UNIVERSIDADE DE AVEIRO | Departamento de Eletrónica, Telecomunicações e Informática

Segurança Informática e nas Organizações – 1º Semestre 2019-20

Projeto 2: Comunicações Seguras

Curso:

Licenciatura em Engenharia Informática

Data:

18 de Novembro de 2019

Docentes:

Professor João Barraca
Professor Vítor Cunha

Discentes:

João Magalhães - 79923
João Ferreira - 80041

Índice

Introdução

Este trabalho foi realizado no âmbito da Unidade Curricular Segurança Informática e nas Organizações (SIO) e aborda os seguintes tópicos: explorar conceitos relacionados com a troca de chaves, cifras simétricas e controlo de integridade.

Aplicando os tópicos acima referidos, o objetivo é estabelecer uma sessão segura entre dois interlocutores (server e cliente), trocando mensagens entre ambos.

A criptografia é um conjunto de regras que visa codificar a informação trocada de forma que só os dois interlocutores consigam decifrá-la.

Numa primeira abordagem a criptografia devemos referir termos que consideramos essenciais, encriptação e a desencriptação.

- Encriptação
    - Descreve-se como sendo o processo de transformar informação por meio de um algoritmo, i.e. utilizando uma cifra, de forma a que seja impossível a sua leitura, excetuando aos intervenientes da comunicação, ou seja, aqueles que possuam a chave para decifrar o conteúdo da informação.
- Desencriptação
    - Descreve-se como sendo o processo de transformar a informação anteriormente encriptada novamente legível, ou seja, desencriptar a mensagem enviada.

Neste trabalho, apesar de existirem várias formas de encriptar e desencriptar as mensagens, iremos utilizar cifras simétricas, como pedido no enunciado.

- Cifras simétricas
    - As cifras simétricas têm como característica serem detentoras de uma única chave secreta partilhada por todos os interlocutores presentes na comunicação. Assim, ao existir uma única chave esta operação torna-se simples de realizar e mais rápida. No entanto, os algoritmos que utilizam a cifra simétrica não são tão seguros como os que utilizam criptografia assimétrica, uma vez que estas utilizam duas chaves, uma pública para cifrar e uma privada para decifrar.

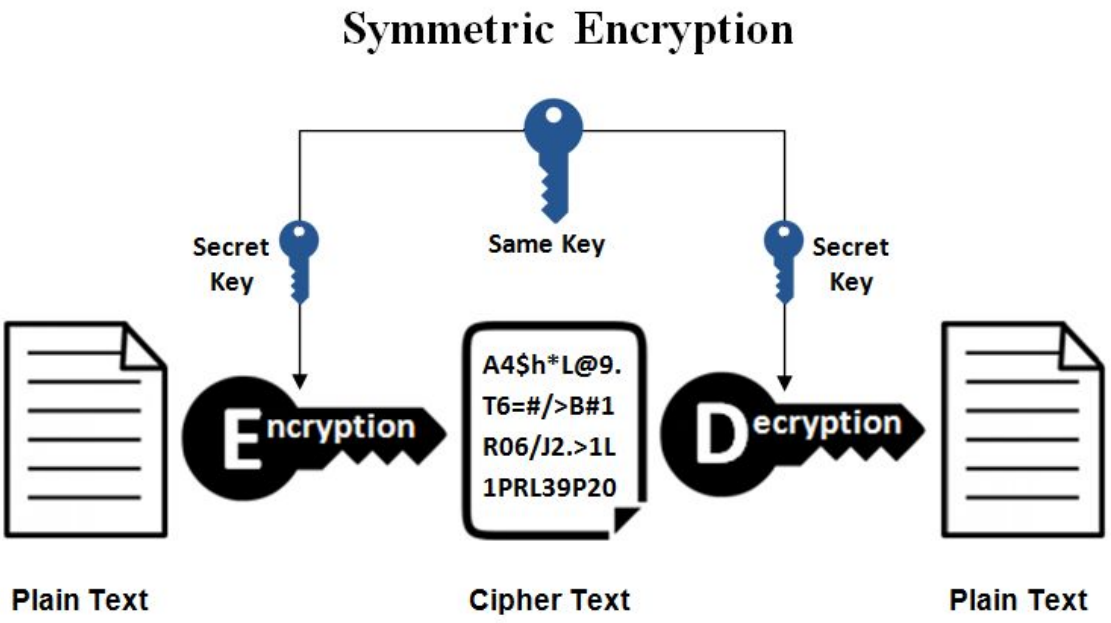

Figura 1 - Criptografia Simétrica

As cifras simétricas podem ser divididas em dois grupos: as cifras simétricas por blocos e as cifras simétricas contínuas.

- Cifras simétricas por blocos
  - Consistem em encriptar um conjunto de blocos de grande dimensão de bytes que têm um tamanho fixo. Se a informação transmitida não for do tamanho ou múltiplo do bloco pode ser preenchida com mais dados não importantes para que o tamanho da informação seja apropriado para o uso de uma cifra desse tipo.

    Usam os princípios básicos de difusão e confusão introduzidos por Shannon. Tal é feito recorrendo às seguintes operações:
    - Aplicação iterativa de uma operação complexa a um bloco de grande dimensão, controlada por uma chave.
    - Operações de permutação, substituição, expansão e compressão de blocos.

