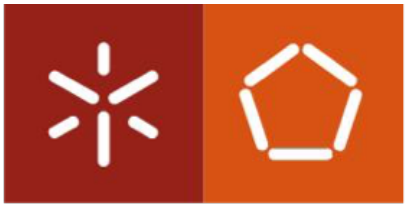

Universidade do Minho
Departamento de Informática
Mestrado Integrado em Engenharia Informática

Engenharia de Segurança: TP1

Aula 2

Diana Lopes, nº a74944
Gabriela Vaz, nº a74899

Conteúdo

Capítulo 1

Números aleatórios/pseudoaleatórios

1.1 P1.1

Nesta pergunta, o objetivo é testar os seguintes comandos e apresentar algumas conclusões sobre o seu funcionamento.

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

Após a execução dos comandos acima apresentados, observou-se que o segundo comando (*urandom*) devolve os 1024 bytes de forma imediata, enquanto que o primeiro comando (*random*) só devolve os 1024 bytes passado bastante tempo e apenas depois de se ter criado entropia suficiente (isto é, apenas depois de se terem aberto várias aplicações para gerar entropia).
Tendo em conta esta observação, facilmente se conclui que o segundo comando não precisa de entropia para funcionar, enquanto que o primeiro comando não funciona sem entropia.

1.2 P1.2

Após instalar a package Haveged, através do comando

- sudo apt-get install haveged,

foram executados novamente os comandos apresentados na questão anterior:

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

Desta vez, ambos os comandos devolveram os 1024 bytes de forma imediata. Consultando a página `http://www.issihosts.com/haveged/index.html`, vemos que o projeto Haveged tem como obejtivo corrigir situações de baixa entropia. Assim, a utilização desta package garante que há entropia suficiente para que o comando /dev/random execute de forma imediata.

1.3 P1.3

Pode consultar-se, na Figura 1.1, o conteúdo do ficheiro *generateSecret-app.py*. Como se pode ver, na linha 49, a função responsável por gerar o segredo aleatório é a função *generateSecret()*.

```
34  import sys
35  from eVotUM.Cripto import shamirsecret
36
37
38  def printUsage():
39      print("Usage: python generateSecret-app.py length")
40
41  def parseArgs():
42      if (len(sys.argv) != 2):
43          printUsage()
44      else:
45          length = int(sys.argv[1])
46          main(length)
47
48  def main(length):
49      sys.stdout.write("%s\n" % shamirsecret.generateSecret(length))
50
51  if __name__ == "__main__":
52      parseArgs()
```

Figura 1.1: Ficheiro *generateSecret-app.py*.

Na Figura 1.2 pode consultar o código da função *generateSecret()*. Como se pode ver, na linha 258, o segredo é composto por um conjunto de carateres. Ora, como se pode ver na linha 256, esses carateres têm de ser letras (*string.ascii_letters*) ou dígitos (*string.digits*).

```
242 #Cripto-4.4.0
243 def generateSecret(secretLength):
244     """
245     This function generates a random string with secretLength characters (ascii_letters and digits).
246     Args:
247         secretLength (int): number of characters of the string
248     Returns:
249         Random string with secretLength characters (ascii_letters and digits)
250     """
251     l = 0
252     secret = ""
253     while (l < secretLength):
254         s = utils.generateRandomData(secretLength - l)
255         for c in s:
256             if (c in (string.ascii_letters + string.digits) and l < secretLength): # printable character
257                 l += 1
258                 secret += c
259     return secret
```

Figura 1.2: Função *generateSecret()*.

Capítulo 2

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

2.1 P2.1

A

Inicialmente, executou-se o seguinte comando

- openssl genrsa -aes128 -out mykey.pem 1024

para gerar um par de chaves. Ao executar este comando, é pedida uma *password*. Neste caso, usamos 'abcd'.
De seguida, executou-se o comando

- python createSharedSecret-app.py 7 3 'abc' mykey.pem

Ao executar este comando é pedida a *password* criada anteriormente (neste caso, 'abcd') e é pedido o segredo (usamos "Agora temos um segredo muito confidencial").

