

Segurança Informática e nas Organizações

Projeto 2: Comunicações Seguras

**Autores:**

André Gual nº 88751

João Laranjo nº 91153

# Índice

[**Objetivos**](#_6bjd106fu096) **3**

[**Introdução Teórica**](#_pg7vovpni30y) **3**

[Cifra Simétrica](#_xwfu3mxgekwm) 3

[Hashing:](#_tg7mxn8c5p3p) 3

[Listas de algoritmos / modos utilizadas no projeto:](#_fd2457greyai) 3

[States:](#_gu2kcynal3io) 4

[**Desenho de um Protocolo**](#_xgxwogm4send) **5**

[**Negociação de Algoritmos Usados**](#_g3f9h8toalyn) **5**

[**Troca de Chaves (Cliente - Servidor)**](#_ggbyupv5mky3) **7**

[**Confidencialidade (Cifra/Decifra)**](#_2ygie8kvd5pi) **11**

[**Controlo de integridade (HMAC)**](#_3rglvgdrv0hw) **14**

[**Rotação de Chaves**](#_wupguwl10b04) **17**

[**Bibliografia**](#_44x693ueuy8n) **21**

# Objetivos

Estabelecimento de uma ligação segura entre dois interlocutores.

Explorar conceitos relacionados com a troca de chaves, cifras simétricas e controlo de integridade.

# Introdução Teórica

#### **Cifra Simétrica**

A cifra/criptografia simétrica é uma forma de ocultar o conteúdo de algo em que o remetente e o destinatário usam a mesma chave secreta. De notar que este tipo de cifra fornece sigilo de informação, mas não autenticidade. Um atacante poderá não ver a mensagem, mas poderá criar mensagens falsas e forçar a decifragem.

#### **Hashing:**

Uma função de hash é uma função que pode ser usada para mapear informação de um tamanho arbitrário num tamanho fixo. O resultado destas funções pode ser chamado de hashes ou síntese.

#### **Listas de algoritmos / modos utilizadas no projeto:**

* Algoritmos de Cifra - CALGORITHMS = ['AES', 'ChaCha20', '3DES']
* Modos - MODES = ['CBC', 'GCM']
* Algoritmos de Hash - HALGORITHMS = ['SHA256', 'SHA512', 'MD5', 'BLAKE2b']

#### **States:**

O controlo do estado do cliente e do servidor é feito através da variável *self.state*.

Quado iniciamos o client.py o estado fica definido para STATE\_CONNECT. Segue-se a ligação e passamos então para o estado onde vai decorrer a negociação dos algoritmos e dos modos a serem utilizados - STATE\_NEGOTIATE. Neste estado vão circular mensagens do tipo NEGOTIATE\_CLI onde, nos argumentos da mensagem, são enviados os métodos/modos suportados pelo cliente. Em seguida, o servidor responde com uma decisão no formato de mensagem OK com a decisão final. Após a recepção desta mensagem são iniciadas as chaves, enviada uma mensagem DH para o servidor e alterado o estado para STATE\_KEY. Deu-se o início à troca de chaves para o algoritmo Deffie-Hellman. O cliente recebe uma mensagem com DH, procede à formação de uma shared\_key e segue para o estado STATE\_OPEN.

Com o STATE\_OPEN é aberto o ficheiro a enviar e encontra-se tudo preparado para o início da comunicação segura, STATE\_DATA. Através de um canal seguro as mensagens vão circular enquanto existir informação a transmitir, ou seja, enquanto STATE\_DATA.

Após a transmissão de todos os pedaços de informação é mudado o estado para STATE\_CLOSE e é enviado um sinal para terminar a ligação e por sua vez o cliente.

Estados:

