PASSWORD AUTHENTICATED KEY EXCHANGE BY JUGGLING

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
University College of Technology
UNIVERSITY OF CALCUTTA

কম্পিউটার বিজ্ঞান ও প্রযুক্তিবিদ্যা বিভাগ, কলিকাতা বিশ্ববিদ্যালয়

92, Acharya Prafulla Chandra Road, Kolkata - 700 009, India.

CERTIFICATION

This is to certify that the dissertation entitled "**Password Authenticated Key Exchange By Juggling**" is the bonafide work of **Dr.Devadatta Sinha** and has been prepared according to the regulations for award of the degree of **Master of Science** in **Computer and Information Science** in **University of Calcutta** for the year **2011** and the candidate has partially fulfilled the requirements for the submission of the dissertation.

………………………………………………

Dr. Devadatta Sinha

Project Supervisor

Comp. Sc. & Engg. Dept., CU

………………………………………………

Prof. Nabendu Chaki

Head of the Dept.

Comp. Sc. & Engg. Dept., CU

………………………………………………

External Examiner

ACKNOWLEDGEMENT

With great pleasure we would like to express our sincere gratitude to the respected teacher Dr. Devadatta Sinha, Department of Computer Science and Engineering, University College of Science and Technology , University of Calcutta who helped us to the fascinating field of 'Password Security' and guided us with constant encouragement for successful completion of the project.

We also take pleasure to acknowledge our indebtness to Prof.Nabendu Chaki, Head of the Department of Computer Science and Engineering, University College of Science and Technology, University of Calcutta.

Hirakjyoti Banerjee
Roll Number. 91/CIS/091012
Registration Number :- 0007312
Session :- 2006-2007

Dipendu Ghosh
Roll Number :- 91/CIS/091007
Registration Number :- 0000439
Session :- 2006-2007

CONTENTS

1. INTRODUCTION:

Keys exchange protocols are cryptographic primitives used to provide a pair of users communicating over a public unreliable channel with a secure session key. In practice, one can find several flavors of key exchange protocols, each with its own benefits and drawbacks. An example of a popular one is the SIGMA protocol used as the basis for the signature-based modes of the Internet Key Exchange (IKE) protocol. The setting in which we are interested here is the 2-party symmetric one, in which every pair of users share a secret key. In particular, we consider the scenario in which the secret key is a password.

Password-Authenticated Key Exchange (PAKE) studies how to establish secure communication between two remote parties solely based on their shared password, without requiring a Public Key Infrastructure (PKI). The username/password paradigm is the most commonly used authentication mechanism in security applications. Alternative authentication factors, including tokens and biometrics, require purchasing additional hardware, which is often considered too expensive for an application. However, passwords are low-entropy secrets, and subject to dictionary attacks. Hence, they must be protected during transmission. The widely deployed method is to send passwords through SSL/TLS. But, this requires a Public Key Infrastructure (PKI) in place; maintaining a PKI is expensive. In addition, using SSL/TLS is subject to man-in-the-middle attacks. If a user authenticates himself to a phishing website by disclosing his password, the password will be stolen even though the session is fully encrypted.

Since passwords are inherently weak, one logic solution seems to replace them with strong secrets, say, cryptographically secure private keys. This approach was adopted by the UK National Grid Service (NGS) to authenticate users. In the UK, anyone who applies to access the national grid computing resource must first generate a private/public key pair of his own, and then have the public key certified by NGS. However, developments in the past ten years reveal that users – most of them are non-computer specialists – encounter serious difficulties in managing their private keys and certificates. This has greatly hindered the wider acceptance of the grid computing technology. Hence, weak passwords are just a fact of life that we must face. Researchers have been actively exploring ways to perform password-based authentication without using PKIs or certificates – a research subject called the Password-Authenticated Key Exchange (PAKE).

Password authenticated key exchange (PAKE) is where two or more parties, based only on their knowledge of a password, establish a cryptographic key using an exchange of messages, such that an unauthorized party (one who controls the communication channel but does not possess the password) cannot participate in the method and is constrained as much as possible from guessing the password. (The optimal case yields exactly one guess per run exchange.) Two forms of PAKE are Balanced and Augmented methods.

Balanced PAKE allows parties that use the same password to negotiate and authenticate a shared key. A balance scheme assumes two communicating parties hold symmetric secret information, which could be a password, or a hashed password. It is generic for any two-party communication, including client-client and client-server. Examples of these are:

- Encrypted Key Exchange(EKE)
- PAK and PPK
- SPEKE (Simple password exponential key exchange)
- J-PAKE (Password Authenticated Key Exchange by Juggling)

Augmented PAKE is a variation applicable to client/server scenarios, in which an attacker must perform a successful brute-force attack in order to masquerade as the client using stolen server data. It is more customized to the client-server case. It adds the "server compromise resistance" requirement – an attacker should not be able to impersonate users to a server after he has stolen the password verification files stored on that server, but has not performed dictionary attacks to recover the passwords. This is usually realized by storing extra password verification data – such as a (weak) public key derived from the password – together with a hash of the password on the server Examples of these are:

- AMP
- Augmented-EKE
- B-SPEKE
- PAK-Z
- SRP

2. DIFFIE- HELLMAN KEY EXCHANGE

The first published public-key algorithm appeared in the seminal paper by Diffie and Hellman that defined public-key cryptography and is generally referred to as Diffie-Hellman Key Exchange. The purpose of this algorithm is to enable two users to securely exchange a key that can be used for subsequent encryption of messages. The algorithm is itself limited to the exchange of secret values.

The Diffie-Hellman algorithm depends for its effectiveness on the difficulty of computing discrete logarithms. Briefly we can define the discrete logarithm in the following way. First we define a primitive root of a prime number *p* as one whose power modulo *p* generates all integers from 1 to *p*-1. That is if *a* is a primitive root of a prime number *p* then the numbers

$$a \bmod p,\ a^2 \bmod p, \ldots\ldots,\ a^{p-1} \bmod p$$

are distinct and consists of integers 1 through *p*-1 in some permutation. For any integer *b* and a prime root *a* of prime number *p*, we can find a unique exponent *i* such that

$$b = a^i \bmod p \qquad \text{where } 0 \leq i \leq (p-1)$$

The exponent *i* is referred to as the discrete logarithm of *b* for the base *a*, mod *p*.

2.1 Description:-

Diffie–Hellman establishes a shared secret that can be used for secret communications by exchanging data over a public network. A more general description of the protocol is given as follows:

1. Alice and Bob agree on a finite cyclic group *G* and a generating element *g* in *G* (This is usually done long before the rest of the protocol; *g* is assumed to be known by all the attackers). We will write the group G multiplicatively.
2. Alice picks a random natural number *a* and sends g^a to Bob.
3. Bob picks a random natural number *b* and sends g^b to Alice.
4. Alice computes $(g^b)^a$.
5. Bob computes $(g^a)^b$.

Both Alice and Bob are now in possession of the group element g^{ab}, which can serve as the shared secret key. The values of $(g^b)^a$ and $(g^a)^b$ are the same because groups are power associative.

