GKCTF2020_Crypto_复现

原创

M3ng@L 于 2022-02-26 11:01:29 发布 3114 收藏

分类专栏： CTF比赛复现 文章标签： Crypto python

本文链接：https://blog.csdn.net/qq_51999772/article/details/123146784

GFCTF_Crypto_复现

小学生的密码学

keywords: 仿射密码

在线解密网址：CTF在线工具-在线仿射密码加密|在线仿射密码解密|仿射密码算法|Affine Cipher (hiencode.com)

汉字的秘密

keysowrds: 当铺密码， 变异凯撒

解密脚本

```
dic = {"王": 6,"壮": 9,"工": 4,"夫": 7,"中": 2,"由": 1,"井": 8,"土": 5,"口": 0," ": "temp","人": 3,"士": 5}
c = "王壮 夫工 王中 王夫 由由井 井人 夫中 夫夫 井王 土土 夫由 土夫 井中 士夫 王工 王人 土由 由口夫"
ls = []
for i in range(len(c)):
    ls.append(dic[c[i]])
print(ls)
temp = ""
ls2 = []
for i in ls:
    if i == "temp":
        ls2.append(temp)
        temp = ""
        continue
    else:
        temp = temp + str(i)
ls2.append("107")
print(ls2)
flag = ""
for i in ls2:
    flag = flag + chr(int(i))
print(flag)
real_flag = ""
for i in range(len(flag)):
    temp = int(ls2[i]) + i + 1
    real_flag += chr(temp)
print(real_flag.lower())
```

babyCtypto

keywords: 高位攻击

已知`p`的高128位数据，直接套高位攻击的脚本（深入浅出RSA在CTF中的攻击套路 - 先知社区 (aliyun.com)）

```
from Crypto.Util.number import *
import gmpy2

n = 0xb119849bc4523e49c6c038a509a74cda628d4ca0e4d0f28e677d57f3c3c7d0d876ef07d7581fe05a060546fedd7d061d3bc70d679b
6c5dd9bc66c5bdad8f2ef898b1e785496c4989daf716a1c89d5c174da494eee7061bcb6d52cafa337fc2a7bba42c918bbd3104dff62ecc9d
3704a455a6ce282de0d8129e26c840734ffd302bec5f0a66e0e6d00b5c50fa57c546cff9d7e6a978db77997082b4cb927df9847dfffef551
38cb946c62c9f09b968033745b5b6868338c64819a8e92a827265f9abd409359a9471d8c3a2631b80e5b462ba42336717700998ff38536c2
436e24ac19228cd2d7a909ead1a8494ff6c3a7151e888e115b68cc6a7a8c6cf8a6c005
e = 65537
enc = 14225665844801998787146630514681435136679342162133667334420591065294519310782714603633358870541995779506790
10265927017947591110174762512054442926233421448368833211155200453582818242515296522359916012961099003691114602910
70033592055768983427904835395850414634659565092191460875900237711597421272312032796440948509724492027247376113210
86781834432223645316699851280329712567925320150518290412302038140901946110411727753683571978544512012609271177920
27755969020534251543762541779286769228084913953768776391926933782289974692426984769413889916582000416031911874920
98031065800530869562704671435709578921901495688124042302500361
# p>>128<<128
high_p = 0xe4e4b390c1d201dae2c00a4669c0865cc5767bc444f5d310f3cfc75872d96feb89e556972c99ae20753e3314240a52df5dccd
076a47c6b5d11b531b92d901b2b512aeb0b263bbfd624fe3d52e5e238beeb581ebe012b2f176a4ffd1e0d2aa8c4d3a2656573b727d4d3136
513a931428b0000000000000000000000000000000

# sage
# kbits = 128
# PR.<x> = PolynomialRing(Zmod(n))
# f = x + high_p
# low_p = f.small_roots(X=2^kbits, beta=0.3)[0]
# print(low_p)
low_p = 194744276640369236134349576809641082787
p = int(low_p) + high_p
q = n // p
phi_n = (p - 1) * (q - 1)
print(phi_n)
d = gmpy2.invert(e,phi_n)
m = pow(enc,d,n)
print(m)
print(long_to_bytes(m))
```