* STATE\_CONNECT = 0
* STATE\_NEGOTIATE = 1
* STATE\_KEY = 2
* STATE\_OPEN = 3
* STATE\_DATA = 4
* STATE\_CLOSE = 5

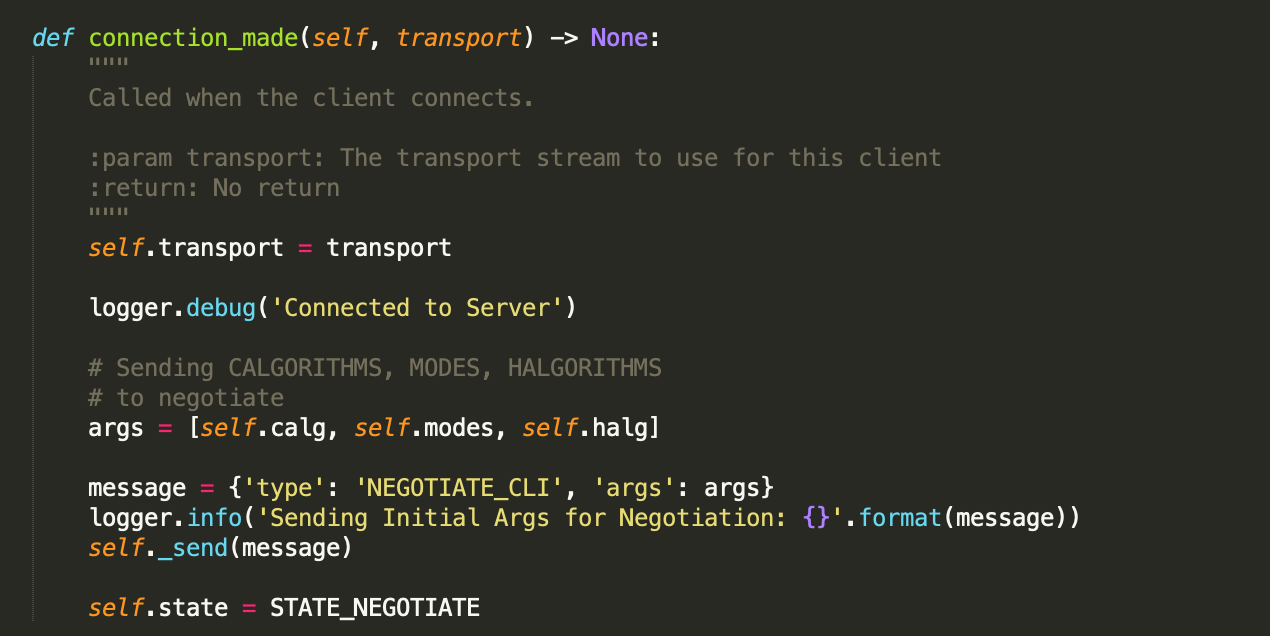
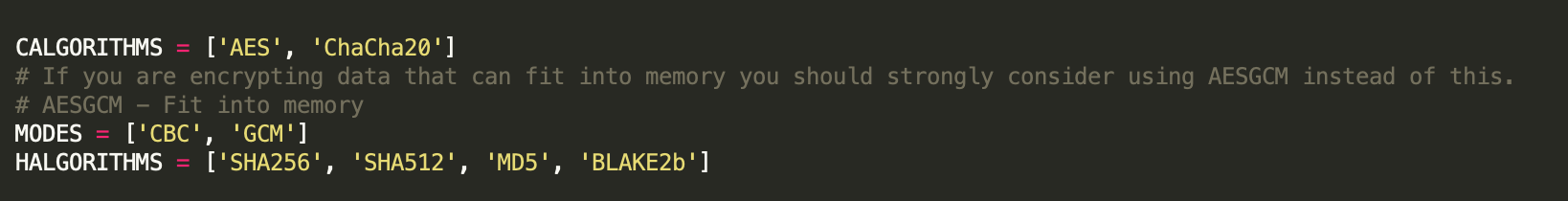
# Desenho de um Protocolo

# Negociação de Algoritmos Usados

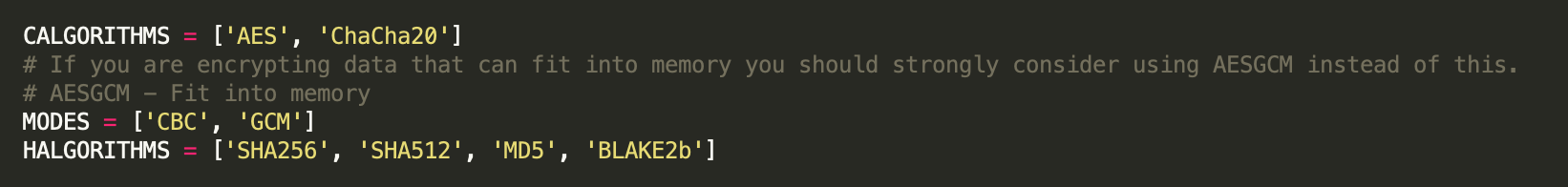
Para o acordo entre Cliente-Servidor dos algoritmos a serem utilizados é enviada uma mensagem do cliente para o servidor com a indicação para o início da negociação e acompanhado com os algoritmos disponíveis pelo utilizador.

Em seguida, o servidor compara a mensagem recebida com o seu “banco” de algoritmos e decide quais o algoritmo de cifra, o modo e o algoritmo hash. Essa decisão é enviada para o cliente através de uma mensagem ‘OK’.

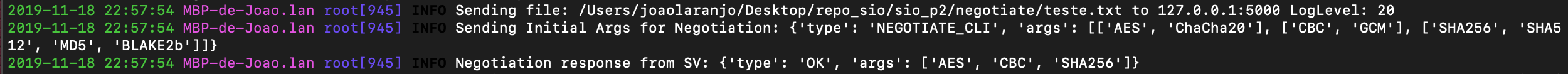
Cliente:



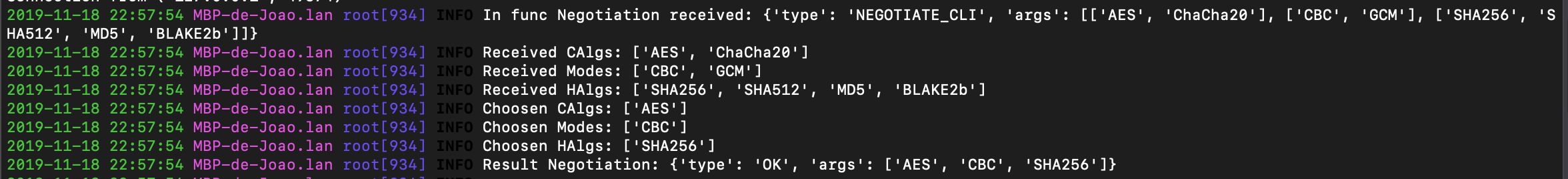
Servidor:



