UNIVERSIDADE DO MINHO

DEPARTAMENTO DE INFORMÁTICA

ESTRUTURAS CRIPTOGRÁFICAS

Trabalho 1

Grupo 10

Autores:

Joel Gama (A82202)

Tiago Pinheiro (A82491)

16 de Março de 2020

Conteúdo

Introdução

O presente relatório surge no âmbito da Unidade Curricular de Estruturas Criptográficas integrada no perfil de Criptografia e Segurança da Informação.

Este trabalho é o segundo de cinco trabalhos que ainda irão ser abordados nesta UC. O trabalho vai ter duas versões, sendo que, em cada uma delas são usados tipos de cifras e acordos de chaves diferentes. Em comum, tem que em ambas é necessário construir uma sessão síncrona de comunicação segura entre dois agentes (o *Emitter* e o *Receiver*), apesar de grande parte do código ser reutilizável.

Na primeira versão é necessário utilizar a cifra simétrica **AES**, usando autenticação de cada criptograma com **HMAC** e um modo seguro contra ataques aos vectores de iniciação. Para além disso, o protocolo acordo de chaves a usar é o **DH** (*Diffie-Hellman*) sendo a autenticação dos agentes feita através do esquema de assinaturas **DSA**.

Na segunda versão, em vez de utilizar-se a cifra simétrica **AES** é necessário usar a **ChaCha20Poly1305**. E, no protocolo de acordo de chaves, deve-se substituir o **DH** e o **DSA** pelo **ECDH** e pelo **ECDH**, respetivamente.

Versão com *AES*, *DH* e *DSA*

Nesta versão da implementação é pedido que estejam presentes três partes essenciais:

- Um gerador de *nounces*: um *nounce*, que nunca foi usado antes, deve ser criado aleatoriamente em cada instância da comunicação
- Utilização da cifra *AES* com autenticação de cada criptograma com *HMAC* e um modo seguro contra ataques aos vectores de iniciação (*iv*)
- Protocolo de acordo de chaves *Diffie-Hellman* (*DH*) com verificação da chave, e autenticação dos agentes através do esquema de assinaturas *DSA*

2.1 Gerador de *nounces*

Para gerar um *nounce* que não tivesse sido utilizado anteriormente seria necessário manter em memória uma grande quantidade de informação. Então tivemos de balancear a decisão abdicando da restrição que garante que o *nounce* nunca foi utilizado. Porém temos o cuidado de gerar um *nounce* suficientemente grande para que a probabilidade de ele se repetir seja baixa.

```
inicial = os.urandom(512)
nonce = random.choices(inicial, k=16)
iv = np.asarray(nonce)
```

Para gerar o *nounce* é criado um vetor inicial com tamanho 512. Em seguida são retirados 16 bytes aleatoriamente desse vetor inicial. O *nounce* utilizado na cifra é o resultado dessa seleção aleatória do vetor inicial.

O tamanho do *nounce* é de 16 pois o modo utilizado recebe tamanhos específicos para os vetores de inicialização.

2.2 *Diffie-Hellman* e *DSA*

Foi implementado o protocolo de acordo de chaves *Diffie-Hellman* com verificação da chave e autenticação mútua dos agente através do esquema de assinaturas *Digital Signature Algorithm* (*DSA*).

Nesta parte da implementação são criados os parâmetros tanto para o *DH* como para o *DSA*. O *Emitter* gera a chave privada, a sua respetiva chave pública e envia ao *Receiver* e o *Receiver* faz o mesmo. No final, ambos geram a chave partilhada e é usada uma autenticação MAC na respetiva chave na comunicação entre os agente.

```
# Receiver
   # Gerar parâmetros do DH
   parameters_dh = dh.generate_parameters(generator=2,
                                key_size=2048,
                                backend=default_backend())

    # Gerar parâmetros do DSA
    parameters_dsa = dsa.generate_parameters(key_size=2048,
                                backend=default_backend())

    dsa_private_key = self.parameters_DSA.generate_private_key()
    dsa_public_key  = dsa_private_key.public_key()

    self.server_private_key = self.parameters_DH.
                                generate_private_key()
    self.public_key         = self.server_private_key.
                                public_key()

    param_dh  = self.parameters_DH.
                            parameter_bytes(Encoding.DER,
                            ParameterFormat.PKCS3)
    pubK_dh   = self.public_key.
                            public_bytes(Encoding.DER,
                            PublicFormat.SubjectPublicKeyInfo)
    pubK_dsa  = dsa_public_key.
                            public_bytes(Encoding.DER,
                            PublicFormat.SubjectPublicKeyInfo)

   sig = dsa_private_key.sign(
         pubK_dh,
         hashes.SHA256()
   )

   new_msg = {
         "parameters_DH": param_dh,
         "public_key_DH": pubK_dh,
         "public_key_DSA": pubK_dsa,
```

```
        "signature": sig
    }

# Emitter
    parameters_dh  = msg_dict['parameters_DH']
    public_key_DH  = msg_dict['public_key_DH']
    public_key_DSA = msg_dict['public_key_DSA']
    sig = msg_dict['signature']

    parameters_DH     = load_der_parameters(parameters_dh,
                                    backend=default_backend())
    server_public_key = load_der_public_key(public_key_DH,
                                    backend=default_backend())
    server_dsa_pub_key = load_der_public_key(public_key_DSA,
                                    backend=default_backend())

    self.client_private_key = parameters_DH.generate_private_key()
    self.client_public_key  = self.client_private_key.public_key()
    self.shared_key         = self.client_private_key.
                                        exchange(server_public_key)

    signature = server_dsa_pub_key.verify(
         sig,
         public_key_DH,
         hashes.SHA256()
    )
```

