2019强网杯 - 密码学-RSA-Coppersmith

原创

Blus.King 于 2019-05-28 19:09:55 发布 6481 收藏 19
分类专栏： 记录 CTF/AWD

本文链接：https://blog.csdn.net/q851579181q/article/details/90645041

Coppersmith 相关攻击

学习资料：

https://ctf-wiki.github.io/ctf-wiki/crypto/asymmetric/rsa/rsa_coppersmith_attack/

https://code.felinae98.cn/ctf/crypto/rsa%E5%A4%A7%E7%A4%BC%E5%8C%85%EF%BC%88%E4%BA%8C%E7%9B%B8%E5%85%B3/

其他writeup:

https://altman.vip/2019/05/27/QWB2019-writeup/

以下是我的Writeup，包括知识点、题目和解题脚本：

强网杯RSA-Coppersmith集锦概览：

1.challenge 1

已知明文的高位，是Stereotyped messages攻击 或 Lattice based attacks

2.challenge 2

已知p的高位，Factoring with High Bits Known

3.challenge 3

已知私钥的512位的低位 Partial Key Exposure Attack(部分私钥暴露攻击)

4.challenge4

Basic Broadcast Attack 低加密指数广播攻击

如果选取的加密指数较低，并且使用了相同的加密指数给一个接受者的群发送相同的信息，那么可以进行广播攻击得到明文

5.challenge5

明文存在线性关系，Related Message Attack和RSA Padding Attack

6.challenge 6

d 较小时，满足 d<N^0.292d<N0.292　Boneh and Durfee attack 比 Wiener's Attack 要强一些

题目和解答脚本：

大部分为sage脚本，需要在https://sagecell.sagemath.org/运行

1.challenge 1

```
[+]Generating challenge 1

[+]n=0x2519834a6cc3bf25d078caefc5358e41c726a7a56270e425e21515d1b195b248b82f4189a0b621694586bb254e27010ee437

[+]e=3

[+]m=random.getrandbits(512)

[+]c=pow(m,e,n)=0x1f6f6a8e61f7b5ad8bef738f4376a96724192d8da1e3689dec7ce5d1df615e0910803317f9bafb6671ffe722e

[+]((m>>72)<<72)=0xb11ffc4ce423c77035280f1c575696327901daac8a83c057c453973ee5f4e508455648886441c0f3393fe4c9
```

注意到有明文的高位，是tereotyped messages攻击，可以使用以下脚本解，

```
n = 0x2519834a6cc3bf25d078caefc5358e41c726a7a56270e425e21515d1b195b248b82f4189a0b621694586bb254e27010ee4376



e = 3



m = randrange(n)

c = pow(m, e, n)



beta = 1

epsilon = beta^2/7



nbits = n.nbits()

kbits = floor(nbits*(beta^2/e-epsilon))

#mbar = m & (2^nbits-2^kbits)

mbar = 0xb11ffc4ce423c77035280f1c575696327901daac8a83c057c453973ee5f4e508455648886441c0f3393fe4c922ef1c3a62



c = 0x1f6f6a8e61f7b5ad8bef738f4376a96724192d8da1e3689dec7ce5d1df615e0910803317f9bafb6671ffe722e0292ce76cca3

print "upper %d bits (of %d bits) is given" % (nbits-kbits, nbits)



PR.<x> = PolynomialRing(Zmod(n))

f = (mbar + x)^e - c



print m

x0 = f.small_roots(X=2^kbits, beta=1)[0]  # find root < 2^kbits with factor = n1

print mbar + x0
```

2.challenge 2

收到：

```
'[+]Generating challenge 2\n' '[+]n=0x5894f869d1aecee379e2cb60ff7314d18dbd383e0c9f32e7f7b4dc8bd47535d4f3512

 '[+]e=65537\n'

'[+]m=random.getrandbits(512)\n'

    '[+]c=pow(m,e,n)=0x284a601c3321fd882d3b64ae27fb587d1714bc18aecc3293169861bcf17678a6e83947aba4f165f22a71

    '[+]((p>>128)<<128)=0x5d33504b4e3bd2ffb628b5c447c4a7152a9f37dc4bcc8f376f64000fa96eb97c0af445e3b2c03926a
```

