UNIVERSITÀ
DEGLI STUDI
DI PALERMO

Secure chat using Diffie Hellman and AES

Salvatore Calderaro 0704378

Indice

1 Introduzione

In questo progetto è stata implementata una chat sicura tra due nodi. Quest'ultimi sono stati simulati tramite due socket (il client ed il server). Il protocollo usato per stabilire la chiave condivisa con cui i due utenti cifreranno i loro messaggi è *Diffie - Hellman*. Come cifrario è stato scelto uno stream cipher in particolare l'*AES-256* in modalità *EAX* che usa lo schema CTR come modalità di cifratura e uno CMAC per quanto concerne l'integrità. Il software è stato implementato mediante il linguaggio di programmazione *Python*.

2 Creazione dei due nodi mediante socket

Per la simulazione dei due nodi che devono comunicare tra di loro sono state create due *socket* una per il client che rappresenta l'utente umano che utilizzerà il software ed una per il server che rappresenta il potenziale utente con cui si vuole avere una conversazione sicura. Di seguito le due funzioni utilizzate:

```
def create_client_socket():
    global host,port,client_socket
    host = socket.gethostname()
    port = 5000
    client_socket = socket.socket()
    client_socket.connect((host, port))
```

```
def create_server_socket():
    global host,port,server_socket,conn,address
    host = socket.gethostname()
    port = 5000
    server_socket = socket.socket()
    server_socket.bind((host,port))
    server_socket.listen(1)
    conn, address = server_socket.accept()
```

Una volta stabilita la connessione tra i due nodi può iniziare la vera e propria comunicazione che prima però presuppone la creazione di una *chiave condivisa* tramite il protocollo di *Diffie - Hellman* da utilizzare per cifrare i messaggi. Tutto ciò verrà descritto dettagliatamente nella sezione successiva.

3 Creazione della chiave condivisa mediante Diffie - Hellman

Per la creazione della chiave condivisa i due utenti utilizzano l'algoritmo di *Diffie - Hellman*. Dati due utenti *Alice* e *Bob* i passi da seguire sono i seguenti:

- *Alice* e *Bob* concordano un numero primo p abbastanza grande e un intero $g \in \{1, \ldots, p\}$. Sia p che g sono valori non confidenziali e possono essere dunque scambiati in chiaro;
- *Alice* sceglie un numero random $a \in \{1, \ldots, p-1\}$ (*chiave privata*) e manda a *Bob* il seguente valore detto *chiave pubblica*:

$$A = g^a \mod p$$

- *Bob* sceglie un numero random $b \in \{1, \ldots, p-1\}$ (*chiave privata*) e manda ad *Alice* il seguente valore detto *chiave pubblica*:

$$B = g^b \mod p$$

- *Alice* avendo ricevuto da *Bob* la *chiave pubblica* B calcolerà la *chiave condivisa* come:

$$k_{AB} = B^a = \left(g^b\right)^a = g^{ab} \mod p$$

- *Bob* avendo ricevuto da *Alice* la *chiave pubblica* A calcolerà la *chiave condivisa* come:

$$k_{AB} = A^b = (g^a)^b = g^{ab} \mod p.$$

Alice e *Bob* ottengono il medesimo valore k_{AB} che sarà dunque la chiave che potranno utilizzare per scambiarsi in modo sicuro i loro messaggi.

Un *avversario* potrebbe conoscere p e g e le chiavi pubbliche A e B, ma non conoscendo le chiavi private a e b non può calcolare in modo efficiente g^{ab}. Tuttavia tale tecnica è vulnerabile ad attacchi di tipo *meet in the middle*.

Per rendere le operazioni di elevamento a potenza più rapide viene utilizzato l'algoritmo *quadra e moltiplica* il quale permette di calcolare la potenza

di un numero per esponenziazioni successive. Considerato di voler calcolare n^p, questo lo si può esprimere come prodotto di potenze di n elevate a potenze di 2 come segue:

$$n^p = \begin{cases} n \cdot (n^2)^{\frac{p-1}{2}} & \text{se } p \text{ è dispari} \\ (n^2)^{\frac{p}{2}} & \text{se } p \text{ è pari} \end{cases}.$$

