digit if this number is too small
*/

    /* beg: find a multiplier & remainder for this dividend (a[...]) block given
divisor (b) */

    i = 0;
    j = 0;

    for (k = 1; k < 10; ++k)
    {
        if (comp(t[k], d) > 0)
        {
            break;
        }

        i = k;
    }

    c[z] = (i + '0');
    ++z;

    if (i > 0)
    {
        free(s);
        s = bsub(d, t[i]);
    }
}

```

```

    }

    /* end: find a multiplier & remainder for this dividend (a[...]) block given
divisor (b) */

    /* beg: exit loop if no multiplier was found or we ran out of dividend (a) digits
*/

    if ((i < 1) || (x >= 1))
    {
        break;
    }

    /* end: exit loop if no multiplier was found or we ran out of dividend (a) digits
*/

    /* beg: calculate the next possible dividend (a[...]) block (without leading 0s)
*/

    strncpy(d, s, n - 4);
    y = strlen(s);

    while ((d[0] < '1') || (y < m))
    {
        if (d[0] < '1')
        {
            d[0] = '0'; d[1] = '\0';
            y = 0;
        }

        if (x >= 1)
        {
            break;
        }

        d[y] = a[x];
        ++x; ++y;

        if (j != 0)
        {
            c[z] = '0';
            ++z;
        }

        ++j;
    }

    d[y] = '\0';

    free(s);
    s = strdup(d);

    /* end: calculate the next possible dividend (a[...]) block (without leading 0s)
*/
}

free(d);

/* end: core division loop (splits the dividend (a) in divisor (b) sized blocks) */

/* beg: free the lookup table */

for (x = 0; x < 10; ++x)
{
    free(t[x]);
}

free(t);

/* end: free the lookup table */

```

```

    if (r == 1)
    {
        free(c);
        return s;
    }

    free(s);
    return c;
}

char *bpow(char *b, char *e, char *m)
{
    /*
     * Source: Applied Cryptography by Bruce Schneier (ISBN 978-0471117094, page 224)
     */

    int l;
    char *r = malloc(2 * sizeof(char));
    char *u, *z = "0", *t = "2";

    /* r = 1 */

    r[0] = '1';
    r[1] = '\0';

    /* while (e > 0) */

    b = strdup(b);
    e = strdup(e);

    while (comp(e, z) > 0)
    {
        /* if ((e & 1) == 1) */

        l = strlen(e);

        if (((e[l - 1] - '0') & 1) == 1)
        {
            /* r = (r * b) % m */

#ifdef DEBUG
            printf("( (%s * %s) %% %s) = ", r, b, m);
#endif

            u = bmul(r, b);
            free(r);
            r = bdiv(u, m, 1);
            free(u);

#ifdef DEBUG
            printf("%s\n", r);
#endif

        }

        /* e = (e >> 1) */

#ifdef DEBUG
        printf("%s / 2) = ", e);
#endif

        u = bdiv(e, t, 0);
        free(e);
        e = u;

#ifdef DEBUG
        printf("%s\n", e);
#endif

        /* b = (b * b) % m */

```

```

#ifdef DEBUG
    printf("( (%s * %s) %% %s) = ", b, b, m);
#endif

    u = bmul(b, b);
    free(b);
    b = bdiv(u, m, 1);
    free(u);

#ifdef DEBUG
    printf("%s\n", b);
#endif

    }

    free(b);
    free(e);

    return r;
}

/* end: EXPERIMENTAL bignum calculator */

#else

#include <gmp.h>

char *bpow(char *b, char *e, char *m)
{
    char *o;
    mpz_t bb, ee, mm, rr;

    mpz_init_set_str(bb, b, 10);
    mpz_init_set_str(ee, e, 10);
    mpz_init_set_str(mm, m, 10);
    mpz_init(rr);

    mpz_powm(rr, bb, ee, mm);
    o = mpz_get_str(NULL, 10, rr);

    mpz_clear(bb);
    mpz_clear(ee);
    mpz_clear(mm);
    mpz_clear(rr);

    return o;
}

#endif

/* beg: input/output objects */

void term(int mode)
{
    static struct termios termiold;
    static struct termios terminew;

    if (mode == 0)
    {
        /*ioctl(0, TCGETS, &termiold);*/
        tcgetattr(0, &termiold);

        terminew = termiold;
        terminew.c_lflag &= ~ECHO;
        terminew.c_lflag &= ~ICANON;