A diagrammatic representation of the working of the Diffie-Hellman Algorithm is shown as follows:

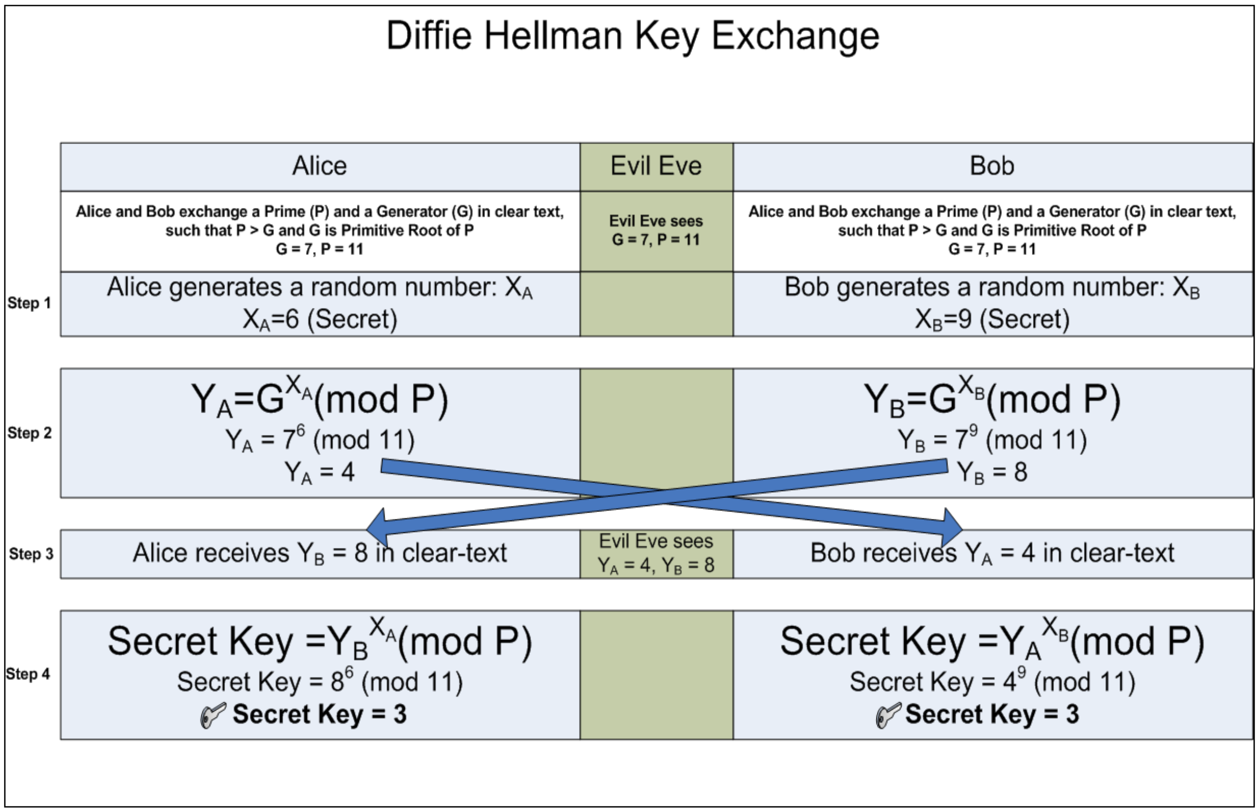
Diffie Hellman Key Exchange
Alice
Evil Eve
Bob
Alice and Bob exchange a Prime (P) and a Generator (G) in clear text, such that P > G and G is Primitive Root of P
G = 7, P = 11
Evil Eve sees G = 7, P = 11
Step 1
Alice generates a random number: X_A
X_A=6 (Secret)
Bob generates a random number: X_B
X_B=9 (Secret)
Step 2
$Y_A=G^{X_A}$(mod P)
$Y_A = 7^6$ (mod 11)
$Y_A = 4$
$Y_B=G^{X_B}$(mod P)
$Y_B = 7^9$ (mod 11)
$Y_B = 8$
Step 3
Alice receives Y_B = 8 in clear-text
Evil Eve sees $Y_A = 4$, $Y_B = 8$
Bob receives Y_A = 4 in clear-text
Step 4
Secret Key $=Y_B^{X_A}$(mod P)
Secret Key = 8^6 (mod 11)
Secret Key = 3
Secret Key $=Y_A^{X_B}$(mod P)
Secret Key = 4^9 (mod 11)
Secret Key = 3

Figure 1: Diffie-Hellman Kay Exchange Algorithm

2.2 Algorithm:-

The simplest, and original, implementation of the protocol uses the multiplicative group of integers modulo *p*, where *p* is prime and *g* is primitive root mod *p*. The following example explains the working of the Diffie-Hellman Key Exchange mechanism with steps showing values exchanged by user A and user B.

Step 1:- **User A** and **User B** agree to use a **prime number p =** 11 and a **base g =** 7. These are the public keys.

Step 2:- **User A** chooses a **secret (private key) integer a** = 3
Sends to **User B**, **$A = g^a$ mod p**
$A = 7^3$ mod 11
A = 343 mod 11
Sends **A = 2**

Step 3:- **User B** chooses a **secret (private key) integer b** = 5
Sends to **User A**, **$B = g^b$ mod p**
$B = 7^5$ mod 11
B = 16807 mod 11
Sends **B = 10**

Step 4:- **User A** computes **$S_A = B^a$ mod p**
$S_A = 10^3$ mod 11
S_A = 1000 mod 11
S_A = 10

Step 5:- **User B** computes **$S_B = A^b$ mod p**
$S_B = 2^5$ mod 11
S_B = 32 mod 11
S_B = 10

Step 6:- **User A** and **User B** now share a secret: **$S_A = S_B = 10$**. This is because **3*5** is the same as **5*3**. So somebody who comes to know both these private integers might calculate S as follows:
$S = 7^{3*5}$ mod 11
$S = 7^{5*3}$ mod 11
$S = 7^{15}$ mod 11
S = 4747561509943 mod 11
S = 10
So the Man in the Middle can stop the attack since the Secret Private Keys are not transferred over the network.

To show who knows what:-

User A		User B		Man in the Middle	
knows	**doesn't know**	**knows**	**doesn't know**	**knows**	**doesn't know**
p = 13	**b = ?**	p = 13	**a = ?**	p = 13	**a = ?**
g = 4		g = 4		g = 4	**b = ?**
a = 3		b = 6			**S = ?**
A = 4^3 mod 13 = 12		B = 4^6 mod 13 = 1		A = 4^a mod 13 = 12	
B = 4^b mod 13 = 1		A = 4^a mod 13 = 12		B = 4^b mod 13 = 1	
S_A = 1^3 mod 13 = 1		S_B = 12^6 mod 13 = 1		S_A = 1^a mod 13	
S_B = 12^b mod 13 = 1		S_A = 1^a mod 13 = 1		S_B = 12^b mod 13	
S_A = 1^3 mod 13 = 12^b mod 13 = S_B		S_B = 12^6 mod 13 = 1^a mod 13 = S_A		S_A = 1^3 mod 13 = 12^b mod 13 = S_B **OR** S_B = 12^6 mod 13 = 1^a mod 13 = S_A	
S_A = 1		S_B = 1			

So here we can see that the Man in the Middle does not know the secret key $S_A = S_B = S$ which is calculated by User A and User B. The only way to know it is using brute force method to guess the values of **a** and **b**. But with large numbers the problem becomes impractical. So the users can send messages encrypted using the secret key and no attacks are made on the message.