Resultado:

Cliente

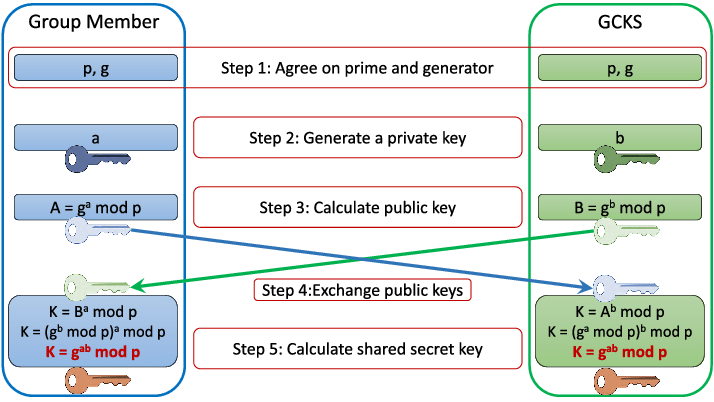
Servidor



# Troca de Chaves (Cliente - Servidor)

Para se obter uma chave partilhada entre as duas entidades implementou-se o algoritmo Diffie-Hellman.

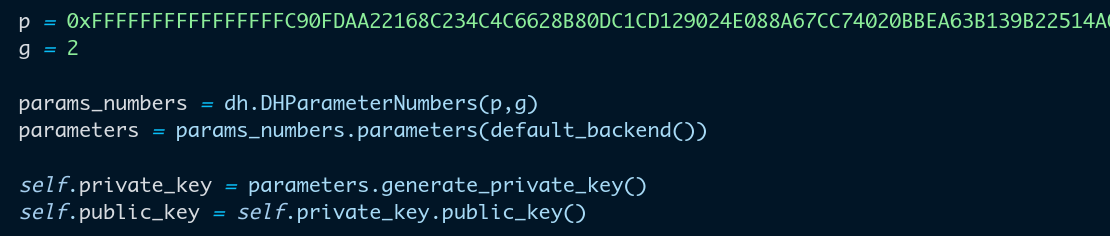
O algoritmo Diffie-Hellman consiste numa troca de chaves entre duas entidades de forma a que nenhuma delas tenha conhecimento da chave privada uma da outra. Após algumas operações obtemos uma chave partilhada por ambas, chave essa que é criada com base em ambas as entidades.



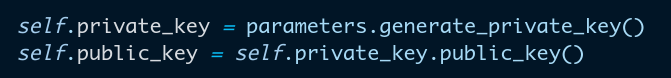
**Nota 1:** Exemplo retirado da internet apenas para ilustrar o processo de troca de chaves.

**Nota 2:** Os processos descritos em baixo são realizados em ambas as entidades de forma igual.

Tal como a imagem demonstra, primeiro temos que obter um *prime* e um *generator.* Ambos estes parâmetros são comuns e iguais para as duas entidades. Procedemos assim à implementação dos mesmos.

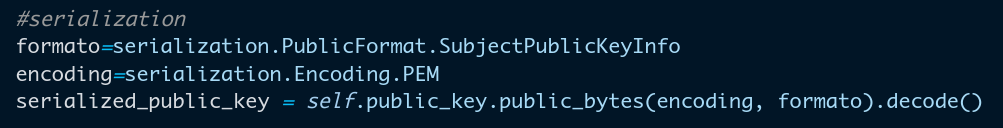


Obtemos então *p (prime)* e *g (generator)*. De seguida transformamos os mesmos em parâmetros para serem utilizados pela biblioteca cryptography.



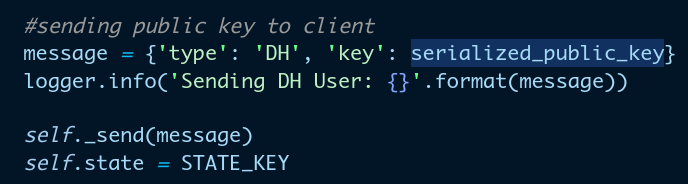
O próximo passo é a obtenção de uma chave privada e da chave pública associada à mesma.

Tendo obtido as chaves, precisamos de serializar as chaves públicas para podermos enviá-las dentro de uma mensagem json, do cliente para o servidor e vice-versa.



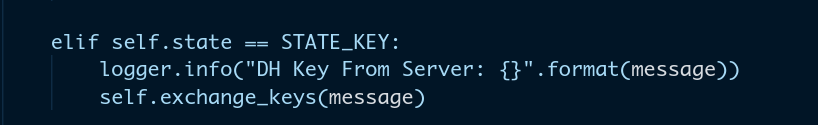
Procedemos então à troca de chaves públicas.

Primeiramente o cliente envia ao servidor a sua chave pública.