        /*ioctl(0, TCSETS, &terminew);*/
        tcsetattr(0, TCSANOW, &terminew);
    }

    else if (mode == 1)

```

```

    {
        /*ioctl(0, TCSETS, &termiold);*/
        tcsetattr(0, TCSANOW, &termiold);
    }
}

int csoc(char *dest, unsigned char *data, unsigned int size)
{
    int sendoptn = 1;

    static int sockdesc = -1;
    static struct sockaddr_in sendaddr;

    if (dest != NULL)
    {
        if ((sockdesc = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
        {
            fprintf(stderr, "error: send: socket(...)\n");
            return -1;
        }

        bzero(&sendaddr, sizeof(sendaddr));
        sendaddr.sin_family = AF_INET;
        sendaddr.sin_addr.s_addr = inet_addr(dest);
        sendaddr.sin_port = htons(SOCKPORT);

        if (setsockopt(sockdesc, SOL_SOCKET, SO_BROADCAST, &sendoptn, sizeof(sendoptn)) <
0)
        {
            fprintf(stderr, "error: send: setsockopt(...)\n");
            return -1;
        }
    }

    if (data != NULL)
    {
        sendto(sockdesc, data, size * sizeof(unsigned char), 0, (struct sockaddr
*)&sendaddr, sizeof(sendaddr));
    }

    return 0;
}

int ssoc(int mode)
{
    int recvoptn = 1;

    static int sockdesc = -1;
    static struct sockaddr_in recvaddr;

    if (mode == 0)
    {
        if ((sockdesc = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
        {
            fprintf(stderr, "error: recv: socket(...)\n");
            return -1;
        }

        bzero(&recvaddr, sizeof(recvaddr));
        recvaddr.sin_family = AF_INET;
        recvaddr.sin_addr.s_addr = htonl(INADDR_ANY);
        recvaddr.sin_port = htons(SOCKPORT);

        if (setsockopt(sockdesc, SOL_SOCKET, SO_REUSEADDR, &recvoptn, sizeof(recvoptn)) <
0)
        {
            fprintf(stderr, "error: recv: setsockopt(...)\n");
            return -1;
        }

        if (bind(sockdesc, (struct sockaddr *)&recvaddr, sizeof(recvaddr)) < 0)

```



```

        {
            fprintf(stderr, "error: recv: bind(...)\n");
            return -1;
        }
    }

    else if (mode == 1)
    {
        return sockdesc;
    }

    return 0;
}

void clrns(int size)
{
    int x;

    for (x = 0; x < size; ++x)
    {
        printf("\b \b");
    }

    fflush(stdout);
}

char *user(int letr, char *pref)
{
    static unsigned int sendindx = 0, prefleng = 0, showchar = 0;
    static char sendbuff[BUFFSIZE];

    if (letr == -1)
    {
        bzero(sendbuff, BUFFSIZE * sizeof(char));
        strncpy(sendbuff, pref, BUFFSIZE - 4);
        strncat(sendbuff, ":", BUFFSIZE - 4);

        sendindx = strlen(sendbuff);
        prefleng = sendindx;

        return NULL;
    }

    else if (letr == -2)
    {
        clrns(sendindx);

        return NULL;
    }

    else if (letr == -3)
    {
        printf("%s", sendbuff);
        fflush(stdout);

        return NULL;
    }

    else if (letr == -4)
    {
        showchar = 1;

        return NULL;
    }

    if ((letr == 3) || (letr == 4))
    {
        term(1);
        exit(0);
    }
}

```

```

else if ((letr == 8) || (letr == 127))
{
    if (sendindx > prefleng)
    {
        if (showchar == 1)
        {
            clrscr(1);
        }

        --sendindx;
        sendbuff[sendindx] = '\0';
    }
}

else if ((letr == 10) || (letr == 13))
{
    return sendbuff;
}

else
{
    if ((letr < 32) || (126 < letr))
    {
        if (letr != 9)
        {
            letr = ' ';
        }
    }

    if (sendindx < (BUFSIZE - 4))
    {
        if (showchar == 1)
        {
            printf("%c", letr);
            fflush(stdout);
        }

        sendbuff[sendindx] = letr;
        ++sendindx;
    }
}

return NULL;
}

/* end: input/output objects */
/* beg: crypto interface */

void rnds(unsigned char *outp, unsigned int olen, int mode)
{
    int n;
    unsigned int x;

    for (x = 0; x < olen; ++x)
    {
        if (mode == 0)
        {
            while (1)
            {
                n = (rand() & 0xf);

                if ((0 <= n) && (n <= 9))
                {
                    break;
                }
            }

