universidade de aveiro

SECURE INSTANT MESSAGING SYSTEM

Relatório

Segurança

Departamento de Eletrónica, Telecomunicações e Informática

6 de novembro de 2016

Professores:

João Paulo Barraca

André Zúquete

Autores:

Miguel Oliveira nº 72638

Tiago Henriques nº 73046

universidade de aveiro

Índice

universidade de aveiro

Introdução

As aplicações de troca de mensagens são muito usadas nas comunicações atuais, através das quais é partilhada informação, detalhes de negócios e até informação privada. Portanto, possíveis falhas de segurança podem ocorrer e podem permitir a um atacante obter compensações monetárias ou até comprometer um sistema em desenvolvimento.
O objetivo do projeto a ser desenvolvido no âmbito da cadeira de Segurança, é permitir a troca de mensagens entre utilizadores através de um sistema seguro composto por um Servidor e múltiplos Clientes que trocam mensagens entre si. Tal sistema tem diversas características comuns com um já existente, Signal, que é uma aplicação para Android ou IOS com um sistema de troca de mensagens instantâneas cifradas, logo seguras. Foi usada a linguagem de programação 'python' e a biblioteca 'cryptography'.

Estrutura:

Existem vários processos que têm de ser suportados no projeto.

- **Connect to Server:** Cliente envia uma mensagem *CONNECT* para o servidor e mais mensagens deste tipo serão trocadas até que o tipo de cifras seja acordado e uma sessão válida segura seja estabelecida.

- **List clients connected:** Cliente envia uma mensagem *LIST* para o servidor. O servidor responde com uma mensagem do mesmo tipo contendo uma lista de clientes disponíveis.

- **Client-to-client connection:** Cliente envia uma mensagem *CLIENT-CONNECT* que é encaminhada pelo servidor para o cliente especificado. Novamente, o tipo de cifras têm de ser acordadas e uma sessão válida e segura tem de ser estabelecida.

- **Client-to-client disconnection:** Cliente envia uma mensagem *CLIENT-DISCONNECT* que é encaminhada pelo servidor para o cliente especificado.

- **Client-to-client communication:** Cliente envia uma mensagem *CLIENT-COM* que é encaminhada pelo servidor para o cliente especificado pelo primeiro cliente.

Cipher Spec

De modo a definir os algoritmos criptográficos a usar no “cipher spec”, foi usada a seguinte notação, [XXXX - AAAA_YY - BBBB], em que:

XXXX - Algoritmo usado na troca de chaves:

- ECDH

AAAA_YY - Algoritmo de cifra simétrica mais o respetivo módulo representado no YY:

- AES_CTR
- AES_OFB

BBBB - Algoritmo de “hash” usado juntamente com o HMAC:

- SHA256
- SHA384

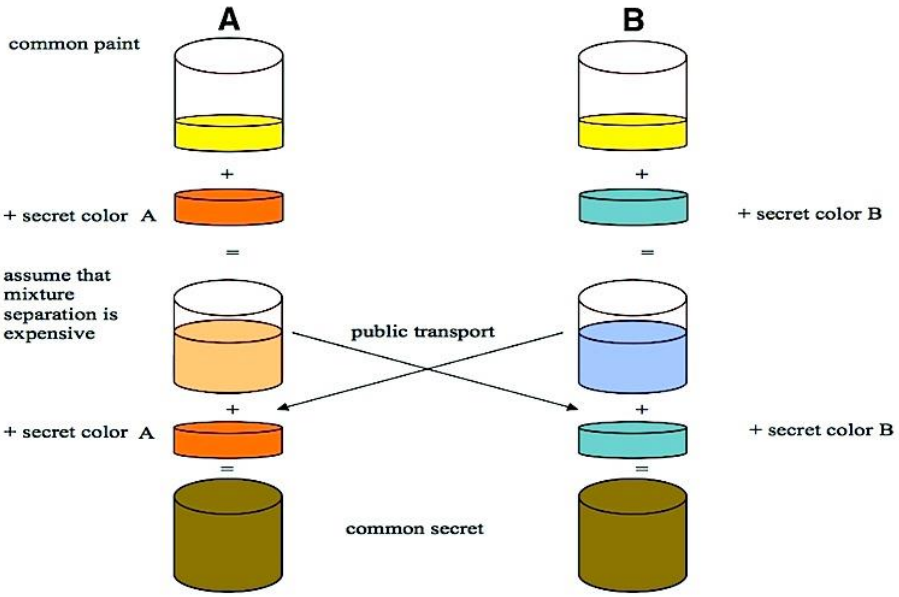
“Cipher Suite” suportado: [‘ECDHE-AES128_CTR-SHA256’, ‘ECDHE-AES256_CTR-SHA384’]

