

Mestrado em Engenharia Informática
Universidade do Minho

Engenharia de Segurança

Aula TP - 17/Fev/2020

João Miranda - PG41845
Sandro Cruz - PG41906

24 de Fevereiro de 2020

Conteúdo

1	Números aleatórios/pseudoaleatórios	2
1.1	Pergunta P1.1	2
1.2	Pergunta P1.2	2
1.3	Experiência 1.3	3
2	Partilha/Divisão de segredo (Secret Sharing/Splitting)	4
2.1	Experiência 2.1	4
2.2	Experiência 2.2	5
2.3	Pergunta P2.1	5
3	Algoritmos e tamanhos de chaves	8
3.1	Experiência 3.1	8
4	Authenticated Encryption	9
4.1	Pergunta P4.1 - Universign	9

Capítulo 1

Números aleatórios/pseudoaleatórios

1.1 Pergunta P1.1

É utilizado o algoritmo yarrow para gerar um número pseudo-aleatório criptograficamente seguro. O `/dev/random` utilizado inicialmente possui um bloqueio que só é libertado quando se consegue entropia ou ruídos do sistema. Neste caso para os 1024 bits o bloqueio foi constante e isto deve-se ao facto da entropia não ser suficiente sendo que neste caso não está disponível. No caso do `/dev/urandom` não existe nenhum sistema de travagem para o caso da entropia não ser o suficiente, sendo que continuará a funcionar normalmente mesmo em níveis críticos de entropia. As principais diferenças nestes duas formas de gerar números pseudo-aleatórios são que no `/dev/random` quando não existe temperatura suficiente não consegue funcionar de forma adequada enquanto no `/dev/urandom` não se efetua o re-seed quando não há temperatura suficiente sendo que continua a funcionar; no `/dev/random` é utilizado o *Entropy Pool* e é mantido um nível de entropia alto mesmo que não possa consumir. Por outro lado, no `/dev/urandom`, consegue-se obter um funcionamento mesmo após o *boot* do sistema, onde se verifica a insuficiência de entropia (sendo grave) e utiliza diretamente o resultado do gerador em vez da *Entropy Pool*.

1.2 Pergunta P1.2

Com a utilização do *haveged* foi possível a diminuição da entropia melhorando a confiabilidade e a adaptabilidade geral, minimizando assim as barreiras. Esta alteração verificou-se sobretudo ao uso do `/dev/random` que com a diminuição da entropia foi possível a temperatura ser o suficiente para funcionar de maneira correta sendo gerado então o número pseudo-aleatório criptograficamente seguro.

1.3 Experiência 1.3

Com a execução do script *generateSecret-app.py* verificamos que o script apenas gerava segredos que não contêm símbolos. Então decidimos ir verificar o código do módulo utilizado e verificamos que existe uma porção do código que gera o segredo que força o apenas ser composto por letras e dígitos.

```
def generateSecret(secretLength):  
    """  
    This function generates a random string with secretLength characters (ascii_letters and digits).  
    Args:  
        secretLength (int): number of characters of the string  
    Returns:  
        Random string with secretLength characters (ascii_letters and digits)  
    """  
    l = 0  
    secret = ""  
    while (l < secretLength):  
        s = utils.generateRandomData(secretLength - 1)  
        for c in s:  
            if (c in (string.ascii_letters + string.digits) and l < secretLength): # printable character  
                l += 1  
                secret += c  
    return secret
```

