

Class: CSG254 Network Security  
Team: Enigma (team 2)  
Kevin Kingsbury  
Tejas Parikh  
Tony Ryan  
Shenghan Zhang  
Assignment: PS3 – Secure IM system

## Overview

Our system uses a server to store the passwords, and authenticate users who 'log on' to the system. All communication between users A and B will be direct, and not go through the server. Users will register with the server when they are 'on-line'. If a user does not communicate with the server for too long, they are timed out and must re-connect. When a user wishes to communicate with another user, it asks the server for their current address, and uses a variation of Ottway-Reese to establish and authenticate a communication channel with the other user. Once the communication channel is established, the server is no longer involved in the communication between the 2 users, and won't even be able to decrypt the communication between them.

## Terminology

Kas - Symmetric encryption key between 'A' and 'S'  
Kab - Symmetric encryption key between 'A' and 'B'  
H(x) - the value x is run through a hash function  
SHA1(x) - the value x is run through the SHA1 hash function  
a | b - the values of a and b are concatenated into a bit-stream  
E{x} - The value x is encrypted with key E  
[x]A - The value x has been signed by A  
A->S - A is sending a message to S

## Password Storage

The server will use a flat file to store usernames, salts and password hashes, using the SHA-1 hashing algorithm. We use two different files which incorporate a mechanism similar to the way a UNIX system uses /etc/passwd and /etc/shadow. We have a file of user details, we call passwd.txt, that is readable by anyone that gives information of the user (and could be expanded to contain more user profile details). The shadow.txt file contains all the hashed passwords for the users of the application and is only readable by the application. This way in the event of a system breach unless the attacker was able to gain the same credentials as the application, they cannot access the password hashes and perform offline dictionary attacks on them.

A entry in the passwd.txt for each user will be in this form:

```
userid:username:passwd
```

The username is variable length up to 32 8-bit chars, which is the maximum length. The allowable characters for usernames will come from the set: A-Za-z0-9\_ (62 possible values). For the password field in this file we show a \* to indicate a password is set.

In the shadow.txt file the format is:

```
userid:SHA1(pw|salt)
```

The password to be at least 8 characters up to a maximum of 128 characters. The allowable characters will be from the set: A-Za-z0-9!@#\$\$%^&\*(){ }[-+\_=<>.,;:'"/?~`|\ (94 possible values). The salt is always 32 bits (4 bytes) and will be the userid.

The userid will be a random number (tested for uniqueness against existing user ID's), to make it harder to predict what values it could take. SHA1 always produces a 160 bit (20 bytes) digest, so each user entry will be exactly  $32+4+20 = 56$  bytes long.

Usernames and passwords will be added off-line to prevent online attacks on account creation.

### ***online vs offline attack resistance***

The passwords will be resistant to both online and off-line attacks.

### **online password attacks**

The system will be resistant to on-line attacks through 2 main mechanisms, assuming an attacker knows at least the user name (which is sent in the clear, so it is easily known). The first is the mathematical complexity of the passwords. Since the minimum length of the passwords is 8 characters, and there are 84 possible values for each character, then even a minimum length password has  $94^8$  (approx  $2^{53}$ ) possible values, which is extremely unlikely to be guessed, and too large to brute force guess all possible combinations given the second mechanism. Additionally, since the attacker doesn't know the actual length of the password, they would need to try all possible values up to 128 characters. The second mechanism is the server will prevent log-in attempts for a period of 5 minutes if a user fails to log in 5 times within 1 minute. This will slow down an online brute-force attack against all combinations enough to make it impossible. Even if they try 4 guesses per minute (avoiding the 5 minute time delay), it would still take  $2^{51}$  minutes (4 billion years) to guess all entries.

### **offline password attacks**

For offline attacks, the user file will be protected to prevent non admin users from reading it. however, it is still possible for an attacker to get the file. In the case that an attacker gets the file the passwords are further protected by adding salt to the hash. The additional 32 bits of salt increases the complexity of the hash input, and makes rainbow table attacks more difficult. Additionally, because the user ID will probably be different on different systems, even if the user has the same password on these systems, the hash will be different, because the salt will be different. This means that if a single server is compromised, others are still secure.

# Communication between a user and the server

Communication between a user and the server will be encrypted using a symmetric key. The encryption algorithm will be AES with a 128 bit key. The exception to this is the authentication and establishment of the session key. The first communication between the user and the server will be to establish the session key.