            outp[x] = (n + '0');
        }
    }
}

```

```

        else
        {
            outp[x] = (rand() & 0xff);
        }
    }
}

void psec(char *outp)
{
    int x;
    unsigned int nsec = time(NULL);

    snprintf(outp, BUFFSIZE - 4, "%d", nsec);

    while (strlen(outp) < SPADSIZE)
    {
        for (x = (BUFFSIZE - 4); x > 0; --x)
        {
            outp[x] = outp[x - 1];
        }

        outp[0] = '0';
    }
}

void ipvf(char *outp, char *iadr)
{
    int x;

    snprintf(outp, BUFFSIZE - 4, "%s", iadr);

    while (strlen(outp) < SPADSIZE)
    {
        for (x = (BUFFSIZE - 4); x > 0; --x)
        {
            outp[x] = outp[x - 1];
        }

        outp[0] = 'x';
    }
}

int next(char **list, int size)
{
    int x, i = -1, o = (SPADSIZE + SPADSIZE);

    for (x = 0; x < size; ++x)
    {
        if ((list[x] != NULL) && (list[x][o] == '-'))
        {
            i = x;
            break;
        }
    }

    return i;
}

int cryp(char *pass, unsigned char *recv, unsigned int rlen, struct sockaddr_in
*sendaddr, char *mesg)
{
    /*
    self-verify: skey = hash(prikey || init)
    self-verify: ciph = encr("verify", skey)
    self-verify: sign = hmac("verify", skey)
    self-verify: send = (mode || init || sign || ciph)

    key-exchange: skey = hash(pass || addr || time || init)
    key-exchange: ciph = encr(pubkey, skey)
    key-exchange: sign = hmac(pubkey, skey)
    key-exchange: send = (mode || time || init || sign || ciph)

```

```

message-exchange: skey = hash(kexg || addr || time || init)
message-exchange: ciph = encr(mesg, skey)
message-exchange: sign = hmac(mesg, skey)
message-exchange: send = (mode || time || init || sign || ciph)

Source: http://www.ietf.org/rfc/rfc3526.txt
*/

int x;
unsigned int tlen;
char seca[BUFFSIZE], secb[BUFFSIZE], sadr[BUFFSIZE];
char radr[BUFFSIZE], rsec[BUFFSIZE], rkey[BUFFSIZE];
char *vrfy = "verify", *temp;
unsigned char ikey[BUFFSIZE], hkey[HASHSIZE], ckey[SKEYSIZE];
unsigned char cpha[BUFFSIZE], cphb[BUFFSIZE], siga[BUFFSIZE], sigb[BUFFSIZE];

unsigned int slen;
unsigned char send[BUFFSIZE];

static int stat = 0;
static int pdes[4];
static char base[BUFFSIZE], pexp[BUFFSIZE], pmod[BUFFSIZE];
static char *pkey = NULL, *addr = NULL;

static unsigned int elen = 0;
static unsigned char ekey[BUFFSIZE];

static int llen = 0;
static char **lkey = NULL;

if (stat == 0)
{
    bzero(base, BUFFSIZE * sizeof(char));
    strncpy(base, "2", BUFFSIZE - 4);

    bzero(pexp, BUFFSIZE * sizeof(char));
    bzero(pmod, BUFFSIZE * sizeof(char));

#ifdef  GPL

    rnds((unsigned char *)pexp, 128, 0);
    push(pexp, 0, 128);

    strncat(pmod,
"3231700607131100730033891392642382824881794124114023911284200975140074170663435422261968
94173635693471179017379097041917546058732091950288537589861856221532121754125149017745202
7023579607823624888424618947758764110592864609941172324542662252219323054091903",
BUFFSIZE - 4);
    strncat(pmod,
"7680524235519125679715870117001058055877651038861847280257976054903569732561526167081339
36179954133647655916036831789672907317838458968063967190097720219416864722587103141133642
9319536193471636533209717077448227988588565369208645296636077250268955505928362",
BUFFSIZE - 4);
    strncat(pmod,
"7511211740969729980684105543595848665832916421362182310789909994486524682624169720359118
52507045361090559", BUFFSIZE - 4);

#else

    rnds((unsigned char *)pexp, 512, 0);

    strncat(pmod,
"1090748135619415929450294929359784500348155124953172211774101106966150168922785639028532
47384883681776971216416907643296922469875267467766273999426578543723359615704597092233804
0698100507861033047312331823982435279475700199860971612732540528796554502867919",
BUFFSIZE - 4);
    strncat(pmod,
"7467769837593914759871425213158787195775191488118308799194269399584870875409657164191674
67499326156226529675209172277001377591248147563782880558861083327174154014975134893125116

```