      Permutação: troca bits de lugar sem alteração do número de bits;

      Substituição: altera conjuntos de bits por outros tantos de valores diferentes segundo uma tabela de substituição;

      Expansão: introduz novos bits replicando alguns blocos de entrada;

      Compressão: passa para a saída apenas alguns dos bits de entrada;

  Exemplos: **AES**, DES(3DES), IDEA

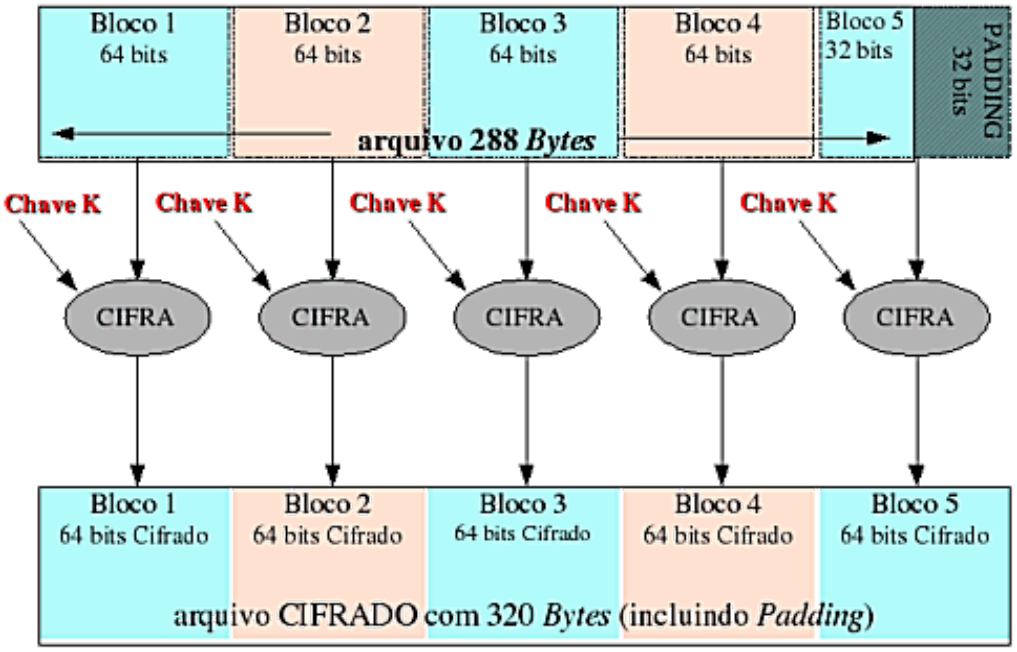

Figura 2 - Cifra por blocos

- Cifras simétricas contínuas
  - Consistem em encriptar gerando uma corrente de bits pseudo aleatória, baseiam-se em LFSRs (linear feedback shift registers), em cifras por bloco, etc. Estas cifras normalmente não têm sincronização e não existe a possibilidade de acesso aleatório rápido.
  - Utilizam apenas o princípio da confusão.
  - As chaves contínuas devem ser o mais próximo possível de one-time pads. Isto significa que:

    i) o período de uma chave contínua deve ser o mais longo possível, de preferência mais longo que o texto a cifrar. Isto depende de três fatores: do algoritmo do gerador, da chave e do texto a cifrar.

    ii) a sequência de bits de uma chave contínua deve aparentar ser verdadeiramente aleatória. Para que isto aconteça, os seus bits consecutivos devem ter valores equiprováveis e imprevisíveis.

    Exemplos: **Salsa20,** RC4, E0

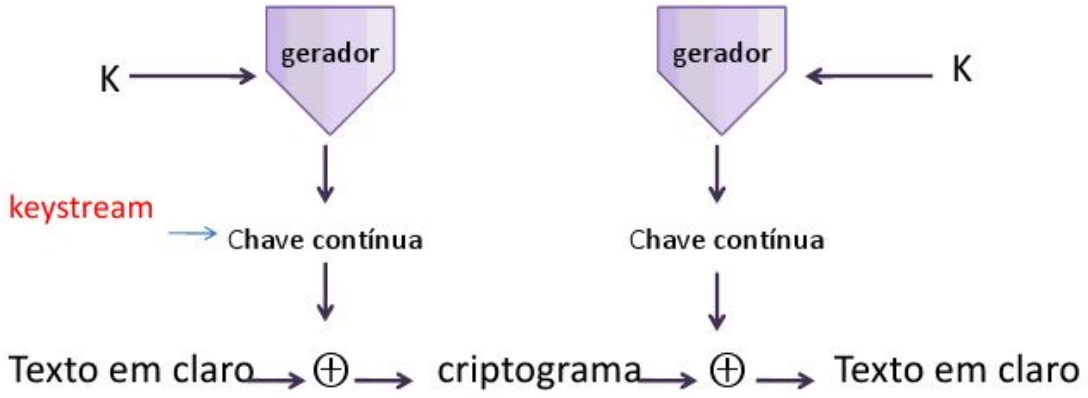

Figura 3 - Cifra Contínua

- AES (Advanced Encryption Standard)
  - É uma cifra por blocos, ou seja, opera em blocos de tamanho fixo, 128 bits. O AES pode trabalhar com chaves de 128, 192 ou 256 bits, isto é, é um algoritmo que recebe uma mensagem com um bloco de 128 bits e uma chave do tamanho escolhido, retornando uma cifra de também 128 bits. A decifra recebe a cifra e retorna como saída um bloco de 128 bits. Se a chave for a chave utilizada para cifrar, a saída será idêntica à mensagem original.

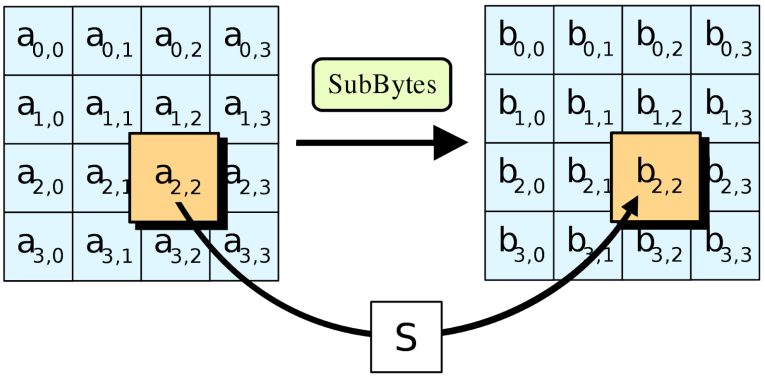

Figura 4 - AES

- Salsa20
  - É uma cifra contínua. Este algoritmo utiliza a adição bit a bit (XOR), adição 32-bit mod 2^32 e uma distância constante de operações de rotação num estado interno de palavras de 32-bits. O estado inicial é formado por 8 palavras-chave, sendo estas: duas palavras de posição de fluxo, duas palavras de nonce e quatro palavras constantes. Como resultado de 20 iterações de mistura, são produzidas 16 palavras da saída da cifra contínua.