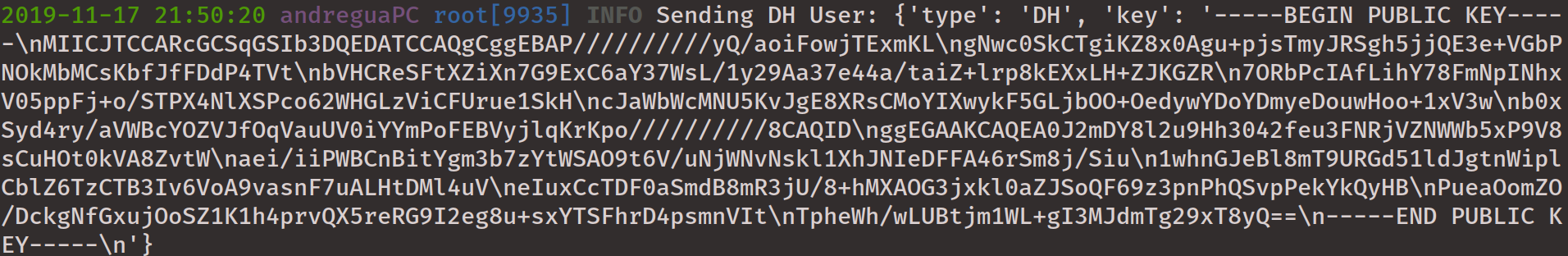


Para esta troca demos atribuímos a string ‘DH’ ao type e enviamos então a chave pública no parâmetro ‘key’. Definimos também um novo STATE denominado STATE\_KEY

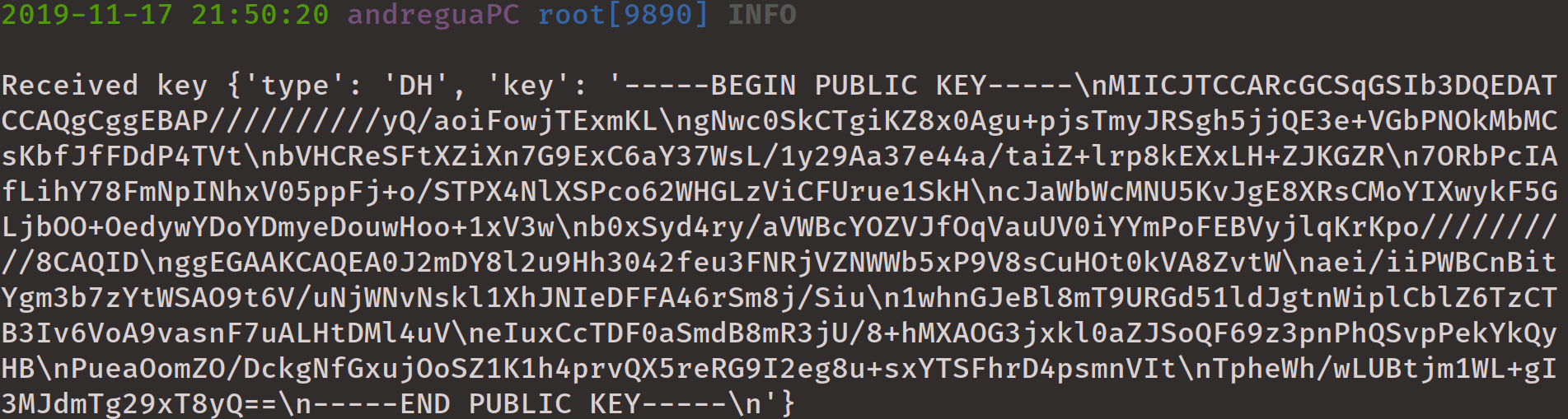
Este state é utilizado na função on\_frame para aquando da resposta do servidor executar a função exchange\_keys que irá calcular o segredo partilhado ( explicado mais à frente).

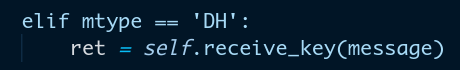


Enviamos então a mensagem:



A mensagem é recebida no servidor:

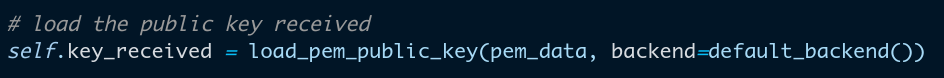
Recebemos a mensagem com o type ‘DH’ e executamos a função receive\_key(message) e enviamos a mensagem por argumento.



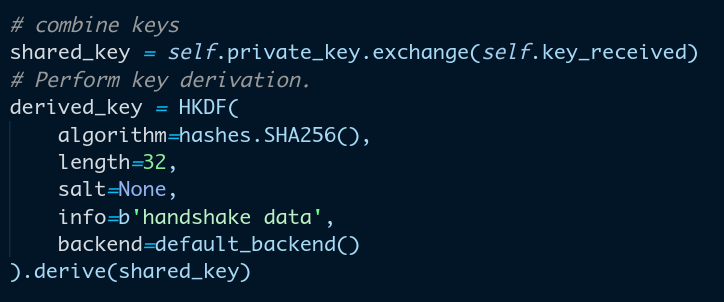
Na função receive\_key é realizado o processo descrito em cima até obtermos a chave pública e colocamos a mesma dentro de uma mensagem para a enviar ao cliente.