已知p的高位，Factoring with High Bits Known

sage脚本：

```
n = 0x5894f869d1aecee379e2cb60ff7314d18dbd383e0c9f32e7f7b4dc8bd47535d4f3512ce6a23b0251049346fede745d116ba8d
p_fake = 0x5d33504b4e3bd2ffb628b5c447c4a7152a9f37dc4bcc8f376f64000fa96eb97c0af445e3b2c03926a4aa4542918c6010




#pbits = 2048
pbits = p_fake.nbits()
#kbits = 900
kbits = 128  #p失去的低位
pbar = p_fake & (2^pbits-2^kbits)
print "upper %d bits (of %d bits) is given" % (pbits-kbits, pbits)

PR.<x> = PolynomialRing(Zmod(n))
f = x + pbar

x0 = f.small_roots(X=2^kbits, beta=0.4)[0]  # find root < 2^kbits with factor >= n^0.3
p= x0 + pbar
print p
```

1. **challenge 3**

```
[+]Generating challenge 3

[DEBUG] Received 0x314 bytes:

[+]n=0xd463feb999c9292e25acd7f98d49a13413df2c4e74820136e739281bb394a73f2d1e6b53066932f50a73310360e5a5c62250

[+]e=3

[+]m=random.getrandbits(512)

[+]c=pow(m,e,n)=0xcaeeb38516d642a19550fa863173f4695c3b44bd5a5554b1e93cfb690d5c1de531b7f1187f7d8c8c11da38af0

[+]d=invmod(e,(p-1)*(q-1))

[+]d&((1<<512)-1)=0x603d033f2ef6c759aec839f132a45215fc8a635b757f3951a731fe60bc6729b3bcf819b57abfcaba3a93e9e

[-]long_to_bytes(m).encode('hex')=
```

已知私钥的512位的低位 Partial Key Exposure Attack(部分私钥暴露攻击)

----------脚本----------

```
def partial_p(p0, kbits, n):

    PR.<x> = PolynomialRing(Zmod(n))

    nbits = n.nbits()



    f = 2^kbits*x + p0

    f = f.monic()

    roots = f.small_roots(X=2^(nbits//2-kbits), beta=0.3)  # find root < 2^(nbits//2-kbits) with factor >=

    if roots:

        x0 = roots[0]

        p = gcd(2^kbits*x0 + p0, n)

        return ZZ(p)



def find_p(d0, kbits, e, n):

    X = var('X')



    for k in xrange(1, e+1):
```

```
        results = solve_mod([e*d0*X - k*X*(n-X+1) + k*n == X], 2^kbits)
        for x in results:
            p0 = ZZ(x[0])
            p = partial_p(p0, kbits, n)
            if p:
                return p



if __name__ == '__main__':
    n = 0xd463feb999c9292e25acd7f98d49a13413df2c4e74820136e739281bb394a73f2d1e6b53066932f50a73310360e5a5c62
    e = 3
    d = 0x603d033f2ef6c759aec839f132a45215fc8a635b757f3951a731fe60bc6729b3bcf819b57abfcaba3a93e9edef766c0d4


    beta = 0.5
    epsilon = beta^2/7


    nbits = n.nbits()
    print "nbits:%d:"%(nbits)
    #kbits = floor(nbits*(beta^2+epsilon))
    kbits = nbits - d.nbits()-1
    print "kbits:%d"%(kbits)
    d0 = d & (2^kbits-1)
    print "lower %d bits (of %d bits) is given" % (kbits, nbits)


    p = find_p(d0, kbits, e, n)
    print "found p: %d" % p
    q = n//p
    print d
print inverse_mod(e, (p-1)*(q-1))
```