- Modos de cifra
  - Os modos de cifra mais antigos como por exemplo o ECB, **CBC**, OFB e CFB, são modos que só asseguram a confidencialidade da mensagem. No entanto existem outros que para além de assegurarem a confidencialidade também asseguram a integridade da mensagem como por exemplo (**GCM**, OCB).

Todos os modos referidos anteriormente necessitam do vetor de inicialização (IV) que inicializa o processo para o primeiro bloco. Assim garantimos a aleatoriedade ao processo, isto é, o mesmo texto que queremos enviar para outro interlocutor é encriptado várias vezes.

- Modo de cifra - CBC(Cipher-block chaining)
  - Neste modo na cifra de cada bloco é introduzida uma realimentação: o texto em claro a cifrar é previamente somado, XOR, com o bloco anterior do criptograma. O valor do vetor de inicialização (IV) usado para processar o primeiro bloco pode ser secreto ou não.

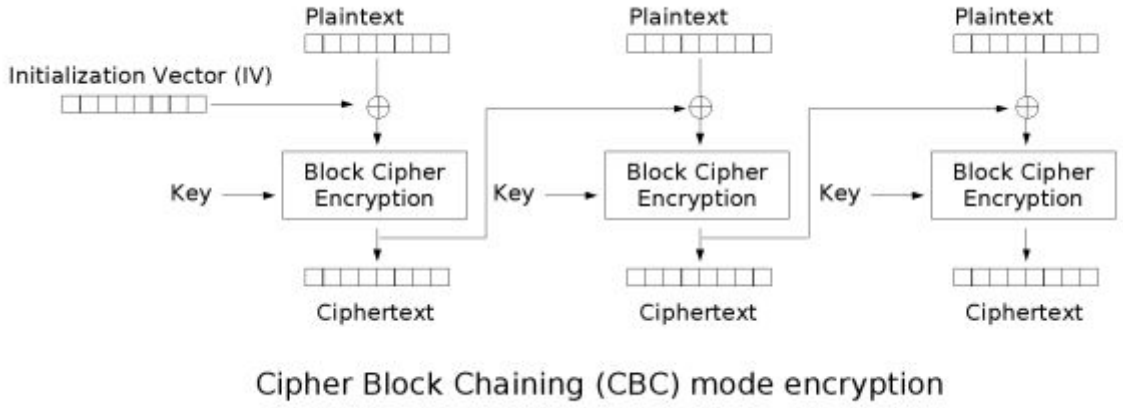

Figura 5 - CBC encriptação

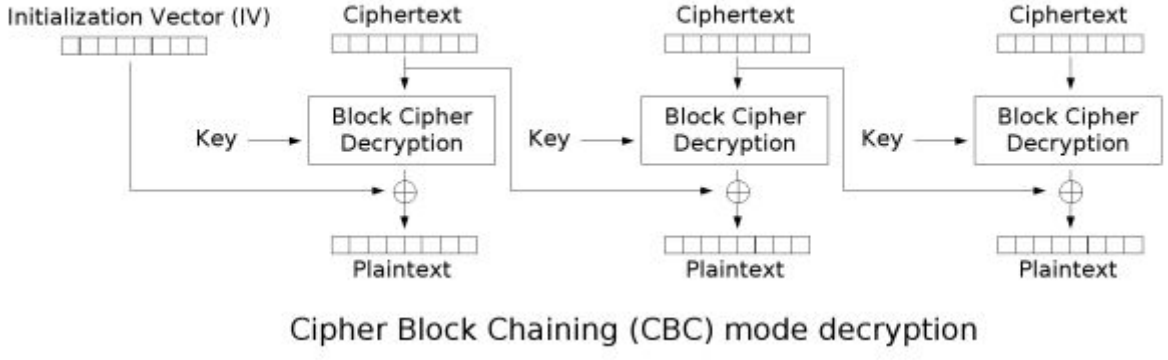

Figura 6 - CBC desencriptação

- Modo de cifra - GCM(Galois/Counter Mode)
    - O GCM é definido para cifras de bloco com um tamanho de bloco de 128 bits. Como no modo normal do CTR, os blocos são sequencialmente numerados e esse número é posteriormente combinado com o IV e criptografado com uma cifra de bloco. Os blocos de texto cifrado são tratados como coeficientes de um polinómio que é então avaliado num ponto dependente da chave.

  O resultado é então criptografado, produzindo uma marca de autenticação que pode ser usada para verificar a integridade dos dados. O texto criptografado contém a etiqueta IV, texto cifrado e autenticação.

