Semana 8

Utilização de Certificados

O recurso a criptografia assimétrica pressupõe tipicamente a utilização de certificados que estabeleçam a autenticidade das chaves públicas utilizadas. Vamos por isso estender a aplicação que tem vindo a ser construída com certificados X509.

Os certificados que iremos utilizar irão conter chaves RSA. Em concreto, são fornecidos os ficheiros[1]:

- `VAULT_CA.crt` - certificado da autoridade de certificação;
- `VAULT_CLI1.p12` - ficheiro PKCS#12 com a chave privada e certificado de CLI1 (obs: ficheiro protegido com password vazia), e que vamos designar por *ALICE*;
- `VAULT_SERVER.p12` - ficheiro PKCS#12 com a chave privada e certificado do servidor (obs: ficheiro protegido com password vazia), e que vamos designar por *BOB*.

Para extrair o certificado de um ficheiro `.p12`, podem executar o seguinte comando:

```
openssl pkcs12 -in xxx.p12 -clcerts -nokeys -out yyy.crt
```

Para extrair a chave privada, podem executar os seguinte comandos:

```
openssl pkcs12 -in xxx.p12 -nodes -nocerts | openssl rsa -out yyy.key
```

Para imprimirem o conteúdo de um certificado no ecrã e verem os diferentes campos, podem usar o comando (para um certificado `xxx.crt`):

```
$ openssl x509 -text -noout -in xxx.crt
```

De igual forma, se pretenderem visualizar o conteúdo da chave privada podem usar:

```
$ openssl rsa -text -noout -in xxx.key
```

QUESTÃO: Q1

Como pode verificar que as chaves fornecidas nos ficheiros mencionados (por exemplo, em `VALUT_SERVER.p12`) constituem de facto um par de chaves RSA válido?

Validação de certificados

Naturalmente que a utilização de certificados pressupõe que estes sejam devidamente **validados**. Por regra, a validação de certificado passa por, para cada certificado da cadeia de certificação:

[1] Na realidade, os ficheiros fornecidos são precisamente os adotados no trabalho prático.

1. validar período de validade estabelecido no certificado;
2. validar o titular do certificado;
3. validar a aplicabilidade do certificado (i.e. se o conteúdo indica que o certificado é aplicável para a utilização pretendida);
4. validar assinatura contida no certificado – passo este que, ao necessitar da chave pública (certificado) da EC emitente, pode requerer subir recursivamente na cadeia de certificação até atingir uma *trust-anchor* (uma entidade em quem se deposita confiança).

Infelizmente, o suporte da biblioteca `cryptography` para a validação de certificados é muito insipiente! Uma alternativa seria recorrer a bibliotecas alternativa, mas acaba por não se justificar atendendo que a utilização que necessitamos é muito básica. Por isso, sugere-se a adopção/adaptação dos seguintes métodos que, sendo em grande medida uma simplificação do mecanismo, acabam para validar os campos essenciais para o fim em vista.

```
from cryptography import x509
import datetime


def cert_load(fname):
    """lê certificado de ficheiro"""
    with open(fname, "rb") as fcert:
        cert = x509.load_pem_x509_certificate(fcert.read())
    return cert


def cert_validtime(cert, now=None):
    """valida que 'now' se encontra no período
    de validade do certificado."""
    if now is None:
        now = datetime.datetime.now(tz=datetime.timezone.utc)
    if now < cert.not_valid_before_utc or now > cert.not_valid_after_utc:
        raise x509.verification.VerificationError(
            "Certificate is not valid at this time"
        )


def cert_validsubject(cert, attrs=[]):
    """verifica atributos do campo 'subject'. 'attrs'
    é uma lista de pares '(attr,value)' que condiciona
    os valores de 'attr' a 'value'."""
    print(cert.subject)
    for attr in attrs:
        if cert.subject.get_attributes_for_oid(attr[0])[0].value != attr[1]:
            raise x509.verification.VerificationError(
                "Certificate subject does not match expected value"
```

```
            )


def cert_validexts(cert, policy=[]):
    """valida extensões do certificado.
    'policy' é uma lista de pares '(ext,pred)' onde 'ext' é o OID de uma extensão e 'pred'
    o predicado responsável por verificar o conteúdo dessa extensão."""
    for check in policy:
        ext = cert.extensions.get_extension_for_oid(check[0]).value
        if not check[1](ext):
            raise x509.verification.VerificationError(
                "Certificate extensions does not match expected value"
            )


def valida_certALICE(ca_cert):
    try:
        cert = cert_load("ALICE.crt")
        # obs: pressupõe que a cadeia de certifica só contém 2 níveis
        cert.verify_directly_issued_by(ca_cert)
        # verificar período de validade...
        cert_validtime(cert)
        # verificar identidade... (e.g.)
        cert_validsubject(cert, [(x509.NameOID.COMMON_NAME, "ALICE")])
        # verificar aplicabilidade... (e.g.)
        # cert_validexts(
        #     cert,
        #     [
        #         (
        #             x509.ExtensionOID.EXTENDED_KEY_USAGE,
        #             lambda e: x509.oid.ExtendedKeyUsageOID.CLIENT_AUTH in e,
        #         )
        #     ],
        # )
        print("Certificate is valid!")
        return True
    except Exception as e:
        print("Certificate is invalid!")
        print(e)
        return False
```

QUESTÃO: Q2

Visualize o conteúdo dos certificados fornecidos, e refira quais dos campos lhe parecem que devam ser objecto de atenção no procedimento de verificação.

Protocolo *Station-to-Station* simplificado

Pretende-se complementar o programa com o acordo de chaves *Diffie-Hellman* para incluir a funcionalidade análoga à do protocolo *Station-to-Station*. Recorde que nesse protocolo é adicionado uma troca de assinaturas:

1. Alice → Bob : gx
2. Bob → Alice : gy, SigB(gy, gx), CertB
3. Alice → Bob : SigA(gx, gy), CertA
4. Alice, Bob : K = g(x*y)

PROG: `Client_sts.py` e `Server_sts.py`

Algumas observações:

- O algoritmo de assinatura que iremos utilizar é o RSA, que pressupõe a utilização de um mecanismo de *padding* (e.g. `PSS`). Esse *padding* tem uma "natureza" diferente do *padding* simétrico usado noutros guiões.
- Os pares de chaves a utilizar na assinatura são os fornecidos nos ficheiros `VAULT_CLI1.{key,crt}` e `VAULT_SERVER.{key.crt}`.
- Uma possível dificuldade neste guião resulta de gerir a troca de mensagens envolvendo várias componentes cujos tamanhos não são fáceis de prever. Para isso, sugere-se que utilizem as funções apresentadas abaixo, que incluem informação dos tamanhos na **serialização** de um par de *bytestrings*:

```
def mkpair(x, y):
    """produz uma byte-string contendo o tuplo '(x,y)' ('x' e 'y' são byte-strings)"""
    len_x = len(x)
    len_x_bytes = len_x.to_bytes(2, "little")
    return len_x_bytes + x + y

def unpair(xy):
    """extrai componentes de um par codificado com 'mkpair'"""
    len_x = int.from_bytes(xy[:2], "little")
    x = xy[2 : len_x + 2]
    y = xy[len_x + 2 :]
    return x, y
```

Note que agora a função `unpair` recupera cada componente do par sem necessitar de se passar informação de tamanhos (e.g. `unpair(mkpair(b'abcde',b'99ijjhh'))` = `(b'abcde', b'99ijjhh')`).