Backdoor

keywords: ROCA漏洞

ROCA 漏洞是一个加密漏洞，它允许从具有漏洞的设备生成的密钥中的公钥中恢复密钥对的私钥。"ROCA"是"Return of Coppersmith's attack"的缩写。ROCA漏洞 - 维基百科，自由的百科全书 (wikipedia.org)

就是形如

$p = k * M + (65537^a \bmod M)$

漏洞利用思路：弱素数生成器 （RSALib） (asecuritysite.com)

github有现成的攻击脚本：GitHub - FlorianPicca/ROCA: A Sage implementation of the ROCA attack

解题代码（来自github）

```
from sage.all import *
from tqdm import tqdm

def solve(M, n, a, m):
    # I need to import it in the function otherwise multiprocessing doesn't find it in its context
    from sage.functions.coppersmith_howgrave_univariate
```

```
    from sage_functions import coppersmith_howgrave_univariate

    base = int(65537)
    # the known part of p: 65537^a * M^-1 (mod N)
    known = int(pow(base, a, M) * inverse_mod(M, n))
    # Create the polynom f(x)
    F = PolynomialRing(Zmod(n), implementation='NTL', names=('x',))
    (x,) = F._first_ngens(1)
    pol = x + known
    beta = 0.1
    t = m+1
    # Upper bound for the small root x0
    XX = floor(2 * n**0.5 / M)
    # Find a small root (x0 = k) using Coppersmith's algorithm
    roots = coppersmith_howgrave_univariate(pol, n, beta, m, t, XX)
    # There will be no roots for an incorrect guess of a.
    for k in roots:
        # reconstruct p from the recovered k
        p = int(k*M + pow(base, a, M))
        if n%p == 0:
            return p, n//p

def roca(n):

    keySize = n.bit_length()

    if keySize <= 960:
        M_prime = 0x1b3e6c9433a7735fa5fc479ffe4027e13bea
        m = 5

    elif 992 <= keySize <= 1952:
        M_prime = 0x24683144f41188c2b1d6a217f81f12888e4e6513c43f3f60e72af8bd9728807483425d1e
        m = 4
        print("Have you several days/months to spend on this ?")

    elif 1984 <= keySize <= 3936:
        M_prime = 0x16928dc3e47b44daf289a60e80e1fc6bd7648d7ef60d1890f3e0a9455efe0abdb7a748131413cebd2e36a76a355c
1b664be462e115ac330f9c13344f8f3d1034a02c23396e6
        m = 7
        print("You'll change computer before this scripts ends...")

    elif 3968 <= keySize <= 4096:
        print("Just no.")
        return None

    else:
        print("Invalid key size: {}".format(keySize))
        return None

    a3 = Zmod(M_prime)(n).log(65537)
    order = Zmod(M_prime)(65537).multiplicative_order()
    inf = a3 // 2
    sup = (a3 + order) // 2

    # Search 10 000 values at a time, using multiprocess
    # too big chunks is slower, too small chunks also
    chunk_size = 10000
    for inf_a in tqdm(range(inf, sup, chunk_size)):
        # create an array with the parameter for the solve function
        inputs = [((M_prime, n, a, m), {}) for a in range(inf_a, inf_a+chunk_size)]
```

```
        # the sage builtin multiprocessing stuff
        from sage.parallel.multiprocessing_sage import parallel_iter
        from multiprocessing import cpu_count

        for k, val in parallel_iter(cpu_count(), solve, inputs):
            if val:
                p = val[0]
                q = val[1]
                print("found factorization:\np={}\nq={}".format(p, q))
                return val

if __name__ == "__main__":
    # Normal values
    #p = 883110349387302985825786603878910566950708630745132761591801993671753009231113
    #q = 122706669547814628745942441166902931145718723658826773278715872626636030375109
    #a = 551658, interval = [475706, 1076306]
    # won't find if beta=0.5
    # p = 80688738291820833650844741016523373313635060001251156496219948915457811770063
    # q = 69288134094572876629045028069371975574660226148748274586674507084213286357069
    # #a = 176170, interval = [171312, 771912]
    # n = p*q
    n = 155189610416250748761824045853940987814871410592854559270243212767838311221687450763597803430780112164805875750724797848292586786917 39
    # For the test values chosen, a is quite close to the minimal value so the search is not too long
    roca(n)
```

```
from sage.all_cmdline import *

def coppersmith_howgrave_univariate(pol, modulus, beta, mm, tt, XX):
    """
    Taken from https://github.com/mimoo/RSA-and-LLL-attacks/blob/master/coppersmith.sage
    Coppersmith revisited by Howgrave-Graham

    finds a solution if:
    * b|modulus, b >= modulus^beta , 0 < beta <= 1
    * |x| < XX
    More tunable than sage's builtin coppersmith method, pol.small_roots()
    """
    #
    # init
    #
    dd = pol.degree()
    nn = dd * mm + tt

    #
    # checks
    #
    if not 0 < beta <= 1:
        raise ValueError("beta should belongs in [0, 1]")

    if not pol.is_monic():
        raise ArithmeticError("Polynomial must be monic.")

    #
    # calculate bounds and display them
    #
    """
    * we want to find g(x) such that ||g(xX)|| <= b^m / sqrt(n)
```

```
    * we know LLL will give us a short vector v such that:
    ||v|| <= 2^((n - 1)/4) * det(L)^(1/n)

    * we will use that vector as a coefficient vector for our g(x)

    * so we want to satisfy:
    2^((n - 1)/4) * det(L)^(1/n) < N^(beta*m) / sqrt(n)

    so we can obtain ||v|| < N^(beta*m) / sqrt(n) <= b^m / sqrt(n)
    (it's important to use N because we might not know b)
    """
    #
    # Coppersmith revisited algo for univariate
    #

    # change ring of pol and x
    polZ = pol.change_ring(ZZ)
    x = polZ.parent().gen()

    # compute polynomials
    gg = []
    for ii in range(mm):
        for jj in range(dd):
            gg.append((x * XX) ** jj * modulus ** (mm - ii) * polZ(x * XX) ** ii)
    for ii in range(tt):
        gg.append((x * XX) ** ii * polZ(x * XX) ** mm)

    # construct lattice B
    BB = Matrix(ZZ, nn)

    for ii in range(nn):
        for jj in range(ii + 1):
            BB[ii, jj] = gg[ii][jj]

    BB = BB.LLL()

    # transform shortest vector in polynomial
    new_pol = 0
    for ii in range(nn):
        new_pol += x ** ii * BB[0, ii] / XX ** ii

    # factor polynomial
    potential_roots = new_pol.roots()

    # test roots
    roots = []
    for root in potential_roots:
        if root[0].is_integer():
            result = polZ(ZZ(root[0]))
            if gcd(modulus, result) >= modulus ** beta:
                roots.append(ZZ(root[0]))
    return roots
```