2.3 Man-in-the-Middle Attack:-

However the Diffie-Hellman algorithm is insecure against the man-in-the-middle attack. Suppose User A and User B wish to exchange keys, and User X is the adversary. The attack proceeds as follows:

1. User X prepares for attack by generating two random private keys **c** and **d** and compute corresponding public keys $\mathbf{X_C = g^c \bmod p}$ and $\mathbf{X_D = g^d \bmod p}$.
2. User A transmits **A** to User B.
3. User X intercepts **A** and transmits $\mathbf{X_C}$ to User B. User X then also calculates $\mathbf{S_A = (A)^d \bmod p}$.
4. User B receives $\mathbf{X_C}$ and calculates $\mathbf{S_B = (X_C)^b \bmod p}$.
5. User B transmits **B** to User A.
6. User X intercepts **B** and transmits $\mathbf{X_D}$ to User A. User X then also calculates $\mathbf{S_B = (B)^c \bmod p}$.
7. User A receives $\mathbf{X_D}$ and calculates $\mathbf{S_A = (X_D)^a \bmod p}$.

At this point User B and User A think that they share a secret key, but instead User B and User X share key $\mathbf{S_B}$ and User A and User X share secret key $\mathbf{S_A}$. Thus if User X wants then he can eavesdrop the conversation without altering it or he can modify the message going from User A to User B and vice versa.

3. MODEL OF THE NETWORK FOR KEY EXCHANGE

We assume the key exchange is carried out over an unsecured network. In such a network, there is no secrecy in communication, so transmitting a message is essentially no different from broadcasting it to all. And the broadcast is unauthenticated. An attacker can intercept a message, change it at will, and then relay the modified message to the intended recipient. First of all, we formulate the security requirements that a PAKE protocol should fulfill.

1. Off-line dictionary attack resistance – It does not leak any password verification information to a passive attacker.
2. Forward secrecy – It produces session keys that remain secure even when the password is later disclosed.
3. Known-key security – It prevents a disclosed session key from affecting the security of other sessions.
4. On-line dictionary attack resistance – It limits an active attacker to test only one password per protocol execution.

In our threat model, we do not consider the Denial of Service (DoS) attack, which is rare but powerful. Since almost all PAKE protocols are built upon public key cryptography, they are naturally vulnerable to DoS attacks, in which an attacker's sole purpose is to keep a server performing expensive public key operations. When this becomes a threat in some applications, there are well-established solutions – for example, we can add a client-puzzle protocol before engaging in any key exchange.

The scheme discussed here is a balanced Password Authenticated Key Exchange Scheme which is Password Authenticated Key Exchange by Juggling (J-PAKE).

4. PASSWORD AUTHETICATED KEY EXCHANGE BY JUGGLING

In cryptography, the **Password Authenticated Key Exchange by Juggling** (or J-PAKE) is a password=authenticated key agreement protocol. Password-authenticated key agreement generally encompasses methods such as:

- Balanced password-authenticated key exchange
- Augmented password-authenticated key exchange
- Password-authenticated key retrieval
- Multi-server methods
- Multi-party methods

This technique allows two parties to establish private and authenticated communication solely based on their shared (low-entropy) password without requiring a Public Key Infrastructure. It provides mutual authentication to the key exchange, a feature that is lacking in the Diffie-Hellman Key Exchange protocol. J-PAKE achieves this mutual authentication in two steps: first, two parties send ephemeral public keys to each other; second, they encrypt the shared password by juggling the public keys in a verifiable way.

4.1 Algorithm showing Two Step Exchange:-

Let G denote a subgroup of Z_p^* with prime order q in which the Decision Diffie-Hellman problem (DDH) is intractable. Let g be a generator in G. The two communicating parties, Alice and Bob, both agree on (G, g). Let s be their shared password, and s ≠ 0 for any non-empty password. Since s has low entropy, we assume the value of s falls within [1, q − 1].

Alice selects two secret values x1 and x2 at random: x1 ЄR Z_q and x2 ЄR Z_q^*. Similarly, Bob selects x3 ЄR Z_q and x4 ЄR Z_q^*. Note that since q is prime, Z_q only differs Z_q^* in that the later excludes ‘0’. Hence, x2, x4 ≠ 0; the reason will be evident in security analysis.

Step 1: Alice sends out g^{x1}, g^{x2} and knowledge proofs for x1 and x2. Similarly, Bob sends out g^{x3}, g^{x4} and knowledge proofs for x3 and x4. The above communication can be completed in one step as neither party depends on the other.

When this step finishes, Alice and Bob verify the received knowledge proofs, and also check g^{x2}, g^{x4} ≠ 1.

Step 2: Alice sends out A = $g^{(x1+x3+x4).x2.s}$ and a knowledge proof for x2.s. Similarly, Bob sends out B =$g^{(x1+x2+x3).x4.s}$ and a knowledge proof for x4.s.

When this step finishes, Alice computes K = $(B/g^{x2.x4.s})^{x2} = g^{(x1+x3).x2.x4.s}$, and Bob computes K = $(A/g^{x2.x4.s})^{x4} = g^{(x1+x3).x2.x4.s}$. With the same keying material K, a session key can be derived κ = H (K), where H is a hash function. Before using the session key, Alice and Bob may perform an additional key confirmation process as follows: Alice sends H (H (κ)) to Bob, and Bob then replies H (κ). This process is the same as in. It gives explicit assurance that the two ends derived the same session key.

4.2 Schnorr protocol:-

One of the simplest and frequently used proofs of knowledge, the *proof of knowledge of a discrete logarithm*, is due to Schnorr. The knowledge proofs mentioned in the algorithm of J-PAKE are done in this way. The protocol is defined for a cyclic group G_q of order q with generator g.

In order to prove knowledge of $x = \log_g y$, the prover interacts with the verifier as follows:

1. In the first round the prover commits herself to randomness r; therefore the first message $t = g^r$ is also called *commitment*.
2. The verifier replies with a *challenge* c chosen at random.
3. After receiving c, the prover sends the third and last message (the *response*) $s = r + cx$.

The verifier accepts, if $g^s = ty^c$.

4.3 SECURITY ANALYSIS

In this section, we analyze the protocol's resistance against both passive and active attacks. First, we consider a passive attacker who eavesdrops on the communication between Alice and Bob. Alice's cipher text A contains the term (x1 + x3 + x4) on the exponent. The following two cases show the security properties of (x1 + x3 + x4) and A.

