Segurança em Sistemas de Computação

Desafio 06 - Parte 2
(Diffie-Hellman com AES e RSA)

INF01045 – Comunicação – Turma U – Prof. Raul Weber

Luiz Gustavo Frozi de Castro e Souza - Cartão 96957
Mário César Gasparoni Jr. - Cartão 151480

Setembro de 2013

Texto Cifrado (notação hexadecimal):

```
45518D4B3B2518E31433F73774B68093
1ACCBEEB4CAEA4FDF628E998FCDE1A56
A9B8E8FCF8BF06397AFA1A790BF789A3
55424C30DB0A7234F03269C9520D7D2A
829279996DA2B361F92AF6E31BC34B22
1D6607B0C78288B5F9069DA7E3FAE145
6B923611B489CB04F122A7FD6AEBFA59
09F4821CA9B511C05DDF335BB4246A39
9DEB838A06D7A2DD6C333ADE60E3B565
BCBC6053CDD2BF08299CC127C7E44E9C
94DAC960741A0756D481C7A509D4189A
3FB887E65A77954E6A2B3E2D7CE47736
95FB155BCF74E8FD30AD45C25D5B3D08
F69F5DB34906563D673585D0F3C9AA67
25366AA2B236CF9FB51C2DA0D233260A
7BC5FAA7C0213C4D5BBDA831409D17D1
21CD990066C10E00679FFA0DC2C22871
CF218773C81EA3F5A9B5E2415E9496A6
BE5C0B917AF7445A4E95A64A9E3A0D67
408C9D79636D87C433CAC5AB5AECDF53
3BB95707E93763060A2704A818D6F8DB
A89A7E9086DA497282D89BCE421ECAC3
948C03159C0B8ECF98A70FE955487F3B
61A79056CDD48E5DDFC97E5983E1720D
D6C63810B8DE8AE970D9DCCC43CBF802
4ED84B42E2FA2DB43B62446D0BACB2EB
BCB3ECD2DFD3B44DF0B5C8911A9FABB1
06BACC5CF9F0C0D63186CA1948C1BF42
E2A32BA5B85BAB2FA14758A1D7544E7E
864B1AEECF1C423CA237499241B43719
A69FE964A04445BAA2087B645033FB08
34A06BFAC16CB9807C35FF6E049BFEC2
FF38DA78840B1738062B16E4CA7F9B3F
EAF95F56E120B7DA55EA4747650A2F92
4208CCB73E114B4CB5575AA569117EE7
A04A9FB9A9331BD2AD6EAA2E816D8344
99BE100E62C76CA8C88CFAFCAC009052
3FF976DE5C55845ACECE623CC436CC33
2041B5EF6E61085EA6418843D54C3C19
2EFB07C387226AF2AE5EFAC30B008F0C
2E5657418020327E11286A0B612E58D9
5CF287063F59561CC9B2DCA92683C20E
4E4E57EBC34A09BAB3F7DB6E40D7A093
97FF5D0DD8944ABBADC67BB89EDACBE2
7EED3188CF4D893B3635FB1993070FFB
```

6D3763BB2760051C6B30D20FA48BA92A
109D395E44EE2A0C1EA1B53B6F59F46B
DAF6F729A02546A4118D8EFE941F6B29
4AEC8A6153FCA42B97034F4A1FBA4E0C
43248618C17BA30D612BB66F8C3FAFD5
7A6743C2900845603D817CC761137E05
50167F1D7104803462B5F750D3465F12
FBEA95B6199BB8A169B772A494B803A9
83AFC2E308A78FEA8A4AA166CDBBBF3E
FD762E7BBFC2B4A0D6BE49E9C7952A84
3C4B65AFA9093D176EBD15C709DD7A4E
044276F3DE0E3FAFBA31A5B558C58F47
8B3CA6AECDF0A309321239E0AFE5399D
683F3E24449497459EDFC54E1971519D
86CD3446527452847965039BD2B2D6C3
CFE9B62F258945BB7826BD2A09D70671
962A78815E7A06A1338CF4142B2A357B

Texto Decifrado:

Se vocês chegaram até aqui, parabéns! O mais difícil já passou.

O restante é somente um exercício sobre o RSA.