Nota: Nesta fase do trabalho não foi tido em conta nenhum algoritmo de autenticação (assinatura), como por exemplo, o RSA.

universidade de aveiro

Cliente-Servidor

Connect



Neste processo, foi usado um procedimento baseado no protocolo de criptografia que estabelece uma comunicação segura sobre a Internet, o TLS (Transport Layer Security). Ou seja, trata-se de fazer um "*handshake*" entre o cliente e servidor para iniciar a ligação entre os mesmos.
Para realizar o *handshake* entre duas entidades que não possuem conhecimento uma da outra, foi utilizado Diffie–Hellman (DH) para a troca de chaves públicas sobre um canal inseguro, para que fim do processo, ambas a entidades possuam um segredo partilhado, e deste modo, estejam preparadas para trocar mensagens usando um esquema de cifra de chave simétrica.

Processo de Handshake:

Este processo está dividido em 4 fases (onde cada fase é incrementada uma unidade após o envio da mensagem)

- **Fase 1**

  Cliente envia para o servidor:

1. Cipher spec suportado;
2. O nome e o ID.

- **Fase 2**

  Servidor:

1. Compara o cipher spec recebido do cliente com o seu suportado, caso haja algum *match*, enviará o cipher spec de novo para o Cliente;
2. É gerada uma chave pública-privada com o cipher spec escolhido (ECDH).

  Cliente:

1. Recebe o cipher spec enviado pelo servidor;
2. Gera uma chave pública-privada com o cipher spec recebido (ECDH);
3. Gera um número random para prevenir ataques de repetição;
4. Envia para o Servidor o valor random e a chave pública geradas dentro do campo 'data' da mensagem.

```
msg['data'] = { 'public_key': self.public_key,
                'client_random': base64.b64encode(self.rand_client)
              }
```

- **Fase 3**:

  Servidor:

1. Recebe a chave pública enviada pelo cliente;
2. Recebe o valor random enviado pelo cliente;

3. Gera um valor random com a mesma função do random do cliente;
4. Gera o segredo comum (master_secret) que resulta na combinação da chave pública recebida com a sua chave privada;
5. Gera o segredo comum final que resulta da combinação do segredo comum (master_secret) com a soma dos dois valores random gerados pelo cliente e servidor.
6. Envia para o Cliente no campo 'data' da mensagem a sua chave pública e o valor random criado.

```
self.master_secret = keys.generate_master_secret(self.server_public_key, self.private_key)
```

