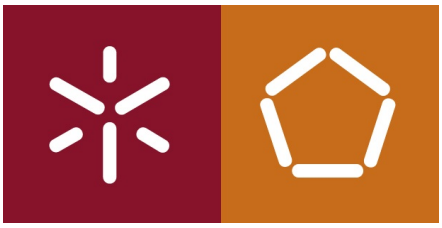

Engenharia de Segurança
Grupo 7
Aula 2

Bruno Machado - A74941
Diogo Gomes - A73825
Francisco Mendes - A75097

Fevereiro 2018

1 Números aleatórios/pseudoaleatórios

1.1

- `head -c 1024 /dev/random | openssl enc -base64`
- `head -c 1024 /dev/urandom | openssl enc -base64`

A primeira diferença que se encontra ao executar ambos os comandos é que o `/dev/random` bloqueia o terminal, o que não acontece com o segundo comando. Ao procurar explicações para este acontecimento, chegou-se à conclusão que este estava a bloquear devido à falta de entropia no sistema. Posto isto, procedeu-se à experimentação com tamanhos de chaves menores e já se obteve output utilizando o primeiro comando.

A conclusão que se pode tirar daqui é que para um uso mais prático e que não necessite de um número pseudo-aleatório seguro faz mais sentido utilizar o segundo comando dado que não é bloqueante. Em contra partida, este comando não se revela tão seguro quanto o primeiro pois não garante a mesma entropia, estando mais sujeito a ataques criptográficos que utilizem as chaves geradas para calcular a próxima.

1.2

O pacote **HAVEGE** é utilizado com o intuito de remediar situações em que exista baixa entropia. Assim, os números gerados por ambos são mais próximos de cumprir os requisitos que lhes são impostos quanto à aleatoriedade e segurança criptográfica. Como o **HAVEGE** adiciona entropia ao sistema, o primeiro comando já não bloqueia, devolvendo output de imediato.

1.3

O programa `generateSecret-app.py` chama a função `generateSecret` do módulo `shamir-secret`. O modo como o programa está implementado força a que apenas sejam retornados caracteres que sejam letras (maiúsculas ou minúsculas) ou dígitos pois, tem uma condição que sempre que seja calculado um caracter aleatório que não seja nenhum dos acima referidos vai descartar este e calcular um novo até preencher por completo o tamanho da *string* que é passado como argumento à função `generateSecret`.

2 Partilha/Divisão de segredo (Secret Sharing/Splitting)

2.1 A

Nesta primeira fase vamos correr o programa `createSharedSecret-app.py`, em que foi dividido o segredo em 7, e o seu quorum é 3.

Para executar este programa, também foi criado um par de chaves, chamado *mykey.pem*, e o correspondente certificado, *mykey.crt*, necessário para correr os programas de *recover*.

Após a execução deste, foi obtido 7 componentes, onde com o programa `recoverSecretFromComponents-app.py`, e apenas 3 desses componentes, foi possível obter o segredo antes decifrado. Já o programa `recoverSecretFromAllComponents-app.py` necessita de todos os componentes, neste caso de 7.

2.2 B

É possível observar que a diferença entre os dois programas em questão, é a quantidade de componentes necessários para obter o segredo.

Esta diferença depende da situação em que se necessita do segredo, ou seja, por vezes é necessário apenas uma maioria para revelar o segredo, nesse caso não é preciso fornecer todos os componentes, apenas a maioria deles. Por outro lado, às vezes é necessário um consenso de todos os membros, dessa forma é preciso todos os componentes para revelar o segredo.

3 Authenticated Encryption

3.1

Optou-se por utilizar uma abordagem onde se pressupõe que:

- a função `gerar_chave` retorna sempre a mesma chave enquanto a empresa que utiliza o serviço pagar a anuidade deste
- os textos cifrados possuem caraterísticas IND_CCA, fazendo com que um atacante conhecendo os textos cifrados dificilmente conseguirá obter informação sobre a chave ou os textos limpos
- a variável `etiq` é utilizada afim de etiquetar cada texto cifrado, tornando possível saber do que trata o conteúdo sem que se revele o segredo nele contido

```
def Cifrar(segredo, etiq)
    chave = gerar_chave() // chave derivada a partir da data
    ctext = cifra(segredo, chave)

    hchave = getRandomBytes(16)
    hash = hmac(hchave, etiq+ctext)

    ret = {'etiq': etiq, 'ctext': ctext,
           'hchave': hchave, 'hash': hash}
    return ret


def Decifrar(ret)
    if(ret['hash'] == hmac(ret['hchave'], ret['etiq']+ret['ctext']):
        chave = gerar_chave() // chave derivada a partir da data
        ptext = decifra(ctext, chave)

    else:
        return None
```

4 Algoritmos e tamanhos de chaves

4.1

Nome	Algoritmo de Chave Pública	Algoritmo de Hash
Ministerie van Defensie	RSA(4096 bits)	SHA-256
QuoVadis TrustLink B.V	RSA(4096 bits)	SHA-256
KPN Corporate Market	RSA(4096 bits)	SHA-256

Na nossa opinião, os algoritmos e chaves utilizadas por estas entidades certificadoras são seguras para as capacidades computacionais existentes atualmente. No entanto, o tamanho das chaves utilizadas assim como o tamanho das hash geradas podem não ser suficientes para um futuro não muito longínquo, especialmente com advento da computação quântica.

O resultado dos comandos `openssl x509 -in "certificado".crt -text -noout` dos três certificados está no ficheiro `certificados.txt`.