Case 1:- Under the Discrete Logarithm (DL) assumption, Bob cannot compute (x1 + x3 + x4).

Suppose if we reveal x2 to Bob and the knowledge proofs of the protocol shows that Bob knows x3 and x4 then Bob knows $\{g^{x1}$, x2, x3, x4} (based on which he can compute A, B). If we assume that Bob is able to compute (x1 + x3 + x4) then he is able to compute x1. This however contradicts the DL assumption, which states that one cannot compute x1 from g, g^{x1}. Therefore, even with x2 revealed, Bob will not be able to compute (x1 + x3 + x4).

Case 2:- Under the Decision Diffie-Hellman (DDH) assumption, Bob cannot distinguish Alice's cipher text A =$g^{(x1+x3+x4)\cdot x2\cdot s}$ from a random element in the group.

We know that (x1 +x3 + x4) is a random value over Z_q unknown to Bob. Also, x2.s is a random value over Z_q, unknown to Bob. Based on the Decision Diffie-Hellman assumption, we can say that Bob cannot distinguish A from a random element in the group. Based on the protocol symmetry we can say from Alice's perspective that Alice cannot compute (x1 + x2 + x3) nor distinguish B from a random element in the group.

Case 3:- (Off-line dictionary attack resistance) Under the DDH assumption, the cipher texts A = $g^{(x1+x3+x4)\cdot x2\cdot s}$ and B = $g^{(x1+x2+x3)\cdot x4\cdot s}$ do not leak any information for password verification.

From the previous cases it is implied that Bob cannot computationally correlate A to the cipher text he can compute: B. We also know that Bob cannot distinguish A from a random value in G. Hence a passive attacker also cannot distinguish A from a random value in G, nor can he computationally correlate A to B. Based on the protocol symmetry, the passive attacker cannot distinguish B from a random value in G either. Therefore, to a passive attacker, A and B are two random and independent values; they do not leak any useful information for password verification.

Case 4:- (Forward secrecy) The past session keys derived from the protocol remain secure even when the secret s is later disclosed.

We now consider the case when a session key is compromised. Compared with other ephemeral secrets xi (i = 1 . . . 4) – which can be immediately erased after the key bootstrap phase – a session key lasts longer throughout the session, which might increase the likelihood of its exposure. An exposed session key should not cause any global effect on the system. In our protocol, a known session key does not affect the security of either the password or other session keys. From an explicit key confirmation process, an attacker learns H (H (κ)) = H(H(H(K))) and H(κ) = H(H(K)). It is obvious that learning κ = H (K) does not give the attacker any additional knowledge about K, and therefore, the security of the password encrypted at that session remains intact. Also, the session key $\kappa = H(g^{(x1+x3)\cdot x2\cdot x4\cdot s})$ is determined by the (fresh) ephemeral inputs from both parties in the session. We know that x2, x4≠0 by definition and (x1+x3) is random over Z_q; hence the obtained session key is completely different from keys derived in other sessions. Therefore, compromising a session key has no effect on other session keys.

Case 5:- (On-line dictionary attack resistance) An active attacker cannot compute the session key if he chose a value s'≠ s.

Here we assume that Alice is honest but Bob is compromised (i.e. an attacker). Then the cipher text sent by Bob is given by $B' = g^{(x1+x2+x3).x4.s'}$ where s' is a value that Bob (the attacker) can choose freely.

After receiving B', Alice computes,

$$
\begin{aligned}
K' &= (B'/g^{x2\cdot x4\cdot s})^{x2} \\
&= (g^{(x1+x2+x3).x4.s'}/ g^{x2\cdot x4\cdot s})^{x2} \\
&= g^{x1\cdot x2\cdot x4\cdot s'}. g^{x2\cdot x3\cdot x4\cdot s'}. g^{x2.x2.x4.(s'-s)}
\end{aligned}
$$

Now Bob computes K' as,

$$
\begin{aligned}
K' &= (A/g^{x2.x4.s'})^{x4} = g^{(x1+x3).x2.x4.s'} \\
&= (g^{(x1+x3+x4).x2.s} /g^{x2.x4.s'})^{x4} \\
&= g^{x1\cdot x2\cdot x4\cdot s}. g^{x2\cdot x3\cdot x4\cdot s}. g^{x2.x4.x4.(s-s')}
\end{aligned}
$$

Hence we see that the values of K' computed by Alice and bob are different, hence an active attacker cannot compute the session key if he chooses a value s' ≠ s.

From the above case it shows that our protocol is zero-knowledge. Because of the knowledge proofs, the attacker is left with the only freedom to choose an arbitrary s'. If s' ≠ s he is unable to derive the same session key as Alice. During the later key confirmation process, the attacker will learn one-bit information: whether s' and s is equal. This is the best that any PAKE protocol can possibly achieve, because by nature we cannot stop an imposter from trying a random guess of password. However, consecutively failed guesses can be easily detected and thwarted accordingly.

The J-PAKE scheme provides security against all the attacks mentioned earlier. The following table describes J-PAKE properties classified according to modules. For each module the type of attacker and the assumptions are also mentioned.

Modules	Security Properties	Attacker Type	Assumptions
Schnorr signature	Leak 1-bit ; whether sender knows discrete logarithm	passive/active	DL and random oracle
Password Encryption	Indistinguishable from random	passive/active	DDH
Session Key	incomputable	passive	CDH
	incomputable	passive (known s)	CDH
	incomputable	passive (known other session keys)	CDH
	incomputable	active (if s ≠ s’)	CDH
Key confirmation	leak nothing	passive	-
	leak 1-bit: whether if s ≠ s’	active	CDH

Here s’ is a value that the attacker can choose freely in case of active attacks.

5. SOFTWARES USED FOR IMPLEMENTATION

The softwares that are used to implement the Diffie-Hellman Key Exchange algorithm and Password Authenticated Key Exchange by Juggling are given:

- **O.S. :** Mandriva Linux 2010
- **C compiler:** Inbuilt C compiler in Linux.

6. IMPLEMENTATION

In this section we illustrate the steps of the algorithms discussed earlier in an informative way. Here we have shown the way they are implemented using C programming language. The following section contains steps for implementing the Diffie-Hellman Key Exchange algorithm and its short coming in the face of the man-in-the-middle attack. We have also given the code for implementing the Password Authenticated Key Exchange by Juggling. Here we have shown only a simulated environment and how the algorithms function in such an environment. Code snippets are provided along with the steps of the algorithms.

6.1 Algorithm of Diffie-Hellman Key Exchange:-

Step 1:- The prime number and the generator are initialized and the choice of the user, whether he wants to perform the key exchange by Normal Exchange of key or Key Exchange infiltrated by attacker.