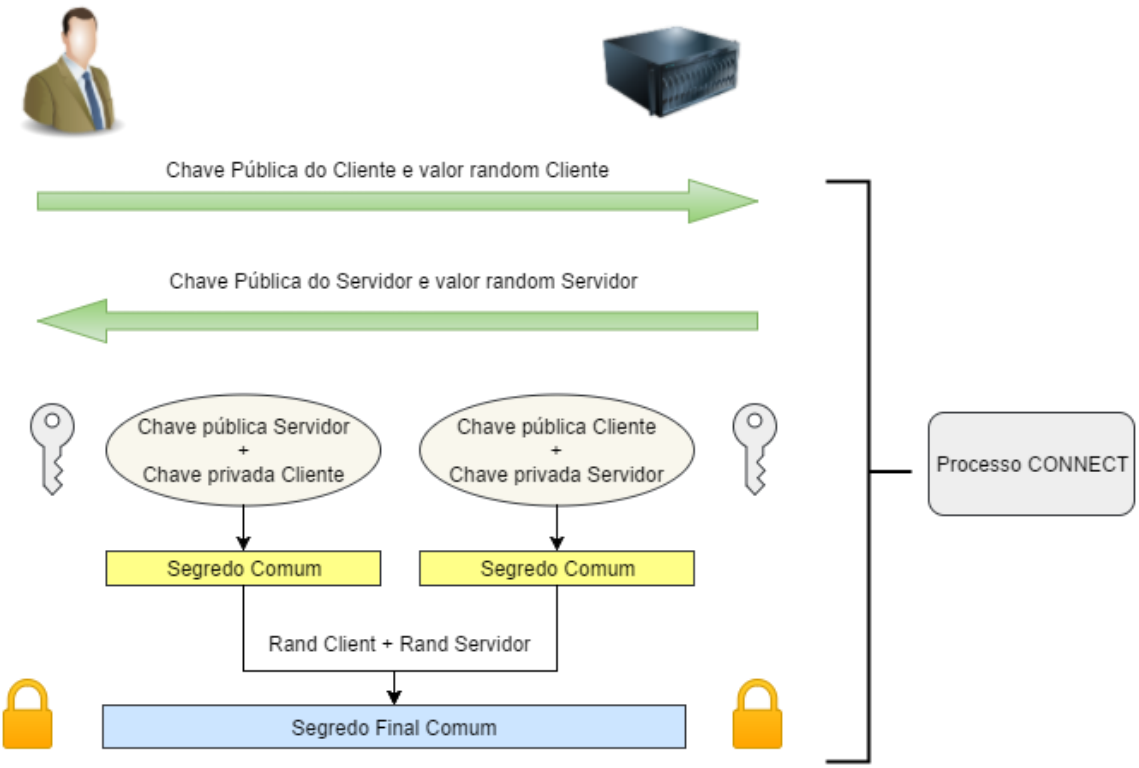
```
self.final_master_secret =
keys.generate_final_master_secret(self.master_secret,self.rand_client + self.rand_srv)
```

```
def generate_final_master_secret(master_secret, rand_client_server):
        backend = default_backend()
        info = "hkdf-example"
        hkdf = HKDF(
                algorithm=hashes.SHA256(),
                length=32,
                salt=rand_client_server,
                info=info,
                backend=backend)
        key = hkdf.derive(master_secret)
        return key
```

- **Fase 4**

1. Recebe a chave pública enviada pelo servidor;
2. Recebe o valor random enviado pelo servidor;
3. Gera o segredo comum (master_secret) que resulta na combinação da chave pública recebida com a sua chave privada;
4. Gera o segredo comum final, que resulta da combinação do segredo comum (master_secret) com a soma dos dois valores random (o gerado anteriormente + o recebido pelo servidor).

Assim, no final das 4 fases, as duas entidades possuem um segredo em comum partilhado, estando, por fim, estabelecida a sessão.
Agora sim é possível enviar mensagens do tipo "secure" cifradas com o segredo final comum, tendo em conta as outras primitivas do cipher spec, como por exemplo, o AES (algoritmo responsável pelo encrypt/decrypt) e SHA.


Chave Pública do Cliente e valor random Cliente
Chave Pública do Servidor e valor random Servidor
Chave pública Servidor
+
Chave privada Cliente
Chave pública Cliente
+
Chave privada Servidor
Processo CONNECT
Segredo Comum
Segredo Comum
Rand Client + Rand Servidor
Segredo Final Comum

Secure

As mensagens "secure" só poderão ser trocadas após o estabelecimento de uma sessão segura entre cliente e servidor e são utilizadas para trocar conteúdos cifrados entre eles. Sempre que um dos dois pretenda enviar uma mensagem deste tipo, deverá seguir um conjunto de operações:

1. Calcular o novo segredo comum partilhado a partir do valor público antigo da outra entidade e do seu novo valor privado;
2. Gerar um novo valor de random, um valor aleatório (32 bytes);
3. Calcular o novo segredo final comum partilhado a partir do novo segredo comum partilhado e dos novos valores de random, usando KDF;
4. Gerar uma nova chave derivada a partir do novo segredo final comum partilhado e do novo valor de random gerado, utilizando HMAC;
5. Utilizando a chave derivada, cifrar a mensagem, de acordo com o algoritmo indicado na "cipher specs";
6. Usar um função de digest para gerar a síntese da mensagem cifrada utilizando HMAC com a função de hash indicada na "cipher specs" e utilizando a chave derivada, de modo a testar a integridade;
7. Incluir no payload a mensagem cifrada (campo 'ciphertext') e o valor da chave pública atualizada (campo 'public_key') e incluir no sa-data o valor random (campo 'random'), o valor da síntese (campo 'hash'), e o iv (campo 'iv') que verifica se o mesmo valor está a ser usado nos processos de cifra e decifra.