Para isto vamos precisar de dois primos. Um deles já foi entregue no primeiro arquivo. O segundo é:
358945627520162596891515025409134475035260832441 27

Não precisam fazer o que "BOB" pede no primeiro arquivo. Sigam estas instruções para gerar chaves RSA:

1 - Calculem o "n" multiplicando estes dois primos (o que está neste arquivo e o que estava no primeiro arquivo)

2 - Calculem (n,e) e (n,d), as chaves pública e privada de vocês, de acordo com o indicado pelo RSA.

3 - Assinem o "desafio" abaixo com a chave secreta de vocês:
796572834081677019613832428014336707 16

4 - Coloquem no Moodle a chave pública (o par (n,e)) e o desafio assinado (ou seja, o resultado de (desafio^d mod n))

E se vocês já tinham resolvido com sucesso todos os demais desafios, parabéns! Estão dispensados da primeira prova!

Assinado: Weber (Não, não sou Alice nem Bob...)

Chave Pública (n, d) e Desafio Assinado:

Chave Publica (n, e) =
(20462536081563409499587300075414581861388047817074870768045590693178547723119584387302417021438546 3L,
56802099736161919460605163470534259004747353981842584379338318138540601370375894864916096109086890816 5885954219331375644252771557177239463690442487141962435422030273818470496722197643965161068462862761641 4707630468926468940638510803799793171852699624085133429977545507612555653909293828027008327126177064538 5L)

Assinatura:
16863164630876401389370106880563851144651763039440300015287437509544960326514527210437189258342758 6

Passo a Passo da Análise efetuada:
Quase todo o ataque foi planejado com base nas mensagens capturadas durante a conversa. Os seguintes dados serviram de base para o protocolo de Diffie-Hellman:

1. Número Primo **n = 340282366920938463463374607431768211297**
2. Segundo número **g = 2**
3. Número **X = 20329610005192804707236986561532170670 8** passado pela Alice para Bob, resultante do cálculo $X = g^x \bmod n$, onde **x** é o número escolhido aleatoriamente por Alice.
4. Número **Y = 111378665949503964756483212252284796659** passado por Bob para Alice, resultante do cálculo $Y = g^y \bmod n$, onde **y** é o número escolhido aleatoriamente por Bob.
5. Variante **exponencial** do protocolo Diffie-Hellman.
6. A mensagem será cifrada com base no algoritmo AES de 128 bits.
7. A chave para cifragem do AES será composta por 128 bits extraídos da chave de sessão conforme o algoritmo:
   - for i := 0 to 15
   - begin
   - bytechave[i] := n mod 256;
   - n := n div 256;
   - end;
   - n é o número da chave de sessão e bytechave[i] são os bytes da chave AES
8. Bob resolveu usar um gerador de números randômicos próprio, seguindo a fórmula $y = 2^a * 3^b * 5^c * 7^d$, onde **y** é o número escolhido aleatoriamente, que é o mesmo número aplicado no cálculo de **Y** do Diffie-Hellman.
9. O número randômico de Bob tem menos de 50 dígitos.

Sabemos que o protocolo não requer que essas informações estejam ocultas, podem ser públicas e/ou transmitidas em texto claro.
A força criptográfica do protocolo baseia-se que o atacante deve realizar resolver um problema NP-Difícil para encontrar a chave, além dos números escolhidos terem que ser grandes, eles devem ser realmente aleatórios e escolhidos ao acaso. Com base nisso, podemos ver que o problema aqui foi o algoritmo que o Bob escolheu para gerar números aleatórios, ele usou uma fórmula determinística simples, que permite encontrar todos os números aleatórios que ela gera. Além disso, Bob resolveu usar um número aleatório não muito grande. Também há o fato da base exponencial ser **2**, um número pequeno.
Para encontrar a chave de sessão, apenas foi necessário encontrar o número aleatório escolhido por Bob, ou seja, calcular $\mathbf{y = \log_g Y \bmod n}$. Como o logaritmo em aritmética de módulo é um problema NP-Difícil, o que foi feito foi criar um algoritmo de força-bruta que calculava os possíveis valores de **y** gerados pelo gerador de números aleatórios de Bob e testava qual era o primeiro que atendia à condição $\mathbf{Y = g^y \bmod n}$.
De posse do número escolhido por Bob, foi possível calcular a chave de sessão $\mathbf{K = X^y \bmod n}$ trocada entre Alice e Bob e aplicar o algoritmo também trocado entre eles para obter os 128 bits da chave do AES.
Com o texto decifrado foi só seguir o passo-a-passo e gerar uma par de chaves RSA com base nos primos trocados nas duas mensagens e assinar o desafio.