Code Snippet for Main:

```
int main()
{
    //prime number(p)
    p=11;//23
    //generator(g)
    g=7;//5

    printf("\nPrime number p = %llu",p);
    printf("\nGenerator g = %llu",g);

    //Performing the key exchange depending on user's choice of Normal
            Exchange of key or Key Exchange infilterated by attacker
    do
    {
            printf("\n\nDEFFIE-HELLMAN KEY EXCHANGE:-");
            printf("\n1. Normal Exchange of key.");
            printf("\n2. Key Exchange infilterated by attacker.");
            printf("\n3. Exit.");
            printf("\nEnter your choice (1-3):");
            scanf("%d",&ch);

            switch(ch)
            {
                  case 1:
                        user_a();
                        break;
                  case 2:
                        attacker();
                        break;
                  case 3:
                        exit(0);
                        break;
                  default:
                        printf("Invalid Choice!");
                        break;
            }
    }while(1);

    return 0;
}
```

Step 2:- For choice 1 of user, user_a() is invoked with x=0 and y=0. The private key of User A is entered and the public key is generated (x = gen_no(g,p,a)) and since it is a normal key exchange user_b() is invoked. The private key of User B is entered and the public key is generated (y = gen_no(g,p,b)). Then common private key is calculated by User B (sb = gen_no(x,p,b)) and common private key is calculated by User A (sa = gen_no(y,p,a)).

```
Code Snippet for User A:

//To take user A's private key and generate the public key
void user_a()
{
    do
    {
        //Private key of user A(6,6)
        printf("\nEnter User A's private key :- ");
        scanf("%llu",&a);

        if(a<p)
                break;
        else
                printf("\nReenter");
    }while(1);

    //Public key for user A
    x = gen_no(g,p,a);
    printf("\nPublic key for User A: %llu",x);

    //Start actions of User B
    if(y==0)
        user_b();

    //Private key for both the users
    sa = gen_no(y,p,a);
    printf("\nCommon Private Key as calculated by User A: %llu",sa);
}
```

```
Code Snippet for User B:

//To take user B's private key and generate the public key
```

```
void user_b()
{
    do
    {
        //Private key of user B(9,15)
        printf("\nEnter User B's private key :- ");
        scanf("%llu",&b);

        if(b<p)
                break;
        else
                printf("\nReenter");
    }while(1);

    //Public key for user B
    y = gen_no(g,p,b);
    printf("\nPublic key for User B: %llu",y);

    //Private key for both the users
    sb = gen_no(x,p,b);
    printf("\nCommon Private Key as calculated by User B: %llu",sb);
}
```

Step 3:- For choice 2 of user, attacker() is invoked with x=0 and y=0. The private key of User A is entered and the private key of User B is entered through the attacker. The Attacker generates the public key to be transmitted to User A and invokes user_a(). The private key entered by User A and the public key generated by User A and the common private key generated by User A are now known to the Attacker. Then the does the same thing for User B. Here to User A the Attacker poses as User B and to User B the Attacker poses as User A.

Code Snippet:

```
//To take user A's and B's private key and generate the public key
void attacker()
{
    //For User A
    printf("\nEnter the 1st private key for attacker:");
    scanf("%llu",&c);

    //For User B
    printf("\nEnter the 2nd private key for attacker:");
    scanf("%llu",&d);

    //Generating Public Key to be transmitted to User A
    y=gen_no(g,p,c);
    user_a();

    //Calculating common private key between User A and Attacker
    sa= gen_no(x,p,c);
    printf("\nCommon Private Key as calculated for User A by Attacker:
%llu",sa);

    //Generating Public Key to be transmitted to User B
    x=gen_no(g,p,d);
    user_b();

    //Calculating common private key between User B and Attacker
    sb=gen_no(y,p,d);
    printf("\nCommon Private Key as calculated for User B by Attacker:
%llu",sb);
}
```

6.2 Algorithm of Password Authenticated Key Exchange by Juggling:-

Step 1:- Instantiating JPakeParameters and JPakeUsers and Initializing their variables.

```
Code Snippet:

      //Declaring the structure variables for the parameters and the uers
      struct JPakeParameters param;
      struct JPakeUser alice,bob;

      //Initializing the parameters
      JPakeParameters_Init(&param);

      //Initializing the variables for user Alice
      alice.name = "Alice";
      alice.base=5;
      alice.key=0;

      //Initializing the variables for user Bob
      bob.name = "Bob";
      bob.base=7;
      bob.key=0;
```

Step 2:- Generating the secret key(≠ 0) randomly using the genrand(n)function for both the users and initialising them.

```
Code Snippet:

From Main:
//The secret key for the users are created and shared by both the users
      alice.secret=genrand(1);
      bob.secret=alice.secret;

Function Declaration:
//To calculate random numbers starting from the given number
unsigned long long genrand(unsigned long long n)
{
      //Defining the starting point
      srand(n*rand());
      //Generating the random number
      x=rand()%100000;

      return x;
}
```

Step 3:- Selecting the secret values of x_1 and x_2 for Alice and the secret values of x_3 and x_4 for Bob by the genknp(&user) function, which generates the values at random, where $x_1,x_3 \in_R Z_q$ and $x_2,x_4 \in_R Z^*_q$ and $x_2,x_4 \neq 0$.

```
Code Snippet:

From Main:
     //Generate secret values for Alice and Bob
     //Alice's x1 and x2
      genknp(&alice);
     //Bob's x3 and x4
      genknp(&bob);

Function Declaration:
//To select the secret values for the users
//Alice selects x1 and x2
//Bob selects x3 and x4
void genknp(struct JPakeUser *user)
{
      //If user is Alice the x1 and x2 are selected
      if(strcmp(user->name,"Alice"))
      {
                    do
                    {
                              user->xa=genrand(3);
                              user->xb=genrand(5);
                    }while(user->xa!=0 && user->xb!=0);
      }
      //Else If user is Bob the x3 and x4 are selected
      else if(strcmp(user->name,"Bob"))
      {
                    do
                    {
                              user->xa=genrand(13);
                              user->xb=genrand(17);
                    }while(user->xa!=0 && user->xb!=0);
      }
}
```

Step 4:- Sending Step 1 for both Alice and Bob. Alice sends g^{x1}, g^{x2} and knowledge proofs for x_1 and x_2. Bob sends g^{x3}, g^{x4} and knowledge proofs for x_3 and x_4.

i) Function sendstep1 (&from, &to, &param) sends g^{x1}, g^{x2} for Alice and sends g^{x3}, g^{x4} for Bob.

ii) Function zkp (from, to, param, step number) sends knowledge proofs for x_1 and x_2 of Alice and knowledge proofs for x_3 and x_4 of Bob.

iii) Function prover(from, to, param, step number, variable order) calculates the knowledge proofs for x1 and x2 of Alice or knowledge proofs for x3 and x4 of Bob by generating the commitment(r) and receiving the challenge(c) from verifier and then sends s=r + c * x, where x Є { x_1, x_2, x_3, x_4}.

iv) Function verifier(u, step number, variable order,0,0,0,0,0) calculates the challenge for the given x, where x Є { x_1, x_2, x_3, x_4}.