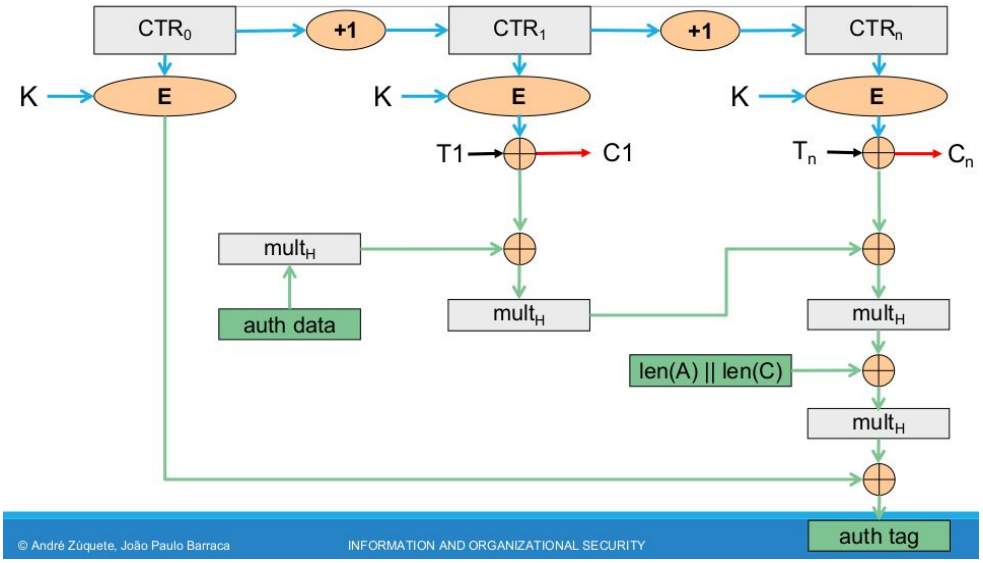

Figura 7 - Modo GCM

- Algoritmos de Síntese SHA256 e SHA512
    - São funções hash computadas com palavras de 32 bits no caso do SHA256 e 64 bits no caso do SHA512. Estes utilizam quantidades de deslocamento e constantes aditivas diferentes, mas as estruturas são praticamente iguais, diferindo apenas no número de iterações.

- Diffie-Hellman
    - Este método consiste na troca de chaves entre dois intervenientes que não possuem conhecimento prévio de cada uma, compartilhando uma chave secreta num canal de comunicação inseguro. Esta chave é usada para encriptar mensagens que são posteriormente enviadas usando o método de cifra de chave simétrica.

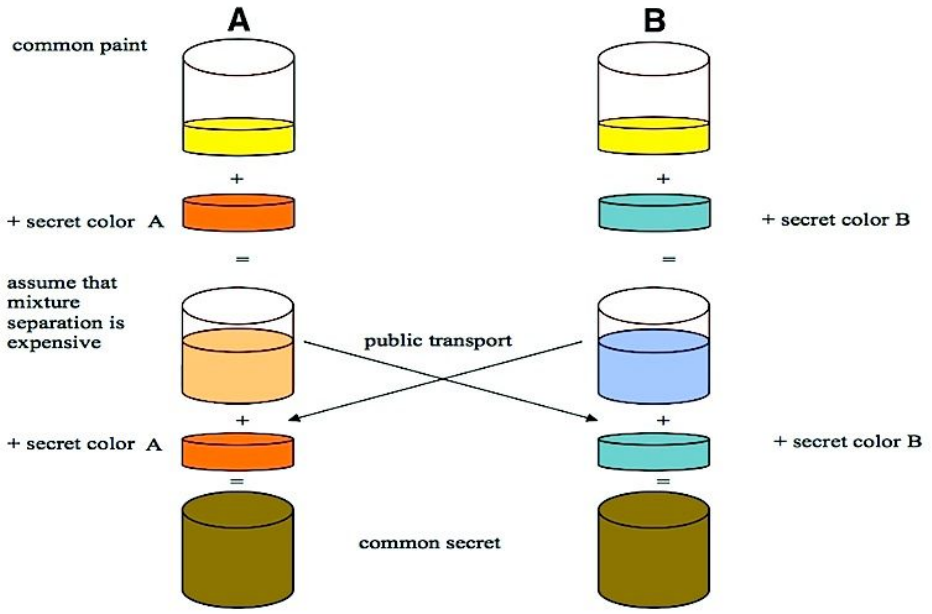

Figura 8 - Esquema Diffie-Hellman

- Message Authentication Code (MAC) é um valor produzido a partir de uma mensagem e de uma chave simétrica partilhada pelo emissor e pelo recetor da mesma. Um MAC apenas pode ser gerado e validado por estas duas entidades. O valor do MAC protege tanto a integridade dos dados da mensagem bem como a sua autenticidade, permitindo aos detentores da chave secreta detetar quaisquer mudanças no conteúdo da mensagem.

- Um MAC pode ser produzido através de:
  - funções de cifra por blocos
  - funções de cifra contínua
  - funções de síntese

O MAC é utilizado para autenticar uma mensagem cifrada, há várias maneiras de enquadrar a cifra da mensagem e a produção de um MAC da mesma:

- Produzir um MAC a partir da mensagem em claro e enviá-lo em claro juntamente com a mensagem cifrada (Encrypt-and-MAC)
- Produzir um MAC a partir da mensagem em claro e cifrá-lo em conjunto com a mensagem (MAC-then-Encrypt)
- Cifrar a mensagem e produzir um MAC a partir do criptograma resultante (Encrypt-then-MAC)

Análise do código

- Client.py

No ficheiro relativo ao Cliente começamos por definir estados (STATE_x) para além dos definidos no código fornecido. Definimos os seguintes estados: STATE_TO_CIPHER, STATE_KEY, STATE_DIFFIE_PARAMETERS e STATE_SECRET. Atribuímos números a estes estados que estão compreendidos entre os números dos estados STATE_OPEN e STATE_DATA uma vez que as operações correspondentes são efectuadas entre estes dois estados ocorrem entre estes dois estados.

O cliente fica no estado STATE_TO_CIPHER quando é necessário escolher o tipo de cifra a ser utilizado para a troca de mensagens. Para definirmos o universo de algoritmos possíveis definimos uma lista à qual chamamos ALGORITHMS e, para além disso, outra lista com os algoritmos de síntese possíveis, à qual chamamos SYNTHESIS.

```
ALGORITHMS = ["AES128-GCM", "AES128-CBC", "AES192-GCM", "AES192-CBC", "AES256-GCM", "AES256-CBC"]
SYNTHESIS = ["SHA256", "SHA512"]
```

Após atingir o estado mencionado no parágrafo anterior, o cliente é transportado para o estado seguinte, STATE_DIFFIE_PARAMETERS, estado correspondente à iniciação do algoritmo Diffie-Hellman. Para o sucesso deste algoritmo é necessário definirmos alguns parâmetros que irão ser definidos enquanto o cliente se encontra neste estado.

Após iniciarmos o algoritmo, iniciamos então a troca de chaves entre o cliente e o servidor. Posteriormente, é fundamental definir o segredo (*secret)* para a troca de mensagens, passando assim o cliente a estar no estado STATE_SECRET.