2.3 Cifra *AES*

Para a comunicação entre os agentes foi implementada a cifra *AES* com o modo *Cipher Feedback* (*CFB*) como forma adicionar de combate a ataques ao vetor de inicialização. Mesmo que o atacante saiba de início o valor do *iv* ele apenas sabe o primeiro bloco que será apresentado ao bloco da cifra. Desde que o vetor de inicialização não se repita (o que é assegurado com o gerador de *nounces*) podemos concluir que este modo é seguro contra ataques ao vetor de inicialização.

```
# Emitter
    inicial = os.urandom(512)
    nonce = random.choices(inicial, k=16)
    nounce = np.asarray(nonce)

    if len(self.shared_key) not in (16, 24, 32):
        key = hashlib.sha256(self.shared_key).digest()

    mac = hmac.HMAC(key,hashes.SHA256(),default_backend())

    cipher = Cipher(algorithms.AES(key), modes.CFB(nounce),
                                    backend=default_backend())
    encryptor = cipher.encryptor()

    print('Input message to send (empty to finish)')
    data = input()
    message = data.encode('utf-8')

    ct = encryptor.update(message) + encryptor.finalize()
    print('Received (%d): %r' % (self.msg_cnt,data))

    new_msg = {
        "nonce": nonce,
        "mac": mac.finalize(),
        "ct": ct
    }

# Receiver
    if len(self.shared_key) not in (16, 24, 32):
        key = hashlib.sha256(self.shared_key).digest()

    mac = hmac.HMAC(key,hashes.SHA256(),default_backend())
    mac = mac.finalize()

    if (mac == msg_dict['mac']):
        nonce = msg_dict['nonce']
        nounce = np.asarray(nonce)
```

```
        cipher = Cipher(algorithms.AES(key), modes.CFB(nounce),
                                    backend=default_backend())

        decryptor = cipher.decryptor()
        mensagem = decryptor.update(msg_dict['ct'])

        print('%d : %r' % (self.id, mensagem.decode('utf-8')))
        msg_dict['ct'] = mensagem

        msg = msg_dict

    else:
        print("Erro no MAC!")
```

Versão com *ChaCha20Poly1305*, *ECDH* e *ECDSA*

Na segunda versão deste trabalho pedia-se para usar a cifra simétrica **ChaCha20Poly1305** e, no acordo de chaves, substitui o **DH** pelo **ECDH** e o **DSA** pelo **ECDSA**.

3.1 *Emitter*

Essencialmente,

3.1.1 Primeira parte

Na primeira parte, o *Emitter* vai enviar uma mensagem simples ao *Receiver* apenas para assinalar a sua presença e dizer que está pronto para começar o acordo de chaves.

```
new_msg = bytes("Hello".encode('utf-8'))
self.msg_cnt += 1
```

3.1.2 Segunda Parte

O seguinte código é executado quando o *Emitter* envia a segunda mensagem, ou seja, quando a variável `msg\_cnt` é igual a `1`.