Code Snippet:

From Main:
```
     //Step 1 send
      sendstep1(&alice,&bob,&param);
      sendstep1(&bob,&alice,&param);
```

Function sendstep1 declaration:
```
//Step 1 Send
//Alice sends g^x1 and g^x2 and zero knowlwdge proofs of x1 and x2 to Bob
//Bob sends g^x3 and g^x4 and zero knowlwdge proofs of x3 and x4 to Alice
void sendstep1(struct JPakeUser *from,struct JPakeUser *to,struct
      JPakeParameters *param)
{
       //Alice sends g^x1 to Bob and Bob sends g^x3 to Alice
       to->stp1c=powdef(param->g,from->xa);
       //Alice sends g^x2 to Bob and Bob sends g^x4 to Alice
       to->stp1d=powdef(param->g,from->xb);

       //Alice sends knowledge proofs of x1 and x2 to Bob and Bob sends
      knowledge proofs of x3 and x4 to Alice
       f=zkp(from,to,param,1);
}
```

Function zkp declaration:
```
//zkp(Alice or Bob,parameter,send step)
unsigned long long zkp(struct JPakeUser *from,struct JPakeUser *to,struct
      JPakeParameters *param,unsigned long long ss)
{
      //It is step 1 send then calculating the knowlwdge proofs of x1 and x2
      for Alice or the knowlwdge proofs of x3 and x4 for Bob
      if(ss==1)
      {
          //Knowledge proof for x1 of Alice or knowledge proof for x3 of Bob
          prover(from,to,param,1,1);
          //Knowledge proof for x1 of Bob or knowledge proof for x3 of Alice
          prover(from,to,param,1,2);

          return 0;
      }
}
```

Function prover declaration:

```
//prover(Alice or Bob,parameters,send step,substep)
void prover(struct JPakeUser *from,struct JPakeUser *to,struct
      JPakeParameters *param, unsigned long long ss, unsigned long long sss)
{
      if(strcmp(from->name,"Alice"))
          u=1;
      else if(strcmp(from->name,"Bob"))
          u=2;

      if(strcmp(from->name,"Alice"))
          u=1;
      else if(strcmp(from->name,"Bob"))
          u=2;

      //Calculating Knowledge Proofs for step 1 send
      if(ss==1)
      {
          //Knowledge proof of x1 for Alice or Knowledge proof of x3 for Bob
          if(sss==1)
          {
              if(u==1)
                  r=genrand(7);
              else if(u==2)
                  r=genrand(11);

              //Random number for Commitment by prover
              to->r1=r;

              //Random number which is the challange by the verifier
              to->c1=verifier(u,1,1,0,0,0,0);

              //The last message by the prover s=r+c*x
              to->s1=to->r1+to->c1*from->xa;
          }
          //Knowledge proof of x3 for Alice or Knowledge proof of x4 for Bob
          else if(sss==2)
          {
              if(u==1)
                  r=genrand(43);
              else if(u==2)
                  r=genrand(47);

            //Random number for Commitment by prover
              to->r2=r;

            //Random number which is the challange by the verifier
              to->c2=verifier(u,1,2,0,0,0,0);
            //The last message by the prover s=r+c*x
              to->s2=to->r2+to->c2*from->xb;
          }
      }
}

Function verifier declaration:
//verifier(Alice or Bob,send step,verification
      step,generator,r,c,s=r+c^x,y=g^x)
unsigned long long verifier(unsigned long long user, unsigned long long ss,
      unsigned long long sss, unsigned long long r, unsigned long long c,
      unsigned long long s, unsigned long long x)
{
      //For send step 1
      if(ss==1)
      {
          //Challenge of x1 for Alice or Challenge of x3 for Bob
          if(sss==1)
          {
              if(user==1)
                  c=genrand(29);
              else if(user==2)
                  c=genrand(31);
          }
          //Challenge of x2 for Alice or Challenge of x4 for Bob
          else if(sss==2)
          {
              if(user==1)
                  c=genrand(37);
              else if(user==2)
                  c=genrand(41);
          }

          c=c;

          return c;
      }
}
```

Step 5:- Verifying Step 1 for both Alice and Bob. Alice and Bob verify the received knowledge proofs, and also checks g^{x2} , $g^{x4} \neq 1$.

i) Function verifysendstep1 (&from,&to,&param) verifies if the sending step 1 is successful.

ii) Function zkp (from, to, param, step number) verifies knowledge proofs for x_1 and x_2 of Alice and knowledge proofs for x_3 and x_4 of Bob.

iii) Function verifier(u, step number, variable order, generator, commitment, challenge, s, x) verifies the knowledge proof for the given x, where x Є { x_1, x_2, x_3, x_4}.

Code Snippet:

From Main:
```
//Verify step 1 send
      verifysendstep1(&alice,&bob,&param);
      verifysendstep1(&bob,&alice,&param);
```

Function verifysendstep1 declaration:
```
//Verify step 1 send
//Verifies the step 1 send to check if the transmission was successful
void verifysendstep1(struct JPakeUser *from,struct JPakeUser *to,struct
      JPakeParameters *param)
{
      //Verifying the sending of knowledge proofs of x1 and x2 by Alice to
      Bob or knowledge proofs of x3 and x4 by Bob to Alice
       f=zkp(from,to,param,2);

      if(f2==1)
          printf("\nStep 1 successful\n");
      else
      {
          printf("\nStep 1 Fail\nExitting\n");
          exit(0);
      }
}
```

Function zkp declaration:
```
//zkp(Alice or Bob,parameter,send step)
unsigned long long zkp(struct JPakeUser *from,struct JPakeUser *to,struct
      JPakeParameters *param, unsigned long long ss)
{
      //It is the verification of step 1 send then calling verifier to verify
      if g^s=t*y^c for Alice or Bob
      else if(ss==2)
      {
          if(strcmp(from->name,"Alice"))
              u=1;
          else if(strcmp(from->name,"Bob"))
              u=2;

          //Verifying the knowledge proof of x1 for Alice or x3 for Bob
          f1=verifier(u,2,0,to->r1,to->c1,to->s1, from->xa);
          //Verifying the knowledge proof of x4 for Alice or x4 for Bob
          f2=verifier(u,2,0,to->r2,to->c2,to->s2, from->xb);

          if(f1==1 && f2==1)
              return 1;
      }
}
```

```
Function verifier declaration:
//verifier(Alice or Bob,send step,verification
      step,generator,r,c,s=r+c^x,y=g^x)
unsigned long long verifier(unsigned long long user, unsigned long long ss,
      unsigned long long sss, unsigned long long r, unsigned long long c,
      unsigned long long s, unsigned long long y)
{
      //For step 1 send verification
      else if(ss==2)
      {
        //Checking if g^=t*y^c of x1 and x2 for Alice and Checking if
      g^=t*y^c of x3 and x4 for Bob
          t1=r+c*x;
          t2=s;

          if(t1==t2)
              return 1;
      }
}
```