4.challenge4

```
e=3

m=random.getrandbits(512)

n1=0x5797fdb74bcea6788212fb2c32c5d98d308c617f893d1f375d0e611b424d5656df4465772e278c25e7d1d5fd73b0fdfdac4a78

c1=pow(m,e,n1)=0x25e206422ea328f1b295dd121970b874b002789b2419a57584b2f1a682a36312a45efe22bb68694b9c9dfdc63c

n2=0x50998a3cf7f86a3044fe3c1fda2f6df7050383833279ebdbfe943f83faae3ada1bb6e684e48efd0487056849d47552d8052144

c2=pow(m,e,n2)=0x448f88bec6795e11b06a7810faf617931bc6d99d1628cafecff1e933154ce575caaf752c3daf50b288ad7759ea

n3=0x15ed9002077c66e48a6fc80ce744f16b87e237ddd9a4efb4ffa2f9f89d09af382dddfc259dbf932728c23757957638f3ec9327

c3=pow(m,e,n3)=0xc9ddbb9478d0b64086091aac64efd51eb37b5067feb380995d39a917c0c927b26902f06dc449b53d80cd59c5d9

long_to_bytes(m).encode('hex')=
```

Basic Broadcast Attack 低加密指数广播攻击

如果选取的加密指数较低，并且使用了相同的加密指数给一个接受者的群发送相同的信息，那么可以进行广播攻击得到明文

----------脚本----------

```
# coding:utf8



from struct import pack,unpack

import zlib

import gmpy
```

```
def my_parse_number(number):
    string = "%x" % number
    #if len(string) != 64:
    #    return ""
    erg = []
    while string != '':
        erg = erg + [chr(int(string[:2], 16))]
        string = string[2:]
    return ''.join(erg)
def extended_gcd(a, b):
    x,y = 0, 1
    lastx, lasty = 1, 0
    while b:
        a, (q, b) = b, divmod(a,b)
        x, lastx = lastx-q*x, x
        y, lasty = lasty-q*y, y
    return (lastx, lasty, a)
def chinese_remainder_theorem(items):
  N = 1
  for a, n in items:
    N *= n
  result = 0
  for a, n in items:
    m = N/n
    r, s, d = extended_gcd(n, m)
    if d != 1:
      N=N/n
      continue
      #raise "Input not pairwise co-prime"
```

```
    result += a*s*m

  return result % N, N

sessions=[

{"c": 0x25e206422ea328f1b295dd121970b874b002789b2419a57584b2f1a682a36312a45efe22bb68694b9c9dfdc63c4f10b746a



{"c":0x448f88bec6795e11b06a7810faf617931bc6d99d1628cafecff1e933154ce575caaf752c3daf50b288ad7759ea8133f7dc9c



{"c":0xc9ddbb9478d0b64086091aac64efd51eb37b5067feb380995d39a917c0c927b26902f06dc449b53d80cd59c5d912fb5a5f45



data = []

for session in sessions:

    e=session['e']

    n=session['n']

    msg=session['c']

    data = data + [(msg, n)]

print "Please wait, performing CRT"

x, n = chinese_remainder_theorem(data)

e=session['e']

realnum = gmpy.mpz(x).root(e)[0].digits()

#print my_parse_number(int(realnum))

print my_parse_number(int(realnum)).encode('hex')
```

challenge-5

```
[+]Generating challenge 5

n=0xf9526aad4d41c9b28f8bae279c7ef6b07d729d1f56e530219851f656ad521218815bdccb15167a25633a2f76969fccd3fe1ef37

e=3
m=random.getrandbits(512)
c=pow(m,e,n)=0x798841c574b7c88ce1430d4b02bac01fc9368c71a7966176b22f9dc2e0c2f6d4d5b8a9e10dbcaa4584e667ef1afd

x=pow(m+1,e,n)=0xe92c4c99052fa3c4bb5e54477b0afe8e18da37255269f070ffa6824492a87153e428fa4ed839b7f3249966259a

"[-]long_to_bytes(m).encode('hex')="
```