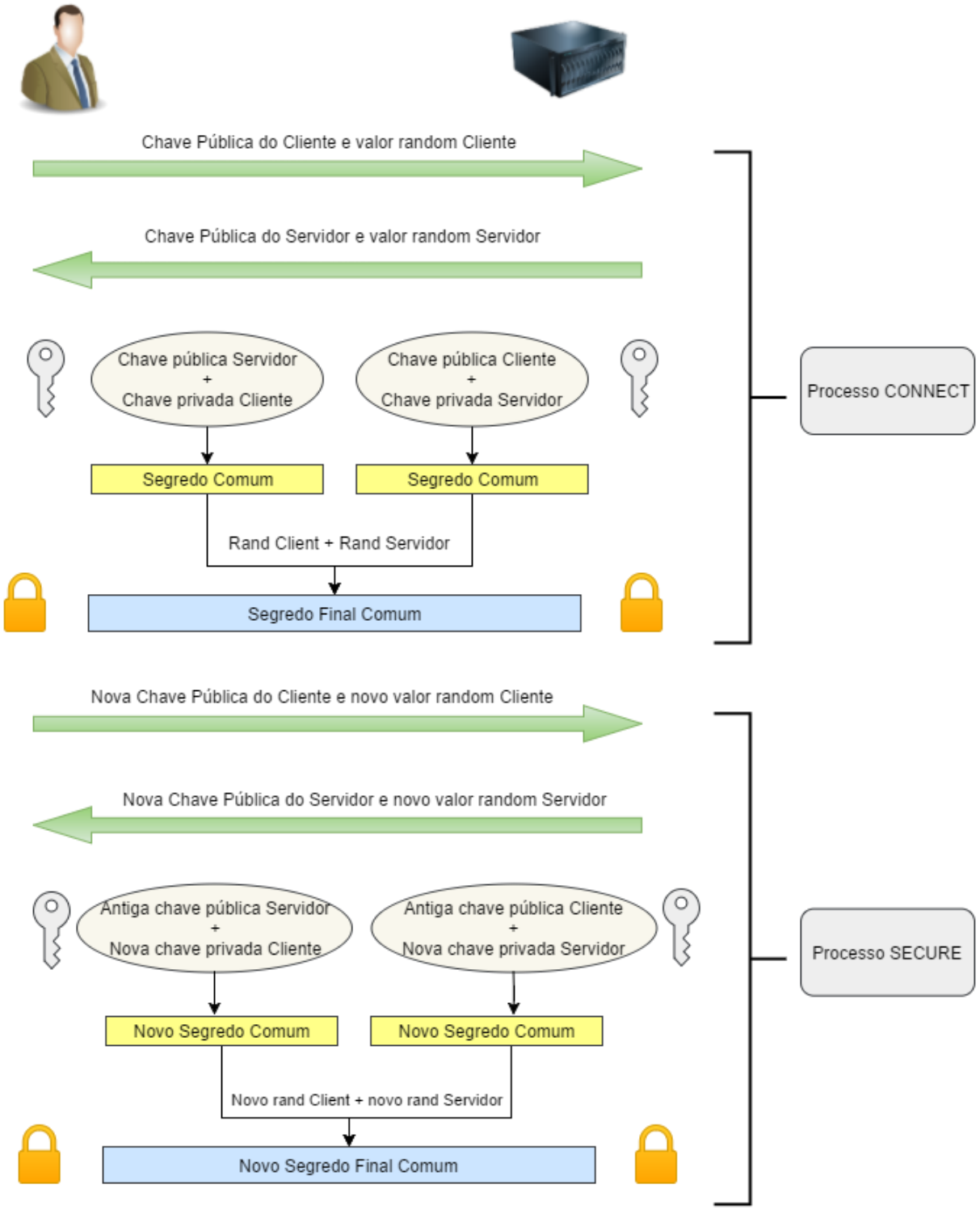
```
    #This is a secure message
def secure_send(self, msg_json):
            """ Process a secure message from server """
            # Cipher: new private key + old public key
            msg = {'type': 'secure', 'sa-data':{}, 'payload':{}}
            new_public_key, self.new_private_key = keys.verify_cipher_suite(self.cipher_suite)

            new_master_secret = keys.generate_master_secret(self.server_public_key, self.new_private_key)
            self.rand_client = os.urandom(32)
            new_final_master_secret = keys.generate_final_master_secret(new_master_secret, self.rand_client + self.rand_srv)
            ciphertext , client_iv = keys.encrypt(self.cipher_suite,new_final_master_secret,msg_json)

            msg['sa-data'] = {'hash': base64.b64encode(keys.message_authentication(self.cipher_suite, new_final_master_secret, ciphertext)),
                    'iv': base64.b64encode(client_iv), 'random':base64.b64encode(self.rand_client)}
            msg['payload'] = {'ciphertext': base64.b64encode(ciphertext),'public_key': new_public_key}

            self.sock.send(msg)
```