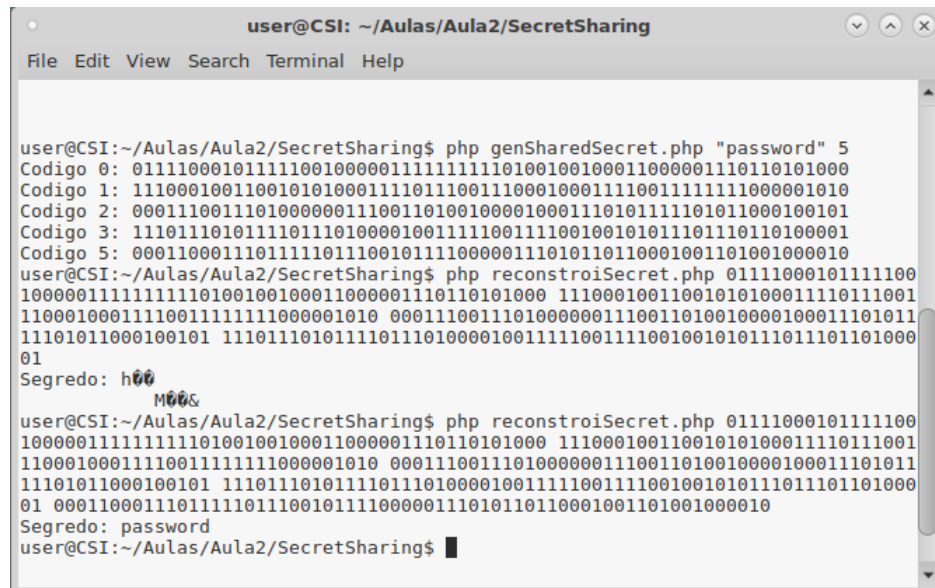
Figura 1.1: eVotUM.Cripto - Função generateSecret.

Retirando o ciclo *for* e o *if* da função deve fazer com que o segredo gerado contenha símbolos, dado que o segredo é obtido a partir do módulo *os* do python utilizando o *urandom*, sendo que este *urandom* é igual ao existente do */dev/urandom* e dado que seja gera uma string *random* com símbolos, com a remoção do *for* e do *if* o segredo deve conter símbolos.

Capítulo 2

Partilha/Divisão de segredo (Secret Sharing/Splitting)

2.1 Experiência 2.1

A terminal window titled "user@CSI: ~/Aulas/Aula2/SecretSharing" with a menu bar (File, Edit, View, Search, Terminal, Help). The terminal shows the execution of a PHP script to generate a shared secret and then reconstruct it. The output shows five binary-coded parts (Codigo 0 to 5) and the reconstructed secret "password".

```
user@CSI:~/Aulas/Aula2/SecretSharing$ php genSharedSecret.php "password" 5
Codigo 0: 01111000101111100100000111111111010010010001100000110110101000
Codigo 1: 111000100110010101000111101110011100010001111001111111000001010
Codigo 2: 0001110011101000000111001101001000010001110101111101011000100101
Codigo 3: 11101110101110111010000100111110011110010010101110110110100001
Codigo 5: 0001100011101111101110010111100000111010110110001001101001000010
user@CSI:~/Aulas/Aula2/SecretSharing$ php reconstroiSecret.php 01111000101111100
1000001111111110100100100011000001110110101000 11100010011001010100011110111001
1100010001111001111111000001010 00011100111010000001110011010010000100011101011
11101011000100101 11101110101111011101000010011111001111001001010111011101101000
01
Segredo: h00
M00G
user@CSI:~/Aulas/Aula2/SecretSharing$ php reconstroiSecret.php 01111000101111100
1000001111111110100100100011000001110110101000 11100010011001010100011110111001
1100010001111001111111000001010 00011100111010000001110011010010000100011101011
11101011000100101 11101110101111011101000010011111001111001001010111011101101000
01 000110001110111101110010111100000111010110110001001101001000010
Segredo: password
user@CSI:~/Aulas/Aula2/SecretSharing$
```

Figura 2.1: Secret Sharing/Splitting - Esquema Simples.

Como se pode ver pela figura acima, foi executado no terminal o comando **php genSharedSecret.php "password"5** que repartiu o segredo em 5 partes no formato binário. Com a utilização do comando **php reconstroiSecret.php**, foi possível a conclusão de que caso não sejam utilizadas todas as partes de

código do segredo em que foram repartidas, será impossível revelar o segredo sendo que este foi perdido definitivamente. Somente utilizando todos códigos é que se consegue verificar a representação do segredo que é o XOR de todos os números aleatórios.