---脚本--

```
import gmpy2
import libnum


def getM2(a,b,c1,c2,n):
    a3 = pow(a,3,n)
    b3 = pow(b,3,n)
    first = c1-a3*c2+2*b3
    first = first % n
    second = 3*b*(a3*c2-b3)
    second = second % n
    third = second*gmpy2.invert(first,n)
    third = third % n
    fourth = (third+b)*gmpy2.invert(a,n)
    return fourth % n
#y=ax+b
a=1
#b=1
b=-1
padding2=b
```

```
n=0xf9526aad4d41c9b28f8bae279c7ef6b07d729d1f56e530219851f656ad521218815bdccb15167a25633a2f76969fccd3fe1ef37

c1=0x798841c574b7c88ce1430d4b02bac01fc9368c71a7966176b22f9dc2e0c2f6d4d5b8a9e10dbcaa4584e667ef1afd213b78c2bd

c2=0xe92c4c99052fa3c4bb5e54477b0afe8e18da37255269f070ffa6824492a87153e428fa4ed839b7f3249966259a0c8864118559

m = getM2(a,b,c1,c2,n)-padding2

#m=m-2

#不知道为什么还要减2

#m = getM2(a,b,c1,c2,n)-padding2

#print libnum.n2s(m)

print m
```

challenge 6

```
[+]Generating challenge 6

[+]n=0xbadd260d14ea665b62e7d2e634f20a6382ac369cd44017305b69cf3a2694667ee651acded7085e0757d169b090f29f3f86fe

[+]d=random.getrandbits(1024*0.270)

[+]e=invmod(d,phin)

[+]hex(e)=0x11722b54dd6f3ad9ce81da6f6ecb0acaf2cbc3885841d08b32abc0672d1a7293f9856db8f9407dc05f6f373a2d92467

[+]m=random.getrandbits(512)

[+]c=pow(m,e,n)=0xe3505f41ec936cf6bd8ae344bfec85746dc7d87a5943b3a7136482dd7b980f68f52c887585d1c7ca099310c4d

[-]long_to_bytes(m).encode('hex')=
```