Ovvero si costruisce la successione $k = \{k_0, \ldots, k_m\}$ dove $k_0 = n$ e si procede nella successione calcolando k_i come $(k_{i-1})^2$ se i è pari altrimenti $k_i = k_{i-1} \cdot k_0$. Questo algoritmo può essere utilizzato anche quando si effettuano operazioni in aritmetica modulare. Quello che si deve fare ad ogni passo k_i è ridurre il risultato modulo in modulo. Il codice dell'algoritmo è mostrato di seguito:

```
def exponentiation(bas, exp,N):
  if (exp == 0):
    return 1
  if (exp == 1):
    return bas % N

  t = exponentiation(bas, int(exp / 2),N)
  t = (t * t) % N

  if (exp % 2 == 0):
    return t
  else:
    return ((bas % N) * t) % N
```

I parametri p e g - che sono valori che vanno concordati - tra i due nodi vengono scelti dal *Client*. Per quanto concerne p verrà creato un numero primo formato da 1024 bit e poi $g \in \{1, \ldots, p\}$. Il *Client* una volta scelti tali valori li invia al *Server*: Di seguito la funzione utilizzata dal *client* per la creazione dei due parametri sopra menzionati:

```
def create_p_g():
    while True:
        p=getPrime(SIZE_PG, randfunc=get_random_bytes)
        if p !=0:
            break

    while True:
        g=getRandomInteger(SIZE_PG,randfunc=get_random_bytes)
        if g>=1 and g<=p and g.bit_length()==SIZE_PG:
            break
```

A questo punto inizia la costruzione della *chiave condivisa*. Per poter farlo *Client* e *Server* devono creare le rispettive *chiavi private* che avranno dimensione 512 bit. La funzione usata per la creazione della *chiave privata* è la seguente:

```
def create_private_key(p,g):
    while True:
        private_key=getRandomInteger(SIZE_PK)
        if(private_key >=1 and private_key <= p-1):
            break
    return private_key
```

A questo punto il *Client* crea la *chiave pubblica* mediante la seguente funzione che per velocizzare l'operazione di elevamento a potenza utilizza l'algoritmo *quadra e moltiplica* descritto in precedenza:

```
def create_public_key(g,p,a):
    public_key=exponentiation(g,a,p)
    return public_key
```

A questo punto invia la *chiave pubblica* al *Server* e attende che gli venga inviata quella del *Server* per poter calcolare la *chiave condivisa*. Quest'ultima viene calcolata mediante la seguente funzione:

```
def create_shared_key(x,y,p):
    shared_key=exponentiation(x,y,p)
    return shared_key
```

Il *Server* non appena riceve la *chiave pubblica* dal *Client* gli invia la propria e dopo procede con il calcolo della *chiave condivisa*. Il *Server* inoltre invia al *Client* un username che sceglie randomicamente da una lista prefissata di nomi.

Di seguito il codice sorgente del *Client*:

```
def init_comm():
    global a,A,B,K,user
    print("
    ==================================================")
    print("********* CLIENT *********")
    print("
    ==================================================")
    create_client_socket()
    parameters=dh.create_p_g()
    set_p_g_parmeters(parameters)
    print("Send to Server p,g;")
    print_parameters()
    client_socket.send(pickle.dumps(parameters))
    a=dh.create_private_key(p,g)
    print("Private Key", a)
    A=dh.create_public_key(g,p,a)
    print("Public Key",A)
    print("Sending Public Key")
    client_socket.send(str(A).encode())
    while True:
        print("Waiting the server public key:")
        B = client_socket.recv(512).decode()
        B = int (B)
        print("I recivied:",B)
        user=client_socket.recv(512).decode()
        print("I recivied:",user)
        break
    K=dh.create_shared_key(B,a,p)
    print("Shared Key",K)
    K=AES.apply_sha256(K)
    print("Shared Key (Byte)",K)
```

Di seguito il codice sorgente del *Server*:

```
def init_comm():
    global b,A,B,K,name
    names=["Alice","Bob","Claudia","Giuseppe"]
    print("
    ==================================================")
    print("********* SERVER *********")
    print("
    ==================================================")
    create_server_socket()
    while True:
        print("Waiting p,g from Client...")
        data = conn.recv(2048)
        parameters = pickle.loads(data)
        print("I recivied:")
        set_p_g_parmeters(parameters)
        print_parameters()
        break
    b=dh.create_private_key(p,g)
    print("Private Key", b)
    B=dh.create_public_key(g,p,b)
    print("Public Key",B)
    print("Connection from: " + str(address))
    while True:
        print("Waiting the client publick key:")
        A = conn.recv(512).decode()
        A = int (A)
        print("I recivied :",A)
        print("Sending Public Key")
        conn.send(str(B).encode())
        name=choice(names)
        print("Send my name to client...")
        conn.send(name.encode())
        break
    K=dh.create_shared_key(A,b,p)
    print("Shared Key",K)
    K=AES.apply_sha256(K)
    print("Shared Key (Byte)",K)
```

La *chiave condivisa* prodotta avrà una lunghezza in bit di 1024 e dato che il cifrario che si vuole utilizzare è *AES-256* che richiede in input una chiave di 256 bit si applica sulla chiave l'algoritmo *SHA-256* in modo tale che la lunghezza in bit della *chiave condivisa* da 1024 passi a 256 bit.

