Universidade do Minho
Criptografia e Segurança da Informação
Departamento de Informática

Engenharia de Segurança

TP1 - Aula 2

Grupo 7
Carlos Pinto Pedrosa A77320
José Francisco Gonçalves Petejo e Igreja Matos A77688
17 de Fevereiro de 2019

Conteúdo

1 Números Aleatórios/Pseudoaleatórios

1.1 /dev/random vs /dev/urandom

```
user@CSI:~$ time head -c 1024 /dev/random | openssl enc -base64
```

Figura 1: Processo /dev/random Bloqueado

Figura 2: Resultado e Tempo de Execução do /dev/random

Figura 3: Resultado e Tempo de Execução do /dev/urandom

Tanto o */dev/random* como o */dev/urandom* são geradores de números pseudoaleatórios bastante usados no ambiente *Unix-Like*. A principal diferença entre os dois, como se pode verificar pela imagens acima, é que o primeiro bloqueia até que exista entropia suficiente para a geração do número, enquanto que no último este processo é imediato.

1.2 haveged - Daemon de Entropia

Figura 4: Tempos de Execução de /dev/random e /dev/urandom com o daemon de entropia ativo

Com a instalação e ativação de um *daemon* de entropia, a execução destes dois comandos continua a ter a diferença em cima mencionada embora não se note na sua execução. Acontece que o processo */dev/random*, que bloqueia quando não existe entropia suficiente, nunca irá bloquear pois o *haveged* gera a entropia necessária para o processo completar sem qualquer *delay*.

1.3 Geração de Segredos

Depois de analisar o código do ficheiro *generateSecret-app.py* verifica-se que a geração do número pseudoaleatório é realizada através do método *generateSecret* da biblioteca *shamirsecret*.

Assim, o próximo passo foi analisar o método *generateSecret* da biblioteca cuja definição é a seguinte:

```
def generateSecret(secretLength):
    ...
    l = 0
    secret = ""
    while (l < secretLength):
        s = utils.generateRandomData(secretLength - l)
        for c in s:
            if (c in (string.ascii_letters + string.digits)
            and l < secretLength): # Alterar esta secção
                l += 1
                secret += c
    return secret
```

Podemos verificar pelo código acima que existe uma validação de cada carácter gerado (*c in (string.ascii_letters + string.digits)*) antes deste ser adicionado ao resultado e caso este não seja visível não é adicionado à string final.

Assim, para permitir que o output não sejam só letras e dígitos apenas é necessário remover a validação anterior, dando como resultado o seguinte código:

```python
def generateSecret(secretLength):
    ...
    l = 0
    secret = ""
    while (l < secretLength):
        s = utils.generateRandomData(secretLength - l)
        for c in s:
            if (l < secretLength):
                l += 1
                secret += c
    return secret
```

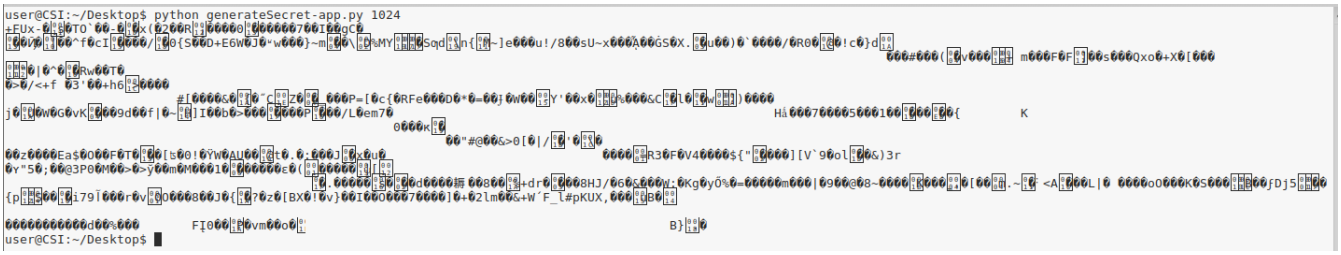

Figura 5: Resultados com o Código Acima

2 Partilha/Divisão do segredo

2.1 ShamirSecret

2.1.1 Partilhar Segredo

O primeiro passo para poder dividir um segredo através do ficheiro *createSharedSecret-app.py* é gerar uma chave privada e o respetivo certificado (para depois poder realizar a operação inversa) através dos comandos:

```
openssl genrsa -aes128 -out mykey.pem 1024
openssl req -key mykey.pem -new -x509 -days 365 -out mykey.crt
```

O segundo e último passo é executar o programa, como pode ser visto pela figura abaixo.

Figura 6: Criar Componentes com createSharedSecret-app.py

Para recuperar o segredo com 5 quaisquer partes apenas é necessário fazer executar o ficheiro *recoverSecretFromComponents-app.py* e ir completando com o que o programa pede, como mostra a figura abaixo.

Figura 7: Exemplo de recoverSecretFromComponents-app.py

A utilização do programa *recoverSecretFromAllComponents-app.py* é muito semelhante à de *recoverSecretFromComponents-app.py*, mudando-se apenas o número de partes a utilizar.

2.1.2 Recuperar Segredo

A diferença entre os 2 ficheiros é que o *recoverSecretFromComponents-app.py* recupera o segredo com apenas um subconjunto das partes, enquanto o *recoverSecretFromAllComponents-app.py* necessita de todas as partes para

recuperar o segredo. É também importante notar que ambos os programas verificam as assinaturas dos componentes através do certificado passado com argumento.

De facto, o segundo pode ser bastante útil quando é necessário validar todos os fragmentos do segredo. Podemos tomar como exemplo uma tomada de decisão. Se for necessário consenso absoluto, isto é, todas as partes terem de concordar utiliza-se o *recoverSecretFromAllComponents-app.py*, mas se só for necessário maioria (+ que 50%) utiliza-se o que permite recuperar o segredo apenas com um subconjunto dos componentes.

3 Authenticated Encryption

3.1

Com base na biblioteca criptográfica *Cryptography* de *Python* pediria-se à equipa de desenvolvimento que utiliza-se o algoritmo *AES* em *Counter-Mode* com *Authenticated Encryption*.

```
def cifra (segredo_plaintext):
    aesgcm = AESGCM(key)
    nonce = os.urandom(12)
    ct = aesgcm.encrypt(nonce, data, nonce)
    // nonce, data, authenticated but unencrypted data
    // Garantir que o nonce não é alterado

    hmac = hmac(k, ct)

    return (ct, hmac, nonce)

def decifra (segredo_cyphertext, chave_cifra, hmac, nonce):
    hash = hmac(k, segredo_cyphertext)
    if ( hash != hmac ):
        return "Authentication Error"
    else:
        aesgcm = AESGCM(key)
        segredo_plaintext = aesgcm.decrypt(nonce, ct, nonce)
        return segredo_plaintext
```

4 Algoritmos e Tamanhos de Chaves

4.1 Holanda

Depois de devidamente analisado o conteúdo dos certificados das ECs *Ministerie van Defensie* e *QuoVadis Trustlink B.V.*, que podem ser encontrados na diretoria atual deste documento, pode-se verificar que ambos usam o *RSA*, como Algoritmo de Chave Pública com tamanhos de chave de $4096 bits$ e o *SHA-256* como algoritmo de *Hash*. De facto, hoje em dia, tanto os algoritmos como os tamanhos de chave são apropriados, no entanto e num futuro próximo, com o avanço da computação quântica, podem vir a ser obsoletos muito rapidamente.