Finalmente, é realizada a comunicação desejada entre ambos os interlocutores, transferindo o ficheiro (STATE_DATA). No fim da transferência, é activado o estado STATE_CLOSE, terminando a comunicação.

Relativamente às funções definidas na classe definida no Cliente, começamos por ir actualizando a função on_frame e actualizando-a à medida em que íamos acrescentando novos tipos de mensagens. Esta função é utilizada para processar mensagens em formato JSON. Como primeira verificação, analisamos o tipo de mensagem enviada pelo servidor (deve ser uma mensagem do tipo 'OK', sinalizando que está preparado). Após esta verificação, verificamos em qual dos estados se encontra o cliente, optando por realizar uma acção que corresponda ao passo seguinte a cada uma delas.

No caso do estado do cliente ser STATE_OPEN chamamos a função pick_cipher(), utilizada para escolher a cifra a ser utilizada, e mudamos o estado para STATE_TO_CIPHER.

No caso do estado do cliente ser STATE_TO_CIPHER chamamos a função pick_diffie_parameters, utilizada para definir os parâmetros do algoritmo Diffie-Hellman, e mudamos o estado para STATE_DIFFIE_PARAMETERS.

No caso do estado do cliente ser STATE_DIFFIE_PARAMETERS chamamos a função build_keys, utilizada para gerar e definir as chaves do cliente, e mudamos o estado para STATE_KEY.

No caso do estado do cliente ser STATE_KEY decodificamos a mensagem recebida com a chave pública do servidor e guardamo-la. Respondemos posteriormente ao servidor com uma mensagem do tipo SECRET. Finalmente, alteramos o estado do cliente para STATE_SECRET.

No caso do estado do cliente ser STATE_SECRET chamamos a função pick_secret, utilizada para construir o segredo para a comunicação desejada e, após a execução desta função, chamamos a função encrypt para encriptarmos o ficheiro pretendido. Por fim, alteramos o estado do cliente para STATE_DATA.

```
def on_frame(self, frame: str) -> None:
    """
    Processes a frame (JSON Object)

    :param frame: The JSON Object to process
    :return:
    """
    #logger.debug("Frame: {}".format(frame))
    try:
        message = json.loads(frame)
    except:
        logger.exception("Could not decode the JSON message")
        self.transport.close()
        return

    mtype = message.get('type', None)

    if mtype == 'OK':  # Server replied OK. We can advance the state
        if self.state == STATE_OPEN:
            logger.info("Channel open")
            self.pick_cipher()
            logger.info("Ciphersuite has been sent.")
            self.state = STATE_TO_CIPHER
        elif self.state == STATE_TO_CIPHER:
            self.pick_diffie_parameters()
            logger.info("Diffie-Hellman Parameters have been sent.")
            self.state = STATE_DIFFIE_PARAMETERS
        elif self.state == STATE_DIFFIE_PARAMETERS:
            self.build_keys()
            logger.info("Public key has been sent.")
            self.state = STATE_KEY
        elif self.state == STATE_KEY:
            self.pub_key_server = base64.b64decode(message.get('data', "").encode())
            logger.info("Server Public Key has been received.")
            self._send({'type':'SECRET'})
            self.state = STATE_SECRET
        elif self.state == STATE_SECRET:
            self.pick_secret()
            self.encrypt()
            self.state = STATE_DATA
        elif self.state == STATE_DATA:  # Got an OK during a message transfer.
            self.send_file('encrypt.txt')
```

De seguida, definimos a função pick_cipher. Nesta função utilizamos o módulo random para escolher aleatoriamente um dos algoritmos presentes na lista ALGORITHMS. Guardamos o tipo de cifra e o respectivo modo, efectuando um *split* por hífen (decidimos separar a cifra por hífen mas poderíamos ter definido de várias maneiras como, por exemplo, espaço). Definimos também a função de síntese seguindo a mesma lógica, através de uma escolha aleatória. Com estes 3 elementos podemos formalizar o tipo de cifra a utilizar. Construímos então uma mensagem JSON, codificamos e enviamos para o servidor.

```
def pick_cipher(self) -> None:
    pick = random.choice(ALGORITHMS)
    self.cipher, self.mode = pick.split("-")[0], pick.split("-")[1]
    self.sintese = random.choice(SYNTHESIS)
    ciphersuite = (self.cipher + "-" + self.mode + "-" + self.sintese).encode()
    print(ciphersuite)
    msg = {'type': 'TO_CIPHER', 'data': None}
    msg['data'] = base64.b64encode(ciphersuite).decode()
    self._send(msg)
```

Na função pick_diffie_parameters geramos os parâmetros necessários para o algoritmo Diffie-Hellman e enviamo-los para o servidor.

```
def pick_diffie_parameters(self) -> None:
    self.diffie = dh.generate_parameters(generator=2, key_size=512, backend=default_backend())
    diffie_pem = self.diffie.parameter_bytes(Encoding.PEM, ParameterFormat.PKCS3)
    msg = {'type': 'DIFFIE_PARAMETERS', 'data': None}
    msg['data'] = base64.b64encode(diffie_pem).decode()
    self._send(msg)
```

Na função build_keys geramos as chaves privada e pública do cliente, utilizando o algoritmo descrito no parágrafo anterior. Posteriormente, enviamos uma mensagem JSON codificada para o servidor que contém apenas a chave pública do cliente.

```
def build_keys(self) -> None:
    self.priv_key_client = self.diffie.generate_private_key()
    pub_key_client = self.priv_key_client.public_key()
    pub_key_to_bytes = pub_key_client.public_bytes(Encoding.PEM, PublicFormat.SubjectPublicKeyInfo)
    msg = {'type': 'KEYS', 'data': None}
    msg['data'] = base64.b64encode(pub_key_to_bytes).decode()
    self._send(msg)
```