Para além disso é também já calculado o segredo partilhado.

Para efetivamente obtermos chave pública recebida temos que inverter a serialização e para isso, visto que a chave está com encoding PEM, podemos simplesmente utilizar a função load\_pem\_public\_key passando pem\_data por argumento, sendo pem\_data a chave pública recebida serializada.



Este processo de cálculo do segredo partilhado tem duas operações:

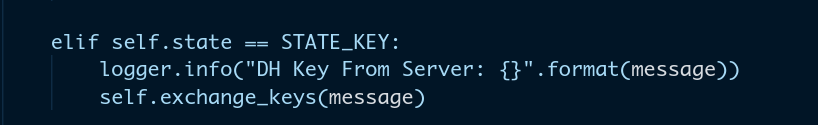


Primeiro através da biblioteca cryptography utilizamos o método exchange que nos irá combinar a nossa chave privada neste caso do servidor com a chave recebida na mensagem que será a chave pública do cliente.

De seguida é necessário fazer a derivação da mesma através da função HKDF.

O servidor de seguida envia a sua chave pública calculada antes para o cliente.

O cliente ao recebê-la irá executar a função exchange\_keys visto que em cima definimos o estado atual para STATE\_KEY.



A função exchange\_keys segue exatamente o mesmo padrão apresentado em cima para o cálculo do “segredo partilhado”. No entanto é utilizado a chave privada o cliente e a chave pública do servidor.

As chaves derivadas obtidas em ambas as entidades são **iguais** e daí o nome “segredo partilhado”.

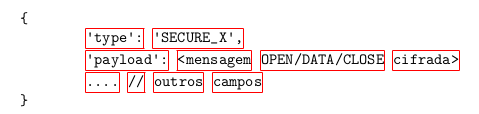
O algoritmo Diffie-Hellman está assim implementado e cada entidade tem acesso a uma chave partilhada que será utilizada para proceder às cifras e decifras das mensagens.

# Confidencialidade (Cifra/Decifra)

Para garantir o suporte para confidencialidade implementámos dois algoritmos de cifra simétrica. O AES(cifra por blocos), e o ChaCha20(cifra contínua).

O AES possui uma implementação que suporta os modos CBC e GCM. (Cifra Por Blocos)

Para implementar o envio e receção de mensagens encriptadas criámos um novo formato para as mensagens encriptadas. Tal como especificado no enunciado o formato utilizado foi o seguinte :

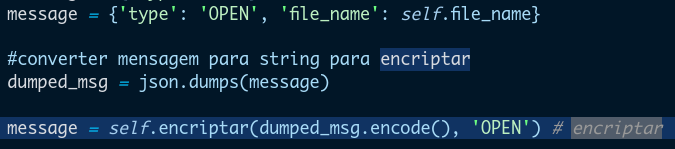


Ou seja as mensagens ‘OPEN’, ‘DATA’, ‘CLOSE’ deixaram de ser enviadas e passaram as ser enviadas mensagens ‘SECURE\_OPEN’, ‘SECURE\_DATA’, ‘SECURE\_CLOSE’ em que o payload corresponde às mensagens ‘OPEN’, ‘DATA’, ‘CLOSE’ encriptadas. Em certos casos são também enviados outros parâmetros para ser utilizados nos processos de cifra e decifra do payload.

Começando pela cifra, esta é feita obviamente do lado do cliente. Para a implementação da mesma criámos uma função ‘encriptar(self,data,operation)’. A função possui 2 argumentos (ignorando o self) :

* data : este argumento representa os dados que queremos encriptar.
* operation: este argumento representa o tipo de mensagem que é para se formar, ou seja a operação que iremos fazer. No nosso caso este argumento toma apenas 3 valores difrentes - ‘OPEN’, ‘DATA’, ‘CLOSE’. Desta forma saberemos qual das mensagens secure teremos que enviar.

Exemplo de chamada da função (para a operação open):



Como podemos verificar na imagem em cima, o data que vai ser encriptado é o dicionário que corresponderia à mensagem open.

A função encriptar retorna um valor que corresponde à mensagem que iremos enviar para o servidor:

A mensagem contém então o type e o payload.

A mensagem é mais tarde enviada para o servidor.

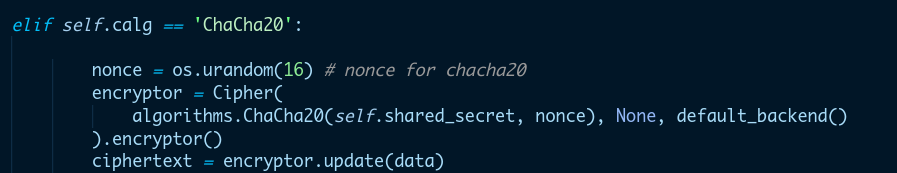
Passando à implementação em si da função encriptar(), primeiramente é identificado o algoritmo que foi definido na negociação e que irá ser usado para a encriptação.

De seguida é verificado o modo para o caso de o algoritmo ser o ‘AES’.

De seguida são implementadas as cifras para cada algoritmo de acordo com o modo. Apenas uma cifra vai ser utilizada. Para a implementação dos algoritmos utilizamos a biblioteca cryptography.