d 较小时，满足 d<N^0.292d<N0.292　Boneh and Durfee attack 比 Wiener's Attack 要强一些

----------脚本----------

```
import time

##############################################
# Config
##############################################

"""
Setting debug to true will display more informations
```

```
about the lattice, the bounds, the vectors...
"""
debug = True

"""
Setting strict to true will stop the algorithm (and
return (-1, -1)) if we don't have a correct
upperbound on the determinant. Note that this
doesn't necesseraly mean that no solutions
will be found since the theoretical upperbound is
usualy far away from actual results. That is why
you should probably use `strict = False`
"""
strict = False

"""
This is experimental, but has provided remarkable results
so far. It tries to reduce the lattice as much as it can
while keeping its efficiency. I see no reason not to use
this option, but if things don't work, you should try
disabling it
"""
helpful_only = True
dimension_min = 7 # stop removing if lattice reaches that dimension

##############################################
# Functions
##############################################

# display stats on helpful vectors
def helpful_vectors(BB, modulus):
    nothelpful = 0
    for ii in range(BB.dimensions()[0]):
        if BB[ii,ii] >= modulus:
            nothelpful += 1

    print nothelpful, "/", BB.dimensions()[0], " vectors are not helpful"

# display matrix picture with 0 and X
def matrix_overview(BB, bound):
    for ii in range(BB.dimensions()[0]):
        a = ('%02d ' % ii)
        for jj in range(BB.dimensions()[1]):
            a += '0' if BB[ii,jj] == 0 else 'X'
            if BB.dimensions()[0] < 60:
                a += ' '
        if BB[ii, ii] >= bound:
            a += '~'
        print a

# tries to remove unhelpful vectors
# we start at current = n-1 (last vector)
def remove_unhelpful(BB, monomials, bound, current):
    # end of our recursive function
    if current == -1 or BB.dimensions()[0] <= dimension_min:
        return BB

    # we start by checking from the end
    for ii in range(current, -1, -1):
        # if it is unhelpful:
```

```
        # if it is unhelpful:
        if BB[ii, ii] >= bound:
            affected_vectors = 0
            affected_vector_index = 0
            # let's check if it affects other vectors
            for jj in range(ii + 1, BB.dimensions()[0]):
                # if another vector is affected:
                # we increase the count
                if BB[jj, ii] != 0:
                    affected_vectors += 1
                    affected_vector_index = jj

            # level:0
            # if no other vectors end up affected
            # we remove it
            if affected_vectors == 0:
                print "* removing unhelpful vector", ii
                BB = BB.delete_columns([ii])
                BB = BB.delete_rows([ii])
                monomials.pop(ii)
                BB = remove_unhelpful(BB, monomials, bound, ii-1)
                return BB

            # level:1
            # if just one was affected we check
            # if it is affecting someone else
            elif affected_vectors == 1:
                affected_deeper = True
                for kk in range(affected_vector_index + 1, BB.dimensions()[0]):
                    # if it is affecting even one vector
                    # we give up on this one
                    if BB[kk, affected_vector_index] != 0:
                        affected_deeper = False
                # remove both it if no other vector was affected and
                # this helpful vector is not helpful enough
                # compared to our unhelpful one
                if affected_deeper and abs(bound - BB[affected_vector_index, affected_vector_index]) < abs(
                    print "* removing unhelpful vectors", ii, "and", affected_vector_index
                    BB = BB.delete_columns([affected_vector_index, ii])
                    BB = BB.delete_rows([affected_vector_index, ii])
                    monomials.pop(affected_vector_index)
                    monomials.pop(ii)
                    BB = remove_unhelpful(BB, monomials, bound, ii-1)
                    return BB
    # nothing happened
    return BB

"""
Returns:
* 0,0   if it fails
* -1,-1 if `strict=true`, and determinant doesn't bound
* x0,y0 the solutions of `pol`
"""
def boneh_durfee(pol, modulus, mm, tt, XX, YY):
    """
    Boneh and Durfee revisited by Herrmann and May

    finds a solution if:
    * d < N^delta
    * |x| < e^delta
```

```
* |y| < e^0.5
whenever delta < 1 - sqrt(2)/2 ~ 0.292
"""

# substitution (Herrman and May)
PR.<u, x, y> = PolynomialRing(ZZ)
Q = PR.quotient(x*y + 1 - u) # u = xy + 1
polZ = Q(pol).lift()

UU = XX*YY + 1

# x-shifts
gg = []
for kk in range(mm + 1):
    for ii in range(mm - kk + 1):
        xshift = x^ii * modulus^(mm - kk) * polZ(u, x, y)^kk
        gg.append(xshift)
gg.sort()

# x-shifts list of monomials
monomials = []
for polynomial in gg:
    for monomial in polynomial.monomials():
        if monomial not in monomials:
            monomials.append(monomial)
monomials.sort()

# y-shifts (selected by Herrman and May)
for jj in range(1, tt + 1):
    for kk in range(floor(mm/tt) * jj, mm + 1):