Na função pick_secret, utilizada para gerar o secret corresponde à comunicação entre as duas entidades, começamos por ler a chave pública do servidor e geramos a síntese. Definimos também o tamanho da chave conforme o tipo de cifra escolhido. Terminamos ao derivar a chave.

```
def pick_secret(self) -> None:
    load_server_pub_key = load_pem_public_key(self.pub_key_server, backend=default_backend())
    secret = self.priv_key_client.exchange(load_server_pub_key)

    if (self.sintese == "SHA512"):
        sintese = hashes.SHA512()
    else:
        sintese = hashes.SHA256()

    if ("CHACHA20" in self.cipher):
        size = int(int(self.cipher.split("CHACHA20")[1])/8)
    else:
        size = int(int(self.cipher.split("AES")[1])/8)
    kdf = HKDF(algorithm=sintese, length=size, salt=None, info=b'handshake data', backend=default_backend())
    self.key = kdf.derive(secret)
    print(self.key)
```

Na função encrypt encriptamos o ficheiro pretendido de acordo com o algoritmo e respectivo modo. Para a encriptação seguimos os passos descritos no site https://cryptography.io/. No caso do algoritmo ser ChaCha20, geramos um vetor inicial (iv) e posteriormente um objecto Cipher e terminamos com a encriptação. No caso do algoritmo ser AES temos dois modos possíveis: GCM e CBC. Em ambos os modos começamos por gerar um vetor inicial, aleatório, e um objeto cipher. A diferença encontra-se no tamanho do vetor inicial.

Por fim geramos um MAC e juntamo-lo ao vetor inicial, enviando-o para o servidor.

```
def encrypt(self) -> None:
    to_encrypt = open(self.file_name, 'rb').read()

    if "CHACHA20" in self.cipher:
        self.iv = os.urandom(16)
        algorithm = algorithms.ChaCha20(self.key, self.iv)
        c = Cipher(algorithm, mode=None, backend=default_backend())
        text = self.chacha_encrypt(to_encrypt, c)

    elif "AES" in self.cipher and self.mode == 'GCM':
        self.iv = os.urandom(12)
        c = Cipher(algorithms.AES(self.key), modes.GCM(self.iv), backend=default_backend())
        text = self.aes_encrypt(to_encrypt, c)

    elif "AES" in self.cipher and self.mode == 'CBC':
        self.iv = os.urandom(16)
        c = Cipher(algorithms.AES(self.key), modes.CBC(self.iv), backend=default_backend())
        text = self.aes_encrypt(to_encrypt, c)

    mac = self.generate_mac(text)
    message = {'type':'IV & MAC', 'data': None}
    message['data'] = base64.b64encode(self.iv + mac).decode()
    self._send(message)
```

Para a encriptação final utilizando o algoritmo ChaCha utilizamos a função chacha_encrypt, e terminamos escrevendo o resultado final no ficheiro encrypt.txt. Para a função aes_encrypt, seguimos a mesma lógica, considerando que o modo pode ser GCM, e caso não seja, será o modo CBC. Nesta função também escrevemos o resultado num ficheiro.

```python
def chacha_encrypt(self, message, cipher):
    to_write = open('encrypt.txt', 'wb')
    encryptor = cipher.encryptor()
    c = encryptor.update(message)

    to_write.write(c)
    return c

def aes_encrypt(self, message, cipher):
    to_write = open('encrypt.txt', 'wb')
    encryptor = cipher.encryptor()
    if self.mode == 'GCM':
        c = encryptor.update(message)
    else:
        c = encryptor.update(self.padding_text(message)) + encryptor.finalize()

    to_write.write(c)
    return c
```

Sendo uma cifra por blocos é necessário que o tamanho do texto a cifrar tenha um tamanho específico, do tamanho dos blocos ou de um múltiplo de um tamanho. Para essa finalidade utilizamos a função padding_text.

Para gerar o MAC utilizamos a função generate_mac, que gera um MAC conforme o algoritmo de síntese escolhido (SHA512 ou SHA256).

```python
def padding_text(self, text):
    padder = padding.PKCS7(128).padder()
    data = padder.update(text)
    data += padder.finalize()

    return data

def generate_mac(self, text):
    if (self.sintese == "SHA512"):
        algorithm = hashes.SHA512()
    else:
        algorithm = hashes.SHA256()

    h_mac = hmac.HMAC(self.key, algorithm, backend=default_backend())
    h_mac.update(text)

    return h_mac.finalize()
```

- Server.py

Como aconteceu com o ficheiro Client.py, começamos por atualizar a função on_frame e acrescentamos também novos tipos de mensagens, o seu funcionamento é idêntico ao explicado acima para o ficheiro Client.py.

```
def on_frame(self, frame: str) -> None:
    """
    Called when a frame (JSON Object) is extracted

    :param frame: The JSON object to process
    :return:
    """
    #logger.debug("Frame: {}".format(frame))

    try:
        message = json.loads(frame)
    except:
        logger.exception("Could not decode JSON message: {}".format(frame))
        self.transport.close()
        return

    mtype = message.get('type', "").upper()

    if mtype == 'OPEN':
        ret = self.process_open(message)
    elif mtype == 'TO_CIPHER':
        ret = self.process_cipher(message)
        logger.info("Ciphersuite has been received.")
    elif mtype == "DIFFIE_PARAMETERS":
        ret = self.process_diffie_parameters(message)
        logger.info("Diffe-Hellman has been received.")
    elif mtype == "KEYS":
        ret = self.process_keys(message)
        logger.info("Public Key has been sent.")
    elif mtype == "SECRET":
        ret = self.process_secret(message)
        logger.info("Building secret...")
    elif mtype == "IV & MAC":
        ret = self.process_iv_mac(message)
        logger.info("IV & MAC have been received.")
    elif mtype == 'DATA':
        ret = self.process_data(message)
    elif mtype == 'CLOSE':
        self.decrypt()
        ret = self.process_close(message)
```

De seguida, definimos a função process_cipher. Nesta função verificamos se o self.state é igual ao STATE_OPEN. Descodificamos a mensagem JSON que nos foi enviada pelo cliente na função pick_cipher e guardamos o tipo de cifra, o respectivo modo, e que algoritmo de síntese utilizado efetuando um *split* por hífen.