SHA-256 è una funzione hash che utilizza una *compression function* con schema *Davies-Meyer* e con un block cipher *SHACAL-2* che usa chiavi da 512 bit e blocchi da 256 bit. Le *compression function* vengono concatenate

seguendo il paradigma di *Merkle-Damgard* in questo modo si ottiene una funzione applicabile a messaggi/input di qualunque lunghezza. L'obbiettivo del paradigma di *Merkle-Demgard* è quello di ottenere delle funzioni hash collision resistant per lunghi messaggi, a partire da funzioni hash collision resistant per messaggi brevi dette *Compression Function*. Una *compression function* è una funzione hash $h : T \times X \to T$ che richiede in input un tag e un blocco di messaggio e restituisce in output un altro tag. Dato un messaggio di L blocchi, se le *compression function* vengono concatenate si ottiene una funzione hash definita come segue:

$$H : X^{\leq L} \to T$$

Gli output delle H_i prendono il nome di *Chaining Variable*. La prima, H_0 corrisponde ad un *Initialization Vector*, l'ultima $H[L-1]$ sarà il tag finale. Nel blocco finale verrà aggiunto un *Padding Block* composto da una parte fissata e da una parte contenente la lunghezza del messaggio. Lo schema di *Davies - Mayer* utilizza un block cipher $E : K \times \{0,1\}^n \to \{0,1\}^n$. La *compression function* sarà dunque definita come segue:

$$h(H, m) = E(m, H) \oplus H$$

Si noti che per il cifrario la chiave sarà il blocco del messaggio, mentre per il messaggio sarà la *chaining variable*. La funzione utilizzata sia dal *Client* che dal *Server* per fare ciò è la seguente:

```
def apply_sha256(key):
    key=str(key)
    h = SHA256.new()
    h.update(key.encode())
    k = h.digest()
    return k
```

4 Comunicazione sicura con cifratura dei messaggi tramite AES

Una volta che *Client* e *Server* hanno stabilito la chiave comune con cui è possibile cifrare/decifrare i loro messaggi inizia il vero è proprio scambio di messaggi. Inoltre nell'interfaccia grafica del programma usato dal *Client*, la chiave negoziata, sarà visualizzata a schermo. Come cifrario è stato scelto *AES-256* in modalità *EAX* che applica lo schema *Encrypt - then - MAC* ed utilizza:

- *CTR*: per quanto concerne l'encryption;
- *CMAC*: per l'integrità.

AES lavora con blocchi di 128 bit e in questo caso con una chiave da 256 bit. L'algoritmo si basa su una *substitution-permutation network* che è una rete a più livelli che si basa sui seguenti principi:

- *principio di diffusione*: un cambiamento ad un bit dell'input causa mediamente un cambiamento della meta dei bit di output;
- *principio di confusione*: non deve mai esserci una correlazione tra la chiave e l'output ottenuto.

Una rete di questo tipo è formata da un certo numero di round e ad ognuno di essi si applica una *round key* k_i che è un'estensione della chiave k. A sua volta ogni round è formato da un layer di sostituzione e da uno di permutazione. L'input prima di attraversare i vari round viene messo in XOR con la prima *round key*. Per ogni round l'input viene spezzato in un certo numero di parti che vengono fornite in input al layer di sostituzione il quale è formato da altrettanti *S-Box*. Gli output delle *S-box* verranno riunificati e passati attraverso il *layer di permutazione* che scambierà opportunamente i bit tra di loro. Infine l'output viene messo in *XOR* con la *round key*. Queste operazioni si ripetono ogni round e l'output dell'ultimo blocco sarà il messaggio cifrato. In *AES-256* vengono utilizzati 14 round ed ogni blocco viene inserito in una matrice 4×4 byte, il *layer di sostituzione* di ogni round viene implementato mediante una funzione chiamata *ByteSub* che per ogni byte della matrice applica una sostituzione secondo quanto riportato in una lookup table, il *layer di permutazione* viene implementato mediante due funzioni che agiscono una dopo l'altra: *ShiftRow* che non modifica la prima riga della matrice, ma ruota le rimanenti di $1, 2, 3$ posizioni verso sinistra rispettivamente e *MixColumn* che applica sulle colonne della matrice una combinazione lineare invertibile

che è mappata all'interno di una matrice. Nell'ultimo round non si applica la funzione *MixColumn*. Alla fine di ogni round dopo la funzione *MixColumn* l'output viene messo in XOR con la *round key*.