No processo de “receive” da mensagem, será necessário fazer o processo "inverso" para decifrar a mensagem, o que implica:

1. Calcular o novo segredo comum partilhado a partir do valor antigo público recebido da outra entidade e do seu novo valor privado gerado;
2. Gerar o novo segredo final comum partilhado a partir do novo segredo comum partilhado e dos novos valores de random, usando KDF;
4. Calcular a partir da função de digest, o valor do hash da mensagem cifrada utilizando HMAC e parte da chave derivada e verificar se esta é igual ao valor de hash recebido, garantindo deste modo a integridade da mensagem;
5. Utilizando o novo segredo final comum e o valor de iv recebido, decifrar a mensagem, de acordo com o algoritmo indicado na "cipher specs";
6. Guardar internamente o novo valor público da outra entidade juntamente com o valor de random;
7. Dependendo do tipo (campo ‘type’) da mensagem, o conteúdo do payload será enviado como argumento para a função responsável pelo tratamento desse tipo.

```
def secure_receive(self, msg_json):
            """ Process a secure message from server """
            # Decipher: public key received + old private key
            ciphertext = base64.b64decode(msg_json['payload']['ciphertext'])
            public_key = keys.deserialize_public_key(base64.b64decode(msg_json['payload']['public_key']))
            server_hash = base64.b64decode(msg_json['sa-data']['hash'])
            server_iv = base64.b64decode(msg_json['sa-data']['iv'])
            server_random = base64.b64decode(msg_json['sa-data']['random'])
            new_master_secret = keys.generate_master_secret(public_key, self.new_private_key)
            new_final_master_secret = keys.generate_final_master_secret(new_master_secret,self.rand_client + server_random)
            client_hash = keys.message_authentication(self.cipher_suite, new_final_master_secret, ciphertext)
            self.server_public_key = public_key
            self.rand_srv = server_random

            msg_json['payload'] = keys.decrypt(self.cipher_suite,new_final_master_secret,ciphertext,server_iv)

            if (server_hash != client_hash):
                    logging.warning('Client Hash and Server Hash do not match!')

            if msg_json['payload']['type'] == 'list':
                    self.receive_list_clients(msg_json['payload'], self.id)

            elif msg_json['payload']['type'] == 'client-connect':
                    self.receive_client_connect(msg_json['payload'])

            elif msg_json['payload']['type'] == 'client-disconnect':
                    self.receive_client_disconnect(msg_json['payload'])

            elif msg_json['payload']['type'] == 'client-com':
                    self.receive_client_com(msg_json['payload'])

            elif msg_json['payload']['type'] == 'ack':
                    self.receive_client_ack(msg_json['payload'])
```

Logo, sempre que uma mensagem é enviada, a entidade que envia vai gerar novos valores de chaves público-privadas. Isto permite que o segredo comum partilhado esteja constantemente a ser modificado, uma vez que as mensagens entre cliente e servidor são trocadas repetidamente entre si, ou seja, depois de um ciclo completo de troca de informação, o novo segredo comum partilhado é completamente independente do par de chaves usadas na troca das mensagens anteriores, garantindo o “Forward secrecy” e o “Backward secrecy”.