No final, atualizamos o state para STATE_TO_CIPHER, se tudo correr como desejado enviamos uma mensagem para o cliente a confirmar que tudo correu como esperado.

```python
def process_cipher(self, message: str) -> bool:
    logger.debug("Process Cipher: {}".format(message))

    if self.state == STATE_OPEN:
        # First Packet
        data = base64.b64decode(message['data']).decode("utf-8").split("-")
        print(data)
        self.cipher = data[0]
        self.mode = data[1]
        self.sintese = data[2]
        self.state = STATE_TO_CIPHER

    else:
        logger.warning("Invalid state. Discarding")
        return False

    try:
        data = message.get('data', None)
        if data is None:
            logger.debug("Invalid message. No data found")
            return False

        bdata = base64.b64decode(message['data'])
    except:
        logger.exception("Could not decode base64 content from message.data")
        return False

    self._send({'type':'OK'})
    return True
```

A função process_diffie_parameters recebe os parâmetros da função pick_diffie_parameters para realizarmos o processamento de todos os campos necessários para reproduzirmos o algoritmo.No final, atualizamos o state para STATE_DIFFIE_PARAMETERS, se tudo correr como desejado enviamos uma mensagem para o cliente a confirmar que tudo correu como esperado.

```python
def process_diffie_parameters(self, message: str) -> bool:
    logger.debug("Process Diffie-Hellman Parameters: {}".format(message))

    if self.state == STATE_TO_CIPHER:
        data = base64.b64decode(message.get('data', "").encode())
        self.diffie_parameters = load_pem_parameters(data, backend=default_backend())
        self.state = STATE_DIFFIE_PARAMETERS

    else:
        logger.warning("Invalid state. Discarding")
        return False

    try:
        data = message.get('data', None)
        if data is None:
            logger.debug("Invalid message. No data found")
            return False

        bdata = base64.b64decode(message['data'])
    except:
        logger.exception("Could not decode base64 content from message.data")
        return False

    self._send({'type':'OK'})
    return True
```

Na função process_keys codificamos as chaves que são geradas na build_keys do cliente, posteriormente transformamos a chave no formato PEM, posteriormente atualizamos o state para STATE_KEY, se tudo correr como desejado enviamos uma mensagem para o cliente a confirmar que tudo correu como esperado.

```
def process_keys(self, message: str) -> bool:
    logger.debug("Process Keys: {}".format(message))

    if self.state == STATE_DIFFIE_PARAMETERS:
        self.pub_key_client = base64.b64decode(message.get('data', "").encode())
        logger.info("Client Public Key has been received.")
        self.priv_key_server = self.diffie_parameters.generate_private_key()
        pub_key_server = self.priv_key_server.public_key()
        pub_key_server_to_bytes = pub_key_server.public_bytes(Encoding.PEM, PublicFormat.SubjectPublicKeyInfo)
        self.state = STATE_KEY


    else:
        logger.warning("Invalid state. Discarding")
        return False

    try:
        data = message.get('data', None)
        if data is None:
            logger.debug("Invalid message. No data found")
            return False

        bdata = base64.b64decode(message['data'])
    except:
        logger.exception("Could not decode base64 content from message.data")
        return False

    message = {'type': 'OK', 'data': None}
    message['data'] = base64.b64encode(pub_key_server_to_bytes).decode()
    self._send(message)
    return True
```

Na função process_secret lemos a chave pública do cliente e criamos o “secret” através da chave privada do servidor. De seguida verificamos que algoritmo de síntese e cifra está a ser utilizado. Definimos utilizar o HKDF para conseguirmos derivar o nosso “secret”. Posteriormente atualizamos o state para STATE_SECRET, se tudo correr como desejado enviamos uma mensagem para o cliente a confirmar que tudo correu como esperado.

```
def process_secret(self, message: str) -> bool:
    logger.debug("Process Secret: {}".format(message))

    if self.state == STATE_KEY:
        load_client_pub_key = load_pem_public_key(self.pub_key_client, backend=default_backend())
        secret = self.priv_key_server.exchange(load_client_pub_key)

        if (self.sintese == "SHA512"):
            sintese = hashes.SHA512()
        else:
            sintese = hashes.SHA256()

        if ("CHACHA20" in self.cipher):
            size = int(int(self.cipher.split("CHACHA20")[1])/8)
        else:
            size = int(int(self.cipher.split("AES")[1])/8)

        kdf = HKDF(algorithm=sintese, length=size, salt=None, info=b'handshake data', backend=default_backend())
        self.key = kdf.derive(secret)
        print(self.key)
        self.state = STATE_SECRET
    else:
        logger.warning("Invalid state. Discarding")
        return False

    self._send({'type':'OK'})
    return True
```

Na função process_iv_mac, decodificamos a mensagem e verificamos que cifra e modo de cifra para sabermos que valor de IV e MAC devemos utilizar. Posteriormente, se tudo correr como desejado enviamos uma mensagem para o cliente a confirmar que tudo correu como esperado.

```
def process_iv_mac(self, message: str) -> bool:
    logger.debug("Process IV & MAC: {}".format(message))

    if self.state == STATE_SECRET:
        data = base64.b64decode(message['data'])
        if "AES" in self.cipher and self.mode == "CBC":
            self.iv = data[:16]
            self.mac = data[16:]
        elif "AES" in self.cipher and self.mode == "GCM":
            self.iv = data[:12]
            self.mac = data[12:]
        else:
            self.iv = data[:16]
            self.mac = data[16:]

    else:
        logger.warning("Invalid state. Discarding")
        return False

    try:
        data = message.get('data', None)
        if data is None:
            logger.debug("Invalid message. No data found")
            return False

        bdata = base64.b64decode(message['data'])
    except:
        logger.exception("Could not decode base64 content from message.data")
        return False

    self._send({'type':'OK'})
    return True
```

Na função decrypt desencriptamos o ficheiro pretendido de acordo com o algoritmo e respectivo modo. Para a desencriptação seguimos os passos descritos no site https://cryptography.io/. E ajustamos a desencriptação aos algoritmos e ao respetivo modo.