Cada algoritmo tem uma implementação diferente e todos eles necessitam de parâmetros adicionais que precisam de ser enviados juntamente com as mensagem para que seja possível fazer a decifragem do lado do servidor.

Por exemplo o modo ‘CBC’ para o algoritmo ‘AES’ necessita que seja transmitido um ‘iv’, ou por exemplo o algoritmo ‘ChaCha20’ em que é necessário enviar um nonce.



Este parâmetros são enviados na mensagem SECURE\_X como parâmetros adicionais.

Exemplo para a mensagem SECURE\_OPEN que utiliza o algoritmos ChaCha20.



Do lado do servidor iremos proceder à decifragem das mensagens (argumento payload) que são enviadas pelo cliente.

Para decifrar implementámos uma função “decrypt(message)” que recebe como argumento uma mensagem recebida pelo cliente, e que devolve a mensagem decifrada.



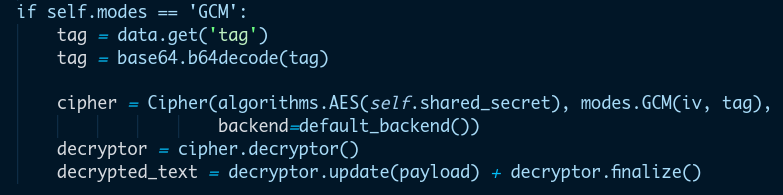
Esta função é chamada quando recebemos um type correspondente a um dos três seguintes: ‘SECURE\_OPEN’, ‘SECURE\_DATA’, ‘SECURE\_CLOSE’

A função decrypt em si segue uma estrutura semelhante à função encriptar(). É verificado primeiramente o algoritmo e modo a ser utilizado para a decifragem seguido da implementação de cada decifra através da biblioteca cryptography.

Antes de ser efetuada a decifragem os argumentos adicionais presentes na mensagem recebida são tratados de modo a poderem ser utilizados na decifragem.

De acordo com o algoritmo e modo é colocado numa variável o texto desencriptado que é depois devolvido no return.

Exemplo de implementação da decifragem para o algoritmo ‘AES’ modo ‘GCM’:



O texto decifrado é depois utilizado para chamar o processo que será efetuado (é passado por argumento):



# Controlo de integridade (HMAC)

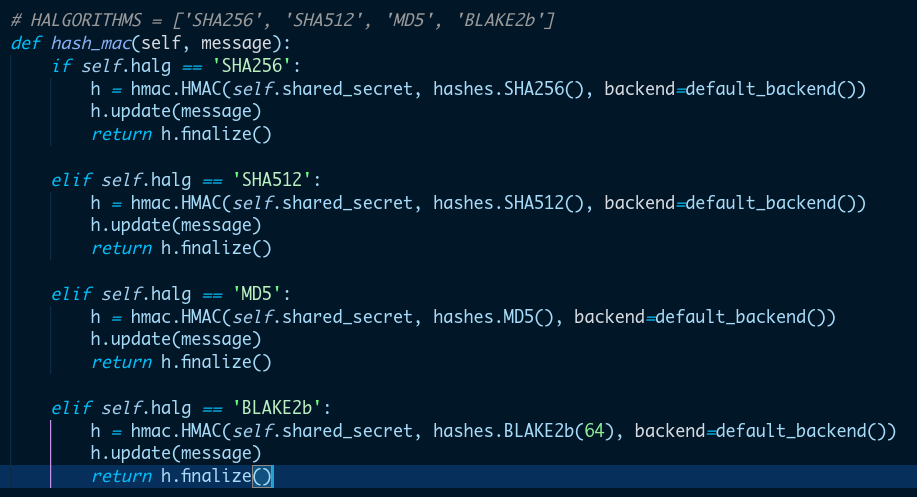
Para implementar o suporte para o controlo de integridade, utilizamos o HMAC da biblioteca cryptography.

As funções de hash que disponibilizamos para o controlo de integridade são as seguintes:

* SHA-256
* SHA-512
* MD5
* BLAKE2b

Para este processo implementámos uma função hash\_mac(self, message) que aceita como argumento uma string e que retorna a síntese (bytes) da string.

Função hash\_mac(self,message):



Decidimos utilizar o hash\_mac pois este utiliza uma chave, uma hash e uma string para a criação de uma síntese que irá garantir então a integridade das mensagens. Para além da integridade este mecanismo garante também a autenticidade.

Parâmetros utilizados:

* Chave: shared\_secret obtido através do Diffie-Hellman.
* Hash: Função de hash definida através da negociação.
* String: O texto cifrado.

Os três parâmetros são possíveis de obter tanto do lado do cliente como do lado do servidor(tendo recebido a mensagem do cliente).

O mac é enviado pelo cliente para o servidor como um parâmetro das mensagens SECURE\_x. É calculado dentro da função encriptar pois segue a lógica Encrypt-then-Mac.

Primeiro ciframos o texto e depois calculamos a síntese do texto encriptado. Desta forma conseguimos do lado do servidor fazer a verificação de integridade antes de decifrar a mensagem.