Códigos-fonte utilizados:

Programa de Ataque (desafio6-p2.py):

```
## Universidade Federal do Rio Grande do Sul - Instituto de Informatica
## Departamento de Informatica Aplicada
## Seguranca em Sistemas de Computacao - 2013/2
## Professor: Raul Fernando Weber
## Alunos: Luiz Gustavo Frozi e  Mario Gasparoni Junior
##
## Python 2.7
##
## Desafio 6 - Parte 2
##
## Script que executa encontra a chave de sessao de um algoritmo Diffie-Hellman
## exponencial.
##
from Crypto.Cipher import AES
#from Crypto import Random
import array
#
# Infomacoes Interceptadas
#

# Numero Primo
n = 340282366920938463463374607431768211297

# Segundo Numero
g = 2

# Bob -> Alice : Y = g^y mod n
Y = 111378665949503964756483212252284796659

# Alice -> Bob : X = g^x mod n
X = 203296100051928047072369865615321706708

# Mensagem Cifrada
cyphered = [] #cada bloco eh um elemento da lista
cyphered.append("45518D4B3B2518E31433F73774B68093".decode("hex"))
cyphered.append("1ACCBEEB4CAEA4FDF628E998FCDE1A56".decode("hex"))
cyphered.append("A9B8E8FCF8BF06397AFA1A790BF789A3".decode("hex"))
cyphered.append("55424C30DB0A7234F03269C9520D7D2A".decode("hex"))
cyphered.append("829279996DA2B361F92AF6E31BC34B22".decode("hex"))
cyphered.append("1D6607B0C78288B5F9069DA7E3FAE145".decode("hex"))
cyphered.append("6B923611B489CB04F122A7FD6AEBFA59".decode("hex"))
cyphered.append("09F4821CA9B511C05DDF335BB4246A39".decode("hex"))
cyphered.append("9DEB838A06D7A2DD6C333ADE60E3B565".decode("hex"))
cyphered.append("BCBC6053CDD2BF08299CC127C7E44E9C".decode("hex"))
cyphered.append("94DAC960741A0756D481C7A509D4189A".decode("hex"))
cyphered.append("3FB887E65A77954E6A2B3E2D7CE47736".decode("hex"))
cyphered.append("95FB155BCF74E8FD30AD45C25D5B3D08".decode("hex"))
cyphered.append("F69F5DB34906563D673585D0F3C9AA67".decode("hex"))
cyphered.append("25366AA2B236CF9FB51C2DA0D233260A".decode("hex"))
cyphered.append("7BC5FAA7C0213C4D5BBDA831409D17D1".decode("hex"))
cyphered.append("21CD990066C10E00679FFA0DC2C22871".decode("hex"))
cyphered.append("CF218773C81EA3F5A9B5E2415E9496A6".decode("hex"))
cyphered.append("BE5C0B917AF7445A4E95A64A9E3A0D67".decode("hex"))
cyphered.append("408C9D79636D87C433CAC5AB5AECDF53".decode("hex"))
cyphered.append("3BB95707E93763060A2704A818D6F8DB".decode("hex"))
cyphered.append("A89A7E9086DA497282D89BCE421ECAC3".decode("hex"))
cyphered.append("948C03159C0B8ECF98A70FE955487F3B".decode("hex"))
cyphered.append("61A79056CDD48E5DDFC97E5983E1720D".decode("hex"))
cyphered.append("D6C63810B8DE8AE970D9DCCC43CBF802".decode("hex"))
cyphered.append("4ED84B42E2FA2DB43B62446D0BACB2EB".decode("hex"))
cyphered.append("BCB3ECD2DFD3B44DF0B5C8911A9FABB1".decode("hex"))
cyphered.append("06BACC5CF9F0C0D63186CA1948C1BF42".decode("hex"))
cyphered.append("E2A32BA5B85BAB2FA14758A1D7544E7E".decode("hex"))
cyphered.append("864B1AEECF1C423CA237499241B43719".decode("hex"))
cyphered.append("A69FE964A04445BAA2087B645033FB08".decode("hex"))
cyphered.append("34A06BFAC16CB9807C35FF6E049BFEC2".decode("hex"))
cyphered.append("FF38DA78840B1738062B16E4CA7F9B3F".decode("hex"))
cyphered.append("EAF95F56E120B7DA55EA4747650A2F92".decode("hex"))
cyphered.append("4208CCB73E114B4CB5575AA569117EE7".decode("hex"))
cyphered.append("A04A9FB9A9331BD2AD6EAA2E816D8344".decode("hex"))
cyphered.append("99BE100E62C76CA8C88CFAFCAC009052".decode("hex"))
cyphered.append("3FF976DE5C55845ACECE623CC436CC33".decode("hex"))
cyphered.append("2041B5EF6E61085EA6418843D54C3C19".decode("hex"))
```

```
cyphered.append("2EFB07C387226AF2AE5EFAC30B008F0C".decode("hex"))
cyphered.append("2E5657418020327E11286A0B612E58D9".decode("hex"))
cyphered.append("5CF287063F59561CC9B2DCA92683C20E".decode("hex"))
cyphered.append("4E4E57EBC34A09BAB3F7DB6E40D7A093".decode("hex"))
cyphered.append("97FF5D0DD8944ABBADC67BB89EDACBE2".decode("hex"))
cyphered.append("7EED3188CF4D893B3635FB1993070FFB".decode("hex"))
cyphered.append("6D3763BB2760051C6B30D20FA48BA92A".decode("hex"))
cyphered.append("109D395E44EE2A0C1EA1B53B6F59F46B".decode("hex"))
cyphered.append("DAF6F729A02546A4118D8EFE941F6B29".decode("hex"))