## *User-Server authentication and Session Key establishment*

For User-server mutual authentication, we combined two protocols from the book (Network Security): protocol 24-2 on page 596, which is the NetWare V3 authentication, and protocol 12-2 on page 296, which is the basic EKE protocol. Algorithm 24-2 is used to establish the weak secret, that is then used in the Diffie-Hellman exchange in protocol 12-2. The reason for combining these 2 is 24-2 by itself does not guarantee perfect forward secrecy. If an attacker records an exchange between a user and the server, then later finds out  $X$  (which is the  $\text{hash}(\text{password} \mid \text{salt})$ , then they key  $Z$  can be recalculated because the random number  $R$  is sent in the clear. However, using this session key  $Z$  as the weak secret ( $W$  in protocol 12-2), allows the Diffie-Hellman exchange to be encrypted with that secret.

Server  $S$  knows  $X = H(\text{password} \mid \text{salt})$ , and knows the salt value used (user ID)

| Message                                      | Notes  |
|--|--|
| 1 A->S: 'alice'                              |  |
| 2 S->A: R,salt                               | R = random number<br>A computes $X' = \text{hash}(\text{password}, \text{salt})$<br>$Z' = H(X', R, k)$ for the key<br>k = constant string  |
| 3 A -> S: $Z'\{g^a \text{ mod } p\}$         | 'a' is a random number generated on client A.<br>S computes $Z = H(X,R,k)$ for the key<br>k = constant string<br>S verifies SHA1-MAC before calculating<br>$g^b \text{ mod } p$ and encrypting |
| 4 S -> A: $Z\{g^b \text{ mod } p\}$          | 'b' and C1 are random numbers generated on the server.<br><br>both A and S calculate the key<br><b><math>K = f(g^{ab} \text{ mod } p, R)</math></b>  |
| 5 A -> S: $K\{C1, C2, \text{SHA1}(C1, C2)\}$ | C2 is a random number generated by the client  |
| 6 S -> A: $K\{C2, \text{SHA1}(C2)\}$         |  |

At this point Both the server and the user A have authenticated each other. The server will list user 'A' as being 'on-line' with the current IP address for 'A'. user 'A' must periodically send a message to the server to indicate that they are still online. After a period of time without a message, the server will time out the entry for 'A' and assume they are no longer on line.

## DoS resistance

This algorithm is DoS resistant, because the server does not do any calculations (other than generating a random number) until the user has first done some calculations and encryption for message 3. Once the server gets the response it does the minimal calculation to get the weak secret  $Z$ , and verifies that the user is 'legitimate' by checking the values of  $Y$  and  $Y'$ . Only then does the server continue with the subsequent calculations. This minimizes the more expensive calculations ( $g^b \bmod p$ , and encrypting then  $g^{ab} \bmod p$ , and encrypting) until the user has at least proven they have done some calculations.

## Perfect Forward Secrecy

This algorithm provides perfect forward secrecy. Because it uses a variation of Diffie-Hellman, the session key used, and the random values used to establish the session key are not remembered by either the client or the server. One of the key features of the Diffie-Hellman protocol is the numbers exchanged ( $g^a \bmod p$ , and  $g^b \bmod p$ ) look like random numbers, so a brute force attack on the encryption key (the key  $Z$  in the protocol above) does not provide the attacker with any clues as to when they have the 'correct' key. Even if the password file is stolen and later cracked-off-line, the session key cannot be recovered in an efficient manner, because it would require the attacker to solve the discrete log problem. Solving the discrete log is considered too hard to be done efficiently.

## Data Integrity

Any encrypted message that is sent between the user and server (in either direction) will also have an SHA1 hash included in the encrypted data. This will prevent an attacker from modifying any of the messages without being detected. In particular, message 4, where the server is sending ' $g^b \bmod p$ ,  $C1$ ' encrypted, the user has no idea what these numbers should be since they are both essentially random numbers. Since  $b$  is a random number,  $g^b \bmod p$  will also be random.  $C1$  is defined as being random. By including an SHA1 hash, the client will immediately know if the message has been tampered with. While it is true that the protocol would detect this in message 5 when the server decrypts with the wrong key and does not get the correct value for  $C1$ , it is better to detect the attack early, and not waste the server's compute time. For non-encrypted messages it is not necessary to provide a hash, since the message is sent in the clear, and attacker could also forge a valid hash for the forged message.