需要在`sagemath`中运行这两个攻击脚本（且放在同一目录下，因为有引入模块的部分）

可以参考paper：(12条消息) 在sagemath中运行python文件_M3ng@L的博客-CSDN博客

得到结果

```
p=338661997705111463730332851917362716581783217984521264076719700194 1
q=458243356112785531080529445665799328178266264511654302453705168247 9
```

那么剩下的就是常规RSA解密了

```
from Crypto.PublicKey import RSA
from Crypto.Util.number import *
import base64
import gmpy2

# with open('C:\\Users\\Menglin\\Desktop\\pub.pem' ,'r') as f:
#     key = RSA.import_key(f.read())
#     e = key.e
#     n = key.n
#     print(key.size_in_bits())
# with open("C:\\Users\\Menglin\\Desktop\\flag.enc") as f:
#     c = base64.b64decode(f.read())
#     c = bytes.decode(c)
# print("e = {}".format(e))
# print("n = {}".format(n))
# print("c = {}".format(c))

e = 65537
n = 155189610416250748761824045853940987814871410592854559270243212767838311221687450763597803430780112164805875750724797848292586786917 39
c = 0x02142af7ce70fe0ddae116bb7e96260274ee9252a8cb528e7fdd29809c2a6032727c05526133ae4610ed944572ff1abfcd0b17aa22ef44a2
# print(n.bit_length())
p=338661997705111463730332851917362716581783217984521264076719700194 1
q=458243356112785531080529445665799328178266264511654302453705168247 9

phi_n = (p-1)*(q-1)
d = gmpy2.invert(e,phi_n)
m = pow(c,d,n)
print(long_to_bytes(m))
```