cyphered.append("4AEC8A6153FCA42B97034F4A1FBA4E0C".decode("hex"))
cyphered.append("43248618C17BA30D612BB66F8C3FAFD5".decode("hex"))
cyphered.append("7A6743C2900845603D817CC761137E05".decode("hex"))
cyphered.append("50167F1D7104803462B5F750D3465F12".decode("hex"))
cyphered.append("FBEA95B6199BB8A169B772A494B803A9".decode("hex"))
cyphered.append("83AFC2E308A78FEA8A4AA166CDBBBF3E".decode("hex"))
cyphered.append("FD762E7BBFC2B4A0D6BE49E9C7952A84".decode("hex"))
cyphered.append("3C4B65AFA9093D176EBD15C709DD7A4E".decode("hex"))
cyphered.append("044276F3DE0E3FAFBA31A5B558C58F47".decode("hex"))
cyphered.append("8B3CA6AECDF0A309321239E0AFE5399D".decode("hex"))
cyphered.append("683F3E24449497459EDFC54E1971519D".decode("hex"))
cyphered.append("86CD3446527452847965039BD2B2D6C3".decode("hex"))
cyphered.append("CFE9B62F258945BB7826BD2A09D70671".decode("hex"))
cyphered.append("962A78815E7A06A1338CF4142B2A357B".decode("hex"))

#
# Funcoes de Biblioteca
    #
    # Algoritmo para gerar a chave citado nas mensagens
    def makeKey128(key):
        bytechave = [None]*16
        for i in range(0, 16):
            bytechave[i] = key % 256
            key = key // 256
        return bytechave

    # Versao do gcd/mdc modificada entre "a" e "b"
    def egcd(a, b):
        if a == 0:
            return (b, 0, 1)
        else:
            g, y, x = egcd(b % a, a)
            return (g, x - (b // a) * y, y)

    # Calcula o inverso multiplicativo de "a" em modulo "m"
    def modinverse(a, m):
        g, x, y = egcd(a, m)
        if g != 1:
            raise Exception('modular inverse does not exist')
        else:
            return x % m

    # Calcula o numero randomico usando a regra de formacao citada na mensagem 1
    def fakeRandom(x, y, w, z):
        return 2**x * 3**y * 5**w * 7**z

    # Tenta encontrar o logaritmo discreto usando a regra de formacao citada
    # Versao mais otimizada
    def modlogInverse(X, g, n):
        print "modlogInverse(", X, ", ", g, ", ", n, ")"
        for d in range(14, 161): # Ajuste feito para executar diretamente
            for c in range(1, 161):
                for b in range(1, 161):
                    for a in range(1, 161):
                        x = fakeRandom(a, b, c, d)
                        if len(str(x)) > 50:
                            break
                        else:
                            #print "(x, a, b, c, d) = ", (x, a, b, c, d)
                            if X == pow(g, x, n):
                                print "(x, a, b, c, d) = ", (x, a, b, c, d)
                                return x
            #print "(x, a, b, c, d) = ", (x, a, b, c, d)
        raise Exception('modular log does not exist')
```

```
147.
148.    #
149.    # Inicio do Ataque
150.    #
151.
152.    # O ataque deve ser feito no numero do Bob que possui uma regra de formacao fraca
153.    y = modlogInverse(Y, g, n) # depois de (a, b, c, d) = (1, 160, 160, 14) na iteracao, depois
    de 28 horas, Core 2 Duo com 2,4GHz e 4GB de RAM
154.    #y   =   695097467712814559879534227200000000000000000000000   #   Resultado   do   logaritmo   =
    randomico do Bob
155.    print "y:", y
156.    arquivo_random = open('desafio6-p2.random', 'w')
157.    arquivo_random.writelines(str(y))
158.    arquivo_random.close()
159.
160.
161.    K = pow(X, y, n)
162.    #K = 283903288088203077050299572641376932758L
163.    print "K:", K
164.    arquivo_k = open('desafio6-p2.k', 'w')
165.    arquivo_k.writelines(str(K))
166.    arquivo_k.close()
167.
168.    print "Key:"
169.    key = makeKey128(K)
170.    print key
171.
172.    strkey = "".join(chr(x) for x in key) #key eh uma lista de inteiros,
173.    #converte cada int em char e depois da um join nessa lista de string com a string ""
174.
175.    print "Texto:"
176.
177.    arquivo = open('desafio6-p2.unciphered', 'w')
178.
179.    cipher = AES.new(strkey, AES.MODE_ECB)
180.    for block in cyphered:
181.        plain = cipher.decrypt(block) #decifra bloco
182.        print(plain)
183.        arquivo.writelines(plain)
184.
185.    arquivo.close()
```