        yshift = y^jj * polZ(u, x, y)^kk * modulus^(mm - kk)
        yshift = Q(yshift).lift()
        gg.append(yshift) # substitution

# y-shifts list of monomials
for jj in range(1, tt + 1):
    for kk in range(floor(mm/tt) * jj, mm + 1):
        monomials.append(u^kk * y^jj)

# construct lattice B
nn = len(monomials)
BB = Matrix(ZZ, nn)
for ii in range(nn):
    BB[ii, 0] = gg[ii](0, 0, 0)
    for jj in range(1, ii + 1):
        if monomials[jj] in gg[ii].monomials():
            BB[ii, jj] = gg[ii].monomial_coefficient(monomials[jj]) * monomials[jj](UU,XX,YY)

# Prototype to reduce the lattice
if helpful_only:
    # automatically remove
    BB = remove_unhelpful(BB, monomials, modulus^mm, nn-1)
    # reset dimension
    nn = BB.dimensions()[0]
    if nn == 0:
        print "failure"
        return 0,0

# check if vectors are helpful
```

```
    if debug:
        helpful_vectors(BB, modulus^mm)

    # check if determinant is correctly bounded
    det = BB.det()
    bound = modulus^(mm*nn)
    if det >= bound:
        print "We do not have det < bound. Solutions might not be found."
        print "Try with highers m and t."
        if debug:
            diff = (log(det) - log(bound)) / log(2)
            print "size det(L) - size e^(m*n) = ", floor(diff)
        if strict:
            return -1, -1
    else:
        print "det(L) < e^(m*n) (good! If a solution exists < N^delta, it will be found)"

    # display the lattice basis
    if debug:
        matrix_overview(BB, modulus^mm)

    # LLL
    if debug:
        print "optimizing basis of the lattice via LLL, this can take a long time"

    BB = BB.LLL()

    if debug:
        print "LLL is done!"

    # transform vector i & j -> polynomials 1 & 2
    if debug:
        print "looking for independent vectors in the lattice"
    found_polynomials = False

    for pol1_idx in range(nn - 1):
        for pol2_idx in range(pol1_idx + 1, nn):
            # for i and j, create the two polynomials
            PR.<w,z> = PolynomialRing(ZZ)
            pol1 = pol2 = 0
            for jj in range(nn):
                pol1 += monomials[jj](w*z+1,w,z) * BB[pol1_idx, jj] / monomials[jj](UU,XX,YY)
                pol2 += monomials[jj](w*z+1,w,z) * BB[pol2_idx, jj] / monomials[jj](UU,XX,YY)

            # resultant
            PR.<q> = PolynomialRing(ZZ)
            rr = pol1.resultant(pol2)

            # are these good polynomials?
            if rr.is_zero() or rr.monomials() == [1]:
                continue
            else:
                print "found them, using vectors", pol1_idx, "and", pol2_idx
                found_polynomials = True
                break
        if found_polynomials:
            break

    if not found_polynomials:
        print "no independant vectors could be found. This should very rarely happen..."
```

```
        return 0, 0

    rr = rr(q, q)

    # solutions
    soly = rr.roots()

    if len(soly) == 0:
        print "Your prediction (delta) is too small"
        return 0, 0

    soly = soly[0][0]
    ss = pol1(q, soly)
    solx = ss.roots()[0][0]

    #
    return solx, soly

def example():
    ############################################
    # How To Use This Script
    ############################################

    #
    # The problem to solve (edit the following values)
    #

    # the modulus
    N = 0xbadd260d14ea665b62e7d2e634f20a6382ac369cd44017305b69cf3a2694667ee651acded7085e0757d169b090f29f3f8
    # the public exponent
    e = 0x11722b54dd6f3ad9ce81da6f6ecb0acaf2cbc3885841d08b32abc0672d1a7293f9856db8f9407dc05f6f373a2d9246752

    # the hypothesis on the private exponent (the theoretical maximum is 0.292)
    #delta = .18 # this means that d < N^delta
    delta = .18
    #
    # Lattice (tweak those values)
    #

    # you should tweak this (after a first run), (e.g. increment it until a solution is found)
    m = 4 # size of the lattice (bigger the better/slower)

    # you need to be a lattice master to tweak these
    t = int((1-2*delta) * m)  # optimization from Herrmann and May
    X = 2*floor(N^delta)  # this _might_ be too much
    Y = floor(N^(1/2))    # correct if p, q are ~ same size

    #
    # Don't touch anything below
    #

    # Problem put in equation
    P.<x,y> = PolynomialRing(ZZ)
    A = int((N+1)/2)
    pol = 1 + x * (A + y)

    #
    # Find the solutions!
    #
```

```
    # Checking bounds
    if debug:
        print "=== checking values ==="
        print "* delta:", delta
        print "* delta < 0.292", delta < 0.292
        print "* size of e:", int(log(e)/log(2))
        print "* size of N:", int(log(N)/log(2))
        print "* m:", m, ", t:", t

    # boneh_durfee
    if debug:
        print "=== running algorithm ==="
        start_time = time.time()

    solx, soly = boneh_durfee(pol, e, m, t, X, Y)

    # found a solution?
    if solx > 0:
        print "=== solution found ==="
        if False:
            print "x:", solx
            print "y:", soly

        d = int(pol(solx, soly) / e)
        print "private key found:", d
    else:
        print "=== no solution was found ==="

    if debug:
        print("=== %s seconds ===" % (time.time() - start_time))

if __name__ == "__main__":
    example()
```