Il cifrario utilizza come modalità di encryption la *CTR*. In questa modalità ogni blocco cifrato c_i si otterrà effettuando lo XOR tra il blocco di messaggio $i - esimo$ e l'output di un *PRF* sicuro che prende in input un *Iinizialization Vector (IV)* che verrà incrementato di 1 ad ogni blocco. L'IV essendo random deve essere incluso nel ciphertext in modo tale che in fase di decifratura possa essere utilizzato. Se si usa l'IV deve essere cambiato ad ogni messaggio per mantenere l'impredicibilità. Per la realizzazione di questo software è stata utilizzata la variante *nonce based*. In questo caso il contatore che si utilizza come input del *PRF* è formato da due parti: un *nonce* impredicibile e un contatore che riparte da 0 ad ogni nuovo messaggio. Se si suppone che le due parti abbiano la stessa lunghezza e questa sia $N/2$ il nonce dovrà essere cambiato dopo la codifica di $2^{N/2}$ blocchi.

Per l'integrità è stato utilizzato un *CMAC* che è una variante dell'*ECBC-MAC*. Quest'ultimo si basa su un *PRP* sicuro ed una costruzione simile a quella di un cifrario in modalità *CBC* con la differenza che l'output dell'ultimo blocco viene cifrato utilizzando un'altra chiave k_1 indipendente dalla prima. Nel *CMAC* vengono usate tre chiavi:

- k per criptare i blocchi;
- k_1 derivata da k utilizzata per criptare i blocchi di messaggi non multipli della lunghezza del blocco;
- k_2 derivata da k utilizzata nell'ultimo blocco di messaggi multipli della lunghezza del blocco.

Nell'ultimo blocco oltre al semplice XOR tra il blocco di messaggio e l'output del blocco precedente, si effettua un'ulteriore XOR con k_1 o k_2.

La funzione utilizzata per l'*encryption* è la seguente:

```
1 def encrypt(key,msg):
2     cipher = AES.new(key, AES.MODE_EAX)
3     nonce = cipher.nonce
4     print("Nonce",nonce)
5     ciphertext, tag = cipher.encrypt_and_digest(msg.encode('
    utf-8'))
6     return nonce, ciphertext, tag
```

La funzione utilizzata per la *decryption* è la seguente:

```
def decrypt(nonce, ciphertext, tag,key):
    print("Nonce",nonce)
    cipher = AES.new(key, AES.MODE_EAX, nonce=nonce)
    plaintext = cipher.decrypt(ciphertext)
    try:
        cipher.verify(tag)
        print("The message is authentic")
        return plaintext.decode('utf-8')
    except:
        print("Key incorrect or message corrupted")
        return False
```

La comunicazione viene avviata dal *Client*. La funzione che gestisce l'invio dei messaggi nel *Client* è la seguente:

```
def send():
    global client_socket,scrollbar,root
    nonce, ciphertext, tag=AES.encrypt(K,edit_text.get())
    print("Ciphertext",ciphertext)
    client_socket.send(pickle.dumps([nonce,ciphertext,tag]))
    listbox.insert(END, name+":")
    listbox.insert(END, "Plaintext: "+ edit_text.get())
    listbox.insert(END, "Ciphertext: "+ str(ciphertext))
    listbox.see('end')
    if edit_text.get() == "Bye":
        edit_text.delete(0, END)
        print("End of comunication !")
        listbox.insert(END, "Fine comunicazione, chiudi la
   finestra !")
        listbox.insert(END, "
   **************************************************")
        listbox.see('end')
        sleep(5)
        root.destroy()
    else:
        edit_text.delete(0, END)
        listbox.insert(END, "
   **************************************************")
        listbox.see('end')
```

.