Chave Pública do Cliente e valor random Cliente
Chave Pública do Servidor e valor random Servidor
Chave pública Servidor
+
Chave privada Cliente
Chave pública Cliente
+
Chave privada Servidor
Segredo Comum
Segredo Comum
Rand Client + Rand Servidor
Segredo Final Comum
Processo CONNECT
Nova Chave Pública do Cliente e novo valor random Cliente
Nova Chave Pública do Servidor e novo valor random Servidor
Antiga chave pública Servidor
+
Nova chave privada Cliente
Antiga chave pública Cliente
+
Nova chave privada Servidor
Novo Segredo Comum
Novo Segredo Comum
Novo rand Client + novo rand Servidor
Novo Segredo Final Comum
Processo SECURE

Cliente - Cliente:

Todas as mensagens trocadas entre clientes são “rotuladas” com o tipo Secure, ou seja, usam a sessão segura estabelecida anteriormente, onde passarão sempre pelo Servidor, que tratará de decifrar o conteúdo da mensagem, e reencaminhar para o destino correto.

Client-Connect

Para estabelecer uma ligação entre clientes, foi implementada uma classe Client, onde serão guardados todos os dados relacionados com o respetivo cliente como: o seu ID, cipher spec suportado, os seus valores públicos-privados, segredo comum, etc.
Um cliente pode ver os clientes disponíveis para conexão através do comando [‘list’] que listará todos os clientes aptos para comunicação. Quando é estabelecida uma conexão entre dois ou mais clientes ([‘connect-id’]), é chamada a função addClient que adicionará a uma lista (clients), o cliente destino. Essa lista serve para listar todos os clientes conectados com esse cliente e posteriormente, será usada no Client-disconnect.

Assim, quando dois clientes pretendem comunicar, necessitam de estabelecer uma ligação segura entre eles, sessão que usará o mesmo mecanismo utilizado na ligação do Cliente-Servidor, diferenciando apenas, no facto, da mensagem enviada ser do tipo “client-connect”.

Client-Com

A comunicação entre dois clientes, apenas pode acontecer após ambos se terem conectado com sucesso.

Após a conexão, o processo de comunicação, é semelhante ao realizado no Secure, porém, apenas a entidade que envia a mensagem, ou seja, a entidade que cifra a mensagem, é que vai gerar novos pares públicos-privados. Neste processo, a entidade que recebe a mensagem, limita-se a receber a chave pública e o novo random, necessários para realizar a decifra da mensagem recebida.

Visto que a troca de mensagens entre clientes é feita sempre pela sessão segura criada no início, ou seja, é sempre encapsulada como mensagem do tipo ‘secure’, podemos garantir que ao fim do ciclo completo, a nova chave partilhada é independente da chave partilhada trocada nas mensagens anteriores.

Client-Disconnect

As mensagens do tipo ‘client-disconnect’ tem o intuito de terminar uma sessão segura entre dois clientes. Para isso, foi usada a função delClient, que terminará a sessão anteriormente estabelecida entre ambos os clientes. Portanto, a lista de clientes conectados de cada cliente (clients) será atualizada, sendo retirado o id do cliente da sessão terminada.

Client-Ack

As mensagens do tipo ‘client-ack’ são também encapsuladas no Secure, tendo apenas a funcionalidade de fornecer a confirmação ao cliente emissor de que a mensagem foi recebida com sucesso no cliente recetor.

Conclusão

O projeto a desenvolver tinha de suportar algumas funcionalidades de segurança e foi concluído que os objetivos mais relevantes de segurança foram assegurados.
- Message confidentiality, integrity and Authentication: As mensagens são enviadas cifradas juntamente com uma função de digest que é verificada no destino. A fase de autenticação só será implementada na segunda parte do projeto.
- Multiple Cipher Support: Em todos os processos (Cliente-Servidor e Cliente-Cliente), existe a fase de negociação e posterior acordo do "Cipher Spec", contendo dois métodos que foram implementados (['ECDHE-AES128_CTR-SHA256', 'ECDHE-AES256_CTR-SHA384'])
- Forward and backward secrecy: Sempre que ocorre um ciclo completo de troca de dados, o par de chaves utilizadas no processo de cifra das mensagens seguintes, é independente do par de chaves anteriores (e consequentemente das seguintes). Caso seja descoberta o valor de uma chave, não é possível saber o que aconteceu no passado nem o que vai acontecer no futuro.

universidade de aveiro

Referências

https://cryptography.io/en/latest/

https://www.openssl.org/

https://en.wikipedia.org/wiki/Transport_Layer_Security#Security

https://en.wikipedia.org/wiki/Cipher_suite

https://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_key_exchange