## User to User session establishment

Once a user A has registered with the server as being on-line, they can use a modification of the Otway-Rees protocol to establish communication with B. There are 2 main differences between this protocol and Otway-Rees protocol. The first difference is the first 2 messages between user A and the Server, which is simply a request/response for B's current address. If B is off-line, then the server would send a message indicating B is off-line.

The second difference allows users A and B to establish a session key without the server being able to determine what the session key is. Instead of sending random numbers Nonce a (from user A) and Nonce b (from user B), Nonce A is replaced with  $g^a \text{ mod } p$ , where a is a random number generated by A, and Nonce b is replaced with  $g^b \text{ mod } p$ , where b is a random number generated by B. g and p are the same numbers used in the user-server session establishment. This is simply a variation of the Diffie-Hellman protocol, with the main difference being that the server decrypts the two pieces from A and B ( $g^a \text{ mod } p$ , and  $g^b \text{ mod } p$ ) and re-encrypts them with the other user's key. However, the Server does not gain any knowledge about the session key that A and B are establishing.

message

Notes

all encrypted portions of messages will contain a SHA1-MAC, which is a hash of the contents of the encrypted portion, not including the SHA1-MAC.

1 A -> S: Kas{want to talk to B, SHA1-MAC}

2 S -> A: Kas { B not online, Ns SHA1-MAC}

Ns is a random number to prevent replay attacks, B is not online, so A terminates attempt to

-OR-

2 S -> A: Kas { B's address, Ns SHA1-MAC}

Communicate with B

Ns is a random number to prevent

Replay attacks

3 A -> B: Nc A, B, Kas{Na, Nc, Ns, A, B, SHA1-MAC}

A calculates Nc = random number  
a = random number  
Na =  $g^a \text{ mod } p$

4 B -> S: Kas{Na, Nc, Ns, A, B, SHA1-MAC} Kbs{Nb, Nc, A, B, SHA1-MAC}

B calculates b = random number  
Nb =  $g^b \text{ mod } p$   
Server remembers Ns's sent out as pending connection authentications, and will verify that the Ns in the encrypted messages is still pending. Once the authentication is approved or rejected, the Ns is no longer pending, and is forgotten.  
Server verifies Nc, A and B from both pieces

5 S -> B: Nc, Kas{Na, Nb, SHA1-MAC}, Kbs{Na, Nb, SHA1-MAC}

6 B -> A: Kas{Na, Nb, SHA1-MAC}

B calculates  $(g^a)^b \text{ mod } p$   
A calculates  $(g^b)^a \text{ mod } p$   
**Kab = f( $g^{ab} \text{ mod } p$ , Nc)**

A <-> B: Kab{message, SHA1-MAC}

## **Distrust of the Server**

This protocol protects the users from even the server eavesdropping on the conversation. In message 4, the server will see the values  $(g^a \bmod p)$  and  $(g^b \bmod p)$  as essentially random numbers. Only A and B can calculate the key  $g^{ab} \bmod p$  because only they know 'a' and 'b' respectively. For the server to get the session key  $K_{ab}$ , it would have to solve the discrete log problem.

## **Perfect Forward Secrecy**

This algorithm provides PFS because, as is the case in the user-server authentication, it uses a variation of the Diffie-Hellman protocol. Even if the server session keys were compromised (which is a hard problem itself), or the user passwords were cracked (off-line or otherwise), the user-user session key is still not known.

## **Data Integrity**

All encrypted messages, or encrypted portions of messages will contain an SHA1-MAC, which is a SHA1 hash of the data in the encrypted message, not including the hash itself. This allows either client or the server to detect altered messages and break off the communication. If the server receives an invalid message, it will block that endpoint, by dropping all packets for a period of 5 minutes. Before blocking it will notify the user only once that it has been blocked (although the user may not get the message). Note that this is more restrictive than the user-server session establishment limit of 5 failed attempts in 1 minute. The main reason for this is to prevent a type of DoS attack where an attacker, pretending to be A simply sends bad packets to user B, who then puts a request in to the server. The server would then need to decrypt bad data, which wastes time. Alternately, an attacker could just keep sending a replay of message 4, posing as user B trying to authenticate a session between user A and itself. The server would see that  $N_s$  is stale, and block that user for 5 minutes.