Chamada dentro da função encriptar após a cifra:

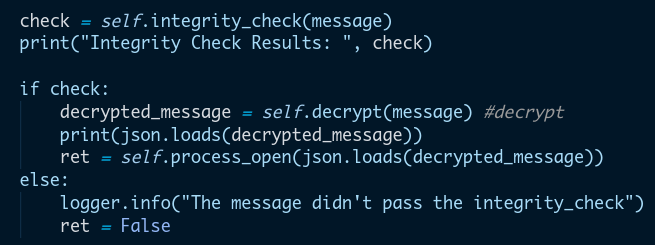


O mac é encoded para base64 de forma a podermos enviar o mesmo na mensagem para o servidor da seguinte forma(último parâmetro):



Quando a mensagem chega ao lado do servidor o primeiro processo a realizar antes da desencriptação é a verificação da integridade da mensagem(HMAC).

Isto é feito da seguinte forma:

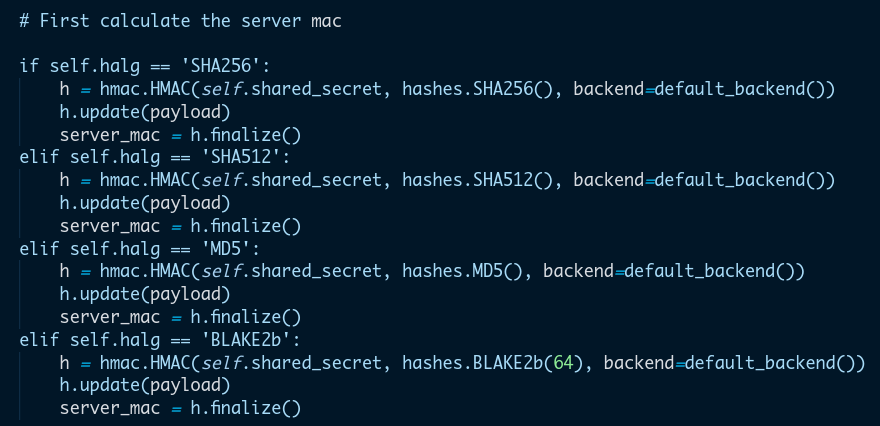


Para a verificação de integridade implementámos a função integrity check(self,message) que recebe como argumento a mensagem vinda do cliente e retorna um valor boolean que é atribuído à variável check.

Caso check seja‘ True’, ou seja a mensagem passou no teste de integridade, é feita a decifragem da mensagem.

Caso check seja ‘False’ a mensagem não passa no teste integridade e por isso o processo é abortado e a decifragem não é feita. A mensagem é descartada.

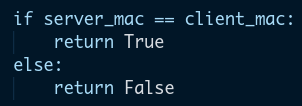
Relativamente à forma como o teste de integridade é feito na função integrity check, primeiramente segue-se um processo idêntico ao cliente e calcula-se o HMAC do servidor, utilizando o shared\_secret do servidor,a função de hash e o texto cifrado vindo do cliente.



De seguida é que é realizado o teste. Para isso obtemos o mac do cliente que vem na mensagem vinda do cliente e comparamos o mesmo com o mac calculado no servidor.



Comparação:



Se os macs forem iguais é retornado o valor True e a mensagem passa no teste de integridade.

Se os macs forem diferentes é retornado o valor False e a mensagem não passa no teste de integridade.

# 

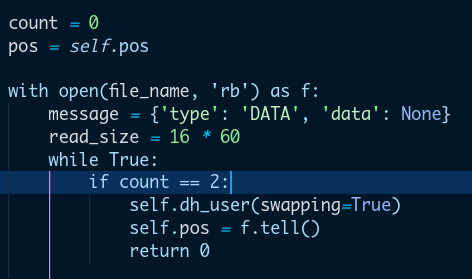
# Rotação de Chaves

O mecanismo de rotação de chaves é importante pois caso um atacante consiga obter uma chave que lhe permita decifrar as mensagens este apenas o conseguirá fazer para um número reduzido por mensagens. A chave, ao fim de um certo número de iterações é trocada por uma nova, calculada através do mecanismo de Diffie-Hellman.

Na função send\_file() no cliente onde o ficheiro é enviado por chunks, definimos um contador que irá representar o número de iterações sobre o ficheiro, ou seja, o número de vezes que são enviadas chunks para o servidor.

Para fazer a rotação de chaves definimos um limite para esse contador (2 iterações por motivos de legibilidade, mas na verdade o número deveria ser muito maior).

Quando o contador atinge o valor 2, em vez do envio de uma *chunk* para o servidor, iremos executar novamente o mecanismo Diffie-Hellman.



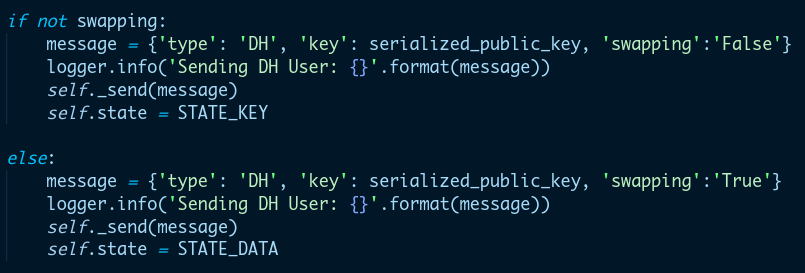
* count : contador de iterações.
* pos: posição do ponteiro de leitura do ficheiro.