2.2 Experiência 2.2

```
user@CSI:~/Aulas/Aula2/ShamirSharing$ echo "password" | perl shares.pl 3 5
3:1:845afa91db166bf6:
3:2:ae4946958e20724a:
3:3:ee2e597f918d8762:
3:4:4309324fe45caa3d:
3:5:afdbd205868eddbc:
user@CSI:~/Aulas/Aula2/ShamirSharing$ perl reconstruct.pl <<EOF
> 3:1:845afa91db166bf6:
> 3:4:4309324fe45caa3d:
> EOF
too few shares at reconstruct.pl line 77, <STDIN> line 2.
user@CSI:~/Aulas/Aula2/ShamirSharing$ perl reconstruct.pl <<EOF
> 3:1:845afa91db166bf6:
> 3:2:ae4946958e20724a:
> 3:3:ee2e597f918d8762:
> 3:4:4309324fe45caa3d:
> 3:5:afdbd205868eddbc:
> EOF
Ignoring share 4...
Ignoring share 5...
password
user@CSI:~/Aulas/Aula2/ShamirSharing$ perl reconstruct.pl <<EOF
> 3:3:ee2e597f918d8762:
> 3:4:4309324fe45caa3d:
> 3:5:afdbd205868eddbc:
> EOF
password
```

Figura 2.2: Secret Sharing/Splitting - Esquema de Shamir.

Como se pode verificar na figura acima primeiramente executou-se **echo "password" | perl shares.pl 3 5** para o segredo ser dividido em 5 partes e poder ser obtido apenas com 3 destas. Portanto é natural verificar-se que com o comando **perl reconstruct.pl** quando se colocam menos de 3 partes não se observa o segredo; por outro lado, quando se colocam partes a mais (no caso da figura as 5) ignora 2 dessas partes e é possível a revelação do segredo; por fim, com 3 partes obtém-se diretamente o segredo.

2.3 Pergunta P2.1

A parte A consistiu na execução do comando que foi indicado no enunciado. O output pode ser verificado na imagem abaixo.

Figura 2.5: Secret Sharing/Splitting - Parte B - recoverSecretFromAllCompo-
nents.

Capítulo 3

Algoritmos e tamanhos de chaves

3.1 Experiência 3.1

A autenticação de mensagens por códigos (ou, MAC tags) são uma maneira de providenciar integridade dos dados com autenticação, neste caso usando a encriptação simétrica. A melhor forma para a autenticação de mensagens é a utilização do esquema Encrypt-then-MAC. Primeiro, a mensagem é encriptada e depois a mensagem cifrada é autenticada. O MAC resultante é acrescentado à mensagem cifrada. Isto permite a integridade da mensagem cifrada e até da própria mensagem. A maior vantagem desta abordagem é que se a MAC tag não coincidir com a nova MAC tag calculada, durante a verificação, a mensagem cifrada não precisa de ser decifrada. A cifra utilizada deve ser uma cifra por blocos como por exemplo a CBC-MAC.

Capítulo 4

Authenticated Encryption

4.1 Pergunta P4.1 - Universign

Utilizando o website fornecido no enunciado *<https://webgate.ec.europa.eu/tl-browser/>* encontramos a Entidade de Certificação(EC) Universign, onde podemos verificar que o algoritmo de utilizado para gerar as chaves, é o SHA256 ou seja SHA-2 em que o output terá o tamanho de 265 bits, e o algoritmo utilizado para o certificado é o x509v3, que é o *standard* definido para certificados de chave pública.