```

0157763188902959606980116141577212825275394688165193193333375031147771923604122",
BUFSIZE - 4);
    strcat(pmod,
"8172101895583437761548046847925274886732036238535559660179512280675621771357981987063432
15619078132551537039507952712326524048949838694921744816523038034988813662105086472636683
7651413103110233683748899977574404673365182723939535354034841487285463971929469",
BUFSIZE - 4);
    strcat(pmod,
"4323450186884189822544540647226987292160693184734654941906936646576130260972193280317171
69641897155395416144619175909371952495111670557736207348131929604120128351615426904438925
7727700289684119460283480452306204130024913879981135908026983868205969318167819",
BUFSIZE - 4);
    strcat(pmod,
"6808509986496944169079527129049624049377757896989172073563552274550661838158476691355305
49755439819480321732925869069136146085326382334628745456398071603058051634209386708703306
5459031996085238245137296251366591282211009677354505199524042481982628138310973",
BUFSIZE - 4);
    strcat(pmod,
"7426165038001727791697532413484657468130733701738083035368062321633694947130619168643824
93056864133802310460964509535940893755402850372924709293951140283055474525849620743094381
5182543790297601289174935519867842060372203490031136489304649576140433393868614",
BUFSIZE - 4);
    strcat(pmod,
"0037848030916292543273684533640032637639100774502371542479302473698388692892420946478947
73380038778274141778648477019010886787977899163321862864053398261932246615488301145229189
0252336487236086654396093853898628805813177559162076363154436494477507871294119",
BUFSIZE - 4);
    strcat(pmod,
"8416378677017221666098312018454840780705180413368698083984546255869212013081856388880826
99408686536045192649569198110353659943111802300636106509865023943661829436426563007917282
0508944293888417488853982907077430529736053592775157496197308237732158947551217",
BUFSIZE - 4);
    strcat(pmod,
"6146788786532770711557380426451920634921585019519536481338752681174247413154980213024650
63412070203357977067807054069452754388062659785162097067957025792440753804902317410308626
1496878330620786968786810842363997198320907762475808049998827559139278726762718",
BUFSIZE - 4);
    strcat(pmod,
"2442892809646874228263172435642368588260139161962836121481966092745325488641054238839295
138992979335446110090325230955276870524611359124918392740353154294858383359", BUFSIZE -
4);

#endif

    pkey = bpow(base, pexp, pmod);
    pdes[0] = 0; pdes[1] = 0;

    stat = 1;
}

if (pass == NULL)
{
    return 0;
}

bzero(seca, BUFSIZE * sizeof(char));
psec(seca);

if (recv != NULL)
{
    bzero(sadr, BUFSIZE * sizeof(char));

    if (sendaddr != NULL)
    {
        ipvf(sadr, inet_ntoa(sendaddr->sin_addr));
    }

    /* beg: self-verify recv */

    slen = (1 + HASHSIZE + HASHSIZE + strlen(vrfy));

```

```

    if ((slen <= rlen) && (rlen <= (BUFSIZE - 4)) && (recv[0] == 'v'))
    {
        slen = 1;
        bcopy(&(recv[slen]), ikey, HASHSIZE * sizeof(unsigned char)); slen +=
HASHSIZE;
        bcopy(&(recv[slen]), siga, HASHSIZE * sizeof(unsigned char)); slen +=
HASHSIZE;
        bcopy(&(recv[slen]), cpha, strlen(vrfy) * sizeof(unsigned char)); slen +=
strlen(vrfy);

        bzero(send, BUFSIZE * sizeof(unsigned char)); slen = 0;
        bcopy(pkey, &(send[slen]), strlen(pkey) * sizeof(char)); slen +=
strlen(pkey);
        bcopy(ikey, &(send[slen]), HASHSIZE * sizeof(unsigned char)); slen +=
HASHSIZE;
        hash(hkey, send, slen);

        cksa(ckey, hkey, HASHSIZE);
        bzero(cphb, BUFSIZE * sizeof(unsigned char));
        encr(cphb, cpha, strlen(vrfy), ckey);
        hmac(sigb, cphb, strlen(vrfy), hkey, HASHSIZE);

        if (strncmp((char *)cphb, vrfy, strlen(vrfy)) == 0)
        {
            if (bcmp(siga, sigb, HASHSIZE * sizeof(unsigned char)) == 0)
            {
                if (addr == NULL)
                {
                    addr = strdup(sadr);