Chamamos então a função que implementa o DH e acrescentamos um argumento à função que nos permite indicar se estamos a fazer rotação de chaves (swapping=True).

Atualizamos o ‘pos’ para quando formos ler novamente o ficheiro, começarmos onde parámos, ou seja num certo offset. No fim retornamos 0 para sair da função send\_file().

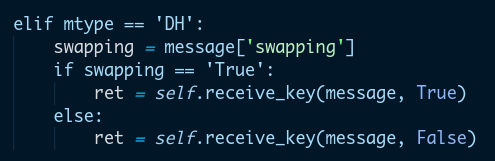
É necessário adaptar a função dh\_user para enviar nas mensagens de DH um parâmetro swapping que indica se estamos a fazer a rotação das chaves.

Para além disso a variável state no caso da rotação de chaves terá o valor STATE\_DATA e não STATE\_KEY. O STATE\_DATA lida com as mensagens precisamente vindas do servidor quando estamos a fazer rotação das chaves e suspendemos o processo de enviar chunks para o servidor.

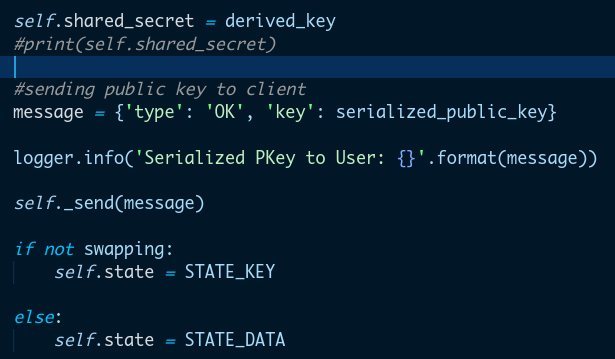


É enviada uma mensagem com ‘type’:‘DH’ para o servidor com o parâmetro swapping.

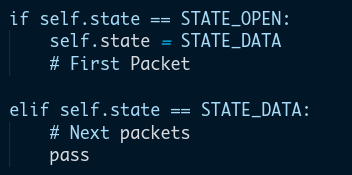
Do lado do servidor adaptamos a receção de mensagens do tipo ‘DH’ para que este chame a função receive\_key() (que calcula o shared\_secret e envia a chave pública do servidor para o cliente) agora também com um argumento swapping.



Dentro da função receive\_key():

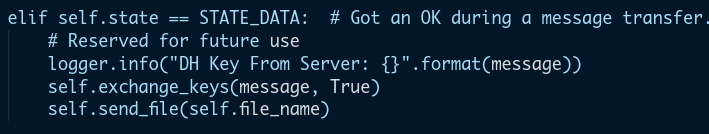


Enviamos a mensagem com a chave pública para o cliente mas no entanto quando estamos a fazer a rotação das chaves temos que definir também o estado como STATE\_DATA, para mais tarde quando chamarmos a função process\_data() recebermos o resto dos chunks vindos do cliente.



Se não atribuirmos o state\_data ao estado obtemos um erro de estado inválido.

É no state data que recebemos a resposta enviada pelo servidor(com a public key do mesmo):



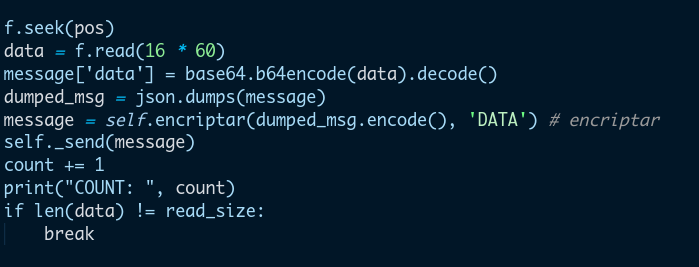
Ao recebermos nova public key do servidor chamamos a função exchange\_keys (agora também com um argumento swapping idêntico ao do dh\_user) que irá calcular o novo shared\_secret.

No exchange\_keys é calculado normalmente o novo shared\_secret, mas visto que estamos a fazer rotação de chaves adicionamos uma condição para o cliente não enviar uma mensagem open para o servidor.



De seguido no STATE\_DATA é simplesmente chamado novamente o send\_file()

Ao entrarmos novamente no send\_file() desta vez iremos começar a ler o ficheiro a partir da posição presente na variável ‘pos’. Assim retomamos a leitura do ficheiro a partir do sítio onde tínhamos suspendido anteriormente. A função do python seek() permite definir a posição onde se começa a ler um certo ficheiro.



Assim está implementado o mecanismo de rotação de chaves. O processo descrito anteriormente repete-se até o ficheiro ser lido totalmente.

# 

# Bibliografia

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

<https://cryptography.io/en/latest/hazmat/primitives/symmetric-encryption/>

<https://cryptography.io/en/latest/hazmat/primitives/mac/>

<https://www.tutorialspoint.com/python/file_tell.htm>

<https://www.tutorialspoint.com/python/file_seek.htm>