O algoritmo SHA-2 é de hash que é considerado seguro apesar de que se aconselha a passagem para o SHA-3, mas dado que o certificado foi emitido em 2013 o SHA-3 ainda não estava disponível para a utilização. Se a Universign voltar a emitir um certificado de EC nos pensamos que deveriam utilizar o SHA-3.

```

user@CSI:~$ openssl x509 -in cert.crt -text -noout
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            69:da:6c:43:7f:07:77:91:b9:91:db:e7:99:18:8a:96
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: C = FR, O = Cryptolog International, OU = 0002 43912916400026, CN = Universign Primary CA hardware
        Validity
            Not Before: May 31 14:54:32 2017 GMT
            Not After : May 31 14:54:32 2027 GMT
        Subject: C = BE, O = Universign, 2.5.4.97 = NTRBE-0673755466, CN = Universign CA hardware
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
            Public-Key: (2048 bit)
            Modulus:
                00:ac:20:b9:5a:e9:33:3f:67:0c:e1:99:f4:12:
                fa:7d:f1:bc:d5:dd:30:e9:e8:ab:d7:fc:bd:b4:6b:
                ac:f7:e4:5a:aa:55:9f:80:04:54:fc:21:4c:19:b3:
                22:81:93:ac:08:4c:99:d1:09:c0:34:8a:8b:b8:c3:
                1d:c3:09:4e:94:5b:fa:a0:ce:f4:e0:2a:76:f6:a9:
                3e:d0:af:dc:b4:62:64:35:24:88:6a:5d:45:41:53:
                35:66:56:5b:aa:39:45:a1:9e:f9:88:78:5b:e9:a5:
                0b:25:26:55:53:b7:6b:2a:50:6b:96:05:27:54:
                a5:73:0f:07:61:a8:ce:80:99:5e:d3:21:38:b9:6d:
                af:f0:77:04:49:b9:ea:30:91:b1:6b:7a:55:4b:ce:
                20:ac:44:0b:88:44:b0:5a:fb:1b:7b:42:bc:fc:7e:
                b4:52:d0:61:f3:3c:bb:3a:e3:f3:24:0e:a7:c2:ae:
                6b:6d:f8:23:bc:d7:38:58:ce:a9:43:82:a9:e6:e0:
                1b:da:d0:15:0d:85:a4:75:65:ed:7f:1a:f9:62:6d:
                d1:59:ad:f3:97:6d:f6:57:05:5b:b8:57:28:1e:70:
                56:74:8b:97:13:a4:70:12:a4:79:64:ab:99:89:66:
                0c:ac:56:86:3a:51:fa:ba:74:46:f4:96:4d:4d:de:
                6e:05
            Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Key Usage: critical
                Certificate Sign, CRL Sign
            X509v3 CRL Distribution Points:

                Full Name:
                URI:http://crl.universign.eu/universign_primary_ca_hardware.crl

            X509v3 Subject Key Identifier:
                3E:42:FC:11:D1:40:0C:09:D0:3A:4C:EF:AB:3B:4C:57:15:3A:AB:0D
            X509v3 Certificate Policies:
                Policy: X509v3 Any Policy
                CPS: http://docs.universign.eu/

            X509v3 Basic Constraints: critical
                CA:TRUE, pathlen:0
            X509v3 Authority Key Identifier:
                keyid:4D:D9:FC:A8:2D:C7:C8:5A:A4:AD:5F:49:AE:68:A4:DC:9E:8A:12:22:
                fe:b8:aa:a3
    Signature Algorithm: sha256WithRSAEncryption
        02:66:ad:af:77:73:b1:86:f8:b0:81:eb:86:df:a8:43:8d:5f:
        27:4d:fd:c2:c8:87:7a:73:25:d6:91:cf:17:eb:04:b3:98:c8:
        67:91:d7:98:66:c2:e0:ad:10:8a:86:d6:76:94:7d:58:c2:0b:
        d4:0f:2b:c8:dd:81:51:a7:83:72:0b:54:31:fe:d9:92:ad:6e:
        23:cf:5b:bb:89:32:37:f7:c4:83:e6:d8:a9:d3:7e:02:58:00:
        c8:fd:f3:85:5a:1f:e5:fe:a5:75:2c:b2:3a:08:df:16:9d:27:
        5f:f5:37:09:93:de:d3:01:66:84:3c:6c:d4:6e:5d:7b:cc:cc:
        db:f8:08:c1:b4:c7:51:93:47:9f:dc:53:4d:2d:5a:8b:c5:24:
        f7:dc:f4:22:08:b0:be:19:1b:ef:99:60:dd:52:96:a5:c0:c5:
        c5:b9:d8:19:cf:24:1b:6f:1a:97:a2:50:1b:79:bc:8d:91:83:
        c6:0c:00:27:70:18:78:2e:d6:ef:ba:d7:b1:9e:29:9a:cc:95:
        97:6d:b9:aa:60:1c:07:ab:5e:d0:30:6e:36:4d:e5:85:eb:35:
        0f:4e:ba:07:7b:78:1c:a9:35:e2:b2:cf:1b:c7:7b:7f:12:4e:
        b2:7b:ac:51:2c:6a:ff:04:b6:15:7d:d9:75:10:28:30:1e:7e:
        fe:b8:aa:a3

```

Figura 4.1: Output da execução do comando - openssl x509 -in cert.crt -text -noout