Step 6:- Sending Step 2 for both Alice and Bob. Alice sends $A=g^{(x1+x3+x4)*x2*secret}$ and knowledge proof x_2*secret. Bob sends $B=g^{(x1+x2+x3)*x4*secret}$ and knowledge proof for x_4*secret.

i) Function sendstep2 (&from, &to, &param) sends $A=g^{(x1+x3+x4)*x2*secret}$ for Alice and sends $B=g^{(x1+x2+x3)*x4*secret}$ for Bob.

ii) Function zkp (from, to, param, step number) sends knowledge proof for x_2*secret of Alice and knowledge proof for x_4*secret of Bob.

iii) Function prover (from, to, param, step number, variable order) calculates the knowledge proof for x_2*secret of Alice or knowledge proof for x_4*secret of Bob by generating the commitment(r) and receiving the challenge(c) from verifier and then sends s=r + c * x, where x Є {x_2*secret, x_4*secret}.

iv) Function verifier (u, step number, variable order, 0,0,0,0,0) calculates the challenge for the given x, where x Є {x_2*secret, x_4*secret}.

```
Code Snippet:

From Main:
      //Step 2 send
      sendstep2(&alice,&bob,&param);
      sendstep2(&bob,&alice,&param);
```

Function sendstep2 declaration:
```
//Step 2 Send
//Alice sends A=g^{(x1+x3+x4)*x2*secret} and zero knowlwdge proof of
      x2*secret to Bob
//Bob sends B=g^{(x1+x2+x3)*x4*secret} and zero knowlwdge proof of x4*secret
      to Alice
void sendstep2(struct JPakeUser *from,struct JPakeUser *to,struct
      JPakeParameters *param)
{
      //Alice sends A=g^{(x1+x3+x4)*x2*secret} to Bob and Bob sends
      B=g^{(x1+x2+x3)*x4*secret} to Alice
      to->stp2=(powdef(param->g,from->xa)*from->stp1c*from->stp1d);
      to->stp2=powdef(to->stp2,from->xb*from->secret);

      //Alice sends knowledge proof of x2*secret to Bob and Bob sends
      knowledge proof of x4*secret to Alice
      f=zkp(from,to,param,3);
}
```

Function zkp declaration:
```
//zkp(Alice or Bob,parameter,send step)
unsigned long long zkp(struct JPakeUser *from,struct JPakeUser *to,struct
      JPakeParameters *param, unsigned long long ss)
{
      //It is step 2 send then calculating the knowlwdge proof of x2*secret
      for Alice or the knowlwdge proof of x4*secret for Bob
      else if(ss==3)
      {
          //Knowlwdge proof of x2*secret for Alice or knowlwdge proof of
      x4*secret for Bob
          prover(from,to,param,3,0);

          return 0;
      }
}
```

Function prover declaration:
```
//prover(Alice or Bob,parameters,send step,substep)
void prover(struct JPakeUser *from,struct JPakeUser *to,struct
      JPakeParameters *param, unsigned long long ss, unsigned long long sss)
{
      if(strcmp(from->name,"Alice"))
          u=1;
      else if(strcmp(from->name,"Bob"))
          u=2;

      //Calculating Knowledge Proofs for step 2 send
       //Knowledge proof of x2*secret for Alice or Knowledge proof of
      x4*secret for Bob
       else if(ss=3)
      {
          if(u==1)
              r=genrand(19);
          else if(u==2)
              r=genrand(23);
          //Random number for Commitment by prover
          to->r3=r;

          //Random number which is the challange by the verifier
          to->c3=verifier(u,3,0,0,0,0,0);

          //The last message by the prover s=r+c*x
          to->s3=to->r3+to->c3*from->xb*from->secret;
      }
}

Function verifier declaration:
//verifier(Alice or Bob,send step,verification
      step,generator,r,c,s=r+c^x,y=g^x)
unsigned long long verifier(unsigned long long user, unsigned long long ss,
      unsigned long long sss, unsigned long long r, unsigned long long c,
      unsigned long long s, unsigned long long x)
{
      //For send step 2
      //Challenge of x2*secret for Alice or Challenge of x4*secret for Bob
      else if(ss==3)
      {
          if(user==1)
              c=genrand(51);
          else if(user==2)
              c=genrand(53);

          c=c;

          return c;
      }
}
```

Step 7:- Verifying Step 2 for both Alice and Bob. Alice and Bob verify the received knowledge proof.

i) Function verifysendstep2 (&from,&to,&param) verifies if the sending step 2 is successful.
ii) Function zkp (from, to, param, step number) verifies knowledge proof for x_2*secret of Alice and knowledge proofs for x_4*secret of Bob.
iii) Function verifier(u,step number,variable order,generator,commitment,challenge,s,x) verifies the knowledge proof for the given x, where x Є {x_2*secret, x_4*secret}.

Code Snippet:

From Main:
```
//Verify step 2 send
      verifysendstep2(&alice,&bob,&param);
      verifysendstep2(&bob,&alice,&param);
```

Function verifysendstep2 declaration:
```
//Verify step 2 send
//Verifies the step 2 send to check if the transmission was successful
void verifysendstep2(struct JPakeUser *from,struct JPakeUser *to,struct
      JPakeParameters *param)
{
      //Verifying the sending of knowledge proof of x2*secret by Alice to Bob
      or knowledge proof of x4*secret by Bob to Alice
      f=zkp(from,to,param,4);

      if(f==1)
          printf("\nStep 2 successful\n");
      else
      {
          printf("\nStep 2 Fail\nExitting\n");
          exit(0);
      }
}
```

Function zkp declaration:
```
//zkp(Alice or Bob,parameter,send step)
unsigned long long zkp(struct JPakeUser *from,struct JPakeUser *to,struct
      JPakeParameters *param, unsigned long long ss)
{
      //It is the verification of step 2 send then calling verifier to verify
      if g^s=t*y^c for Alice or Bob
      else if(ss==4)
      {
          if(strcmp(from->name,"Alice"))
              u=1;
          else if(strcmp(from->name,"Bob"))
              u=2;

          //Verifying the knowledge proof of x2*secret for Alice or x4*secret
      for Bob
          f1=verifier(u,4,0,to->r3,to->c3,to->s3,(from->xb*from->secret));

          if(f1==1)
              return 1;
      }
}
```

Function verifier declaration:
```
//verifier(Alice or Bob,send step,verification
      step,generator,r,c,s=r+c^x,y=g^x)
unsigned long long verifier(unsigned long long user, unsigned long long ss,
      unsigned long long sss, unsigned long long r, unsigned long long c,
      unsigned long long s, unsigned long long y)
```

```
{
      //For step 2 send verification
      else if(ss==4)
      {
          //For step 2 send verification
      else if(ss==4)
      {
        //Checking if g^=t*y^c of x2*secret for Alice and Checking if
      g^=t*y^c of x4*secret for Bob
          t1=r+c*x;
          t2=s;

          if(t1==t2)
              return 1;
      }
      return 0;
}
```