Programa para o Cálculo das Chaves do RSA (desafio6-rsa.py):

```
1.  ## Universidade Federal do Rio Grande do Sul - Instituto de Informatica
2.  ## Departamento de Informatica Aplicada
3.  ## Seguranca em Sistemas de Computacao - 2013/2
4.  ## Professor: Raul Fernando Weber
5.  ## Alunos: Luiz Gustavo Frozi e  Mario Gasparoni Junior
6.  ##
7.  ## Python 2.7
8.  ##
9.  ## Desafio 6 - Parte 3
10. ##
11. ## Script que calcula as chaves (n,e) e (n,d), chaves publica e privada
12. ## indicadas no segundo arquivo.
13. ##
14. from Crypto import Random
15. import fractions
16.
17. # Primo do Primeiro Arquivo
18. p = 57007341816453784345611883741305290721948860641 69
19.
20. # Primo do Segundo Arquivo
21. q = 35894562752016259689151502540913447503526083244127
22.
23. # Desafio a ser assinado
24. desafio = 79657283408167701961383242801433670716
25.
```

```
#
# Biblioteca
#
def primoRelativo(n):
    p = int(Random.get_random_bytes(128).encode('hex'), 16)
    i = 1
    while fractions.gcd(p, n) != 1:
        p = int(Random.get_random_bytes(128).encode('hex'), 16)
        i += 1
        if i > 100000:
            return None
    return p

# Versao do gcd/mdc modificada entre "a" e "b"
def egcd(a, b):
    if a == 0:
        return (b, 0, 1)
    else:
        g, y, x = egcd(b % a, a)
        return (g, x - (b // a) * y, y)

# Calcula o inverso multiplicativo de "a" em modulo "m"
def modinverse(a, m):
    g, x, y = egcd(a, m)
    if g != 1:
        raise Exception('modular inverse does not exist')
    else:
        return x % m


#
# Execucao do RSA
#

n = p * q
n_temp = (p - 1) * (q - 1)

e = primoRelativo(n_temp)
public_key = (n, e)
print "Chave Publica (n, e) = ", public_key

d = modinverse(e, n_temp)
private_key = (n, d)
print "Chave Privada (n, d) = ", private_key

assinatura = pow(desafio, d, n)
print "Desafio: ", desafio
print "Assinatura: ", assinatura

verificacao = pow(assinatura, e, n)
if verificacao == desafio:
    print "Verificacao OK"
else:
    print "Assinatura Invalida"
```