                    user(-2, NULL);
                    fprintf(stderr, "self-addr [%s]\n", addr);
                    fflush(stderr);
                    user(-3, NULL);
                }
            }
        }

        return -1;
    }

    /* end: self-verify recv */

    /* beg: key-exchange recv */

    slen = (1 + strlen(seca) + HASHSIZE + HASHSIZE + 1);

    if ((slen <= rlen) && (rlen <= (BUFSIZE - 4)) && ((recv[0] == 'k') || (recv[0]
== 'l'))))
    {
        tlen = (rlen - (slen - 1));
        bzero(secb, BUFSIZE * sizeof(char));

        slen = 1;
        bcopy(&(recv[slen]), secb, strlen(seca) * sizeof(unsigned char)); slen +=
strlen(seca);
        bcopy(&(recv[slen]), ikey, HASHSIZE * sizeof(unsigned char)); slen +=
HASHSIZE;
        bcopy(&(recv[slen]), siga, HASHSIZE * sizeof(unsigned char)); slen +=
HASHSIZE;
        bcopy(&(recv[slen]), cpha, tlen * sizeof(unsigned char)); slen += tlen;

        if ((addr != NULL) && (strcmp(sadr, addr) != 0))
        {
            for (x = 0; x < llen; ++x)
            {
                bzero(radr, BUFSIZE * sizeof(char));
                bzero(rsec, BUFSIZE * sizeof(char));

                slen = 0;
            }
        }
    }

```

```

        bcopy(&lkey[x][slen], radr, strlen(sadr) * sizeof(char)); slen +=
strlen(sadr);
        bcopy(&lkey[x][slen], rsec, strlen(seca) * sizeof(char)); slen +=
strlen(seca);

        if (strcmp(sadr, radr) == 0)
        {
            if (recv[0] == 'l')
            {
                x = -1;
            }

            if (atoi(secb) <= atoi(rsec))
            {
                x = -1;
            }

            break;
        }
    }

    if (x > -1)
    {
        bzero(send, BUFFSIZE * sizeof(unsigned char)); slen = 0;
        bcopy(pass, &(send[slen]), strlen(pass) * sizeof(char)); slen +=
strlen(pass);
        bcopy(sadr, &(send[slen]), strlen(sadr) * sizeof(char)); slen +=
strlen(sadr);
        bcopy(secb, &(send[slen]), strlen(seca) * sizeof(char)); slen +=
strlen(seca);
        bcopy(ikkey, &(send[slen]), HASHSIZE * sizeof(unsigned char)); slen +=
HASHSIZE;
        hash(hkey, send, slen);

        tlen = (rlen - (1 + strlen(seca) + HASHSIZE + HASHSIZE));
        cksa(ckey, hkey, HASHSIZE);
        bzero(cphb, BUFFSIZE * sizeof(unsigned char));
        encr(cphb, cpha, tlen, ckey);
        hmac(sigb, cphb, tlen, hkey, HASHSIZE);

        if (bcmp(siga, sigb, HASHSIZE * sizeof(unsigned char)) == 0)
        {
            bzero(send, BUFFSIZE * sizeof(unsigned char));
            strncat((char *)send, sadr, BUFFSIZE - 4);
            strncat((char *)send, secb, BUFFSIZE - 4);
            strncat((char *)send, "-", BUFFSIZE - 4);
            strncat((char *)send, (char *)cphb, BUFFSIZE - 4);

            if (x < llen)
            {
                free(lkey[x]);
            }

            else
            {
                x = llen; ++llen;
                lkey = realloc(lkey, (llen + 1) * sizeof(char *)); lkey[llen]
= NULL;
            }

            lkey[x] = strdup((char *)send);
            csoc(NULL, ekey, elen);
        }
    }

    return -1;
}

/* end: key-exchange recv */

```

```

/* beg: message-exchange recv */

slen = (1 + strlen(seca) + HASHSIZE + HASHSIZE + 1);

if ((slen <= rlen) && (rlen <= (BUFSIZE - 4)) && (recv[0] == 'm'))
{
    tlen = (rlen - (1 + strlen(seca) + HASHSIZE + HASHSIZE));
    bzero(secb, BUFSIZE * sizeof(char));

    slen = 1;
    bcopy(&(recv[slen]), secb, strlen(seca) * sizeof(unsigned char)); slen +=
strlen(seca);
    bcopy(&(recv[slen]), ikey, HASHSIZE * sizeof(unsigned char)); slen +=
HASHSIZE;
    bcopy(&(recv[slen]), siga, HASHSIZE * sizeof(unsigned char)); slen +=
HASHSIZE;
    bcopy(&(recv[slen]), cpha, tlen * sizeof(unsigned char)); slen += tlen;