B

Inicialmente é gerado o certificado que corresponde ao par de chaves gerado no ponto **A**, através do comando

- openssl req -key mykey.pem -new -x509 -days 365 -out mykey.crt

Este certificado é pedido quer por *recoverSecretFromComponents-app.py* como por *recoverSecretFromAllComponents-app.py*.
Pode consultar-se, nas Figuras 2.1 e 2.2 a sintaxe de uso de *recoverSecretFromComponents-app.py* e de *recoverSecretFromAllComponents-app.py*, respetivamente.
Através da análise dessa parte do código e através de experiências com exemplos concretos, conclui-se que o programa *recoverSecretFromComponents-app.py* precisa

apenas de *quorum* para recuperar o segredo. Isto é, este programa consegue recuperar o segredo tendo acesso apenas ao número de componentes que constituem o *quorum*. Se o número de componentes for inferior ao *quorum*, o programa dá erro ao ser executado. Por sua vez, o programa *recoverSecretFromAllComponents-app.py* só consegue recuperar o segredo se tiver o número total de partes em que o segredo foi dividido.

```
40  def printUsage():
41      print("Usage: python recoverSecretFromComponents-app.py number_of_shares uid cert.pem")
```

Figura 2.1: Função que mostra os parâmetros usados por *recoverSecretFromComponents-app.py*.

```
40  def printUsage():
41      print("Usage: python recoverSecretFromAllComponents-app.py total_number_of_shares uid cert.pem")
```

Figura 2.2: Função que mostra os parâmetros usados por *recoverSecretFromAllComponents-app.py*.

Capítulo 3

Authenticated Encryption

3.1 P3.1

Para garantir a confidencialidade, integridade e autenticidade do segredo considerou-se a utilização de **IND_CPA**, sendo isto uma função de sentido único, garantindo assim a confidencialidade e integridade.

Para cifrar, são passados como argumentos a tag e a mensagem que queremos transmitir. A chave, como tem de ser no formato data podemos utilizar uma função **getChave()**, que nos fornece a data do espaço temporal que nos encontramos e transforma-a na chave que se vai usar.
Em seguida codificamos a mensagem utilizamos **cifra(segredo, chave)** e para obter o valor hash da mensagem codificada utilizamos a função **hmac(chaveHash, tag+data+mensagemCodificada)**.
Por fim o ciphertext vai ser construido juntando o valor de hash, a tag, a data e a mensagem codificada.

Para Decifrar é recebido o ciphertext e começamos por verificar o criptograma e se obtivermos o valor de hash expectável deciframos tanto a chave como a mensagem codificada, obtendo assim a mensagem.

A autenticidade é garantida no teste realizado na função de decifragem comparando os valores de hash.

Capítulo 4

Algoritmos e tamanhos de chaves

4.1 P4.1 (Grupo 2)

ECs que emitem certificados "QCert for ESig":

- Fina RDC 2015 Digital Certificate Registry GOV - QC (CERTIFICADO 1)
- Agencija za komercijalnu djelatnost d.o.o. (CERTIFICADO 2)
- Zagrebačka banka dioničko društvo (CERTIFICADO 3)

CERTIFICADO 1:

- Algoritmo de Hash utilizado: SHA256
- Algoritmo utilizado: RSA
- Tamanho de chave: 4096 bit

CERTIFICADO 2:

- Algoritmo de Hash utilizado: SHA256
- Algoritmo utilizado: RSA
- Tamanho de chave: 4096 bit

CERTIFICADO 3:

- Algoritmo de Hash utilizado: SHA256
- Algoritmo utilizado: RSA
- Tamanho de chave: 4096 bit

Quanto ao tamanho de chave utilizado, é fácil de verificar que é igual em todos os certificados. Portanto concluímos que no caso de SHA256 e RSA são aceitáveis num futuro próximo, mas não são seguros a longo prazo.