mentre quella per la ricezione dei messaggi:

```
def recv():
    global root,scrollbar,client_socket
    while True:
        data = client_socket.recv(1000000)
        try:
            aes_data = pickle.loads(data)
        except:
            break
        print("I recivied:",aes_data[1])
        plaintext = AES.decrypt(aes_data[0],aes_data[1],
    aes_data[2],K)
        print("Plaintext:",plaintext)
        listbox.insert(END, user+":")
        listbox.insert(END, "Ciphertext: "+ str(aes_data[1]))
        listbox.insert(END, "Plaintext: "+ plaintext)
        listbox.insert(END, "
    **************************************************")
        edit_text.delete(0, END)
        listbox.see('end')
        if plaintext == "Bye":
            print("End of comunication !")
            listbox.insert(END, "Fine comunicazione, chiudi
    la finestra !")
            listbox.see('end')
            break
```

.

La funzione che utilizza il *Server* per la ricezione è l'invio dei messaggi è la seguente:

```
def comm():
    while True:
        print("Waiting for a message...")
        data = conn.recv(1000000)
        aes_data = pickle.loads(data)
        print("I recivied",aes_data[1])
        plaintext = AES.decrypt(aes_data[0],aes_data[1],
    aes_data[2],K)
        print("Plaintext:",plaintext)
        if plaintext == "Bye":
            print("End of comunication !")
            break
        else:
            plaintext=create_random_text()
            if plaintext == "Bye":
                nonce, ciphertext, tag=AES.encrypt(K,
    plaintext)
                print("Ciphertext",ciphertext)
                conn.send(pickle.dumps([nonce,ciphertext,tag
    ]))
                print("End of comunication !")
                break
            else:
                nonce, ciphertext, tag=AES.encrypt(K,
    plaintext)
                print("Ciphertext",ciphertext)
                conn.send(pickle.dumps([nonce,ciphertext,tag
    ]))
```

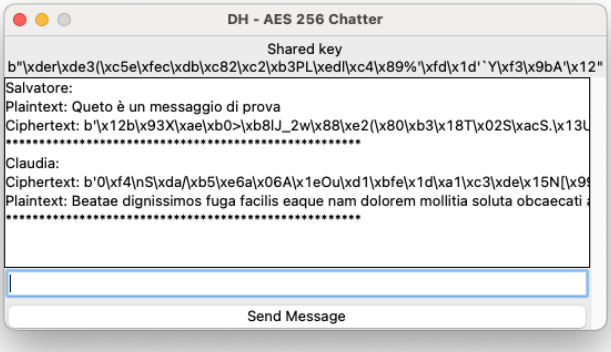
.

Il *Server* risponderà ai messaggi del *Client* (che è l'utente umano che utilizza il programma) generando delle frasi random. La comunicazione verrà interrotta quando una delle due parti invierà la stringa *Bye*.

5 Esempio di funzionamento

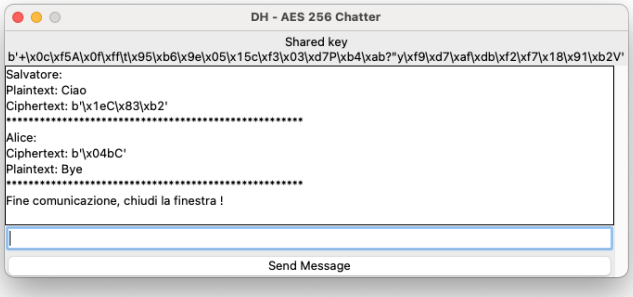
Dopo che le due parti avranno negoziato la chiave condivisa da utilizzare per proteggere i loro messaggi, nella *GUI* del *Client* verrà visualizzata la *chiave condivisa*:

dopo di ciò può avvenire la vera e propria conversazione, di seguito un esempio:



Nella chat sono stampati sia ciphertext che plaintext soltanto per verificare che che tutti i passi vengano effettuati correttamente.

Di seguito un esempio di fine della comunicazione:



6 Conlusioni

È stato realizzato un software per la comunicazione sicura tra due nodi che sono stati simulati mediante due socket, una per il client ed una per il server. La chiave utilizzata per la protezione dei messaggi è stata stabilita utilizzando il protocollo *Diffie-Hellman* e come cifrario è stato utilizzato *AES-256* in modalità *EAX*. Successivamente il software potrebbe modificato opportunamente in modo che il secondo utente non sia simulato ma sia anch'esso reale ed inoltre può essere approfondito l'aspetto inerente l'attacco *Meet in the middle* a cui è vulnerabile l'algoritmo di *Diffie-Hellman*.