    if ((addr != NULL) && (strcmp(sadr, addr) != 0))
    {
        for (x = 0; x < llen; ++x)
        {
            tlen = (strlen(lkey[x]) - (SPADSIZE + SPADSIZE));
            bzero(radr, BUFSIZE * sizeof(char));
            bzero(rsec, BUFSIZE * sizeof(char));
            bzero(rkey, BUFSIZE * sizeof(char));

            slen = 0;
            bcopy(&(lkey[x][slen]), radr, strlen(sadr) * sizeof(char)); slen +=
strlen(sadr);
            bcopy(&(lkey[x][slen]), rsec, strlen(seca) * sizeof(char)); slen +=
strlen(seca);
            bcopy(&(lkey[x][slen]), rkey, tlen * sizeof(char)); slen += tlen;

            if (strcmp(sadr, radr) == 0)
            {
                if (atoi(secb) > atoi(rsec))
                {
                    bzero(send, BUFSIZE * sizeof(unsigned char)); slen = 0;
                    bcopy(rkey, &(send[slen]), strlen(rkey) * sizeof(char)); slen
+= strlen(rkey);
                    bcopy(sadr, &(send[slen]), strlen(sadr) * sizeof(char)); slen
+= strlen(sadr);
                    bcopy(secb, &(send[slen]), strlen(seca) * sizeof(char)); slen
+= strlen(seca);
                    bcopy(ikey, &(send[slen]), HASHSIZE * sizeof(unsigned char));
                    slen += HASHSIZE;
                    hash(hkey, send, slen);

                    tlen = (rlen - (1 + strlen(seca) + HASHSIZE + HASHSIZE));
                    cksa(ckey, hkey, HASHSIZE);
                    bzero(cphb, BUFSIZE * sizeof(unsigned char));
                    encr(cphb, cpha, tlen, ckey);
                    hmac(sigb, cphb, tlen, hkey, HASHSIZE);

                    if (bcmp(siga, sigb, HASHSIZE * sizeof(unsigned char)) == 0)
                    {
                        temp = strstr((char *)cphb, ": ");

                        if ((temp != NULL) && (temp[2] != '\0'))
                        {
                            user(-2, NULL);
                            printf("%s\n", cphb);
                            user(-3, NULL);
                        }

                        slen = strlen(sadr);
                        bcopy(secb, &(lkey[x][slen]), strlen(seca) *
sizeof(char)); slen += strlen(seca);
                    }
                }
            }
        }
    }
}

```



```

        }
    }
}

return -1;
}

/* end: message-exchange recv */

/* beg: key-exchange calc-recv */

slen = (1 + 1);
tlen = ((BUFSIZE - 4) - (SPADSIZE + SPADSIZE));

if ((slen <= rlen) && (rlen <= tlen) && (recv[0] == 'c') && (sendaddr == NULL))
{
    x = pdes[2];

    tlen = (rlen - 1);
    bzero(send, BUFSIZE * sizeof(unsigned char)); slen = 0;
    bcopy(lkey[x], &(send[slen]), (SPADSIZE + SPADSIZE) * sizeof(unsigned char));
slen += (SPADSIZE + SPADSIZE);
    bcopy(&(recv[1]), &(send[slen]), tlen * sizeof(unsigned char)); slen += tlen;

    free(lkey[x]);
    lkey[x] = strdup((char *)send);

    bzero(radr, BUFSIZE * sizeof(char));
    strcat(radr, &(lkey[x][0]), SPADSIZE);

    bzero(rkey, BUFSIZE * sizeof(char));
    strcat(rkey, &(lkey[x][SPADSIZE + SPADSIZE]), SPADSIZE);

    user(-2, NULL);
    fprintf(stderr, "key-exchg [%s]-[%d/%d]=[%s...]\n", radr, x + 1, llen, rkey);
    fflush(stderr);
    user(-3, NULL);

    close(pdes[0]);
    pdes[0] = 0; pdes[1] = 0;
}

/* end: key-exchange calc-recv */
}

/* beg: self-verify send */

if ((addr == NULL) && (stat > 0))
{
    rnds(ikey, HASHSIZE, 1);

    bzero(send, BUFSIZE * sizeof(unsigned char)); slen = 0;
    bcopy(pkey, &(send[slen]), strlen(pkey) * sizeof(char)); slen += strlen(pkey);
    bcopy(ikey, &(send[slen]), HASHSIZE * sizeof(unsigned char)); slen += HASHSIZE;
    hash(hkey, send, slen);