O programa começa por receber e guardar a chave pública enviada pelo *Receiver* e depois gerar as suas chaves pública e privada. No final, é gerada também a *shared key*.

```
msg = msg.decode()
msg_dict = ast.literal_eval(msg)

pub_key = msg_dict['pubKey']

server_public_key = load_der_public_key(pub_key,
                            backend=default_backend())

self.client_private_key = ec.generate_private_key(
                           ec.SECP256K1(), default_backend())
```

```
self.client_public_key = self.client_private_key.public_key()

self.shared_key = self.client_private_key.exchange(
                            ec.ECDH(),server_public_key)
```

De seguida, já é possível enviar uma mensagem cifrada ao *Receiver*. Portanto, começa-se por gerar uma **key**, de 32 *bytes* - tamanho pedido pela cifra, e um **nounce** obrigatório de 12 *bytes*. Depois é espera-se um input, essa mensagem é cifrada com a cifra simétrica *ChaCha20Poly1305*, usando a *key* e o *nounce* gerados anteriormente.

```
key = os.urandom(32)
nounce = os.urandom(12)

cip = ChaCha20Poly1305(key)

print('Input message to send (empty to finish)')
data = input()
message = data.encode('utf-8')

ciphertext = cip.encrypt(nounce,message,None)
```

Por fim, apenas falta assinar o texto cifrado com a chave privada do *Emitter* e os componentes necessários são colocados num dicionário para enviar.

```
signature = self.client_private_key.sign(
                        ciphertext,
                        ec.ECDSA(hashes.SHA256()))

new_msg = {
    "nounce": nounce,
    "key": key,
    "ct": ciphertext,
    "sign": signature,
    "pub_key": self.client_public_key.public_bytes(
        Encoding.DER,PublicFormat.SubjectPublicKeyInfo)
}
```

3.1.3 Terceira Parte

O princípio é o mesmo que a parte anterior mas, nesta fase, apenas se executam os dois últimos blocos de código mostrados em cima. Ou seja, não são geradas chaves nem se envia a chave pública. Apenas cifra-se e assina-se a mensagem.

3.2 *Receiver*

Tal como a classe *Emitter*, o *Receiver* também se encontra dividido em 3 partes distintas.

3.2.1 Primeira parte

Depois de receber a mensagem do *Emitter* a iniciar a conversação, o *Receiver* gera as suas chaves privada e pública, enviando esta para o *Emitter*.

```
self.server_private_key = ec.generate_private_key(
                ec.SECP256K1(), default_backend())

self.server_public_key = self.server_private_key.public_key()

new_msg = {
        "pubKey": self.server_public_key.public_bytes(
        Encoding.DER,PublicFormat.SubjectPublicKeyInfo)
}
```

3.2.2 Segunda parte

Depois de receber a segunda mensagem do *Emitter*, o *Receiver* guarda a chave pública deste e gera a *shared key*. De seguida, o *Receiver* usa a *key* para criar uma cifra simétrica igual à do *Emitter* e, desta forma, utilizando também o *nounce*, decifrar a mensagem. Mas, antes de decifrar, o *Receiver* vai validar o criptograma enviado, que tinha sido assinado pelo *Emitter*

```
msg = msg.decode()
msg_dict = ast.literal_eval(msg)

nounce = msg_dict['nounce']
key = msg_dict['key']
ciphertext = msg_dict['ct']
signature = msg_dict['sign']
pub_key = msg_dict['pub_key']

self.client_public_key = load_der_public_key(pub_key,
                            backend=default_backend())

self.shared_key = self.server_private_key.exchange(
                    ec.ECDH(), self.client_public_key)

cip = ChaCha20Poly1305(key)

    try:
        self.client_public_key.verify(signature,
                ciphertext,ec.ECDSA(hashes.SHA256()))
```

```
        message = cip.decrypt(nounce,ciphertext,None)
    except:
        print("Error while decrypt")
```

3.2.3 Terceira parte

Como acontece no *Emitter*, a terceira parte do *Receiver* também é muito idêntica à anterior, tirando a parte do acordo de chaves - que já não existe.

Conclusão

Este trabalho foi importante para recordar alguma da matéria dada no semestre passado e, também, serviu como forma de aplicar a matéria dada nas aulas ao longo deste semestre.

No trabalho foi possível aplicar conceitos relacionados com os ataques a vetores de inicialização. Para evitar os ataques, foram gerados *nounces* nunca usados e com a escolha de um modo de cifra apropriado. E também acordo de chaves com o protocolo *Diffie-Hellman* e *DSA*.

Para além disso, também foram abordados conceitos de curvas elípticas: foi usado o acordo de chaves *ECDH* e a validação de assinaturas foi feita com *ECDSA*.

Numa análise final, o grupo acha que respondeu com sucesso a todas as perguntas mencionadas no enunciado. Porém não foi possível finalizar o programa na tecnologia pedida pelo docente. Apesar de os ficheiros *jupyter notebook* estarem bastante completos faltou finalizar a parte final. Enquanto isso o programa encontra-se funcional nas classes *python*.