Step 8:- If the sending in two steps (send step1 and send step2) are successful then the keys are computed. compute_key (&user, &param) computes $K=(B/g^{x2*x4*secret})^{x2}= g^{(x1+x3)*x2*x4\cdot secret}$ for Alice and $K=(A/g^{x2*x4*secret})^{x4}= g^{(x1+x3)*x2*x4\cdot secret}$ for Bob.

```
Code Snippet:

From Main:
//Compute keys
      compute_key(&alice,&param);
      compute_key(&bob,&param);

Function compute_key declaration:
//Computeing the keys for Alice and Bob after step 1 send and step 2 send are
successful
void compute_key(struct JPakeUser *user,struct JPakeParameters *param)
{
      //For Alice K=(B/g^{x2*x4*secret})^x2
      //For Bob K=(A/g^{x2*x4*secret})^x4
      temp=powdef(user->stp1d,(user->xb*user->secret));
      temp=user->stp2/temp;
      user->key=powdef(temp,user->xb);
}
```

Step 9:- Performs hashing on the computed keys for the users and maps them in the hash table.

```
Code Snippet:

From Main:
//Generating session key
```

```
      hashing(&alice);
      hashing(&bob);

Function hashing declaration:
//Hashing the computed keys to their positions in the hash table
void hashing(struct JPakeUser *user)
{
      //Using sum of digits until it is between 0 and 10
      while(k>10)
      {
            s=0;
            while(k>0)
            {
                  s=s+k%10;
                  k=k/10;
            }
            k=s;
      }

      //Finding the appropriate hashed position in the hash table
      do
      {
            if(hash[k]==-999)
            {
                   hash[k]=user->key;
                   user->sessionk=k;
                   break;
            }
            else
                   k++;
      }while(1);
}
```

Step 10:- Successful Termination.

```
From Main:
//Displaying the attributes of Alice and Bob
      show(&alice);
      show(&bob);
```

7. OUTPUT

Diffie-Hellman Key Exchange

```
[dipendu@localhost Project]$ cc dhke_all.c
dhke_all.c: In function ‘main’:
dhke_all.c:153: warning: incompatible implicit declaration of built-in function ‘exit’
[dipendu@localhost Project]$ ./a.out

Prime number p = 11
Generator g = 7

DEFFIE-HELLMAN KEY EXCHANGE:-
1. Normal Exchange of key.
2. Key Exchange infilterated by attacker.
3. Exit.
Enter your choice (1-3):1

Enter User A's private key :- 3

Public key for User A: 2

Enter User B's private key :- 5

Public key for User B: 10

Common Private Key as calculated by User B: 10
Common Private Key as calculated by User A: 10

DEFFIE-HELLMAN KEY EXCHANGE:-
1. Normal Exchange of key.
2. Key Exchange infilterated by attacker.
3. Exit.
Enter your choice (1-3):2

Enter the 1st private key for attacker:4

Enter the 2nd private key for attacker:6

Enter User A's private key :- 5
```

```
Public key for User A: 10

Common Private Key as calculated by User A: 1
Common Private Key as calculated for User A by Attacker: 1
Enter User B's private key :- 3

Public key for User B: 2

Common Private Key as calculated by User B: 9
Common Private Key as calculated for User B by Attacker: 9


DEFFIE-HELLMAN KEY EXCHANGE:-
1. Normal Exchange of key.
2. Key Exchange infilterated by attacker.
3. Exit.
Enter your choice (1-3):3
[dipendu@localhost Project]$
```

<u>**Password Authenticated Key Exchange by Juggling**</u>

```
[dipendu@localhost Project]$ cc jpake.
jpake.c   jpake.c~
[dipendu@localhost Project]$ cc jpake.c
jpake.c: In function ‘verifysendstep1’:
jpake.c:391: warning: incompatible implicit declaration of built-in function ‘exit’
jpake.c: In function ‘verifysendstep2’:
jpake.c:424: warning: incompatible implicit declaration of built-in function ‘exit’
[dipendu@localhost Project]$ ./a.out

Enter a prime number :- 11
Enter a generator :- 7

Step 1 successful

Step 1 successful

Step 2 successful

Step 2 successful
```

```
Name    :      Alice
Base    :      5
Secret  :      3434404
Key     :      1
xa      :      2302302
xb      :      0
stp1c   :      1321033202100222413203224004
stp1d   :      1
stp2    :      1
session key     :      1
hash value      :      1

Name    :      Bob
Base    :      7
Secret  :      345222
Key     :      1
xa      :      503400
xb      :      0
stp1c   :      4661511401502654153621
stp1d   :      1
stp2    :      1
session key     :      2
hash value      :      1

Terminating Successfully
[dipendu@localhost Project]$
```

8. CONCLUSION

In this document we showed how the Diffie-Hellman Key Exchange Algorithm works. We have also shown its shortcoming in the face of the man-in-the-middle attack. Finally we have shown how J-PAKE overcomes the shortcomings of the Diffie-Hellman Key Exchange. We have also shown how J-PAKE authenticates a password with zero-knowledge and then subsequently create a strong session key if the password is correct. We also stated that the protocol fulfills the following properties: it prevents off-line dictionary attacks; provides forward secrecy; insulates a known session key from affecting any other sessions; and strictly limits an active attacker to guess only one password per protocol execution. The implementations of the Diffie-Hellman Key Exchange Algorithm and Password Authenticated Key Exchange by Juggling are also given in C. As compared to the de facto internet standard SSL/TLS, J-PAKE is more lightweight in password authentication with two notable advantages: 1). It requires no PKI deployments; 2). It protects users\ from leaking passwords (say to a fake bank website).

9. FUTURE SCOPE

1. Here we have programmed only a simulated environment. So we didn’t compute keys which have huge number of digits.
2. In future we need to implement password authenticated key exchange by juggling in a real-world networked environment.
3. We also need to show by implementing certain cases that a network with J-PAKE implemented on it is secure against an attacker.

10. REFERENCES

1. Simple Password-Based Encrypted Key Exchange Protocols
   By Michel Abdalla and David Pointcheval

2. Password Authenticated Key Exchange by Juggling
   By Feng Hao from Center for Computational Science, University College London
   And Peter Ryan from School Of Computing Science, University of Newcastle upon Tyne

3. Cryptography and Network Security Principles and Practices, 4th Ed
   By William Stallings

4. http://en.wikipedia.org/wiki/Schnorr_signature

5. http://en.wikipedia.org/wiki/Diffie-Hellman_key_exchange

6. http://en.wikipedia.org/wiki/Schnorr_group

7. http://en.wikipedia.org/wiki/Cyclic_group

8. http://en.wikipedia.org/wiki/Proof_of_knowledge

9. http://en.wikipedia.org/wiki/PAKE

10. http://en.wikipedia.org/wiki/J-PAKE

11. http://www.lix.polytechnique.fr/~liberti/public/computing/prog/c/C/FUNCTIONS/rand.html

12. http://www.dreamincode.net/forums/topic/12660-use-of-rand-function/

13. http://pubs.opengroup.org/onlinepubs/009695399/functions/rand.html