    tlen = strlen(vrfy);
    cksa(ckey, hkey, HASHSIZE);
    encr(cpha, (unsigned char *)vrfy, tlen, ckey);
    hmac(siga, (unsigned char *)vrfy, tlen, hkey, HASHSIZE);

    bzero(send, BUFSIZE * sizeof(unsigned char)); slen = 0;
    bcopy("v", &(send[slen]), 1 * sizeof(unsigned char)); slen += 1;
    bcopy(ikey, &(send[slen]), HASHSIZE * sizeof(unsigned char)); slen += HASHSIZE;
    bcopy(siga, &(send[slen]), HASHSIZE * sizeof(unsigned char)); slen += HASHSIZE;
    bcopy(cpha, &(send[slen]), tlen * sizeof(unsigned char)); slen += tlen;

    csoc(NULL, send, slen);
}

/* end: self-verify send */

```

```

/* beg: key-exchange send */

if ((addr != NULL) && (stat == 1))
{
    rnds(ikey, HASHSIZE, 1);

    bzero(send, BUFFSIZE * sizeof(unsigned char)); slen = 0;
    bcopy(pass, &(send[slen]), strlen(pass) * sizeof(char)); slen += strlen(pass);
    bcopy(addr, &(send[slen]), strlen(addr) * sizeof(char)); slen += strlen(addr);
    bcopy(seca, &(send[slen]), strlen(seca) * sizeof(char)); slen += strlen(seca);
    bcopy(ikey, &(send[slen]), HASHSIZE * sizeof(unsigned char)); slen += HASHSIZE;
    hash(hkey, send, slen);

    tlen = strlen(pkey);
    cksa(ckey, hkey, HASHSIZE);
    encr(cpha, (unsigned char *)pkey, tlen, ckey);
    hmac(siga, (unsigned char *)pkey, tlen, hkey, HASHSIZE);

    bzero(ekey, BUFFSIZE * sizeof(unsigned char)); elen = 0;
    bcopy("k", &(ekey[elen]), 1 * sizeof(unsigned char)); elen += 1;
    bcopy(seca, &(ekey[elen]), strlen(seca) * sizeof(char)); elen += strlen(seca);
    bcopy(ikey, &(ekey[elen]), HASHSIZE * sizeof(unsigned char)); elen += HASHSIZE;
    bcopy(siga, &(ekey[elen]), HASHSIZE * sizeof(unsigned char)); elen += HASHSIZE;
    bcopy(cpha, &(ekey[elen]), tlen * sizeof(char)); elen += tlen;

    csoc(NULL, ekey, elen);
    ekey[0] = 'l';

    stat = 2;
}

/* end: key-exchange send */

/* beg: message-exchange send */

if ((addr != NULL) && (mesg != NULL))
{
    for (x = 0; x < llen; ++x)
    {
        rnds(ikey, HASHSIZE, 1);

        tlen = (strlen(lkey[x]) - (SPADSIZE + SPADSIZE));
        bzero(rkey, BUFFSIZE * sizeof(char));
        strncpy(rkey, &(lkey[x][SPADSIZE + SPADSIZE]), tlen);

        if (rkey[0] != '-')
        {
            bzero(send, BUFFSIZE * sizeof(unsigned char)); slen = 0;
            bcopy(rkey, &(send[slen]), strlen(rkey) * sizeof(char)); slen +=
strlen(rkey);
            bcopy(addr, &(send[slen]), strlen(addr) * sizeof(char)); slen +=
strlen(addr);
            bcopy(seca, &(send[slen]), strlen(seca) * sizeof(char)); slen +=
strlen(seca);
            bcopy(ikey, &(send[slen]), HASHSIZE * sizeof(unsigned char)); slen +=
HASHSIZE;
            hash(hkey, send, slen);

            tlen = strlen(mesg);
            cksa(ckey, hkey, HASHSIZE);
            encr(cpha, (unsigned char *)mesg, tlen, ckey);
            hmac(siga, (unsigned char *)mesg, tlen, hkey, HASHSIZE);

            bzero(send, BUFFSIZE * sizeof(unsigned char)); slen = 0;
            bcopy("m", &(send[slen]), 1 * sizeof(unsigned char)); slen += 1;
            bcopy(seca, &(send[slen]), strlen(seca) * sizeof(char)); slen +=
strlen(seca);
            bcopy(ikey, &(send[slen]), HASHSIZE * sizeof(unsigned char)); slen +=
HASHSIZE;

```