Na função check_mac verificamos qual algoritmo de síntese utilizamos e verificamos para validar o mac.

```
def decrypt(self):
    if not self.check_mac(self.text):
        logger.warning("Message integrity violated. Discarding")
        return False

    if "CHACHA20" in self.cipher:
        algorithm = algorithms.ChaCha20(self.key, self.iv)
        c = Cipher(algorithm, mode=None, backend=default_backend())
        self.chacha_decrypt(self.text, c)

    if "AES" in self.cipher:
        if self.mode == 'GCM':
            c = Cipher(algorithms.AES(self.key), modes.GCM(self.iv), backend=default_backend())
        elif self.mode == 'CBC':
            c = Cipher(algorithms.AES(self.key), modes.CBC(self.iv), backend=default_backend())
        self.aes_decrypt(self.text, c)

def check_mac(self, text):
    if (self.sintese == "SHA512"):
        algorithm = hashes.SHA512()
    else:
        algorithm = hashes.SHA256()
    m = hmac.HMAC(self.key, algorithm, backend=default_backend())
    m.update(text)

    try:
        m.verify(self.mac)
        return True
    except:
        return False
```

Para a desencriptação final utilizando o algoritmo ChaCha utilizamos a função chacha_decrypt, e terminamos escrevendo o resultado final no ficheiro done.txt. Para a função aes_decrypt, seguimos a mesma lógica, considerando que o modo pode ser GCM, e caso não seja, será o modo CBC. Nesta função também escrevemos o resultado num ficheiro descodificado.

```
def chacha_decrypt(self, message, cipher):
    to_write = open('done.txt', 'w')
    decryptor = cipher.decryptor()
    dc = decryptor.update(message)
    to_write.write(dc.decode())
    to_write.close()

def aes_decrypt(self, message, cipher):
    to_write = open('done.txt', 'w')
    decryptor = cipher.decryptor()

    if self.mode == 'GCM':
        dc = decryptor.update(message)

    elif self.mode == 'CBC':
        dc = decryptor.update(message) + decryptor.finalize()
        unpadder = padding.PKCS7(128).unpadder()
        dc = unpadder.update(dc) + unpadder.finalize()

    to_write.write(dc.decode())
    to_write.close()
```

Conclusão

Em conclusão, com este trabalho consolidamos conhecimentos adquiridos nas aulas teóricas e também nas aulas práticas, tais como: troca de mensagens seguras, cifras simétricas, funções de síntese, confidencialidade, integridade e negociação de chaves.

Executando o código desenvolvido obtemos o seguinte output:

1) python server.py

```
(venv) user@vm:~/79923_80041$ python server.py
2019-11-18 14:27:45 vm root[2962] INFO Port: 5000 LogLevel: 20 Storage: /home/user/79923_80041/files
[2019-11-18 14:27:45 +0000] [2962] [INFO] Single tcp server starting @0.0.0.0:5000, Ctrl+C to exit
2019-11-18 14:27:45 vm aio-tcpsever[2962] INFO Single tcp server starting @0.0.0.0:5000, Ctrl+C to exit
[2019-11-18 14:27:45 +0000] [2962] [INFO] Starting worker [2962]
2019-11-18 14:27:45 vm aio-tcpsever[2962] INFO Starting worker [2962]
2019-11-18 14:27:53 vm root[2962] INFO
```

Depois de executarmos este comando o server fica à espera de uma resposta do cliente, que ainda não foi executado.

2) python cliente.py poema.txt (poderia ser utilizado outro ficheiro)

```
(venv) user@vm:~/79923_80041$ python client.py poema.txt
2019-11-18 14:27:53 vm root[2971] INFO Sending file: /home/user/79923_80041/poema.txt to 127.0.0.1:5000 LogLevel: 20
2019-11-18 14:27:53 vm root[2971] INFO Channel open
2019-11-18 14:27:53 vm root[2971] INFO Ciphersuite has been sent.
2019-11-18 14:27:53 vm root[2971] INFO Diffie-Hellman Parameters have been sent.
2019-11-18 14:27:53 vm root[2971] INFO Public key has been sent.
2019-11-18 14:27:53 vm root[2971] INFO Server Public Key has been received.
2019-11-18 14:27:53 vm root[2971] INFO File transferred. Closing transport
2019-11-18 14:27:53 vm root[2971] INFO The server closed the connection
(venv) user@vm:~/79923_80041$
```

3) Após a execução do cliente, o servidor produz o seguinte output no servidor:

```
Connection from ('127.0.0.1', 33524)
2019-11-18 14:33:06 vm root[3325] INFO File open
2019-11-18 14:33:06 vm root[3325] INFO Ciphersuite has been received.
2019-11-18 14:33:07 vm root[3325] INFO Diffe-Hellman has been received.
2019-11-18 14:33:07 vm root[3325] INFO Client Public Key has been received.
2019-11-18 14:33:07 vm root[3325] INFO Public Key has been sent.
2019-11-18 14:33:07 vm root[3325] INFO Building secret...
2019-11-18 14:33:07 vm root[3325] INFO IV & MAC have been received.
2019-11-18 14:33:07 vm root[3325] INFO Done.
```

No lado do servidor somos informados que a troca do ficheiro foi concluída através da mensagem “Done”.

Webgrafia

- https://cryptography.io/
- https://pt.wikipedia.org/wiki/Advanced_Encryption_Standard
- https://pt.wikipedia.org/wiki/SHA-2
- https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation
- https://en.wikipedia.org/wiki/Galois/Counter_Mode
- https://pt.wikipedia.org/wiki/Diffie-Hellman
- https://pt.wikipedia.org/wiki/Algoritmo_de_chave_sim%C3%A9trica
- https://pplware.sapo.pt/tutoriais/networking/criptografia-simetrica-e-assimetrica-sabe-a-diferenca/
- https://en.wikipedia.org/wiki/Message_authentication_code
- Conteúdos das aulas teóricas
- Conteúdos das aulas práticas