```

        bcopy(siga, &(send[slen]), HASHSIZE * sizeof(unsigned char)); slen +=
HASHSIZE;
        bcopy(cpha, &(send[slen]), tlen * sizeof(unsigned char)); slen += tlen;
        csoc(NULL, send, slen);
    }
}
/* end: message-exchange send */
/* beg: key-exchange calc-send */
if ((recv == NULL) && (mesg == NULL))
{
    if ((pdes[0] == 0) && (pdes[1] == 0))
    {
        x = next(lkey, llen);

        if (x > -1)
        {
            pipe(pdes);
            pdes[2] = x;

            if (fork() == 0)
            {
                close(pdes[0]);
                tlen = (SPADSIZE + SPADSIZE + 1);
                temp = bpow(&(lkey[x][tlen]), pexp, pmod);

                bzero(send, BUFFSIZE * sizeof(unsigned char)); slen = 0;
                bcopy("c", &(send[slen]), 1 * sizeof(unsigned char)); slen += 1;
                bcopy(temp, &(send[slen]), strlen(temp) * sizeof(unsigned char));
slen += strlen(temp);

                write(pdes[1], send, slen * sizeof(unsigned char));
                free(temp);

                close(pdes[1]);
                exit(0);
            }

            close(pdes[1]);

            return pdes[0];
        }
    }
    else
    {
        return pdes[0];
    }
}
/* end: key-exchange calc-send */

return 0;
}
/* end: crypto interface */

int maxn(int a, int b)
{
    if (a > b)
    {
        return a;
    }

    return b;
}

```

```

int main(int argc, char **argv)
{
    int recvdesc, pipedesc, maxndesc;
    char readchar;
    char *userbuff, *password = NULL;
    unsigned int recvsize;
    unsigned char recvbuff[BUFSIZE];
    socklen_t respsize;
    struct sockaddr_in respaddr;
    fd_set filedesc;

    if (argc < 3)
    {
        printf("Usage: %s <broadcast> <nickname>\n", argv[0]);
        return 1;
    }

    srand(time(NULL));

    term(0);
    csoc(argv[1], NULL, 0);
    ssoc(0);

    cryp(NULL, NULL, 0, NULL, NULL);
    user(-1, "password");
    user(-3, NULL);

    /* beg: infinite loop */

    while (1)
    {
        recvdesc = ssoc(1);
        pipedesc = cryp(password, NULL, 0, NULL, NULL);

        FD_ZERO(&filedesc);
        FD_SET(0, &filedesc);
        FD_SET(recvdesc, &filedesc);

        if (pipedesc > 0)
        {
            FD_SET(pipedesc, &filedesc);
        }

        maxndesc = maxn(recvdesc, pipedesc);
        select(maxndesc + 1, &filedesc, NULL, NULL, NULL);

        if (FD_ISSET(0, &filedesc))
        {
            readchar = getchar();
            userbuff = user(readchar, argv[2]);

            if (userbuff != NULL)
            {
                if (password == NULL)
                {
                    password = userbuff;

                    if (strlen(password) > strlen("password: "))
                    {
                        password = &(password[strlen("password: ")]);
                    }

                    password = strdup(password);
                    user(-4, NULL);
                }

                else
                {
                    cryp(password, NULL, 0, NULL, userbuff);
                    user(-2, NULL);
                    printf("%s", userbuff);
                }
            }
        }
    }
}

```

```

        }

        printf("\n");
        user(-1, argv[2]);
        user(-3, NULL);
    }

    if (password != NULL)
    {
        if (FD_ISSET(recvdesc, &filedesc))
        {
            respsize = sizeof(respaddr);

            bzero(recvbuff, BUFFSIZE * sizeof(unsigned char));
            recvsize = recvfrom(recvdesc, recvbuff, (BUFFSIZE - 4) * sizeof(unsigned
char), 0, (struct sockaddr *)&respaddr, &respsize);

            cryp(password, recvbuff, recvsize, &respaddr, NULL);
        }
    }

    if (pipedesc > 0)
    {
        if (FD_ISSET(pipedesc, &filedesc))
        {
            bzero(recvbuff, BUFFSIZE * sizeof(unsigned char));
            recvsize = read(pipedesc, recvbuff, (BUFFSIZE - 4) * sizeof(unsigned
char));
            cryp(password, recvbuff, recvsize, NULL, NULL);
        }
    }

    /* end: infinite loop */

    return 0;
}

```